Industrialised Application of Combinatorial Optimization

Jesper Hansen

Kgs. Lyngby 2003
IMM-PHD-2003-120

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Contents

Preface v

Resumé vii

Abstract ix

1 Introduction 1
- 1.1 The CIAMM Project 1
- 1.2 Industrial Applications 2
- 1.3 Thesis Structure 6

2 Modelling Combinatorial Optimisation Problems 7
- 2.1 Introduction 7
- 2.2 The Model Concept 8
- 2.3 Model Categorisation 11
- 2.4 Object Oriented Modelling 12
- 2.5 Logical and Mathematical Modelling 16
- 2.6 Simulation Modelling 26
- 2.7 Local Search Modelling 31
- 2.8 Summary 32

3 Solving Optimisation Models 33
- 3.1 Introduction 33
- 3.2 Heuristics 34
- 3.3 Mathematical and Constraint Logic Programming 37
- 3.4 Simulation 45

3.5 Summary 47

4 A Methodology for Combinatorial Optimisation Projects 49
4.1 Background 49
4.2 The Phases of the Methodology 52
4.3 Summary 54

5 Summary of Case Studies 57
5.1 Crane Scheduling for a Plate Storage in a Shipyard 57
5.2 Solving the non-oriented three-dimensional bin packing problem with stability and load bearing constraints 62
5.3 Summary 68

6 Conclusion 69
6.1 Overview 69
6.2 Results 70
6.3 Perspectives and Future Research 72

A An Object-oriented Local Search Framework 75

Bibliography 81

Papers

I Crane scheduling for a Plate Storage in a Shipyard: Modelling 87
1 Introduction 89
2 Problem Description 90
3 Related Research 94
4 Model 98
5 Conclusion 112
References 112

II Crane scheduling for a Plate Storage in a Shipyard: Solving the Problem 115
1 Introduction 117
2 Planning Procedure 119
3 The Control System 148
4 Conclusion 152
References 153

III Crane scheduling for a Plate Storage in a Shipyard: Experiments and Results **155**
1 Introduction 157
2 Comparison of Search Methods 159
3 Experiments and Results 169
4 Conclusion 178
References 179

IV Crane scheduling for a Plate Storage **181**
1 Introduction 183
2 The Problem 183
3 Related Research 185
4 Objective Function 186
5 Solution Procedure 188
6 Results 194
7 Conclusion 196
References 196

V Solving the non-oriented three-dimensional bin packing problem with stability and load bearing constraints **199**
1 Introduction 201
2 Column Generation 204
3 Pricing Problem 205
4 Branch & Price 209
5 Lower Bounds 213
6 The On-Line Algorithm 215
7 Stability Issues 216
8 Load Bearing Constraints 217
9 Experiments and Results 217
10 Conclusion 224
A Construction of Bang & Olufsen instances 226
References 226

VI Constraint Programming versus Mathematical Programming **231**
1 Introduction 233
2 Comparing Branch & Bound and CLP 234
3 Modelling and Solving in ECLiPSe 239
4 Conclusion 248
References 249

Preface

This dissertation is submitted to Informatics and Mathematical Modelling at the Technical University of Denmark in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Professor Jens Clausen has supervised the work.

Acknowledgements

I would like to thank my supervisor Jens Clausen for giving me the opportunity to pursue the Ph.D. title and all the partners in the CIAMM project. Special thanks go to Andrea Carugati, Jesper Saxtorph, Torben F. H. Kristensen, Troels M. Range, Søren Yde and Claus Risager for the co-operation during the three-year project.

The Operations Research section at IMM has been a great place to work with many great colleagues both socially and intellectually. Special thanks go to Thomas Stidsen, Jesper Larsen, Allan Larsen and Tommy Thomadsen. Former members include Tim Hultberg and Brian Kallehauge with whom I enjoyed playing Badminton and Football. Thanks to all the members of ORCG and ORSCG for a lot of fun and cake.

In the summer 2002 I had a very pleasant and enlightening three-month stay at IC-Parc, Imperial College in London. Here I would like to thank all the staff at IC-Parc including Mark Wallace and Helmut Simonis for making it possible.

Thanks to Søren Thomsen and Peter Fausbøll and everybody else who have read and commented on papers and parts of the thesis. Finally I would like to thank my family for all the support and patience you have given me. On July 1st 2000 Britt became my wife and June 3rd this year she gave birth to our first child, Freja. The last three years have truly been an outstanding time of my life.

Kgs. Lyngby, 19th November 2003

Jesper Hansen

Resumé

Denne afhandling er et af resultaterne af CIAMM (Center for Industrialiseret Anvendelse af Matematiske Modeller). Centeret består af firmaer fra industrien bl.a. Odense Stålskibsværft (OSS), Bang & Olufsen (B&O) og Teknologisk Institut samt 4 institutter fra 3 universiteter inklusiv Informatik og Matematisk Modellering ved Danmarks Tekniske Universitet.

Formålet med centeret har været at industrialisere anvendelsen af matematisk modellering og metoder til løsning af disse typer modeller i produktionsvirksomheder. I afhandlingen betragtes to problem cases stillet af firma partnerne og en metodik er udviklet baseret på erfaringer fra arbejdet med de to cases.

Den første case er en undersøgelse af mulige forbedringer i håndteringen af stålpladelageret ved OSS. To portalkraner flytter pladerne ind på, rundt og ud af lageret, når de skal anvendes i produktionen. Forskellige principper for at organisere lageret og forskellige metoder er blevet sammenlignet til løsning af problemet. Vore resultater indikerer en potentiel reduktion i pladeflytninger på 67% og reducering i tid på 39% i forhold til nuværende praksis.

Den anden case stillet af B&O omhandler af pakning af 3-dimensionelle rektangulære emner i rektangulære kasser og kasser på paller. Målet er at reducere det nødvendige antal paller og dermed antallet af nødvendige lastbiler til at transporte pallerne. Rotation af emnerne er tilladt og pakningerne skal være mulige at pakke og stabile. Emnernes bæreevne er også taget i betragtning. En on-line heuristik og en eksakt metode er blevet udviklet og sammenlignet på instanser genereret ud fra virkelige data og på benchmark instanser. On-line algoritmen opnår konsekvent gode løsninger indenfor sekunder. Den eksakte metode er i stand til at forbedre løsningerne, men en anseelig mængde cpu tid er påkrævet.

Den tredje del af afhandlingen beskriver en metodik til udvikling af industriel kombinatorisk optimering software. Den typiske fremgangsmåde i software udvikling er at investere væsentlige resurser i begyndelsen af projektet for at reducere risikoen for at løse det forkerte problem. Her tages der dog ikke hensyn til at det er svært for brugerne at specificere, hvad de ønsker af software systemet, før de faktisk ser det. I projekter med kombinatorisk optimering er det, på grund af optimeringsteknikkernes kompleksitet, specielt svært for brugerne og udviklerne at udveksle den nødvendige viden. For at facilitere vidensudveksling foreslår vi en proces med korte udviklingscykler efterfulgt af diskussioner baseret på software prototyper. Det kræver en signifikant længere tidsperiode at udvikle kernen til kombinatorisk optimering, hvilket ikke er muligt i kombination med vores fremgangsmåde med korte udviklingscykler. Vi foreslår en måde at opdele udviklingen af optimeringskernen i mindre opgaver håndterbare i en udviklingscykel og som kan visualiseres for brugeren.

De tre hovedområder er suppleret med et kapitel, der introducerer modellering i software udvikling, Kombinatorisk Optimering og relationerne mellem de forskellige modeltyper. Endeligt er der inkluderet et kapitel om relevante optimeringsmetoder specielt til løsning af praktiske planlægningsproblemer.

Abstract

This thesis is one of the results of CIAMM (Center for Industrialised Application of Mathematical Modelling). The center is composed of industrial companies e.g. Odense Steel Shipyard (OSS), Bang & Olufsen (B&O), Danish Technological Institute and 4 departments from 3 universities including Informatics and Mathematical Modelling at Technical University of Denmark.

The purpose of the consortium has been to industrialise the application of mathematical modelling and methods for solving these models in the manufacturing industry. Two problem cases posed by the industrial partners have been considered in the thesis and a methodology has been developed based on the experience gained from working on the cases.

The first case is an investigation of possible improvements in methods for handling the storage of steel plates at OSS. Two gantry cranes move the plates into, around and out of the storage when needed in production. Different principles for organising the storage and different methods have been compared for solving the problem. Our results indicate a potential reduction in plate movements by 67% and reduction in time by 39% compared to current practices.

The second case has been posed by B&O concerning packing of 3-dimensional rectangular items in rectangular boxes and boxes on pallets. The goal is to reduce the necessary number of pallets and evidently the number of necessary vehicles transporting the pallets. Rotation of the items is allowed and the packings must be pack-able and stable. The load bearing of the items is taken into account as well. An on-line heuristic and an exact method have been developed and compared on instances generated from real-life data and on benchmark instances. The on-line algorithm consistently reaches good solu-

tions within seconds. The exact method is able to improve the solutions, but a significant amount of computation time is required.

The third part of the thesis describes a methodology for developing industrial Combinatorial Optimisation software. The usual approach in software development is to invest significant resources in the beginning of the project in order to reduce the risk of solving the wrong problem. However, this approach fails to consider that it is difficult for the users to specify what they request from the software before they actually see it. In Combinatorial Optimisation projects it is particularly difficult for the users and developers to exchange the necessary knowledge caused by the complexity of the optimisation techniques. To facilitate knowledge exchange we suggest a process of short development cycles followed by discussions based on software prototypes. It requires a significantly longer period of time to develop the Combinatorial Optimisation core, which is not possible in combination with our approach of short development cycles. We propose a way to subdivide the development of the Combinatorial Optimisation core into smaller tasks manageable in a development cycle and which can be visualised for the user.

The three main parts are supplemented with a general chapter introducing modelling in software development, Combinatorial Optimisation and the relation between the different model types. Finally, a chapter on relevant optimisation methods specifically for solving real-world planning problems is included.

CHAPTER 1

Introduction

The production costs in Denmark are relatively high caused by the high wages. Other countries with relatively low wages (e.g. Asia) threaten the competitive power of Danish industry. The ability to utilise the resources in the best possible way is therefore of prime importance to stay competitive.

In the last decade the progress within Information Technology (IT) has been tremendous. The increase in hardware performance and decrease in prices has led to the IT revolution. PC's are everywhere and they are connected via the Internet. Now it is only a question of implementing the software for solving any imaginable problem. Unfortunately developing software is not as easy as that. History tells that it is difficult and it requires both technical, communicative skills and a significant degree of experience.

The subject of this thesis is the development of software to solve planning problems in industry.

1.1 The CIAMM Project

The Ph.D. project has been carried out as part of the Center for Industrialised Application of Mathematical Models (CIAMM). The CIAMM consortium is composed of the industrial companies Odense Steel Shipyard (OSS), Bang & Olufsen (B&O) and Aage Oestergaard, the consultancy company Danish Technological Institute and university partners from Aalborg University Cen-

ter (AUC), Copenhagen Business School (CBS) and Technical University of Denmark (DTU). Two departments from DTU participated, Informatics and Mathematical Modelling (IMM) and Manufacturing Engineering and Management (IPL).

The purpose of CIAMM is to facilitate technology transfer from the universities to the industry. The particular technology of interest in the project is decision support systems based on mathematical models and methods for solving industrial planning and control problems. The methods are used for utilising the given resources in the best possible way.

IMM has the responsibility of the planning part of the project, which is the subject of this thesis. AUC is responsible for the control part, IPL for investigating possible process and business improvements and finally CBS has attended the organisational problems in connection with integrating decision support systems in industrial companies.

The plan of CIAMM has been for the university partners to work on real-life cases posed by the industrial partners. The experiences gained from the work on the cases should eventually turn into a process description or methodology for solving industrial planning and control problems. No precise requirements for the document were put forward, which obviously makes it difficult to determine whether or not the goal has been achieved. This thesis is considered a step towards fulfilling this goal. The methodology developed is the subject of paper VII: "CIAMM: A Knowledge Driven Methodology for Development of Combinatorial Optimisation based Information Systems". The methodology will also form the structure of the thesis.

1.2 Industrial Applications

Many industrial applications can be solved with planning methods. Examples are Production Scheduling and Sequencing, Job-Shop Scheduling, Staff and Shift Scheduling, Vehicle Routing, Crane and Robot Scheduling as well as Cutting, Nesting and Packing problems. We distinguish between *prototype problems* considered in academia and *real-life problems* from industry. Prototype problems are simplified versions of real-life problems. A number of complicating factors are ignored or assumed unimportant to include.

In the following we introduce two cases considered in the project. The first case is a crane scheduling problem at OSS and the second a packing problem at B&O.

1.2.1 The Crane Scheduling Problem

OSS is the producer of the world's largest container ships similar to the one shown in figure 1.1. These are built from steel plates, cut up in smaller pieces and afterwards welded together into sections, then blocks and finally the blocks are welded together to produce the final ship.

Figure 1.1: Ship produced by OSS.

The case we are considering is the process from arrival of the steel plates, storage in the plate inventory and delivery when the plates are due in production. The plates are usually delivered by ship and placed on the quay by a tower crane as shown in figure 1.2 on the next page. In the inventory the plates are placed in 8×32 stacks and are moved from stack to stack by two gantry cranes sharing tracks. When a plate is due in production it should be placed on a conveyer belt by one of the cranes which is shown in figure 1.3 on page 5. After that the plates are passing through different production processes ending with the cutting up of the plates.

The purpose of our work has been to improve the current practices at the inventory and specifically the planning and control of the cranes in the

Figure 1.2: Arrival of plates to the plate storage.

inventory. The goal has been to reduce the number of times a plate was moved on average from the current 12. The project is considered successful if a target of 6 movements is reached.

Three papers I, II and III with the common title “Crane scheduling for a Plate Storage in a Shipyard” discuss the aspects of “Modelling”, “Solving” and “Experiments and Results” for this particular case. Paper IV is an earlier less detailed paper, which is included for completeness and a few additional interesting results.

1.2.2 The Packing Problems

Transportation costs contribute a significant part of the total costs in today’s manufacturing companies – B&O is no exception. The assembled goods are packed in boxes, which are again packed on pallets. The pallets are finally packed on vehicles transporting the goods to stores all over Europe. A significant amount of money can be saved if the number of necessary trucks is reduced by better solving the sequence of packing problems.

Figure 1.3: The storage of plates.

In the pallet-packing problem the boxes arrive at the packing area on a conveyer belt. The sequence of the boxes is not known a priori and the next box to put on the two available pallets must be picked from the first 20 boxes on the belt. After a box has been picked the next box will arrive, etc. The problem is hence *dynamic* and the planning must be done *on-line*.

The problem of packing goods in boxes is on the other hand *static*: All the goods to be packed are known a priori and they can be picked in any order. The packing plan can hence be found *off-line* or in other words before the plan is actually executed. The time perspective has a significant impact on the suitable set of methods to apply for solving a given problem.

Paper V titled "Solving the non-oriented three-dimensional bin packing problem with stability and load bearing constraints" deals with this case.

1.3 Thesis Structure

The main focus of this thesis is modelling and solving real-life Combinatorial Optimisation problems and providing a methodology for developing software where Combinatorial Optimisation problems are solved. The following 7 papers contribute to fulfilling this goal:

Paper I: *"Crane scheduling for a Plate Storage in a Shipyard: Modelling"*

Paper II: *"Crane scheduling for a Plate Storage in a Shipyard: Solving"*

Paper III: *"Crane scheduling for a Plate Storage in a Shipyard: Experiments and Results"*

Paper IV: *"Crane scheduling for a Plate Storage"*

Paper V: *"Solving the non-oriented three- dimensional bin packing problem with stability and load bearing constraints"*

Paper VI: *"Constraint Programming versus Mathematical Programming"*

Paper VII: *"CIAMM: A Knowledge Driven Methodology for Development of Combinatorial Optimisation based Information Systems"*

The chapters 2 to 5 following this introduction will serve mainly as a supplement and introductory material for the papers. In chapter 2 we discuss modelling aspects in Combinatorial Optimisation where amongst others our industrial cases will be used as examples. Object-Oriented modelling is a fundamental part of software development and also applies for software where Combinatorial Optimisation is a substantial part. Object-Oriented modelling is therefore discussed in chapter 2 as well. In chapter 3 the various optimisation methods suitable for solving Combinatorial Optimisation models is discussed. The methodology is discussed in chapter 4 and the two industrial cases are summarised in chapter 5. Finally the thesis is concluded in chapter 6. The bibliography of the chapters is found on page 81.

CHAPTER 2

Modelling Combinatorial Optimisation Problems

A model is a simplified representation of a system where a system is a collection of objects and processes that interact with each other. This chapter focuses on the use of models when developing software to solve planning problems in the industry.

Section 2.1 is an introduction to mathematical models for describing planning problems. In section 2.2 we discuss different model types and the purposes for using models in software development and in particular for solving planning problems. Section 2.3 describes different categories of models for planning problems. Object-oriented modelling is widely used in all phases of software development. In section 2.4 we describe the basic components and propose supplementary issues to consider when developing software for solving planning problems. After having laid the foundation, the remaining part of the chapter is focused on modelling planning problems with different model types: mathematical and logical models, simulation models and models to be solved with local search heuristics.

2.1 Introduction

When defining a planning problem, it is based on our perception of the real world and our view of the problem to be solved. The defined problem is a

simplification or abstraction of the real world problem in that we only include what is considered relevant to define the problem.

Planning problems are characterised by a set of decisions taken to optimise a given set of objectives. Often the decisions are yes-no decisions or decisions determining the order of tasks to be executed e.g. on machines. Usually the yes-no decisions are modelled with 0-1 variables also called binary variables. These types of problems are referred to as *Combinatorial Optimisation* problems. Generally we assume that the variables are discrete and bounded.

The number of potential solutions grows exponentially with increasing number of variables. For instance a problem of scheduling n tasks can be modelled with $\frac{n(n-1)}{2}$ binary variables – one for each pair of tasks: x_{ij} is 1, if task i is before j, and 0 if j is before i. In the case of 5 tasks, we have 10 binary variables, which result in $2^{10} = 1024$ potential solutions. Doubling the number of tasks to 10 result in 45 variables and 2^{45} potential solutions, which is a 13 digit number. This is an example of the combinatorial explosion.

Besides the decision variables, a Combinatorial Optimisation problem consists of constraints defining the *feasible* set of solutions. We seek to maximise or minimise an objective function or a function composed of several objective functions. The latter type of problem is referred to as a *Multi-criteria Optimisation* problem.

We will in this thesis not restrict ourselves to any special types of constraints or objective functions. Given a constraint over a subset of the decision variables, the only requirement is that for a given solution it should be possible to check if the constraint is satisfied or not. Regarding objective functions, we will throughout the thesis indirectly assume that the objective functions can be transformed into one objective function. The problem setting is the following:

$$\begin{aligned} \min \quad & f(x) \\ & x \in S \\ & x \in D^n \end{aligned}$$

where $D^n = \{x : x, l, u \in \mathbb{Z}^n, l \leq x \leq u\}$. We have already introduced the special type of model where x must be binary, i.e. $D^n = \mathbb{B}^n$.

2.2 The Model Concept

As stated earlier, a model is a simplified representation of a system where a system is a collection of objects and processes that interact with each other.

Models can be divided into different types. Scale models are physical models with the same shape as the original object, but scaled up or down in size. Schematic models use diagrams and symbols to describe aspects of the original object. Mathematical and logical models describe the system of interest with use of variables, relations and equations. Other types of models can undoubtedly be identified, but we are focused on models facilitating the development of software systems where Combinatorial Optimisation is a substantial component. There are several purposes for building models:

- Models may increase the understanding of systems.
- Models facilitate communication between people.
- Models are a way of documenting collected knowledge.
- Models are the basis for further model building and system development.
- Models are used for making decisions.

Figure 2.1 on the next page illustrate the different types of models one might use when developing software with a Combinatorial Optimisation component. Based on the modeller's perception of the real world a so-called *conceptual object model* is developed. The model is built to understand the domain in which the software is going to be used. A conceptual object model should hence describe the relevant objects in the real world and their interaction. This is often done with use of different diagrams such as class/object, state and activity diagrams part of the Unified Modelling Language (UML) [17]. In section 2.4 we elaborate on the object model concept and state diagrams are illustrated in section 2.6 for simulation modelling.

The business problem defines the problem to be solved with use of Combinatorial Optimisation and the software system. Based on the problem definition we can develop a *usage model.* The usage model describes how the software system is going to be used including organisational issues and the actual user interface. *Use cases*, which are part of the UML, or *user stories* described in Beck [4] can be used for modelling the usage of the system. A user story is a description of a set of functionality that adds value to the customer.

The objectives defined by the business problem are also the objectives of the *optimisation model* and the usage and conceptual object model determine the constraints of the system and have major influence on the optimisation model. The optimisation model is usually more formally stated in a *mathematical* or *logical* form.

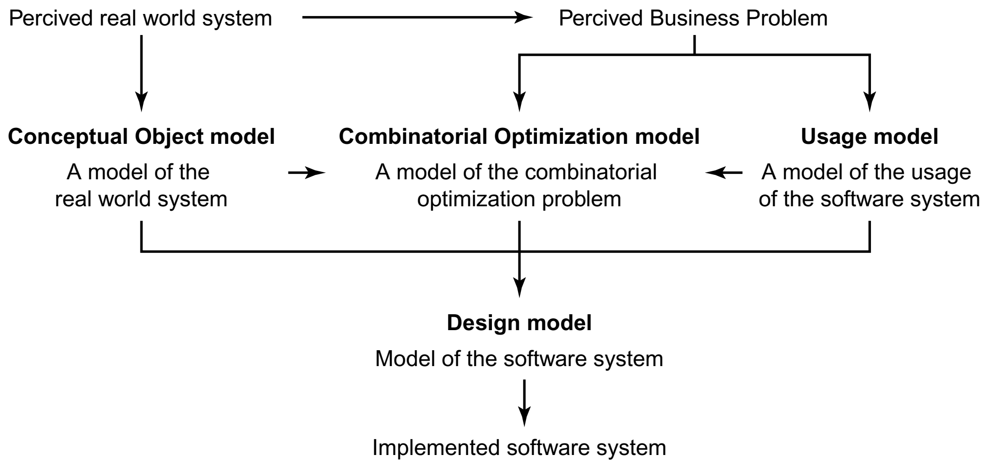

Figure 2.1: Model building in optimisation software development.

The *design model* is a model of the software system facilitating the implementation of the software. The design model is based on the three models mentioned above and is often an object-oriented model. The remaining part of this chapter is focused on conceptual and mathematical/logical modelling.

Usually mathematical/logical modelling is not discussed in connection with object-oriented modelling. In mathematical/logical modelling the problem is described by means of variables and equations while in object-oriented modelling a system is described by interacting objects. We believe that there are several good reasons for building an object-oriented model in connection with building a mathematical/logical model when developing software with a Combinatorial Optimisation component:

- The object model is a basis for discussion and understanding of the real-world system, which is useful when developing an optimisation model.
- The variables of the mathematical/logical model can often be mapped to an object in the object model.
- As the Combinatorial Optimisation component is part of a larger software system, an interface layer is required between the optimisation method and the remaining system.
- Objects are natural building blocks of other methods such as simulation and local search heuristics. The object model is therefore a unifying

model type.

2.3 Model Categorisation

Both researchers and practitioners are often focused on particular optimisation methods and model types. The risk is here that when faced with an industrial problem, the model type does not really fit and is made to fit neglecting some of the relevant issues. When solving industrial problems a broad knowledge of and willingness to use other perhaps more appropriate approaches is necessary to be successful. The purpose of this chapter is to give a broad view on the different model types and modelling aspects available when faced with an optimisation problem.

Models can be divided into categories. The model characteristics we consider are the possible presence of randomness in the model and the time factor. Figure 2.2 illustrates the choice of a deterministic and a stochastic model. In deterministic models everything is predictable and known while in stochastic models there is some degree of uncertainty about the system, or some factors cannot be controlled in the system.

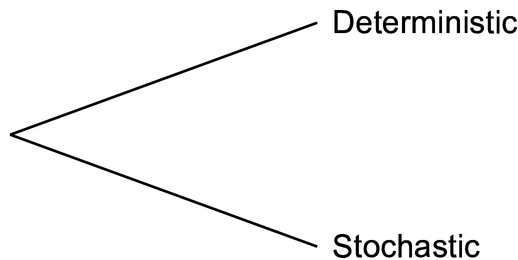

Figure 2.2: Deterministic and stochastic models.

Figure 2.3 illustrates the choice between a static and dynamic model. In dynamic models the time aspect is considered explicitly for capturing the problem characteristics. Note that both static and dynamic models can be either deterministic and stochastic. Dynamic models are subdivided into discrete and continuous models. In discrete models the time is assumed to pass in discrete time intervals either of equal size referred to as time driven or the time is driven by events occurring in the system. In continuous models the state of the system changes continuously. In this thesis we consider dynamic discrete event driven deterministic and stochastic models and static deterministic

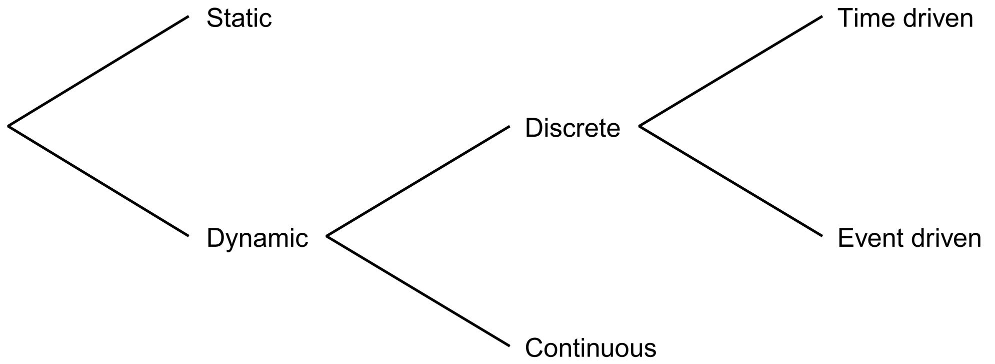

Figure 2.3: Static and dynamic models.

models only.

2.4 Object Oriented Modelling

In this section we briefly introduce the fundamental components and concepts of object-oriented modelling. The material covered is used throughout the thesis. For further details on object-oriented modelling we refer to Bennett et. al. [6].

2.4.1 Objects and Classes

Objects or *entities* are physical and conceptual things we find around us. In our industrial case of scheduling cranes, the following objects can be identified (without being an exhaustive list): gantry cranes, plates, stacks, tower cranes, conveyer belt, due date, production processes, storage and plate movements. Some of them are clearly physical objects while for instance a due date is the concept of a time in the future where the plate is due in production.

Objects are thought of as having a *state*. The state of an object is the condition of the object. For instance a crane can either be holding a plate or not holding a plate and it can additionally be moving or not moving. Another characteristic of an object is its *behaviour*. A crane can move, it can lift a plate, drop a plate, etc. The objects are hence related to each other and the interactions of the objects create the behaviour of the entire system. Attributes of an object define certain useful information to describe the object. For a plate this could for instance be the size.

A *class* can be thought of as a template or pattern for similar types of objects. An object is an *instance* of a class. Another way to explain it is to think of a class as everything necessary to create an object or instance corresponding to the given template. Again in our case, we have a gantry crane class, which is a template for all instances of this class. There are two gantry cranes on the storage, which are then objects corresponding to the template of the gantry crane class.

2.4.2 Encapsulation and Operations

One of the fundamental concepts of object-orientation is *encapsulation*. Basically it is only the creator of the object, who knows about the internals of the object. Users of the object must interact with the object through the *interface* of the object. All other details are encapsulated in the object.

In the interface of an object are *operations* callable from other objects. Typically we distinguish between operations enquiring the state of the object, changing the state and iterating over object components.

2.4.3 Composition, Inheritance and Polymorphism

An object composed of other objects is called a *composite*. For example a plate *has a* due date, a storage is composed of stacks and a stack can hold several plates. Often it is useful to distinguish between objects that can exist outside the composite and objects that cannot. Plates for instance can exist outside a stack.

Inheritance is another relation here between classes instead of objects. The characteristics of a class can be inherited from another class. In the storage case, assume we have a class, let us call it "PlateHolder", describing objects that are able to "hold" plates. Then a crane *is a* plate holder. Stacks and conveyer belts are also plate holders, since they can hold plates as well. A class inheriting characteristics is referred to as a *specialisation* or *subclass* of a *superclass* or *generalisation*.

Figure 2.4 on the next page is a so-called class diagram showing the relations between classes for the crane scheduling case. Note that all plate holders are types of container objects where the container has a specific size. For instance the size of the container of the crane is one, since it can only hold one plate at a time indicated on the arrow from the crane to the plate class. All plate holders can "push" a plate into the container or "pop" a plate out of the container. For stacks the plates are pushed and popped after the last in first out (LIFO) principle while queues has the first in first out (FIFO) principle.

The operation "getPlate" queries the plate holder for the next plate that would be popped. When calling these operations for a plate holder the correct behaviour depends on the actual plate holder and is hence implemented in the corresponding subclasses. When an operation is called on a plate holder the appropriate implementation is called. This is an example of *polymorphism*, which means multiple meanings or forms. When calling the method pop on a plate holder the caller does not need to know whether the plate holder is a stack or a queue for the appropriate operation to be performed. Finally, a plate holder has a position in the storage and a crane can move between positions with the move operation, which is specific for the crane. For more details on the crane scheduling model, we refer to paper I.

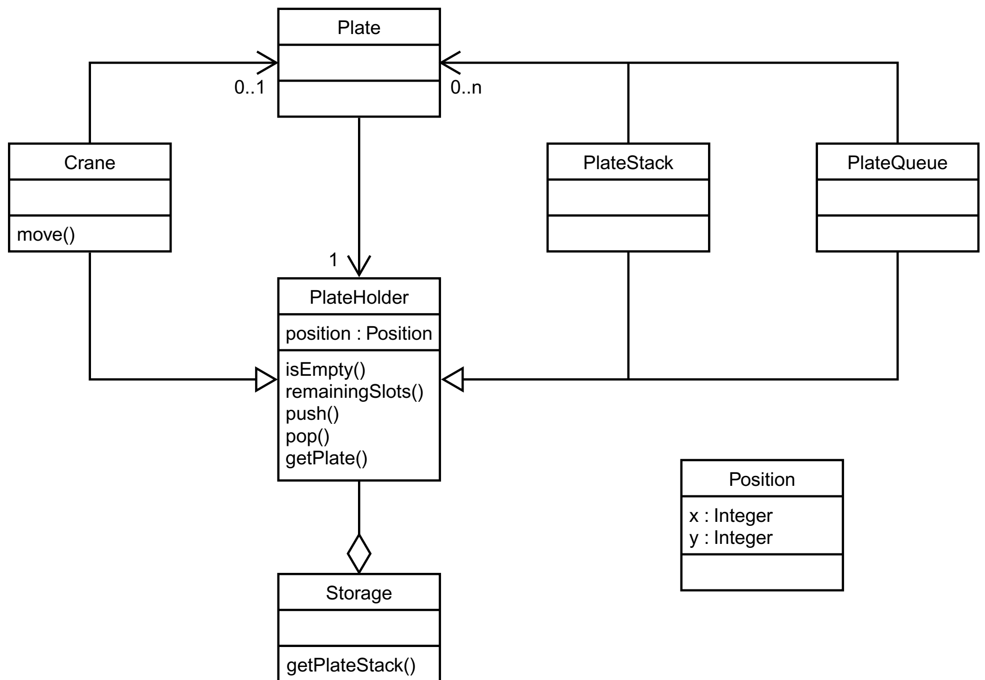

Figure 2.4: Class diagram for the crane scheduling case.

Class diagrams are part of the Unified Modelling Language (UML) described for instance in Fowler and Scott [17]. UML includes several different diagram types for object-oriented modelling.

2.4.4 Supplementary Issues

The class diagram can be used in all phases of software development from the conceptual level to design. When building optimisation models the class diagram is not sufficient for capturing all relevant issues. In table 2.1 we have listed a number of points to consider when collecting information for building optimisation models. Some of the points are documented in the class diagram, while others are new or supplemented with further details. In the table are points related to the crane class listed to give an example of its use.

Class	*Crane*
Deterministic static input	*Start position, costs of lifting/dropping plates and moving crane.*
Deterministic dynamic input	*Request for movement jobs and changes in movement jobs.*
Stochastic static input	
Stochastic dynamic input	*Distributions over movement speed in X and Y direction and lift/drop times.*
Disruptions	*Breakdown/halt of crane.*
Constraints	*Avoid collision with other cranes. Can only hold one plate at a time.*
Output	*Sequence of movements performed by the crane.*
Objectives	*Minimise number of lift/drops, moved distance and duration of movement sequence.*

Table 2.1: Checklist for supplementary modelling issues.

The first four points relate to the input parameters of the class resulting in an indication of the model type to choose discussed earlier. The disruptions relate to any special events or exceptions, which must be handled in the system.

Certain constraints can be handled directly in the class diagram as for instance the fact that a crane can hold zero or one plate. Avoidance of crane collision is more difficult to express. Generally with more complex constraints and more objects one should refrain from expressing constraints in the class

diagram, since the diagram becomes too complicated or impossible to both draw and read.

Output describes how objects of this class relate to the "variables" of the optimisation problem. For this problem the crane performs the sequence of movements, which are to be optimised. Finally it is described how the object of the class influence the objective function.

The checklist is an informal way of ensuring that relevant information is collected, useful in further modelling (not saying that all relevant information is collected in table 2.1 for this problem).

2.5 Logical and Mathematical Modelling

The model class of Combinatorial Optimisation introduced in section 2.1 falls under the paradigm of Constraint Logic Programming (CLP). CLP was originally used for solving *constraint satisfaction* problems, where the goal was to find a solution satisfying all constraints, i.e. a feasible solution. Lately, focus has shifted toward solving optimisation problems as well.

The model constructs used to build Combinatorial Optimisation models solvable with Mathematical Programming (MP) techniques constitute a subset of the constructs of CLP. In *Linear Programming* (LP) problems, the objective function and constraints are all linear. If the variables additionally are restricted to be integral, the problem is called an *Integer Linear Programming* (IP) problem and if the variables are binary it is called a *Binary Integer Programming* problem:

$$\begin{aligned} \min \quad & cx \\ & Ax \geq b \\ & x \in \mathbb{B}^n \end{aligned}$$

More general functions, for instance convex functions, are applicable as well, but we will only consider linear functions.

In the remainder of this section we discuss different modelling issues and possibilities with use of different examples. The Travelling Salesman Problem and The Warehouse Location Problem are used to illustrate the differences between modelling within the MP paradigm and the CLP paradigm. This is also the subject of paper VI. The 3-Dimensional Packing Problem and The Set Partitioning Problem are discussed to introduce the modelling ideas applied in the packing case. In section 3.3 of chapter 3 we present methods for solving IP and CLP models.

2.5.1 The Travelling Salesman Problem

Here we consider an example of modelling the famous Travelling Salesman Problem (TSP) with both an IP model and a CLP model. A salesman starting at city 1 must visit all cities $2, ..., n$, once and return to city 1 again in a tour minimising the distance travelled. The distance from city i to city j is c_{ij}.

The motivation for considering the TSP is that it can be identified as a subproblem of many industrial applications: Robot welding, soldering and cutting, crane and machine scheduling, vehicle routing, etc.

For the IP model, binary variables are used: x_{ij} is 1, if the salesman visits city j directly after city i and 0 otherwise. The objective can then be formulated linearly in the following way:

$$\min \sum_{i,j} c_{ij} x_{ij}$$

The following "assignment" constraints ensure that every city is preceded and followed by exactly one other city:

$$\sum_{i} x_{ij} = 1, \quad \forall j$$

$$\sum_{j} x_{ij} = 1, \quad \forall i$$

To avoid "sub-tours" where the cities are covered by disjoint tours, additional *sub-tour elimination constraints* are necessary. For every disjoint pair of subsets $V, W \subset \{1, ..., n\}$ of cities, we have the following constraint:

$$\sum_{i \in V} \sum_{j \in W} x_{ij} \geq 1$$

Basically for the tour to be connected there must be at least one city from subset V to be visited directly before a city in subset W. Note that the same is true when the roles of subsets V and W are exchanged. Unfortunately there is an exponential number of constraints of this type expressed in the number of cities. Typically the model is solved in the following way: First all or most of the sub-tour constraints are removed and afterwards iteratively adding them when needed. This procedure is called *Branch & Cut* and is described in section 3.3.1 of chapter 3.

A CLP model of the TSP uses variables $y_k \in \{1, ..., n\}$ to represent the k'th city visited for $k = 1, ..., n$. Each city must be visited exactly once, which

can be formulated as a set of dis-equalities:

$$y_k \neq y_l, \quad 1 \leq k < l \leq n$$

All these constraints can be formulated with only one "global" constraint:

$$\text{all}-\text{different}(y_1, ..., y_n)$$

The all-different constraint states that $y_1, ..., y_n$ must all take distinct values. Finally the objective function can be stated in the following way:

$$\min \sum_{k=1}^{n} c_{y_k, y_{k+1}}$$

where additionally $y_{n+1} = y_1$. The formulation is very compact using variable subscripts y_k and y_{k+1}, which is perfectly valid in logic models, but outside the scope of IP. Fewer variables are needed and only one constraint compared to the exponential number shown earlier. The simplicity of the model should however not mislead the reader to think that the CLP model is easier to solve. The TSP is still a difficult problem to solve. We refer to section 3.3.1 of chapter 3 for details on *global constraints* and how CLP models are solved.

2.5.2 The Warehouse Location Problem

Another example discussed in paper VI is the Warehouse Location Problem: Given is a set of customers and a set of warehouse locations. Each open warehouse can only deliver to a limited number of customers and each customer only receives goods from one warehouse. We must now decide which warehouses to open in order to minimise costs for opening the warehouses and delivering the demanded goods to the customers.

The IP model can be formulated in the following way. c_j is the cost of opening warehouse $j \in J = \{1, ..., n\}$ and d_{ij} is the cost of delivering goods to customer $i \in I = \{1, ..., m\}$ from warehouse j. The binary variable y_j is 1, if warehouse j is open and 0 otherwise, while x_{ij} is 1, if warehouse j delivers

to customer i. The number of customers serviced by warehouse j is cap_j.

$$\min \sum_{j\in J} c_j y_j + \sum_{i\in I}\sum_{j\in J} d_{ij}x_{ij} \tag{2.1}$$

$$\sum_{j\in J} x_{ij} = 1, \qquad i \in I, \tag{2.2}$$

$$\sum_{i\in I} x_{ij} \leq cap_j y_j, \qquad j \in J, \tag{2.3}$$

$$y_j \in \mathbb{B}, \qquad j \in J, \tag{2.4}$$

$$x_{ij} \in \mathbb{B}, \quad i \in I,\ j \in J \tag{2.5}$$

Constraint (2.2) ensures delivery from exactly one warehouse to each customer. (2.3) forces delivery from open warehouses only and limits the number of customers serviced by each warehouse.

Below in the CLP model the variable $x_i \in \{1, ..., n\},\ i \in I$, picks the chosen warehouse to deliver to customer i. Actually in (2.7), Y is a *set variable* where the set is constrained to be a subset of the set of warehouses as indicated in (2.6). The "values" Y can take are then all possible subsets of $\{1, ..., n\}$.

$$Y \subseteq \{1, ..., n\} \tag{2.6}$$

$$x_i \in Y, \quad i \in I \tag{2.7}$$

Further the capacity constraints is simply stated with the *atmost* constraint:

$$\text{atmost}(cap_j, [x_1, ..., x_m], j),\ j \in J$$

At most cap_j of the variables $x_1, ..., x_m$ must take the value j. The above constraints are actually sufficient to define the feasible set of solutions.

For the objective function we use two other types of global constraints in (2.9) and (2.10). The *weight* constraint returns in w the sum of weights of the constants $c_1, ..., c_n$ corresponding to the set indices of Y. This is basically the cost of opening the warehouses corresponding to the set Y. For instance if $Y = \{1, 3, 4\}$, then $w = c_1 + c_3 + c_4$.

$$\min\ w + \sum_{i\in I} z_i \tag{2.8}$$

$$\text{weight}(Y, [c_1, ..., c_n], w) \tag{2.9}$$

$$\text{element}(x_i, [d_{i1}, ..., d_{in}], z_i), \quad i \in I \tag{2.10}$$

The *element* constraint in (2.10) assigns to z_i the x_i'th value in the list $d_{i1}, ..., d_{in}$ corresponding to the delivery cost from warehouse x_i to customer

i. The resulting objective function in (2.8) is a sum of both the set-up costs and delivery costs. (2.10) is easily linearised in the following way:

$$\sum_{j \in J} x_{ij} = 1, \quad i \in I \tag{2.11}$$

$$\sum_{j \in J} d_{ij} x_{ij} - z_i = 0, \quad i \in I \tag{2.12}$$

These two constraints are basically the constraint (2.2) and the second term in the objective (2.1).

The purpose of this example was not to reach a more compact model as for the TSP case, but to show the variety of modelling capabilities offered by CLP. Paper VI gives more details on implementing the models in ECL^{i}PS^e [22], which is a Prolog based programming language for implementing and solving both CLP and IP models. Also a limited computational comparison is described.

A very promising research area is combining CLP and IP both in modelling the problem and optimising the achieved model. Rodošek et. al. [33] describe a procedure for transforming a CLP model to both an IP and a combined CLP and IP model. The combined model is then solved by a hybrid of solvers. The result is a more robust method with the modelling expressiveness of CLP. Also Hooker [21] presents a logic-based modelling framework for combining CLP and IP. These issues are discussed in more detail in section 3.3 of chapter 3.

2.5.3 The 3-Dimensional Packing Problem

The third example we discuss on logical and mathematical modelling is the case of packing 3 dimensional items in bins. The specific problem we consider is a generalisation of the knapsack problem: Decide which subset of items to pack in a knapsack with limited capacity, maximising the utility of the chosen items. (2.13) is the 1-dimensional knapsack problem just explained where p_i is the profit of item i and v_i the weight. The binary decision variable u_i is 1, if item i is in the bin and 0 otherwise. The capacity of the bin is denoted by V.

$$\begin{aligned} \max \ & \sum_{i \in I} p_i u_i \\ & \sum_{i \in I} v_i u_i \leq V \\ & u_i, \in \mathbb{B}, \quad i \in I \end{aligned} \tag{2.13}$$

In the 3-dimensional case the bin is characterised by the size of its sides in all three dimensions, length, width and height: (L, W, H). We restrict the bin and items to be rectangular. Similarly the sides of the items are denoted by the tuple: $(l_i, w_i, h_i), i \in I$. Generally we restrict the items to be placed with the sides parallel to the sides of the bin. The items can be rotated 90% degrees in the 3 dimensions resulting in six unique orientations. The orientation of an item is modelled by six binary variables, r_{ij}, where r_{ij} is 1, if item i is rotated in the j'th orientation. The sizes of the sides for a given rotation are then given by $(\alpha_i, \beta_i, \delta_i)$ from equations (2.14) to (2.17).

$$\sum_{j=1}^{6} r_{ij} = 1, \qquad i \in I \tag{2.14}$$

$$h_i r_{i1} + h_i r_{i2} + l_i r_{i3} + l_i r_{i4} + w_i r_{i5} + w_i r_{i6} - \alpha_i = 0, \qquad i \in I \tag{2.15}$$

$$l_i r_{i1} + w_i r_{i2} + h_i r_{i3} + w_i r_{i4} + h_i r_{i5} + l_i r_{i6} - \beta_i = 0, \qquad i \in I \tag{2.16}$$

$$w_i r_{i1} + l_i r_{i2} + w_i r_{i3} + h_i r_{i4} + l_i r_{i5} + h_i r_{i6} - \delta_i = 0, \qquad i \in I \tag{2.17}$$

$$\alpha_i, \beta_i, \delta_i \in \mathbb{Z}_+, \qquad i \in I$$

$$r_{ij} \in \mathbb{B}, \qquad i \in I, \; j = \{1, ..., 6\}$$

Note that the constraints (2.14)-(2.17) can be replaced by three element constraints in the same way as (2.11) and (2.12) on the preceding page. If an item is chosen to be in the bin, we additionally have to decide the exact position in the bin, (x_i, y_i, z_i). The position is the left,back, bottom corner of the item. The position $(0, 0, 0)$ is then assumed to be in the left, back, bottom corner of the bin. Items in the bin cannot overlap each other. To avoid this we introduce the binary variables, g_{ij}, which is 1, if item i is placed to the left of item j, a_{ij}, which is 1, if item i is placed in front of item j and finally f_{ij}, which is 1, if item i is placed on top of item j. Now (2.18) ensures together with (2.19) to (2.21) that, if two items i and j are in the bin, then they do not overlap. Finally the items must be inside the bin, which is assured by the

constraints (2.22) to (2.24).

$$g_{ij} + g_{ji} + a_{ij} + a_{ji} + f_{ij} + f_{ji} - u_i - u_j \geq -1, \qquad i,j \in I,\ i < j \quad (2.18)$$
$$x_i + \alpha_i - x_j + L g_{ij} \leq L, \qquad i,j \in I, i \neq j \quad (2.19)$$
$$y_i + \beta_i - y_j + W a_{ij} \leq W, \qquad i,j \in I, i \neq j \quad (2.20)$$
$$z_i + \delta_i - z_j + H f_{ij} \leq H, \qquad i,j \in I, i \neq j \quad (2.21)$$
$$x_i + \alpha_i \leq L, \qquad i \in I \quad (2.22)$$
$$y_i + \beta_i \leq W, \qquad i \in I \quad (2.23)$$
$$z_i + \delta_i \leq H, \qquad i \in I \quad (2.24)$$
$$x_i, y_i, z_i \in \mathbb{Z}_+, \qquad i \in I$$
$$g_{ij}, a_{ij}, f_{ij} \in \mathbb{B}, \qquad i,j \in I, i \neq j$$

There are a number of issues regarding the model making it very difficult to solve and even create meaningful solutions for the problem.

- The big-M type of constraints in (2.19)-(2.21) makes the problem difficult to solve with MP methods. The LP-relaxation is weak resulting in a relatively large integrality gap (refer to section 3.3 of chapter 3).

- There is a large number of symmetrical solutions in a huge search space.

- There is no guarantee that the achieved solutions are actually packable (den Boef et. al. [13]).

- There are no constraints restricting the items to be sufficiently supported by other items, i.e. they can basically fly hereby defying gravity.

This example illustrates the difficulties of extending prototype problems with real-life industrial issues. An alternative to the above model closer to the requirements of the industry is described in paper V and summarised in chapter 5.

2.5.4 The Set Partitioning Problem

The Set Partitioning Problem is part of a family of problems consisting of two additional: The Set Covering Problem and the Set Packing Problem. Given is a set of columns, $\mathcal{P}$, where each column j covers a set of rows where $a_{ij} = 1$, if row i is covered and 0 otherwise. For each column we have a variable x_j, which

is 1, if the column is used and 0 otherwise. The Set Partitioning Problem can then be formulated in the following way:

$$\begin{aligned} \min \quad & \sum_{j \in \mathcal{P}} c_j x_j \\ & \sum_{i \in \mathcal{P}} a_{ij} x_j = 1, \quad i \in \mathcal{M} \\ & x_j \in \mathbb{B}, \quad j \in \mathcal{P} \end{aligned} \tag{2.25}$$

Basically the goal is to cover each row exactly once with a set of columns with minimum cost. In the Set Covering Problem each row must be covered at least once resulting in a "$\geq$" in the equations. The Set Packing Problem is a bit different:

$$\begin{aligned} \max \quad & \sum_{j \in \mathcal{P}} c_j x_j \\ & \sum_{j \in \mathcal{P}} a_{ij} x_j \leq 1, \quad i \in \mathcal{M} \\ & x_j \in \mathbb{B}, \quad j \in \mathcal{P} \end{aligned} \tag{2.26}$$

Here we are interested in maximising the value of the columns packed where each row must be covered at most once.

Especially the Partitioning and Covering problems are very much used since a significant number of problems can be modelled in that way. Consider for instance the case where the columns represent feasible packings in a knapsack according to section 2.5.3 and the rows correspond to items to be packed in bins or knapsacks. Assume that all feasible packings have been generated and that each item must be packed in a knapsack or bin exactly once. Then the problem is a Set Partitioning Problem, where the goal is to pack all items in a minimum number of bins. The overall problem is called the Bin Packing Problem. Other important application areas of the Partition and Covering formulations are Vehicle Routing, Crew and Personnel Scheduling Problems. For a survey on these types of problems refer to Desrosiers et. al. [14].

2.5.5 Decomposition

The previous sections bring about an important issue in modelling, which is the concept of *decomposition*. In the above mentioned Bin Packing Problem, the problem has been decomposed into the problems of generating feasible packings and afterwards selecting the minimal subset of packings. Generally the reasons for decomposing a problem are better models with regard

to capturing the problem characteristics and the ability to eventually solve the problem. In section 3.3.2 of chapter 3 we discuss the *column generation* method, which can be used for solving certain decomposed problems.

2.5.6 Crane Scheduling

We finalise the section on logical and mathematical modelling with a model for the Crane Scheduling Problem with one crane. A schedule is found for one day at a time. Let the date be *today*. S is the set of stacks and let the set P be all n plates to be moved for that particular day. P are all plates with a due date $d_i \leq today$ and plates above these in the stacks. We define the set $N = P \cup \{0\} \cup \{n+1\}$ where 0 and $n+1$ corresponds to the start and end positions of the crane.

Further, we introduce the following parameters:

- t_{ij}: time to move from the position of plate $i \in N$ to stack $j \in S$.
- t'_{ji}: time to move from stack $j \in S$ to the position of plate$i \in N$.
- d_i: due date of plate $i \in P$.
- e_j: earliest due date of plates remaining in stack $j \in S$ after plates to be moved today from stack j have been removed.
- $a_{il} = 1$ if $d_i > d_l$ and 0 otherwise.
- $p_{il} = 1$ if plate i is above plate l and 0 otherwise.
- m: cost of a movement.
- K is a large constant.

The decisions to be taken are where to move the plates and in which order:

- $x_{ij} = 1$ if plate $i \in P$ is moved to stack $j \in S$ and 0 otherwise.
- $y_{il} = 1$ if plate i is moved directly before plate l and 0 otherwise.

To ease the modelling we introduce the following auxillary variables:

- s_i: start time of movement of plate i.
- s'_j: start time of last movement from stack j. For each stack j we have a plate i corresponding to the last movement. For each such pair we have a constraint $s'_j = s_i$.

- $z_{ijl} = 1$ if plate i is moved to stack j and plate l is moved directly after and 0 otherwise.
- $q_{ilj} = 1$ if both plates i and l are moved to stack j and 0 otherwise.
- $r_{ij} = 1$ if plate i is moved to stack j and $d_i > e_j$. $r_{ij} = 0$ otherwise.

Now we can formulate the model in the following way:

$$\min \sum_{i \in N} \sum_{j \in S} t_{ij} x_{ij} + \sum_{i \in N} \sum_{j \in S} \sum_{i \in N} t'_{jl} z_{ijl} + m \sum_{i \in P} \sum_{j \in S} r_{ij} \tag{2.27}$$

$$\sum_{j \in S} x_{ij} = 1, \qquad i \in P \tag{2.28}$$

$$\sum_{i \in N} y_{il} = 1, \qquad l \in N \tag{2.29}$$

$$\sum_{l \in P} y_{0l} = 1 \tag{2.30}$$

$$\sum_{i \in P} y_{il} - \sum_{h \in P} y_{lh} = 0, \qquad l \in P \tag{2.31}$$

$$\sum_{i \in P} y_{i,n+1} = 1 \tag{2.32}$$

$$s_i + t_{ij} + t'_{jl} - K(1 - z_{ijl}) \leq s_l, \qquad i, l \in N, j \in S \tag{2.33}$$

$$x_{ij} + y_{il} - 1 \leq z_{ijl}, \qquad i \in P, j \in S, l \in N \tag{2.34}$$

$$p_{il} s_i \leq s_l, \qquad i, l \in P \tag{2.35}$$

$$s'_j - K(1 - x_{ij}) \leq s_i, \qquad i \in P, j \in S \tag{2.36}$$

$$d_i x_{ij} - K r_{ij} \leq e_j, \qquad i \in P, j \in S \tag{2.37}$$

$$s_i - K(1 - a_{il} q_{ilj}) \leq s_l, \qquad i, l \in P, j \in S \tag{2.38}$$

$$x_{ij} + x_{lj} - 1 \leq q_{ilj}, \qquad i, l \in P, j \in S \tag{2.39}$$

$$s_i \geq 0, \qquad i \in N \tag{2.40}$$

$$s'_j \geq 0, \qquad j \in S \tag{2.41}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in P, j \in S \tag{2.42}$$

$$y_{il} \in \{0, 1\}, \qquad i, l \in N \tag{2.43}$$

$$r_{ij} \in \{0, 1\}, \qquad i \in N, j \in S \tag{2.44}$$

$$z_{ijl} \in \{0, 1\}, \qquad i \in P, j \in S, l \in N \tag{2.45}$$

$$q_{ilj} \in \{0, 1\}, \qquad i, l \in P, j \in S \tag{2.46}$$

The objective function consists of 3 terms: 1) travelling time with a plate, 2) travelling time without a plate and 3) cost of movements introduced in the future when a plate i is put on a stack j where the minimum due date in the stack e_j is larger than d_i. This relation is forced by the constraint (2.37).

Constraint (2.28) assigns exactly one destination to each plate movement. (2.29)-(2.32) are flow constraints. These together with (2.33) and (2.34) restricts the movement of the crane to be a TSP tour.

(2.35) restricts a plate above another plate to be moved first. (2.36) makes sure that a plate i cannot be moved to a stack j before all plates to be moved from j have been moved. Without this constraint a plate would have to be moved more than once on the same day and the information would not be available apriori – requirering a dynamic model. Finally (2.38) and (2.39) restricts plates moved to the same stack to be placed in due date order. Instead of a constraint one might add a cost term to the objective function, since the condition is only influencing the future number of movemements.

From this formulation it is quite clear that the problem is a type of generalised TSP with different precedence constraints. No attempts have been done to solve the model for several reasons: In reality two cranes can perform the movements and the model would be intractable for real-life instances with up to 1000 movements per day.

2.6 Simulation Modelling

A simulation model is different from other typical mathematical models. The model consists of objects each with their own behaviour interacting with each other. The overall behaviour of the system is a result of these interactions. A simulation model is hence a virtual world of interacting objects. Simulation is an experimental approach where the behaviour of the system is improved by changing the behaviour of the objects or the composition of the model.

The flexible setting of simulation allows modelling of more general systems than for instance is possible within Mathematical Programming. Simulation can easier cope with dynamic problems perhaps including randomness, which often occur in industrial problems. In some respects, simulation can be seen as the last resort when the modelling becomes to complex for Mathematical Programming or Constraint Logic Programming (Pidd [30]).

The use of simulation in Combinatorial Optimisation is mostly considered in the simulation community and is referred to as *Simulation Optimisation* (See e.g. Ólafsson and Kim [29]). Simulation is used for checking legality and evaluating solutions often in a dynamic and stochastic environment.

Simulation has been used in our Crane Scheduling case for different reasons. First of all, due to different technical reasons the real system was not ready to adopt the scheduling software and simulation was hence required for evaluating the software. Secondly, the management of the yard wanted a proof of concept through simulations before changing the real system to use the software. The use of simulations is able to reduce the risks connected to taking optimisation software into production in industry. Finally, the complexity of the planning problem called for the use of simulation, since the schedule of crane movements must take into account the order of the plates in the stacks and avoid collisions of the cranes.

2.6.1 Discrete Event Modelling

In discrete event simulation, the passing of time is driven by the time of occurring events. An event can be described as a significant state change occurring in the system at a certain instant in time. This is the model type most appropriate for modelling industrial problems with simulation. Other types of dynamic models will hence not be discussed.

Discrete event modelling is tightly connected with object-oriented modelling. The following items must be identified when building a simulation model of a system:

1. Objects and classes in the system.
2. Relations between objects.
3. Possible states of the objects.
4. Possible events that change the state of the objects.
5. Operations or activities performed by objects occurring at events transforming the states of the objects.

In simulation we distinguish between *permanent* and *temporary* objects. Permanent objects remain in the system throughout the simulation while temporary objects pass through the system. Examples are cranes and stacks, which are permanent while plates are temporary objects. Further we denote *active* objects as objects that can process other objects, while processed objects are called *passive*. Cranes are clearly active objects, since moving plates is a process. Plates are passive objects, since the cranes process them. Typically objects are either passive or active and not both. Stacks on the other hand are neither active nor passive.

The progress of the simulation is controlled by a *manager* or *executive* object, which handles the overall management of elapsed simulation time, identifying the next event to occur and sequencing the activities of the simulation objects. In section 3.4 of chapter 3 we describe the fundamentals in designing the manager. Several different approaches are discussed based on different principles. In this section we discuss the problem-specific issues of simulation modelling.

Figure 2.5 is a simplified illustration of the operations and state changes of the gantry crane in the crane scheduling case. In UML the diagram is called

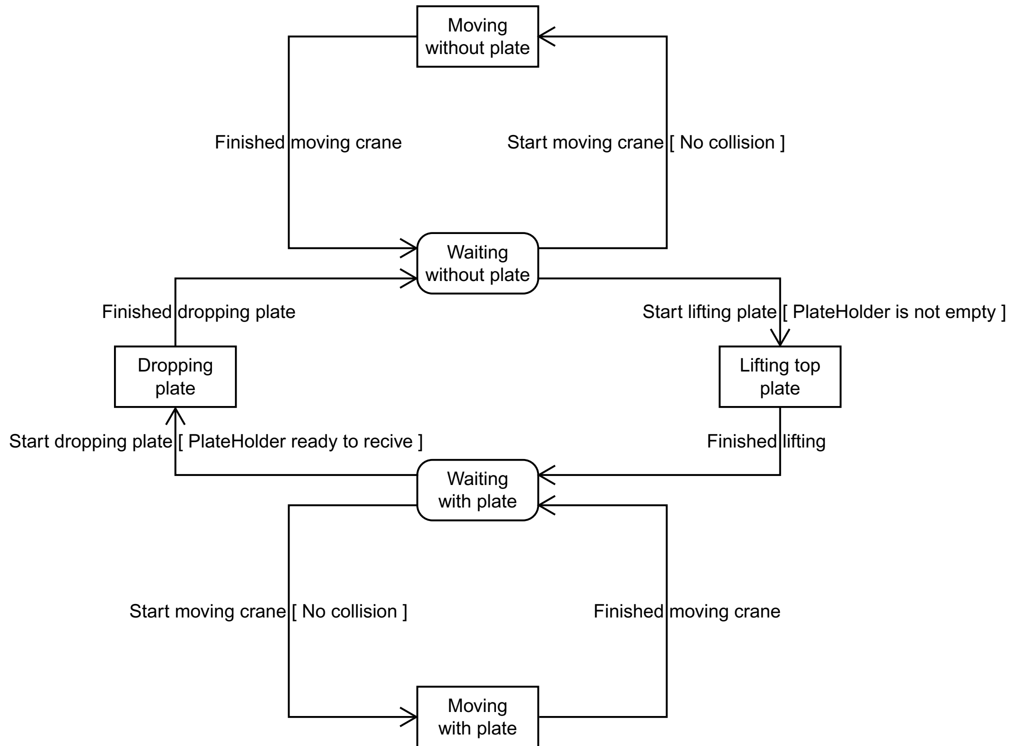

Figure 2.5: State-diagram for the crane class.

a *state-diagram*, while in simulation it is called an *activity cycle diagram*. The boxes are states and arrows indicate state transformations fired by the event described by the legend. In brackets are given the conditions for the state changes. For instance a plate can only be lifted, if a plate is actually placed in the plate holder below the crane. Note that the only difference between state-

diagrams and activity cycle diagrams are that in the activity cycle diagram we distinguish between inactive and active states. Basically the waiting states in the figure are inactive states indicated by rounded corners while the others are active.

We distinguish between two types of operations called by an event:

Bound (B) events: Operations that can be executed directly after the scheduled time is reached. For example operations called directly after a crane has finished dropping a plate. The corresponding events occur directly after active states, where the duration of the state is known in the simulation when the state commences.

Conditional (C) events: Operations only executable, if certain conditions are fulfilled and can hence not be scheduled in advance. For instance the crane cannot be moved, if a collision with the other crane will occur. The corresponding events occur directly after passive states, where the duration of the state is unknown when the state commences.

Notice that the activity of dropping a plate is a simplification of the real-life dropping of a plate. In reality, first the crane drops its hoist, then the plate is released and finally the hoist is raised. Three activities are modelled as one. The state of the crane and the stack can only change when an event occurs. In our model either when the crane *starts* dropping the plate *or* when the crane *has finished* dropping the plate. Which event is the proper event for popping the plate from the crane and pushing it onto the stack? The answer is, that it depends on the logic of the system. The point here is that the modeller should be aware of these simplifications and validate them against the real system.

Not all logic is captured in figure 2.5. When a crane is waiting with and without a plate there are two possible events following, which are moving or lifting/dropping a plate. A decision must be taken either continuing waiting, moving or lifting/dropping a plate. In our case this decision was taken either online or based on a plan. We refer to paper II for details on this.

When the plates are due to leave the storage, the plates are placed on a conveyer belt transporting them to the first production process where the plates are rolled. The duration of the process is depending on the size of the plate and the order in which the plates are processed. In this context we assume that the duration is some constant plus a random number drawn from a negative exponential distribution. The conveyer belt works as a queue and has a capacity of 8 plates. The extension to the previously given state-diagram is shown in figure 2.6 on the next page.

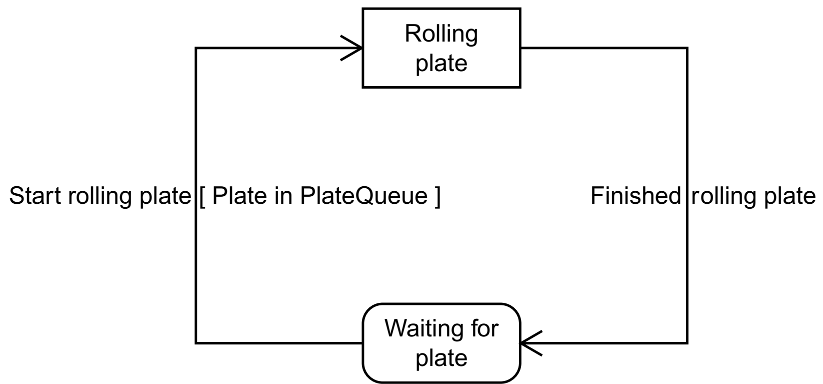

Figure 2.6: State-diagram for the plate roller class.

A plate can only be dropped on the belt, if strictly less than 8 plates are placed on the belt and the plate roller can only start processing a plate, if a plate is placed on the belt.

Brooks and Robinson [7] gives an alternative or supplement to the state-diagram in form of a table as shown in table 2.2. Note especially that the conditional events lead to scheduling of new events. The time of the new events are determined by the duration of the current state.

Event	**Type**	**Conditions for event to occur**	**Change in state caused by event**	**Future events to schedule**
Start *drop plate*	C	Room for plate in queue	Crane change state to *dropping*	Finished *drop plate*
Finished *drop plate*	B		Move plate from crane to queue and change state of crane to *waiting*	
Start *rolling plate*	C	Plate present in queue	Pop plate on queue and change state of roller to *rolling*	Finished *rolling plate*
Finished *rolling plate*	B		Change state of roller to *waiting*	

Table 2.2: Events at the conveyer belt in the crane scheduling case.

The step from the state-diagrams and event tables to the design of the simulator depends on the simulation approach taken. Further discussion will

hence be deferred to section 3.4 of chapter 3.

2.7 Local Search Modelling

Local search is also called improvement heuristics, since they iteratively move from one solution to a new improved solution. The next solution is chosen in the *neighbourhood* of the current solution indicating that the new solution is somewhat close to the current solution. The neighbourhood is defined by the *neighbourhood structure*, which is a set of operators modifying a solution.

For instance in the crane scheduling case a solution is a sequence of movements for each crane, where a movement at least is defined by an origin and destination, possible the plate to move, start time and duration. A neighbourhood structure could then be defined by all possible ways of removing a movement in one sequence and inserting it at another position in the same sequence or inserting it in the sequence of the other crane. With n movements the size of the neighbourhood is $O(n^2)$. Other possible definitions of the neighbourhood structure obviously exist and the choice of structure can have significant impact on the effectiveness of the heuristic.

Local search heuristics are general heuristics applicable to many different problem types. For each type three problem-specific issues must be addressed:

- Representation of a solution.
- Definition of the neighbourhood structure.
- The evaluation of the objective and feasibility of a solution.

The modelling exercise when developing local search heuristics is to build an appropriate object-oriented model for the above three issues. Useful tools for local search modelling are again different UML diagrams, such as the class-diagram. The importance of an efficient evaluation of both the objective value and feasibility of the neighbour solutions cannot be stressed enough. In the crane scheduling case this was however not possible since simulation was used to evaluate the feasibility and objective value. Then either more time is required to reach sufficiently good solutions or other optimisation approaches must be chosen.

Local search heuristics and other types of heuristics are discussed in section 3.2 of chapter 3.

2.8 Summary

In this chapter we have covered different areas of modelling, relevant when building software to solve industrial Combinatorial Optimisation problems. The models include the conceptual object model to describe the perceived real-world system and mathematical/logical, simulation and local search models for describing Combinatorial Optimisation problems. Several of the model types are applied in the industrial cases.

The purposes of models are to facilitate communication and knowledge exchange, documentation and the basis for implementing the optimisation procedures and other software modules.

In industry Combinatorial Optimisation software must be integrated with the surrounding information systems and users must be able to interact with the system. Here modelling must be seen in a broader sense where different models in combination create a foundation for software development. The Combinatorial Optimisation component is part of this and cannot be considered in isolation.

CHAPTER 3

Solving Optimisation Models

3.1 Introduction

Based on the developed model of our problem, a method must be implemented to solve the model and supply solutions to the problem. Certain methods can be applied to certain models while others are not suitable. In the previous chapter, we introduced different types of mathematical and logical models as well as models better described within the object-oriented technology. Generally, an object-oriented model should in our opinion be developed disregarding the chosen solution method, since the interaction between the Combinatorial Optimisation component and the remaining software system is better implemented in an object-oriented fashion. The objects in the implementation are also the building blocks for heuristics and simulation methods. Optimisation methods are based on logical and mathematical methods. Either the objective and constraints are created and posted to a solver returning the optimal solution or a specially tailored search procedure is developed.

Table 3.1 on the next page is used to indicate which approach is best suitable for given problem characteristics and given requirements to the resulting solution. The table obviously gives a very simplistic view of the dependencies between the different approaches, solution quality and necessary computation time. We believe however that most industrial applications can be correctly mapped into this table. Two rows are given, separating methods for solving

problems with static and dynamic data. We are in the following mainly focused on methods for solving problems with static data leaving the dynamic methods somewhat in the background.

Solution Quality **Computation Time**	**Low** **Short**	**Medium** **Medium**	**High** **Long**
Static Data	Construction heuristics	Descent Algorithms	Local Search heuristics and optimisation methods
Dynamic Data	On-line heuristics	Forward-looking on-line heuristics	Re-planning with heuristics or optimisation methods

Table 3.1: Appropriate approaches based on problem and solution characteristics.

The outline of the remainder of the chapter is the following. Section 3.2 gives an overview of different heuristic approaches including construction and different local search heuristics. Section 3.3 discusses different optimisation methods for solving MP and CLP problems. Finally, in section 3.4 solving by simulation is described. Simulation has not been mapped into table 3.1. The reason is that simulation can be used in combination with all the described methods though typically in a dynamic environment.

3.2 Heuristics

Heuristics methods do not guarantee to find an optimal solution. Optimisation methods on the other hand are constructed in such a way that when an optimal solution is found, the method can prove that indeed that solution is the globally best. This might however take an unrealistic amount of time and we must be content with less than a proven optimum. In the following we introduce the different types of heuristics from table 3.1.

3.2.1 Construction heuristics

Construction heuristics are greedy heuristics where planning decisions are done incrementally in a greedy fashion without regretting earlier decisions. On-line heuristics are basically construction heuristics, where decisions are taken online. The plan is in other words being constructed while it is actually being executed.

Construction and on-line heuristics are driven by decisions, which are good from a local or myopic perspective, but which might be very different than the globally best decisions. A forward-looking on-line heuristic tries to avoid this disadvantage by evaluating several steps of decisions and possible scenarios into the future and choosing the best decision of those.

One of the main points in the methodology described in paper VII and chapter 4 is that when developing applications for solving industrial problems simple methods should be developed before more complex methods. A construction heuristic is the simplest form of method to develop and is anyway necessary for other types of heuristics to function. This method should hence be the first to be developed.

Examples of on-line/construction heuristics are given both in paper II for the crane scheduling case and paper V for the packing case.

3.2.2 Local Search Heuristics

Local search heuristics were introduced in section 2.7 of chapter 2. To recapitulate, local search heuristics iteratively move from one solution to a neighbour solution in pursuit of improving solutions. Three components of the heuristics are problem-specific: Representation of a solution, neighbourhood structure and objective function. These must be appropriately modelled and implemented for the heuristic to execute efficiently.

In paper II we discuss the different improvement heuristics implemented for solving the crane scheduling problem. Also [31] by Pirlot gives an introduction to local search heuristics. The central characteristics of the heuristics are summarised here:

Steepest Descent algorithm: The best neighbour solution in the neighbourhood is picked as the next solution. The algorithm terminates when a local optimum is found, since no improving neighbour solution exists.

Simulated Annealing: A random solution is picked from the neighbourhood as next solution. If the solution is improving, it is accepted as the next solution. If the solution on the other hand is worse than the current solution, it is picked with a certain probability. The probability decreases when the difference between the current and new solution increases. In the analogy of annealing the probability is determined by the temperature, which slowly decreases – the *cooling schedule*. For increasing number of iterations the probability hence decreases and eventually the search resembles a descent algorithm.

Tabu Search: The best neighbour solution in the neighbourhood is picked as the next solution, if it is not tabu. Tabu can be defined in a number of ways often in forms of attributes describing the move from one solution to another. The opposite move is then defined tabu in a number of iterations hopefully avoiding the return to already visited local optima.

All the three above methods are local search heuristics and are also referred to as *meta-heuristics*.

From the above description of the heuristics, we can identify different common algorithmic design decisions, which must be taken when developing local search heuristics:

Choice of solution from neighbourhood: Simulated Annealing picks a random neighbour in the neighbourhood while Steepest Descent and Tabu Search picks the best (non-tabu) neighbour. When the size of the neighbourhood becomes too large, the best neighbour can instead be picked from a random or shifting subset. Alternatively the first improving (non-tabu) neighbour can be chosen. Often the neighbourhood is defined by a combination of different neighbourhood structures, which means that the algorithm can shift between the structures.

Stopping criteria: Different possible criteria for stopping the search are maximum amount of time, maximum number of iterations, improvement smaller than $\delta\%$, no improvement of best solution in i iterations, etc.

Escaping local optima: Both Simulated Annealing and Tabu Search enable the escape from local optima by accepting worse solutions. Simulated Annealing escapes local optima by randomisation while Tabu Search deems certain movement attributes tabu avoiding the return to the local optima and hence occasionally forcing the move to a worse solution.

Local search heuristics are working in two opposite directions, which must be balanced in order to reach high quality solutions. Intensification is representing the focus of the heuristic on moving toward good solutions while diversification represents the ability of the heuristics to explore diverse regions of the search space. Some heuristics are able to dynamically change its behaviour in either direction. An example is the Reactive Tabu Search by Battiti and Tecchiolli [3], which dynamically increase or decrease the number of iterations, move attributes are deemed tabu depending on the behaviour of the search. If the search keeps returning to the same solution, the tabu

iterations are increased, and otherwise it is slowly decreased. More details on intensification and diversification in local search are given in paper II.

3.2.3 Object-oriented Local Search Framework

To solve the Crane scheduling problem an object-oriented local search heuristic framework has been implemented in Java. The framework is based on the design of Andreatta et. al. [1] making the framework suitable for solving any type of problem with local search. The object-oriented approach leads to extensive code reuse even though the user of the framework must supply certain components. In Voss and Woodruff [35] several other local search frameworks and class libraries are described. Further we can mention "Open Tabu Search", an open source Tabu Search framework also in Java due to Harder [20]. The developed framework is described in appendix A.

3.3 Mathematical and Constraint Logic Programming

Problems modelled within the paradigm of Mathematical Programming (MP) and Constraint Logic Programming (CLP) were the subject of section 2.5 in chapter 2. In this section we discuss different methods for solving these types of models.

This section of the thesis is somewhat different than others, since it supplements the text in paper VI. The name of the paper is "Constraint Programming versus Mathematical Programming", but besides pointing at the differences of the two paradigms, it also points at the similarities. Hybrid methods combining the best of IP and CLP solvers make up a very promising research area, which we discuss further in section 3.3.1.

MP has been used in the packing case described in paper V and the main issues are summarised in chapter 5. The methods applied include Branch & Price, Branch & Cut and a specially tailored search procedure for testing whether a set of items can be packed in a bin. Branch & Cut is covered in section 3.3.1 and Branch & Price in 3.3.2.

3.3.1 Logic-Based Branch & Bound

Methods combining what is considered the best from different methods are called *hybrid methods*. When considering MP and CLP, hybrid methods combine components from both areas in order to construct even more powerful

optimisation methods. In this section we discuss the possibilities in combining MP and CLP to what Hooker [21] refers to as *Logic-based Branch & Bound*, but first the main components are reviewed.

Both MP and CLP are based on tree search or branching. The idea is that the subproblems after branching are easier to solve than the original. The subproblems are *strengthened* compared to the original. In CLP further strengthening is done by *domain reduction* and *propagation*, which is basically deducing infeasible values in the variable domains. One constraint is considered at a time and domain reductions deduced from one constraint leads to further potential reductions by other constraints in which the same variables participate – propagation. If at any stage a domain of a variable becomes empty, no feasible solution exists for that branch and it can be pruned.

In MP a relaxed problem is solved to give a lower bound (assuming minimisation), z^{LB}, on the optimum value, z^{opt}. This problem should be easier to solve than the original. Assume that we have found a feasible solution for instance with use of a heuristic with value z^{UB}. The purpose of the bound is then that we can prune the branch, if $z^{LB} \geq z^{UB}$, since an improving solution cannot be in that branch. Further if no feasible solution exists to the relaxed problem, then no feasible solution exists for the original problem either and the branch can be pruned. The most used relaxation without comparison is the LP-relaxation in which the integer restrictions on variables are relaxed. The combination of branching and bounding results in the name Branch & Bound.

Cuts or cutting planes originate in MP and are simply constraints added during the search. The purpose of the cuts is in most cases to strengthen the LP-relaxation to better describe the convex hull of the feasible solutions of the original problem. In the TSP problem discussed earlier the sub-tour constraints are first relaxed and afterwards iteratively added when violated. The sub-tour elimination constraints are however not enough to describe the convex hull of feasible solutions. More cuts are needed such as comb inequalities (see e.g. Nemhauser and Wolsey [28]) and eventually branching. The Approach of combining branching and adding of cuts are naturally called Branch & Cut.

Notice the relation of the concepts of strengthening and relaxing. Strengthening adds constraints reducing the solution space while relaxation removes constraints enlarging the solution space. Strengthening hence provides upper bounds and relaxation lower bounds. In the context of a Branch & Bound algorithm, relaxation provides lower bounds in each node, while upper bounds are only found in nodes where a feasible solution is found. Notice also that

for both concepts the goal is easier problems to solve. The relaxation is easy to solve in each node as opposed to series of strengthenings, which evidently result in easy subproblems to solve in the leaves of the tree.

In our logic-based Branch & Bound, **Relaxation** denotes solving the LP problem defined by some or all the linear constraints in the problem where the integer restrictions of variables included in the constraints are relaxed. Cuts might be added based on the relaxation. The modeller can choose to send only a subset of the constraints to the LP solver.

Inference denotes domain reduction and propagation over integer variables and some or all constraints including these variables. Again the modeller might choose to only apply inference algorithms on a subset of the constraints.

Now consider a node in the branching tree. It is processed as shown in algorithm 1. The value of the LP-relaxation is denoted by z^{LB}.

Algorithm 1: Logic-based Branch & Bound

```
Inference
if no infeasibility detected then
    Relaxation
    if relaxation is feasible ∧ z^LB < z^UB then
        if relaxation is integral then
            z^UB = z^LB
        else
            Branching
```

In **Branching** an integer variable with a non-integral value is chosen to branch on. Typically in IP branching is done by splitting the domain in two, while in CLP branching is done on each value in the domain. Any of the two can be applied. It is quite clear that algorithm 1 is very close to the usual Branch & Bound in IP and CLP. The major difference is in the modelling phase where the model must be developed with both logic and linear constraints and objectives in mind.

3.3.1.1 The Warehouse Location Problem Revisited

Consider again the Warehouse Location problem discussed in chapter 2 and the possibilities of modelling the problem in the context of logic-based Branch & Bound. Given below is a combination of the two models where $s_i \in Y, i \in I$

is the variable deciding the warehouse of customer i.

$$\min \sum_{j \in J} c_j y_j + \sum_{i \in I} \sum_{j \in J} d_{ij} x_{ij} \tag{3.1}$$

$$\sum_{j \in J} x_{ij} = 1, \qquad i \in I \tag{3.2}$$

$$x_{ij} \leq y_j, \quad i \in I,\ j \in J \tag{3.3}$$

$$\sum_{i \in I} x_{ij} \leq cap_j, \qquad j \in J \tag{3.4}$$

$$\text{atmost}(cap_j, [s_1, ..., s_m], j), \qquad j \in J \tag{3.5}$$

$$\sum_{j \in J} j\ x_{ij} - s_i = 0, \qquad i \in I \tag{3.6}$$

$$\text{membership_booleans}(Y, [y_1, ..., y_n]) \tag{3.7}$$

$$Y \subseteq \{1, ..., n\} \tag{3.8}$$

$$s_i \in Y, \qquad i \in I \tag{3.9}$$

$$0 \leq y_j \leq 1, \qquad j \in J \tag{3.10}$$

$$0 \leq x_{ij} \leq 1, \quad i \in I,\ j \in J \tag{3.11}$$

The two types of warehouse to customer variables are linked in (3.6). In some constraint solvers this constraint could be modelled with the element constraint instead:

$$\text{element}(s_i, [x_{i1}, ..., x_{in}], 1), \ i \in I$$

Here s_i is the index in the list where the variable must take the value 1. Some solvers however do not allow variables in the list. The variables concerning open warehouses are linked in (3.7), where Y is the set corresponding to indices in $[y_1, ..., y_n]$ where the variables take the value 1.

Now in each node of the Branch & Bound tree **Inference** is performed on all constraints from the CLP model and the link constraints. **Relaxation** is performed on the linear constraints and objective function. Hooker [21] describes a modelling framework, which allows the modeller to specify which constraints are part of the inference phase and which are part of the relaxation. Branching is done on the integer variables, which here are the Y and s_i's. The model has been implemented in ECL^{i}PS^e [22] with the built-in inference algorithms. We note, however, that for this particular example, the formulation is inferior to the IP formulation. The example nevertheless illustrates the different modelling possibilities of logic-based Branch & Bound.

3.3.1.2 Comparing Approaches

Rodošek et. al. [33] compare the performance of CLP, IP and CLP/IP on 4 different problems: The 2-Hoist Scheduling Problem (2-HSP), the Progressive Party Problem, the Cabinet Assignment Problem (CAP) and the Set Partitioning Problem (SPP). For the CAP problem the pure CLP formulation is the best while IP is better suited for the SPP. For the 2-HSP the combined approach makes the difference of solving the problem at all in reasonable time. Overall the performance of the CLP/IP approach is more robust in that variation in computation time over problem types is less than for the pure approaches. Generally the combined approach covers for the deficiencies of the pure approaches. The performance of the Warehouse Location problem is similar to the SPP. When a relatively strong IP model can be formulated nothing is gained from CLP. Modelling requires much experimentation and combining CLP and IP approaches only adds to the vast amount of possible model constructs. The goals are to provide better models for describing the problem and better performance of the combined optimisation algorithms.

3.3.1.3 Inference Algorithms

The success of CLP and hence hybrid approaches depends on the strength of the inference algorithms used. We have given examples of modelling with use of so-called global constraints, e.g. the all-different constraint, which performs domain reduction over a collection of constraints. Other examples are the atmost and element constraints. Refer for instance to Hooker [21] and Marriott and Stuckey [26] for further information on domain reduction algorithms. Some global constraints can be linearised and included in the LP-relaxation as we saw for the element constraint. We believe that the real benefit arises when modelling problems with many complex constraints where linear models are out of the question. In ECL^{i}PS^e and other constraint solvers, the user can construct new global constraints with matching inference algorithms. Here there is an obvious relation between algorithm complexity and domain reduction ability. In some cases, however, the algorithms can deduce nothing until the domains of all variables have been reduced to a single value. This leads to a significant waste of time. Then instead of doing domain reduction *constraint checking* is preferred. Here constraint checking is delayed until the domains of all variables participating in the constraint have been reduced to a single value.

3.3.1.4 Specially-Tailored Branch & Bound

CLP in many ways resembles what is referred to as *specially-tailored Branch & Bound* algorithms. Specially tailored algorithms are as the name indicates constructed for a specific purpose – here solving a specific Combinatorial Optimisation problem. Typically some other relaxation is used instead of LP-relaxation or none altogether and the constraints are checked during the search perhaps supplemented with some domain reduction. To solve the packing problem introduced in section 2.5.3 of chapter 2, a search algorithm was developed to determine if a set of items could be packed in a bin or not. Refer to paper V and chapter 5 for further details.

The advantage of using a CLP system is that the system provides a number of built-in facilities reducing the development time and often it requires a significant amount of knowledge to build a competitive specially tailored algorithm. The flexibility of for instance ECL^{i}PS^e gives the user the power to easily experiment with different domain reduction and search strategies, which are more difficult to do with a specially tailored algorithm. On the other hand, there can be a computational overhead of using a CLP system, which might be avoided.

3.3.2 Column Generation

In section 2.5.5 of chapter 2 we introduced the concept of decomposition. Decomposing a problem can in some instances result in problems, which are easier to solve than the original. The textbook example is the Cutting Stock Problem in Gilmore and Gomory [18, 19] where a set of small items of different sizes must be cut from larger items. Each item has a length and we want to minimise the costs of the large items necessary for cutting the small items. The problem can now be formulated in the following way:

$$\begin{aligned} \min \quad & \sum_{j \in \mathcal{P}} c_j x_j \\ \sum_{j \in \mathcal{P}} a_{ij} x_j & \geq b_i, \quad i \in \mathcal{M} \\ x_j & \in \mathbb{Z}_+, \quad j \in \mathcal{P} \end{aligned} \tag{3.12}$$

For each small item we have a row $i \in \mathcal{M}$ and a b_i indicating the minimum number of item type i needed. Each column $j \in \mathcal{P}$ is a *cutting pattern* and for each row i, a_{ij} indicates how many items of type i is cut from the cutting pattern. Finally c_j is the cost of cutting pattern j according to the type of

large item necessary to cut the pattern. Generally column j is one of the types $t \in \mathcal{T}$ of larger items. The decision variable x_j indicates the number of larger items cut by pattern j in the solution. Solving the model will return an optimum solution given that the set $\mathcal{P}$ consists of all feasible cutting patterns. Generally the size of this set is astronomical. The potential size of the set is $2^{|\mathcal{M}|}$ for the case where $b_i = 1$, for all $i \in \mathcal{M}$. For 10 items the size is only 1024, but for 20 items the size is already more than 1 Mio. patterns. Complete enumeration of all patterns is therefore out of the question for most practical instances.

Instead we can use *column generation*. The outline of the approach is first to solve the problem for a small subset of patterns, $\mathcal{P}'$, and afterwards to iteratively add more patterns until the solution cannot be improved. When relaxing the integer restrictions on variables in (3.12) we get the following *master problem*:

$$\begin{aligned} \min \quad & \sum_{j \in \mathcal{P}'} c_j x_j \\ \sum_{j \in \mathcal{P}'} a_{ij} x_j & \geq b_i, \quad i \in \mathcal{M} \\ x_j & \in \mathbb{R}_+, \quad j \in \mathcal{P}' \end{aligned} \tag{3.13}$$

For each row $i \in \mathcal{M}$ we have a dual variable π_i and we can use the dual solution to direct the search for improving patterns. More formally, we are searching for columns with negative reduced costs:

$$c_t - \sum_{i \in \mathcal{M}} \pi_i y_i < 0$$

The problem of *pricing* columns is in this case the Knapsack Problem where l_i is the length of item i:

$$\begin{aligned} z_t = \max \quad & \sum_{i \in \mathcal{M}} \pi_i y_i \\ \sum_{i \in \mathcal{M}} l_i y_i & \leq L_t \\ y_i & \in \mathbb{Z}_+, \quad i \in \mathcal{M} \end{aligned} \tag{3.14}$$

There is a pricing problem for each large item type $t \in \mathcal{T}$ with length L_t. A decision variable y_i for each item i indicates the number of items cut from the pattern. A pattern is added, if $z_t > c_t$. When no more patterns with negative reduced costs can be generated, the solution is optimal. There is however no

guarantee that the solution is integral and hence feasible to (3.12), but the objective value is a lower bound.

Different approaches can now be undertaken to reach a feasible integer solution:

Rounding procedures: The solution can be rounded up when having covering constraints ($\geq$).

Heuristics: Different local search heuristics can be applied to find a good feasible solution.

The master problem can be solved with integer restrictions over the given subset of generated columns, which is basically solving (3.13) with $x_j \in \mathbb{Z}_+, j \in \mathcal{P}'$.

Branch & Price: The column generation can be embedded in a Branch & Bound procedure resulting in a *Branch & Price* algorithm.

When designing a Branch & Price algorithm, the natural choice of branching strategy would perhaps be to branch directly on the x_j variables in the master problem. The problem with this strategy is that the branching does not push the pricing problem in direction of providing columns useful in reaching integer solutions, since the pricing problem is not changed. Further, an eliminated column ($x_j = 0$) will most likely be priced again and reenter the master problem. Vanderbeck and Wolsey [34] have developed a branching scheme, which works in both the master and the pricing problem. To remove fractional variables in a solution, we add the following two constraints – one in each branch: An upper bound on the sum of variables corresponding to columns covering a specific set of rows in one branch and a lower bound in the other. When the sum, δ, is fractional the upper bound is $\lfloor \delta \rfloor$ and the lower bound is $\lceil \delta \rceil$. The extra constraints result in additional dual variables, which influence the reduced cost of generated columns. For further details on the branching scheme, we refer to paper V covering the packing case.

During column generation, Lagrangian-relaxation can be used for providing a lower bound even though all improving columns have not yet been generated. First we show the Lagrangian assuming that all columns have been generated:

$$L(\pi) = \min \sum_{j \in \mathcal{P}} c_j x_j + \sum_{i \in \mathcal{M}} \pi_i \left(b_i - \sum_{j \in \mathcal{P}} a_{ij} x_j \right), \qquad (3.15)$$
$$x_j \in \mathbb{R}_+,\ j \in \mathcal{P}, \qquad \pi_i \in \mathbb{R}_+,\ i \in \mathcal{M}$$

For $\pi_i \leq 0$ and a solution feasible to (3.13) the last term will be negative and when minimising over a larger region of solutions the lower bound is in place. By rearranging the terms we get:

$$L(\pi) = \min \sum_{j \in \mathcal{P}} \left(c_j - \sum_{i \in \mathcal{M}} a_{ij}\pi_i \right) x_j + \sum_{i \in \mathcal{M}} b_i \pi_i \tag{3.16}$$
$$x_j \in \mathbb{R}_+,\ j \in \mathcal{P}, \qquad \pi_i \in \mathbb{R}_+,\ i \in \mathcal{M}$$

Notice, that the first term is a sum over the reduced costs, but these values are basically the values from the pricing problem. Assume that we have an upper bound, U_t on necessary items of each type $t \in \mathcal{T}$. Further from the last pricing iteration we have the resulting reduced costs, z_t. For each item type t the maximum saving in cost can then be $U_t z_t$ and we reach the lower bound:

$$LB = \min_{t \in \mathcal{T}} U_t z_t + \sum_{i \in \mathcal{M}} b_i \pi_i$$

The lower bound is valid during the generation of columns, since it is independent of the columns used in the solution. We can at any time stop the column generation in the current branching node, if $\lceil LB \rceil \geq LP$, where LP is the solution value to the master problem, since no better bound can be achieved. If $\lceil LB \rceil \geq UB$ where UB is any upper bound to (3.12), then UB cannot be improved in the node and it may be pruned.

The general framework of column generation is widely applicable in industry and therefore a significant tool in the optimisation toolbox.

3.4 Simulation

In section 2.6 of chapter 2 we discussed modelling of problems within a simulation setting: Simulation of objects interacting over time. In the crane scheduling case two simulators have been implemented. Range and Yde [32] developed a graphical simulation while the implementation described in paper II was used for evaluating solutions for the local search heuristics. Both implementations are based on the *event approach* described in section 3.4.2. A number of other approaches exist for instance the *activity approach*, the *process approach* and the *three-phase approach* to mention a few. All of them are described in Pidd [30]. The three-phase approach is actually a simpler and in some sense more intuitive approach than the others. The three-phase approach is described in section 3.4.1.

The main difference between the approaches is how the manager object is designed. The manager has the responsible for the simulation clock, the scheduled bound events (events bound to happen), sequencing of conditional events and collection of statistics. Naturally simulation makes extensive use of random numbers and probability distributions. We will however not discuss this further, but refer to Pidd [30].

3.4.1 The Three-Phase Approach

The Three-Phase Approach consists of the following three phases, which are repeated until the available simulation time is spent or there is nothing more to be simulated:

Phase A: The manager determines when the next event is due and sets the simulation clock to that time.

Phase B: The manager collects due B (bound) events and executes them in a given predefined order.

Phase C: The manager scans all C (conditional) events and execute the events whose conditions are fulfilled. The scan is repeated until no more events are executed.

All scheduled B events are stored in a list type of data structure or possibly a priority queue for fast access (Cormen et. al [12]). When an event is due it is removed from the list. Each event is an individual object, which is either of the class Event or a subclass. In the B phase processEvent() is called on each due event object. The default behaviour of processEvent() then is to pass the message on to the object in the simulation, which should react on the event. For instance the event that dropping a plate on the conveyer belt has finished, calls the operation pop() on the crane and push(plate) on the belt to move the plate from the crane to the belt. For further details, Joines and Roberts describe an object-oriented simulation framework in [24, 25] for simulation in networks.

The C phase can be designed in several different ways. Typically the objects will be asked in turn for a next event. This works fine for instance for the plate roller, which works independent of the cranes. It is simply checked whether a plate is placed on the conveyer belt. The cranes however need some co-ordination in order to avoid deadlocks and collisions. During the simulation, decisions must be taken regarding which crane should yield for the other. Either some co-ordination or negotiation behaviour must be built

into the crane class or more likely a co-ordinating class must handle this task. These issues are discussed in paper II. Note also that optimisation in simulation is exactly for the algorithm to decide the behaviour of the objects optimising the objectives of the system.

3.4.2 The Event Approach

In the event approach all events are B events or in other words bound to occur at a given time. The approach then consists of only A and B phases. This makes the event handling somewhat more complicated. For instance assume that the crane arrives at the conveyer belt to deliver a plate, but the conveyer belt is full. The plate roller must then be rolling a plate. Now in the simulation it is known at which time the rolling process will finish and at that time the plate roller will start processing a new plate, which is then removed from the belt leaving room for a new plate. Now we simply schedule a new event with the exact same time where the crane can then drop its plate. Alternatively the end of rolling event can handle the extra operation of starting the plate drop of the crane, but this seems like an even more complicated approach.

Now assume that the plate roller has finished rolling a plate and the conveyer belt instead is empty. Now the plate roller must wait until a plate is dropped on the belt. The problem here is that it is not known at which time the next plate will be dropped on the belt, and the change of state for the plate roller from waiting to rolling can only be fired by an event. The solution is either to let the "start drop" event schedule a new "start rolling" event with the same scheduled time as the "finished drop" event or to let the "finished drop" event directly change the state of the plate roller and schedule a "finished rolling" event. In the first solution, we must make sure that the "finished drop" event is called before the "start rolling", otherwise the plate held by the crane is not moved to the queue. In table 3.2 on the next page the events from table 2.2 on page 30 are changed according to the event approach.

It is quite clear that more logic is built into the events, which make the events and objects more interrelated and hence more difficult to modify and to add new events to the model. This is definitely one of the conclusions which has been made from the crane scheduling project.

3.5 Summary

In this chapter we have given an overview of different approaches for solving Combinatorial Optimisation models. The span of methods cover a large

Event	**Change in state caused by event**	**Future events to schedule**
Finished *move to queue*	Crane change state to *waiting*	Start drop if room in queue else wait until room
Start *drop plate*	Crane change state to *dropping*	Finished *drop plate* and possibly start *rolling plate*
Finished *drop plate*	Move plate from crane to queue and change state of crane to *waiting*	
Start *rolling plate*	Pop plate on queue and change state of roller to *rolling*	Finished *rolling plate*
Finished *rolling plate*	Change state of roller to *waiting*	

Table 3.2: Events at the conveyer belt in the crane scheduling case.

potential set of models to be solved. This is necessary when faced with industrial applications. An additional constraint necessary to capture the industrial problem at hand may lead to a situation where a previously suitable method becomes inadequate. To be successful the developer of software to solve industrial problems must have an overview of available methods and models. The methods covered include construction and local search heuristics, a logic-based Branch & Bound framework for solving MP, CLP and hybrid models, column generation and finally simulation.

We proposed to begin the development with the simplest form of model and method and iteratively increase the complexity of both model and method. In chapter 4 we further motivate this development approach.

CHAPTER 4

A Methodology for Combinatorial Optimisation Projects

The purpose of this chapter is to present our suggestion for a methodology for software development projects where a substantial part of the business problem includes a Combinatorial Optimisation problem. The methodology has emerged from experiences gained during work on the crane scheduling case. Paper VII with the title "CIAMM: A Knowledge Driven Methodology for Development of Combinatorial Optimisation based Information Systems" is the foundation for this chapter. In CIAMM other relevant papers related to this subject have been produced by Carugati [8, 9, 10].

In section 4.1 we give a summary of the background for the methodology and in section 4.2 we give overview of the different phases in the methodology.

4.1 Background

The ideas build into the methodology have emerged from experiences gained mainly from the Crane scheduling case and existing *agile methodologies* [16] such as Extreme Programming [4] and DSDM [15] both focused on quickly adapting to changes. Also note that the Chic-2 project [11] had a similar focus

of *engineering optimisation projects.*

4.1.1 Problem Issues

We have found that the main problems in the project were due to the following issues:

1. The developers' lack of knowledge about the problem domain.
2. The customer's lack of knowledge about Combinatorial Optimisation.
3. The developers did not have sufficient focus on solving the customer's problem.
4. There was no clear view of the requirements for the software.
5. Both the customer and developer side had too much invested knowledge.

The key to all the above points is, in our opinion, the concept of *knowledge exchange.* The amount of knowledge exchanged between the parties must be high in order for the developers to learn about the business domain and problem and for the customer to learn about the possibilities of Combinatorial Optimisation and the developed software.

In most projects the customer initially lacks a clear picture of the software requirements. Much time is spent in order for the developers to capture the requirements before starting development, but the final product will most likely not fulfill the initial requirements and it will certainly not be what the customer really wants. When the developers finally realise that the software is not what the customer wanted, they are reluctant to change it. Too much time and effort has been invested. They will instead try to convince the customer that the software, perhaps with small changes, can be valuable.

This phenomenon we refer to as the problem of *invested knowledge.* Actually it can be both a problem emerging from the customer and the developer side. The customer organisation has invested a large amount of knowledge in its current way of operating. Information Systems and Combinatorial Optimisation often give new possibilities for organising the work processes, but the organisation is reluctant to change.

From the developer side, a set of tools are learned and used to solve the posed problems. When other problems appear the developer will often try to apply the same set of tools, even though better tools might exist. Perhaps the developer is unaware of the other tools or unwilling to invest more in learning new tools.

4.1.2 Ideas of the Methodology

Several ideas are incorporated in the methodology to facilitate the process of knowledge exchange and to reduce the risk of invested knowledge causing problems:

1. Short iterations.
2. Simple iterations.
3. Discussions based on software.

After each iteration of development, the software is shown to the users or even better integrated and used in production. It must be possible for the users to **use** the software in order to give concrete feedback. Discussions must be based on the software **not** presentations about the software. For the users to be able to comment on the software, the development steps and difference in complexity must be small. This calls for short iterations with limited new functionality. Based on the discussions the customer's knowledge about the software will increase and new requirements will emerge. The developers will gain additional knowledge about the domain and requirements for the software. The short iterations result in more frequent meetings and increased knowledge exchange. The amount of invested knowledge and effort invested in the software will decrease simply because less time has passed since the last meeting. The developers will hence be less reluctant to change the software according to the changed requirements.

Software with a substantial content of Combinatorial Optimisation is very complex black-box type software with limited user interaction and it is often time consuming to build. Depending on the degree of innovation needed to build the system, the degree of risk must be considered as well. Our idea is to split the development into smaller and simpler components, which are visualised for the user. In that way, the software can be used for educating the users in Combinatorial Optimisation and for exchanging knowledge about the problem to be solved – *we need to open the black box.* Two questions emerge:

- How is the development of a Combinatorial Optimisation component split into smaller components to be developed in separate iterations?
- How can these smaller components be visualised for the user?

In the following is given an example of the headlines of the planned development in the first 5 iterations of a development project where the Combinatorial Optimisation components have been split into several development tasks:

1. A database and simple GUI to do manual planning and visual simulation.

2. Constraint checks and objective calculation. The user uses the software manually but is alerted if constraints are violated.

3. Simple construction heuristic that can automate the work of the user. The user can compete against the heuristic in an optimisation game. Here visualisation is essential to create a realistic test environment for the user.

4. The neighbourhood structure is developed and the user can iterate through the possible neighbour moves, which are evaluated by the system.

5. A local search heuristic is developed.

We believe that the above scheme is sufficiently general to be widely applicable in Combinatorial Optimisation projects. Obviously the pace of the project must be adjusted to the complexity of the problem to be solved and the experience of the customer and developers in these types of projects.

4.2 The Phases of the Methodology

The methodology is divided into the 6 phases shown below. All phases are summarised in the remainder of this chapter. For further details we refer to paper VII. In fact, the phases concerned with modelling are considered in chapter 2 as well and chapter 3 has more details on designing optimisation methods. In this chapter we are more focused on the non-technical issues of these types of software development projects.

1. Business Problem

2. Conceptual Modelling

3. Iteration and Release Planning

4. Modelling, Design and Implementation

5. Evaluation

6. Integration

4.2.0.1 Business Problem

First the business problem is identified including the possibility of using Combinatorial Optimisation. The customer is driving the process in this phase, but the developer side is facilitating the process. This phase should result in a vision of the final system and success criteria for the project. Note that these might well change during the project when more knowledge is gained. The initial version of the business problem is defined at a workshop where all stakeholders attend. During the iterations the definition of the business problem is monitored and updated when necessary. The duration of this phase should be less than a week.

4.2.0.2 Conceptual Modelling

In the phase of conceptual modelling, the customer and developers in cooperation determine the requirements of the software system. This calls for substantial knowledge exchange, but the most efficient knowledge exchange occur on the basis of concrete software. The purpose of this phase is hence not to reach a complete understanding of the domain and requirements, but to determine a coarse sketch or model of the domain and the future system.

Splitting up the entire system in smaller and simpler development tasks is a prerequisite for allowing the system to be delivered in smaller iterations. The purpose of user stories reviewed in chapter 2 is not a detailed requirement specification, but rather a subject for discussion and input for the iteration planning discussed later. When the user story is going to be implemented the user will be asked for further details. Beck and Fowler [5] have an extensive discussion on writing user stories.

4.2.0.3 Iteration and Release Planning

Every iteration includes the phases from 1 to 5. In case of a release, the software is integrated as well. The definition of the business problem is usually only done once in the beginning of the project, but it may be revised during the project as well. Each iteration should take from 1 and up to 4 weeks.

In this phase the user stories are prioritised and a detailed plan is made for the current iteration and less detailed for the following iterations. Also releases are planned at this stage. We emphasise that the planning is based on the customer's priority of the user stories and the developers' time estimates for developing the stories (Beck [4]). This is to ensure value for the customer

and a hopefully realistic time schedule for the developers. The plans will be updated after each iteration and during iterations when needed.

4.2.0.4 Modelling, Design and Implementation

The development team initially develop the simplest form of optimisation method for the given optimisation model. Often the simplest thing to do is to develop a construction heuristic. The advantage of this approach is that construction heuristics are fast and easy to develop, easier for the customer to understand and the developers will quickly get feedback from the customer. For further details refer to chapter 3 and paper VII.

4.2.0.5 Evaluation

In this phase the iteration and progress of the entire project is evaluated. This is done at a one-day meeting where the developers present the results of the last iteration. The results are discussed and the software is demonstrated to the users. The following use of the software by the users is the foundation for generating new knowledge and hence new user stories.

Iteration and release planning is done immediately after the evaluation possibly at the same meeting. Hereby a new iteration is initiated.

4.2.0.6 Integration

The best way of getting feedback from the customer is when the software is used in daily operation. The software should hence be released as soon as possible. To achieve that, the customer selects the smallest subset of user stories for the first release, which will be sufficient for taking the software into operation. New versions of the software is released as often as possible giving value to the customer as early as possible in the project.

The costs of preparing integration might involve large initial investments, which might only be justified, if the software system results in a significant value for customer. The risk is that the customer will only accept to integrate the system when it is more or less finished, significantly reducing knowledge exchange.

4.3 Summary

We have in this chapter summarised our proposed methodology for software development projects involving difficult real-life Combinatorial Optimisation

problems. The methodology is focused on facilitating knowledge exchange, which is particularly difficult in these types of projects. Short and simple iterations have been suggested as means for increasing knowledge exchange. This however introduces another problem related to splitting up the development of the optimisation component into smaller development tasks suitable for these short iterations. A suggestion for splitting up the development of a local search heuristic is put forward.

CHAPTER 5

Summary of Case Studies

This chapter summarises the two cases we have considered in the CIAMM project. The outline for both cases is a brief problem definition and modelling issues, chosen solution approaches, experiments and achieved results, and finally we discuss perspectives of the case.

5.1 Crane Scheduling for a Plate Storage in a Shipyard

Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce the ship. The purpose of the project was to investigate possible improvements in handling the storage of steel plates at Odense Steel Shipyard (OSS).

Paper I is concerned with defining and modelling the problem, paper II with the chosen solution approaches and finally paper III reports on the executed experiments and achieved results. Paper IV is an earlier report not as detailed as the other three, but is included for completion and significant additional results are given.

5.1.1 Problem Definition and Modelling

The plate storage contains the plates used for building blocks. The storage is a buffer between the suppliers of the plates and the production at OSS.

The plate storage is organised in 32 times 8 stacks of plates. The size of the storage is 600 × 35 meters. The storage contains 5.000 steel plates on average – approximately 20 plates are stored in each stack on average. One quarter of the stacks are used for plates with special purposes, which we will not consider. The rest of the storage is used for plates, which for the main part have different sizes. Each plate is ordered for a specific purpose and the date on which it will be required in production is known. Changes in the overall production plan, can however change the due dates.

Two identical gantry cranes carry out the movements of plates. The cranes share tracks and hence cannot pass each other. Arriving plates are moved to an appropriate stack in the storage by one of the gantry cranes.

Each day a set of plates must leave the storage to be processed. The plates are put on a conveyer belt called the exit-belt, which has a capacity of 8 plates. The exit-belt is the start of a production line where several processes are performed. The exit-belt works as a queue and plates are drawn from the end of the queue when one of the two machines in the following process becomes idle. The time interval between plates being drawn is depending on the order and dimensions of the plates on the belt.

As mentioned earlier, ships are constructed by cutting up plates and afterwards welding them together to form blocks. Finally the blocks are welded together to produce the ship. Currently the stacks in the plate storage are assigned to different blocks, and plates are placed in stacks according to the blocks for which they will be requested. Often the requested plate is not on top of the stack and a lot of unproductive movements are required to find the plate. The plates requested for production on the next day are placed in so-called sort stacks near the exit-belt, such that they can be put on the exit-belt without delay.

The problem is to develop approaches for scheduling the crane operations better than the current practices of the block storage explained above. Two alternative storage principles are considered:

The due date storage principle assigns a due date interval to each stack. A plate with a due date in the interval of a given stack can be placed on that stack. The storage is divided into zones and each zone is divided into a number of due date intervals. Several stacks are assigned to each due date interval. The user can determine the layout of the storage by

specifying different parameters regarding size of the intervals, stacks per interval, number of intervals and overlap in due dates between zones.

Figure 5.1 on the next page gives an example of a due date layout. The numbers in the fields indicate the due date from today. The 8 stacks (9,1) to (16,1) have the plates due today and stacks (25,1) to (32,1) for example have the plates with a due date in the interval 21 to 24. We refer to the paper I for details on how the due dates of the intervals are updated over time.

The self-regulating storage principle is the second alternative. No specific purpose or due date is assigned to the stacks. The organisation of the storage is determined completely by the planning procedure.

5.1.2 Solution Approaches

Two approaches based on the different principles have been investigated to solve the scheduling problem. The first approach is an *on-line algorithm* or more precisely a *heuristic discrete event feedback control system* where a movement is chosen in real-time. The other approach uses the on-line algorithm off-line as a greedy construction heuristic to get a good initial solution, which is then improved by local search heuristics. The off-line algorithm makes a schedule for the controller to follow. The schedule specifies the order in which the movements are to be executed. A control module dispatches the movements in the sequences and adjusts the schedule if the initial schedule would become infeasible because of the underlying stochastic nature of the system. The local search heuristics implemented are a simple descent algorithm, a Simulated Annealing and a Tabu Search algorithm.

Several objectives are optimised: Number of plate movements, distance moved by the cranes and makespan for all plate movements and plates moved to the exit belt. One working day is solved at a time. For the self-regulating principle, objectives taking future number of movements into account are also included. Basically they seek to reach a storage with as well sorted and low stacks as possible. Further an objective seeks to avoid long travelling distances to the exit-belt. All objectives are weighted together to create a single objective used in the local search procedure. The most significant objective is the number of movements, secondly the makespan of plates moved to the exit belt and thirdly the makespan of all movements.

The complexity of the problem makes it impossible to evaluate feasibility and change in cost for neighbour solutions without simulating the entire

Y \ X	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
8	8							8	22-23							22-23	49-53							49-53
7	7							7	20-21							20-21	45-48							45-48
6	6							6	18-19							18-19	41-44							41-44
5	5							5	16-17							16-17	37-40							37-40
4	4							4	14-15							14-15	33-36							33-36
3	3							3	12-13							12-13	29-32							29-32
2	2							2	10-11							10-11	25-28							25-28
1	1							1	8-9							8-9	21-24							21-24
	Zone 1								Zone 2								Zone 3							

Figure 5.1: Illustration of the due-date storage layout.

sequences of movements. This makes the problem very time consuming to solve and was the reason for conducting experiments in estimating the change in cost. The experiments showed a significant speedup of 352 times, but the estimates were only correct in about 40-50% of the trials. Here correct means that the direction or sign of the change for the estimate is the same as the correct change. In 30-40% of the trials the estimate was too optimistic actually leading to a worse solution. The conclusion is that a local change implies a global change to the solution, which is difficult to predict.

5.1.3 Experiments and Results

The major goal of the investigation has been to compare the three proposed ways of managing the steel plate storage: The block, due date and self-regulating storage. Experiments reported in paper IV show that a change from the block principle to any of the other two would result in a saving of approximately 50% in number of movements and 40% in total makespan. Hence there is a significant potential cost saving or alternatively a potential of almost doubling the productivity with the same cost even though the number of plates on the storage will obviously be much larger.

Then we turned our attention to comparing the different storage principles, storage layouts and objective weights. The experiments indicate that significant improvements can be achieved by adjusting the parameters. No final conclusions were reached on best layouts or weights. The management and users of the system should take these decisions based on the overall goals of the organisation.

To conclude, the on-line control approach is more robust to time disturbances and is superior to control with use of a plan for the due date principle. The self-regulating principle on the other hand is dependent on improving the plan initially achieved with the on-line control approach. Better solution times can be achieved with the self-regulating storage compared to the due date storage. Approximately the same number of movements is required. The user of the software can however adjust the behaviour of the self-regulating storage to focus more on reducing the number of movements than time, reducing the number of movements down to 4 per plate. Compared to current practices this is a reduction by 67% and at the same time a reduction in time by 39% leading to an estimated saving of approximately 1.0 mill. DKr. per year in mainly maintenance costs and salaries. Conversely the production capacity can be increased.

5.1.4 Perspectives

The development of the system and these investigations should not end here. Significant improvements can be made to the control module based on a plan and the planning module itself. One must however take into consideration whether the more complex approach of control with a plan is sufficiently more promising than the more simple on-line control approach, which might be sufficient for the purposes of the yard.

5.2 Solving the non-oriented three-dimensional bin packing problem with stability and load bearing constraints

In section 1.2.2 we introduced the case of packing items in boxes and on pallets. This is also the subject of paper V, which is summarised in the following sections.

5.2.1 Problem Definition

The three-dimensional bin packing problem (3D-BPP) is concerned with orthogonally packing a given set of rectangular items in rectangular bins. The goal is to minimise the number of bins used. We are interested in solving real-life problems, and hence additional constraints must be taken into account. It is generally allowed to rotate the items in the 6 different possible orientations, but restrictions on some items can occur, allowing only a subset of the orientations. The packing of items must be stable and possible to pack by a person or a packing robot. In order to guarantee this we will restrict ourselves to so-called robot-packable packings. Basically items are placed starting from the bottom-left-back corner of the bin and successively placing items in front, on top or to the right of already placed items.

We are also interested in solving the on-line version of the problem introduced above where the order of the items arriving at the packing site is unknown. The items are in a queue, where we can observe and pick an item to pack from the first Q items. At the packing site S bins are available to pack at a time. When no more items can be packed in the bins one or more of the bins must be shipped off and replaced by one or more empty bin(s).

5.2.2 Modelling

To solve the on-line version we used a greedy heuristic while the off-line version was solved with a Branch & Price approach.

In the Branch & Price approach we solve the Restricted Master Problem as given in (5.1):

$$\begin{aligned} \min \sum_{p \in \mathcal{P}'} c_p q_p & \\ \sum_{p \in \mathcal{P}'} a_p^t q_p = d_t, \quad & t \in \mathcal{T} \\ q_p \geq 0, \quad & p \in \mathcal{P}' \end{aligned} \tag{5.1}$$

c_p is the cost of packing p depending on the cost of the used bin type, a_p^t is the number of items of type t in packing p and d_t is the number of items of type t to be packed.

The pricing problem is a 3D Knapsack Problem, which is decomposed into a 1D Knapsack Optimisation Problem and a 3D Knapsack Feasibility Problem. The 1D Knapsack Problem is a relaxation of the 3D Knapsack Problem. Figure 5.2 on the following page gives an overview of the entire column generation scheme. Duals from the Restricted Master Problem are sent to the 1D Knapsack relaxation. The optimal solutions are checked for 3D Knapsack feasibility. Either a cut is sent to the 1D Knapsack or a feasible packing is sent to the Restricted Master Problem. This procedure continues until no feasible packings exist, which will improve the LP solution of the Restricted Master Problem.

5.2.3 Solution Approaches

5.2.3.1 The 3D Knapsack Feasibility

For solving the 3D Knapsack Feasibility problem we use a variant of the ONEBIN procedure by Martello et. al. [27]. The items are packed in so-called corner points. Initially the only corner point is the lower-left-back corner of the bin. At any following stage, items can only be placed to the right, above or in front of already placed items. The procedure is a depth first search where we at a given node consider placing all item types in all possible corner points and with each feasible item rotation. Each time an item is placed, the corner points are updated. Figure 5.3 on the next page is an illustration of available corner points for a given set of packed items.

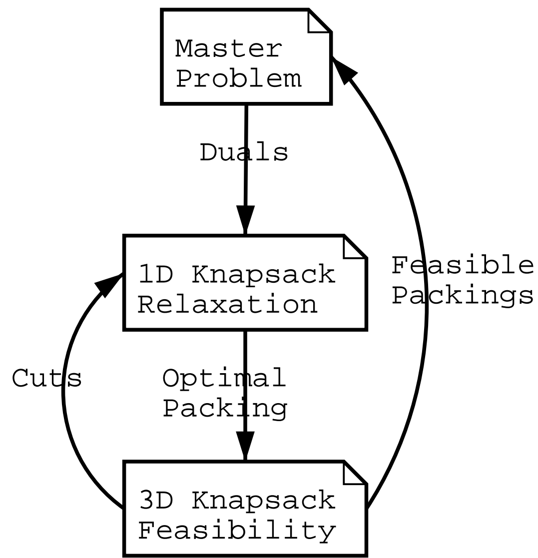

Figure 5.2: Overview of the solution approach.

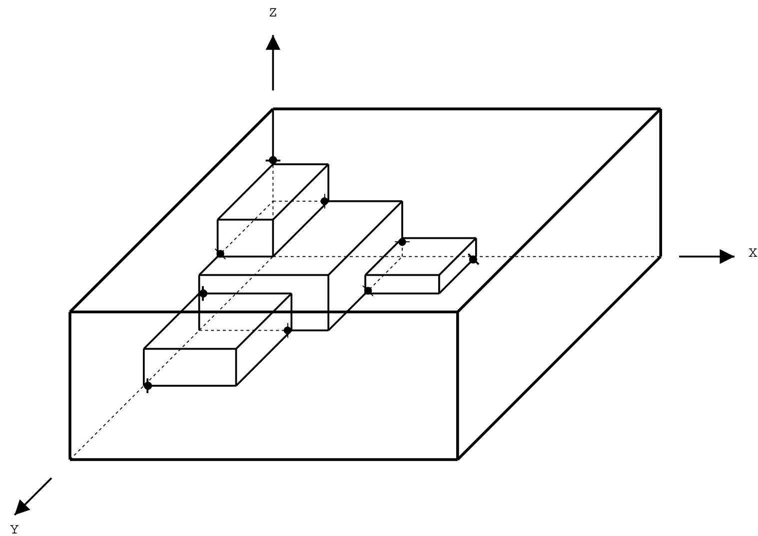

Figure 5.3: Illustration of corner points for a given packing.

In real-life there is requirements on how items may be packed. Items should be placed on the floor of the bin or be somewhat supported underneath by other items. A common measure of support is to require a certain percentage of an item to be supported – for instance at least 90%. Items supported by more than one item, may result in more stable packings. Our packing procedure is able to take item support into consideration.

Also the load bearing ability of the packed items must be taken into account. We assume that a maximum allowed load per square unit for each item type is given. The implemented procedure checks if an item can be placed at a specific location in the bin. First the weight on the contact area of items below is determined. Then it is checked, if the items below can bear the load of the additional item.

5.2.3.2 Branch & Price

For the Branch & Price, we have used the branching scheme due to Vanderbeck and Wolsey [34]. Generally we impose upper and lower bounds on the sums of columns covering certain rows. The extra constraints in the restricted master problem introduce both additional duals and constraints in the 1D Knapsack pricing problem. The model becomes somewhat more complicated and difficult to solve, but the computation time is still insignificant compared to the 3D Knapsack Feasibility problem. We refer to paper V for details on the branching scheme.

5.2.3.3 Lower Bounds

Two types of lower bounds are employed. First, bounds derivable from the instance data are used for finding bounds on the total number of bins and to quickly determine if the set of chosen items to pack cannot be packed in the bin. Second, a Lagrangian bound of the restricted master problem is calculated to possibly improve the bound in the Branch & Bound tree.

5.2.3.4 On-line heuristic

An on-line greedy heuristic is developed to solve the on-line packing problem. When considering which item to place next, we evaluate the placement of each available item in each available bin, in all available corner points and all possible rotations. We primarily choose that combination of item, bin, corner and rotation where the increase in wasted space is minimal and secondarily we pack larger items before smaller items. Wasted space is basically space, which

is not used by an item, but is unavailable for packing after the particular item is placed. When none of the available items can be placed, we replace all filled bins with new empty bins. The procedure continues until all items are placed.

The algorithm is not only used for solving the on-line problem, but is also used off-line to generate an initial upper bound and first columns in the Branch & Price approach.

5.2.4 Experiments and Results

The on-line heuristic and Branch & Price method have been tested on new instances that we have generated from real-life data made available by Bang & Olufsen. The Branch & Price method was also compared against other approaches on available benchmark instances.

The 810 instances due to Martello et. al. [27] showed that our approach was not as competitive as expected. 344 instances was solved to proven optimum while they solved 698 even in less computation time.

47 instances due to Ivancic et. al. [23] were solved with and without 90% support requirement. The gap between lower and upper bounds was only 3.4% without support and 3.8% with support. Actually the support requirement resulted in an increase in upper bound of only 6.5%. Compared to results from competing heuristics, the performance of our implementation was superior. Another important issue is the ability of the Branch & Price approach to supply lower bounds and prove optimum.

5.2.4.1 Bang & Olufsen instances

The Bang & Olufsen instances have been generated from real-life data on item sizes, weight, load bearing ability and allowed orientations. Stability in the form of 90% support was also required. Data on 21 different item types were used for creating 9 instances.

In the following experiment the on-line heuristic could consider 20 items at a time and only place them on one pallet. The packings in the on-line solution was the starting point of the Branch & Price algorithm. Note that the Branch & Price is allowed to consider all packings, i.e. there are no restrictions on the packing order of the items. From another point of view, the improvement of using Branch & Price compared to the on-line heuristic constitute the potential gain from allowing removal of items from the bins for re-packing.

In case of an on-line problem, we can only use the Branch & Price for comparing the performance against the on-line algorithm, and not directly use the solution in operation. The lower bound of the Branch & Price algorithm

is however still valid, but the upper bound might not be achievable for the on-line problem. For these instances a limit of 3600 seconds of computation time was available.

The initial gap between the lower and upper bounds was 35%, which was reduced by the Branch & Price to 28%. Still a quite large gap, indicating that the instances were quite difficult to solve (or prove optimality). The saving by using Branch & Price for these test cases was only 10%. Of the total time approximately 99% of the time was spent in the one bin packing algorithm finding feasible packings or proving infeasibility.

We made several experiments with the on-line algorithm investigating the impact of stability and load bearing requirements. Comparing the case with 0% support **without** load bearing to the case **with** load bearing, the number of pallets increases by approximately 12%. Again requiring 90% support, the increase is 10% in the number of pallets. There is no significant difference between 90 and 100% support.

To test the robustness of the on-line algorithm we did several simulations shuffling the order of item arrivals. The algorithm seems quite robust with a little spread in solution quality, but not that surprisingly the arrival order gets more important when fewer items are visible.

5.2.5 Perspectives

A complex exact approach has been proposed and implemented. For some instances good results were achieved, but the performance needs to be improved to be useful in an industrial environment. The approach can however still be used heuristically without loosing the nice characteristics of an exact method, such as lower bounds.

The Branch & Price algorithm had difficulties improving the solutions of the on-line algorithm for the Bang & Olufsen instances. This indicates that sufficiently good solutions can be expected for the on-line problem.

There is a significant challenge in developing optimisation methods for solving real-life packing problems. Two types of extra requirements were considered: Stability and load bearing. Many other special requirements can be identified and some cannot easily be handled in the methods of today, but to be successful in industry these obstacles must be handled.

5.3 Summary

In this chapter we have summarised the two cases considered in the project. Many details are of course left unmentioned, but we invite the reader to study those in the appropriate papers. Two very difficult real-life problems have been considered in the project in order to gain experience in moving from solving prototype problems with origins in academia to solving industrial problems. We believe that people from both academia and industry will find the reported experiences useful in their own working area.

CHAPTER 6

Conclusion

In this final and concluding chapter we first give an overview of the issues considered in the thesis. We then comment on the achieved results and finally we discuss the perspectives of the work and potential future research topics.

6.1 Overview

We have considered issues related to solving real-life industrial planning problems. Two industrial cases were considered each of them interesting in their own right. In addition these were used to gain experience resulting in general guidelines for developing Combinatorial Optimisation based software.

The research has focused on three major areas:

- A methodology for software development projects where Combinatorial Optimisation is a substantial part.
- Modelling in software development and the relation to modelling of Combinatorial Optimisation problems.
- Solving industrial Combinatorial Optimisation problems.

All three topics are considered in the papers through the two industrial cases and other articles as well as in the preceding chapters.

In the chapter on modelling, the general concept of models and Combinatorial Optimisation models were introduced as well as different categories of models including static/dynamic, deterministic/stochastic. Further, the concepts of object-oriented modelling were briefly introduced. Object-oriented modelling is used in software development and hence is highly relevant when building simulation and local search models. Finally, examples of MP and CLP models were discussed. The purpose of the chapter was to give a broad overview of models to illustrate the diversity in available model types and also bring Combinatorial Optimisation models closer to the software models based on object-orientation. Further, models are often described in connection with the methods to solve them. Each model and corresponding method(s) are described in separate sections, but here we have viewed modelling as a process somewhat disconnected from the methods for solving the models. This is to avoid the temptation to first pick the method and then try to build the best suitable model within that framework. The appropriate method should be chosen based on the developed models.

In the chapter on solution methods the different categories of methods were introduced, all applied in the two industrial cases: Heuristics, optimisation methods and simulation. The examples given are mainly based on the chapter on modelling and the two industrial cases. The goal was again to give an overview of the most relevant methods for solving industrial Combinatorial Optimisation problems.

The chapters on modelling and solving were primarily intended as introductory and supplementary to the the papers and to give an overview of possible approaches to modelling and solving industrial applications. In the following two chapters the methodology and industrial cases were summarised.

6.2 Results

The results are twofold: Results achieved directly in the industrial cases and results achieved based on our experiences from the industrial cases. The first results are discussed in chapter 5 and papers I to V. The second relates to the methodology described in paper VII and chapters 2, 3 and 4 in particular.

The new angle taken in the project was the combination of applying the technology of Combinatorial Optimisation to real-world industrial problems **and** reporting on the achieved experiences, resulting in a new methodology for Combinatorial Optimisation projects. Often in research projects the focus is only on one of these issues. When focus is on applying the technology, the publication of experiences gained from practical projects is lost, and when

the focus is only on developing methodologies, these are primarily built on theoretical considerations.

6.2.1 The Crane Scheduling Case

The goal of the crane scheduling case was to compare current operations at OSS with the potential of using different local search and control approaches in combination with simulation. The results were quite promising showing potential savings of up to 50% in both time and cost. The results have been partly responsible for the decision of the yard to continue the process of implementing a database to create the data foundation for implementing these types of software solutions.

6.2.2 The Packing Case

The goal of the packing case was to develop methods for solving 3-dimensional packing problems. An on-line algorithm was developed to solve a dynamic packing problem and an exact method was also developed to consider the static version for comparison. The exact method was a Branch & Price approach where the pricing problem was a 3-dimensional knapsack problem solved by a Branch & Cut algorithm. Here a 3-dimensional knapsack feasibility problem supplied feasible packings to the master problem or infeasibility cuts to a 1-dimensional knapsack problem. Test instances were generated based on real-world data. The problem is very complex, but the results were quite promising for the on-line heuristic. The exact method was able to improve the upper bound by 10%, but still leaving a gap of 28% to the lower bound. The exact method was also tested on other instances indicating that the method was best suited to instances with relatively few item types. Generally 99% of the computation time was spent in determining whether a combination of items could be packed together. A number of issues of practical importance were taken into account: Stability of the packings were considered by requiring a certain amount of support of the bottom face of the items and also weight bearing of the items were considered.

6.2.3 Methodology

We have proposed a methodology for Combinatorial Optimisation software development based on ideas from existing methodologies and experiences gained from the industrial cases. The crane scheduling case has been the primary source of inspiration. The problem is very complex and too much time was

spent in the beginning of the project gathering information and constructing the first prototype. The result was a very complex first prototype, which was difficult to grasp for the project members at the yard due to sparse communication during the development. At that stage it was difficult to verify the validity of the model and the achieved results. This experience amongst others had a very direct impact on the content of the methodology.

The customer of the software has a lack of knowledge about Combinatorial Optimisation and the developers have a lack of knowledge about the problem domain. Additionally, the customer has a lot of invested knowledge making it difficult for them to consider alternatives to their current situation. Further, the developers often have a lot of invested knowledge in methods they prefer to apply and these methods may not be the most appropriate for the problem at hand. Finally during the software development too much invested effort in the developed software might induce reluctance to redo things.

The solution to these problems is to improve knowledge exchange and to limit the invested knowledge through short and simple development iterations with use of software (prototypes) as an object for discussion. The short and simple iterations imply that the increment in the complexity of the software will be relatively small and hence easier to grasp for the customer and less expensive to redo for the developers.

A prerequisite for this approach is the possibility of decomposing the development of the software into smaller development tasks fitting into short iterations. The challenge here is exactly splitting up the Combinatorial Optimisation component, since it is usually a black-box type of component with limited user interaction, and difficult to visualise for the user. We therefore propose a sequence of development tasks for Combinatorial Optimisation software stressing simplicity, visualisation and user interaction to “open the black-box” for the user.

To summarise, our main contribution has been to identify the main problems in development of tailored Combinatorial Optimisation software and to propose a new methodology for attending these particular types of software development projects focusing on the Combinatorial Optimisation specific issues.

6.3 Perspectives and Future Research

The purpose of the research project described in this thesis was to bridge the gap between academic research in so-called prototype problems and industry requesting solutions for complex real-world problems.

The outcome was a methodology (development process, guideline or cookbook) addressing issues evolving during the process of solving real-world cases in the project. This is obviously only a step in the direction of closing the gap – more research is needed. Experiences from other cases applying the methodology should be gathered, resulting in adjustments of the methodology. Ideas from other methodologies have already been adopted from e.g. Extreme Programming and DSDM, but with our special focus on issues related to Combinatorial Optimisation. More research is needed on software development in connection with Combinatorial Optimisation including issues related to developer/user communication.

On the technical side, we have advocated for using a large tool box containing a multitude of approaches when solving the problems in industry. Further, we suggest attacking the problems with simplicity in mind and gradually increase the complexity of solution methods and models if required. Interesting areas of research on the technical side are in hybrid approaches combining MP and CLP and the combination of simulation and heuristics. Further, the most promising tools for industry right now are local search heuristics/metaheuristics because of their high degree of flexibility in taken complex constraints and objectives into consideration.

The lack of experience in academia of solving real-world problems should be addressed by increased collaboration between academia and industry. Academia must leave the ivory tower and get their hands dirty solving real-world problems instead of always solving the same old "standard" prototype problems. Companies developing systems for solving real-world optimisation problems obviously exist, but many areas in industry have not yet been considered. We believe that an increased focus in academia is necessary to create the impact in industry and other parts of society that we believe the techniques call for. The academic community must take the first step to bridge the gap.

Appendix A

An Object-oriented Local Search Framework

An object-oriented local search heuristic framework has been implemented in Java. The framework is based on the design of Andreatta et. al. [1] though certain changes have been made. In this appendix we describe the components of the framework including what the user of the framework must provide.

Figure A.1 on the following page is a class diagram illustrating the overall structure of the framwork. The user of the framework must create an object of the class HeuristicStrategy, which initiates the construction heuristic and local search heuristic. The class Solution represents a solution of the Combinatorial Optimisation problem and the interface to be used by the construction and improvement heuristics. The Solution class must be subclassed and implemented for the given Combinatorial Optimisation problem. The construction heuristic for the given problem must be a subclass of ConstructionStrategy.

The implemented neighbourhood structures for the given problem must be subclasses of MovementModel. For Tabu Search the tabu handling must also be implemented in the subclass. In the Andreatta et. al framework the tabu handling is generalized and hence handled by the Tabu Search implementation, which leaves less work for the user of the framework.

The improvement heuristics are subclasses of ImprovementStrategy as indicated in figure A.2 on page 77. All subclasses implement the improve method, which is the actual local search heuristic . Generally applicable stop criteria

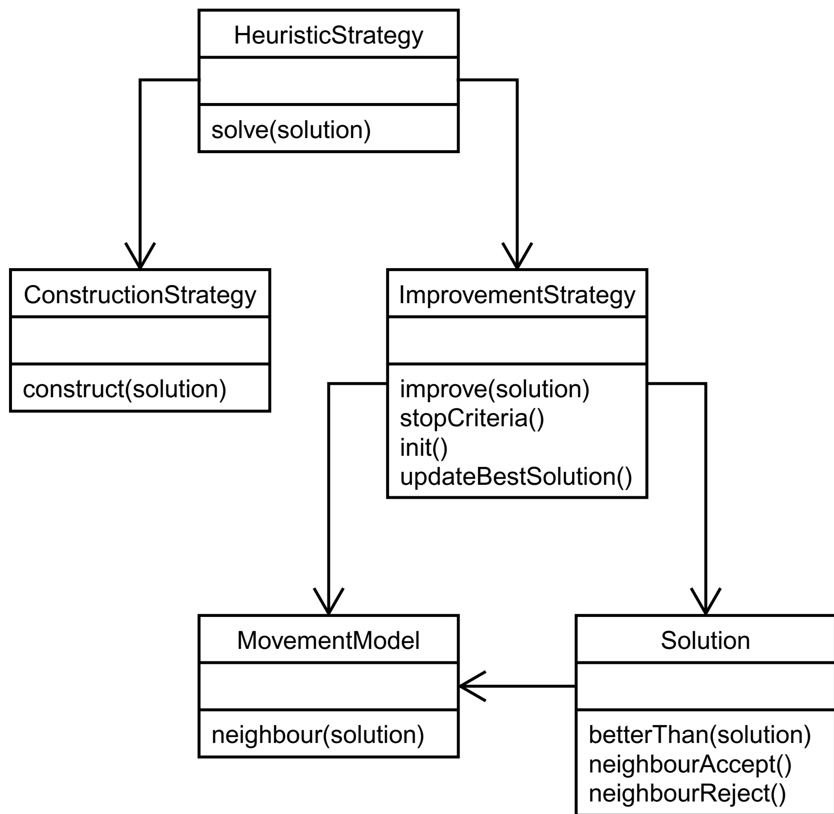

Figure A.1: Overall structure of the local search framework.

are implemented in ImprovementStrategy, which also manages initialization of the search, status messages, a file log, keeping track of the best found solution, iterations and elapsed time.

Algorithm 2 on the facing page shows the improve method of the implemented Tabu Search class. The currentSolution is the object corresponding to the current solution. First initializations of the search are done and afterwards the search begins. Until the stop criteria is fullfilled, the neighbourhood structure object, neightbourhood, is asked to return a neighbour for the current solution. Note, that the neighbourhood structure is required to handle all tabu registration and checking. If the neighbour solution is better than the best solution found so far, the best solution is updated. Finally the solution object is called to make possible updates of its internal structures before

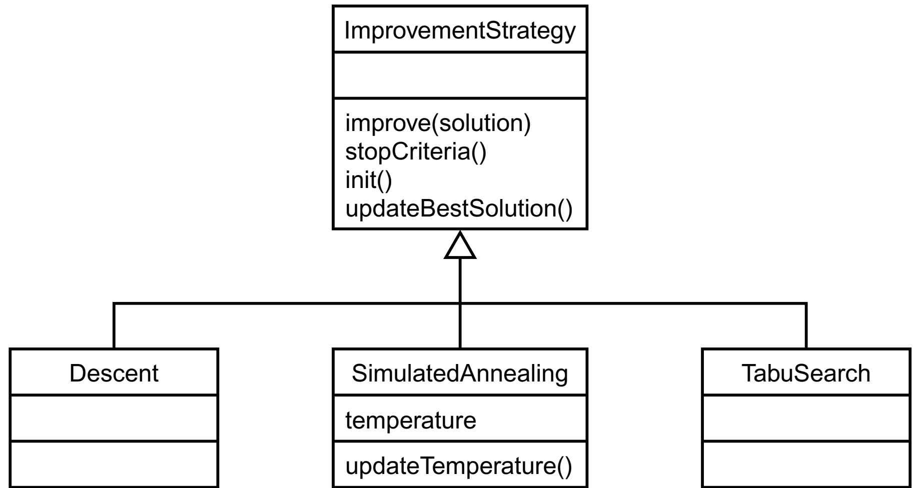

Figure A.2: Improvement heuristics of the local search framework.

continuing with the next iteration.

Algorithm 2: improve(currentSolution) for TabuSearch

Data : model is the object implementing the neighbourhood structure.
Result : Best solution found.
init(currentSolution)
while ¬*stopCriteria()* **do**
  neighbourhood.neighbour(currentSolution)
  if *currentSolution.betterThan(bestSolution)* **then**
    updateBestSolution(currentSolution)
  currentSolution.neighbourAccept()

Algorithm 3 on the next page shows the improve method implemented in the Simulated Annealing class. It is somewhat simplified since in the real implementation the search is divided into two phases: The objective of the first phase is finding a good initial temperature, and the second phase is the usual local search heuristic . In the algorithm we assume that the initial temperature has been set. In the local search loop the neighbourhood structure again returns a new neighbour solution. Note that the neighbourhood object

for Simulated Annealing is different than for Tabu Search, since in Simulated Annealing usually a random neighbour is picked, while in Tabu Search the first improving or best non-tabu neighbour is chosen. The cooling schedule

Algorithm 3: improve(currentSolution) for SimulatedAnnealing

```
Data   : model is the object implementing the neighbourhood structure.
Result : Best solution found.
init(currentSolution)
while ¬stopCriteria() do
    model.neighbour(currentSolution)
    if acceptNeighbour(currentSolution) then
        if currentSolution.betterThan(bestSolution) then
            updateBestSolution(currentSolution)
        currentSolution.neighbourAccept()
    else
        currentSolution.neighbourReject()
    updateTemperature()
```

is implemented in the updateTemperature method. Note, that in Simulated Annealing the neighbour solution can be rejected, which might lead to other internal updates of the solution object. This is handled by the method neighbourReject similar to neighbourAccept when the solution is accepted.

In figure A.3 on the facing page is shown a class diagram including the different classes implementing the MovementModel interface for the crane scheduling case. MovementModelImp is the base class including different methods and attributes useful for the concrete classes, which are MovementModelSA for the Simulated Annealing, MovementModelBest for the Steepest Descent algorithm, MovementModelFirstBest for the Descent algorithm and MovementModelTabu for the Tabu Search.

The framework has been implemented to be applicable to any type of problem solvable with local search heuristics. The user only have to implement subclasses of the types Solution, ConstructionStrategy and MovementModel.

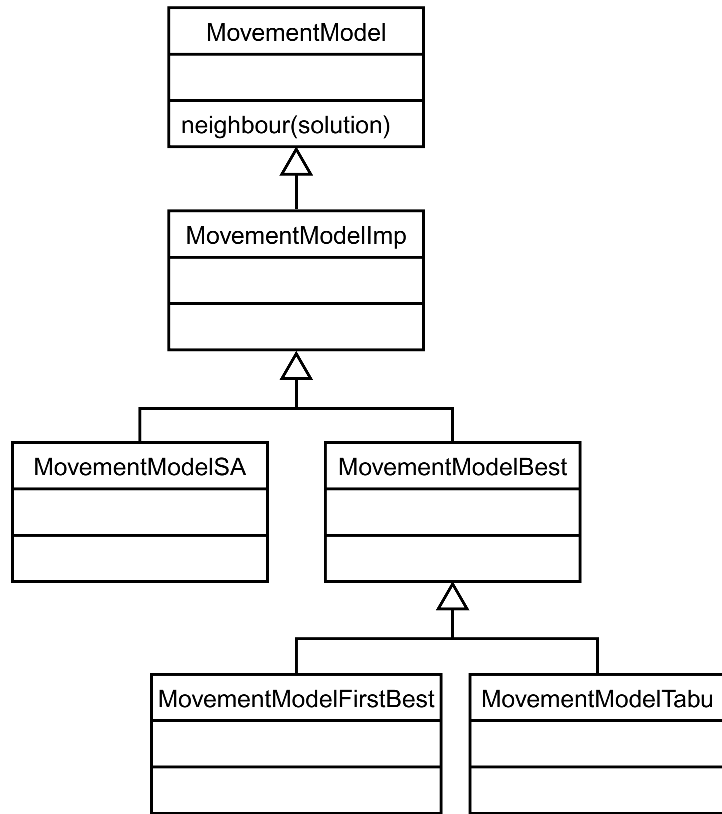

Figure A.3: Classes implementing the MovementModel interface.

Bibliography

[1] A. A. Andreatta, S. Carvalho, and C. Ribeiro, *A framework for local search heuristics for combinatorial optimization problems*, in Voss and Woodruff [35], 2002, ch. 3.

[2] M. Ball, T. Magnanti, C. Monma, and G. Nemhauser, eds., *Handbooks in Operations Research and Management Science*, vol. 8, Elsevier, 1995.

[3] R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing, 6 (1994), pp. 128–140.

[4] K. Beck, *Extreme Programming Explained: Embrace Change*, The XP Series, Addison-Wesley, 1. ed., 1999.

[5] K. Beck and M. Fowler, *Planning Extreme Programming*, The XP Series, Addison-Wesley, 2001.

[6] S. Bennett, S. McRobb, and R. Farmer, *Object-Oriented Systems Analysis and Design using UML*, McGraw-Hill, 1999.

[7] R. Brooks and S. Robinson, *Simulation*, Operational Research Series, Palgrave, 2001.

[8] A. Carugati, *Information system development: Can traditional project management practices be succesful in distributed organizations*, in proceedings of the Seminar in Operation Management and Innovation, 2002.

[9] ——, *New challenges for the management of te development of information systems based on advanced mathematical models*, in proceedings of the Global Information Technology Management Conference, 2002.

[10] ——, *Perspectives of it artifacts: Information systems based on complex mathematical models*, in proceedings of the Business Information Technology Management Conference, 2002.

[11] CHIC-2, *Engineering of optimization projects.* http://www-icparc.doc.ic.ac.uk/chic2/chic2_methodology/, 1999.

[12] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, MIT Press, 2. ed., 2001.

[13] E. DEN BOEF, J. KORST, S. MARTELLO, D. PISINGER, AND D. VIGO, *A note on robot-packable and orthogonal variants of the three-dimensional bin packing problem*, DIKU Technical Report 03/02, (2003).

[14] J. DESROSIERS, Y. DUMAS, M. SOLOMON, AND F. SOUMIS, *Time constrained routing and scheduling*, vol. 8 of Ball et al. [2], 1995, pp. 35–139.

[15] DSDM CONSORTIUM, *Dynamic systems development method.* http://www.dsdm.org, 2003.

[16] M. FOWLER, *Keeping software simple.* Dr. Dobb's TechNetCast, 2000.

[17] M. FOWLER AND K. SCOTT, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, 2. ed., 1999.

[18] P. C. GILMORE AND R. E. GOMORY, *A linear programming approach to the cutting stock problem*, Operations Research, 9 (1961), pp. 849–859.

[19] ——, *A linear programming approach to the cutting stock problem – part ii*, Operations Research, 11 (1963), pp. 863–888.

[20] R. HARDER. OpenTS. In COmputational INfrastructure for Operations Research. http://www-124.ibm.com/developerworks/opensource/coin.

[21] J. HOOKER, *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, Series in Discrete Mathematics and Optimization, Wiley, 2000.

[22] IC-PARC, ECL^{i}PS^e *5.5.* http://www.icparc.ic.ac.uk/eclipse/, 2002.

[23] N. Ivancic, K. Mathur, and B. B. Mohanty, *An integer programming based heuristic approach to the three-dimensional packing problem*, Journal of Manufacturing and Operations Management, 2 (1989), pp. 268–298.

[24] J. A. Joines and S. D. Roberts, *Design of object-oriented simulations in c++*, in Proceedings of the Winter Simulation Conference, 1996, pp. 65–72.

[25] ——, *Fundamentals of object-oriented simulation*, in Proceedings of the Winter Simulation Conference, 1998, pp. 141–149.

[26] K. Marriott and P. J. Stuckey, *Programming with Constraints – An Introduction*, MIT Press, 2. ed., 1998.

[27] S. Martello, D. Pisinger, and D. Vigo, *The three-dimensional bin packing problem*, Operations Research, 48 (2000), pp. 256–267.

[28] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, J. Wiley & Sons, 1988.

[29] S. Ólafsson and J. Kim, *Simulation optimization*, in Proceedings of the Winter Simulation Conference, 2002, pp. 79–84.

[30] M. Pidd, *Computer Simulation in Management Science*, John Wiley & Son, 4. ed., 1998.

[31] M. Pirlot, *General local search heuristics in combinatorial optimization : a tutorial*, Belgian Journal of Operations Research, Statistics and Computer Science, 32 (1992).

[32] T. M. Range and S. Yde, *Storage management at odense steel shipyard. simulation, product placement and control.*, Master's thesis, University of Southern Denmark, 2002. (In Danish).

[33] R. Rodošek, M. G. Wallace, and M. T. Hajian, *A new approach to integrating mixed integer programming and constraint logic programming*, Annals of Operations Research, 86 (1998), pp. 63–87.

[34] F. Vanderbeck and L. A. Wolsey, *An exact algorithm for ip column generation*, Operations Research Letters, 19 (1996), pp. 151–159.

[35] S. Voss and D. Woodruff, eds., *Optimization Software Class Libraries*, Kluwer Academic Publishers, 2002.

Papers

Paper I

Crane scheduling for a Plate Storage in a Shipyard: Modelling

---

Available as IMM Technical Report 2003-4.

Crane scheduling for a Plate Storage in a Shipyard: Modelling

Jesper Hansen[1] **and Torben F. H. Kristensen**[2]

Abstract

This document is the first in a series of three describing an investigation of possible improvements in methods for handling the storage of steel plates at Odense Steel Shipyard (OSS). Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce a ship.
Two gantry cranes move the plates into, around and out of the storage when needed in production. Different principles for organizing the storage and also different approaches for solving the problem are compared. Our results indicate a potential reduction in movements by 67% and reduction in time by 39% compared to current practices. This leads to an estimated cost saving by approx. 1.0 mill. dkr. per year.
This paper describes the modelling aspects of the investigation. Hansen and Kristensen [8] and [9] describe the aspects of solving the model, experiments conducted and the achieved results.

Keywords: Crane scheduling, plate storage

1 Introduction

This document is the first in a series of three describing an investigation of possible improvements in methods for handling the storage of steel plates at Odense Steel Shipyard (OSS). Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce a ship.

Two cranes are used for moving the plates on the storage. Two crane operators control these cranes. A simulator and control systems have been developed to take the decisions normally taken by the crane operators. These control systems are designed such that they can be implemented in the real system and be used as decision support system for the crane operators.

[1]Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark

[2]Department of Production, Aalborg University, 9220 Aalborg Ø, Denmark

The paper is organized as follows. First we describe the problem, including the physical characteristics of the plate storage and the operation of the storage. Section 3 gives an overview of related research. In section 4 we describe our model of the plate storage including the assumptions made for simplifying the model and in section 5 we conclude. The aspects of solving the problem based on the model described here is reported in Hansen and Kristensen [9] and experiments and results are given in Hansen and Kristensen [8].

2 Problem Description

2.1 The purpose of the plate storage

The plate storage contains the plates used for building blocks. The storage is a buffer between the suppliers of the plates and the production at OSS. The flow of plates through the storage can be considered as a dynamic process where new plates are added to the storage and replace plates that are removed from the storage. The plate storage is accumulating plates for a period of time to ensure that the plates needed in the production at a certain time can be delivered. Since steel works from around the world supply the plates, the accumulation of plates for one day's production can run over several weeks. Some plates are delivered a long time before the day and some are delivered close to it. A diagram of the plate storage is shown in figure 1 on the next page. The ellipse in the figure indicates the area which has been zoomed in and shown in figure 2 on page 92.

2.2 Stacks of plates

The plate storage is organized in 32 times 8 stacks of plates. X and Y directions indicate this in figure 1 on the next page. Note that the Y-axis is placed to the left, because of lack of space on the right. Stack $(1,1)$ is indicated in the figure. The size of the storage is 600×35 meters. The storage contains 5.000 steel plates on average – approximately 20 plates are stored in each stack on average. One quarter of the stacks are used for *special purpose plates* and *surplus plates*, for example stacks with identical plates. The rest of the storage is used for plates, which for the main part have different sizes. Each plate is ordered for a specific purpose and the date on which it will be required in production is known. Changes in the production plan, can however influence the due dates of the plates in the storage, which means that the due date of a plate can change several times before it is required in production.

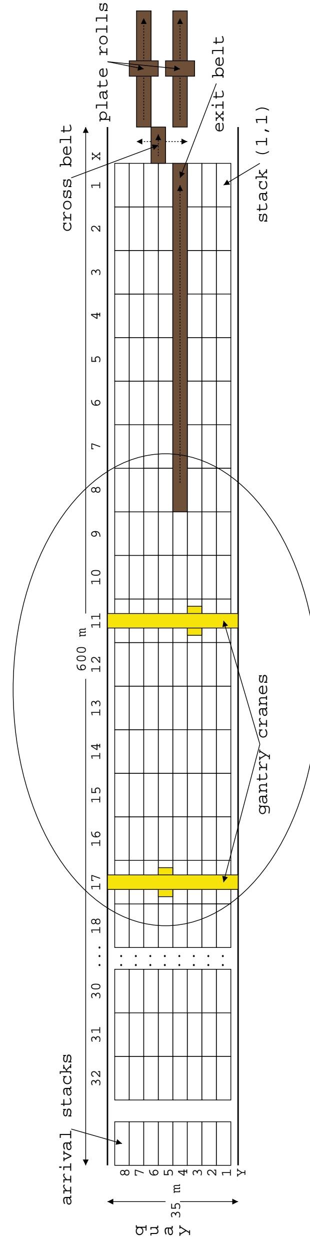

Figure 1: Illustration of the storage layout.

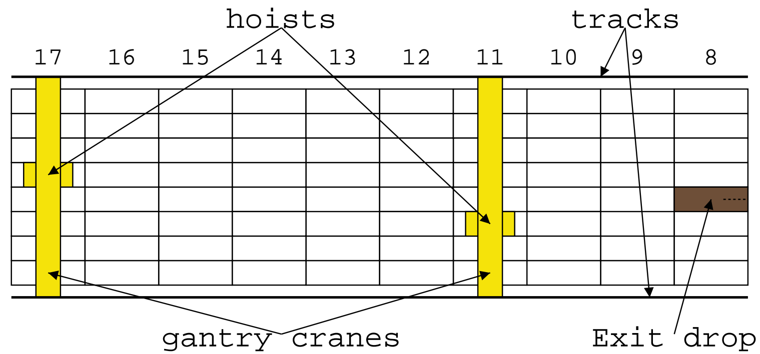

Figure 2: Zoomed illustration of the storage layout.

2.3 Gantry cranes

Two identical gantry cranes carry out the movements of plates. They can move with a speed of 3.3 m/s in both the X and Y directions. I.e. if t_1 is the movement time of a movement in one direction and t_2 in the other direction, the resulting movement time is $\max\{t_1, t_2\}$. A crane operator operates each of the cranes. The cranes share tracks and hence cannot pass each other. For each of the 32 rows the hoist of the cranes can reach each of the 8 stacks. The cranes use electro magnetics to lift the plates. Because of the use of magnetism, it sometimes occur that two plates are lifted at the same time because the magnetism "runs through" the upper plate. When this happens some seconds are lost while the crane operator gets rid of the plate that was not meant to be moved. Since the cranes are gantry cranes there is a limit on the height of the stacks.

2.4 Arrival of new plates

Most steel plates are delivered by sea from the steel works. Trucks also deliver a small number of plates. The new plates are moved from the ships into arrival stacks in the quay area by a tower crane. The identities of the plates are not known before they are manually measured and an identification code is painted on them. When a plate is identified and registered it is moved to an appropriate stack in the storage by one of the gantry cranes. The company policy concerning arrival of new plates to the storage says that plates should be delivered 18 working days before they are requested in production. In reality they are not for various reasons:

- Delays occur because of production and transportation problems.
- Big bulks of plates arrive early because of cheap deals are possible when buying large quantities of plates.
- Special types of plates are only delivered once a year.

2.5 Exit-belt

Each day a set of plates must leave the storage to be processed. The size of the daily set varies from 70 to 150 plates, but to smooth out the production about 100-120 plates are to leave the storage every day. The production planner chooses the set of plates on the basis of needs of plates in the production and how the plates are placed in the stacks. The plates are put on a conveyer belt called the exit-belt, which has a capacity of 8 plates. The plates can be dropped at any vacant slot on the exit-belt from (1,1) to (8,1). The exit-belt is the start of a production line. On the production line several processes are performed. Plates are rolled, sandblasted, painted and finally bar-coded. The exit-belt works as a queue and plates are drawn from the end of the queue. A plate can be drawn from the queue when one of the plate-rollers following the exit-belt is idle. Constraints on the rollers and following processes determine the time interval between plates being drawn. The constraints refer to the dimensions of the plates and required time to adjust the machines according to the processed plates. The time interval between plates being drawn is in other words depending on the order and dimensions of the plates on the belt.

2.6 Operation of the Plate Storage

As mentioned earlier ships are constructed by cutting up plates and afterwards welding them together to produce blocks. Finally the blocks are welded together in the dock to build the ship. Currently the stacks in the plate storage are assigned to different blocks, and plates are placed in stacks according to the blocks for which they will be requested. As indicated in figure 3 on page 95 the two stacks (27,7) and (27,8) are used for block 705. A block is produced over several weeks, which means that the cranes often have to get relatively few plates from particular block stacks. Often the requested plate is not on top of the stack and a lot of unproductive movements are required to find the plate. The plates requested for the production of the next day are put in the sort stacks near the exit-belt. The next day plates due for production can be put on the exit-belt with little delay. Plates arriving a long time before their

due date are put in stacks in the waiting area. Finally standard and surplus stacks are used for identical special purpose plates and plates with no assigned purpose. We call this layout the *Block-storage*. The common opinion is that the storage layout delivers a poor performance. Therefore we introduce two alternative layouts, which have been tested and compared to the performance of the Block-storage described in Hansen and Kristensen [9]. The principles of the two alternative layouts are briefly described in the following.

The *Due-date storage* organizes the storage in three zones as shown in figure 4 on page 96. All stacks have been assigned a specific due date interval given by a start and end date. A Plate is now supposed to be located in stacks where the due date of the plate is in the interval of the stacks. The intervals can be set by the user, but are usually increasing from zone 1 to 3. Zone 1 is closest to the exit-belt and contains plates that have a due date close to the current day, which means that plates due today can be put on the exit-belt with little delay. In zone 1 each stack has an interval of 1 day. Zone 2 consists of plates with a due date from a few days up to some weeks into the future. In zone 2 each stack has an interval around 2-5 days. Stacks in Zone 3 have the longest interval and most of the new plates arriving at the quay are moved into this zone. In figure 4 on page 96 an example of a due date layout is given. The numbers in the fields indicate the due date from today. The 8 stacks (9,8) to (16,8) have the plates due today and stacks (25,1) to (32,1) for example have the plates with a due date in the interval 21 to 24. We will later discuss how these due dates are set up and updated during simulation.

The *Self-regulating storage* uses no stacks which are dedicated to a specific purpose or due date. The storage is organized by use of *objective-functions*. The objective-functions are designed to place plates on stacks where unproductive moves are less likely to be introduced in the future. In other words, the ideal situation is to place plates on stacks where all other plates have a due date later than or equal to the given plate. Off course this is not always possible and may result in long inefficient travelling distances by the cranes. The optimal solution is found by minimizing costs.

3 Related Research

This work is an extension of earlier work of Stidsen et. al. [17] where a simplified problem was solved with only one crane. Hansen and Clausen [7] describe the results of an earlier version of the planning procedure also part of Hansen and Kristensen [9].

Much work has been done on similar or related problems within scheduling

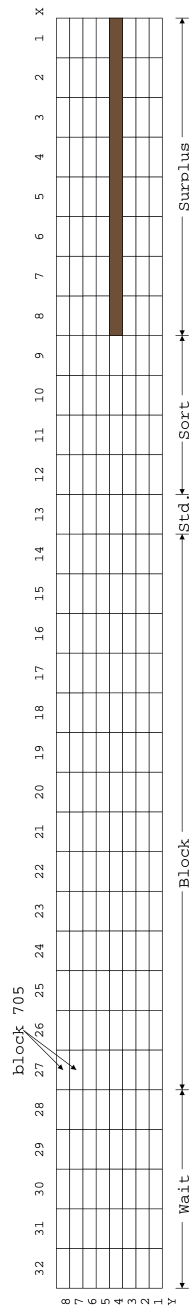

Figure 3: Illustration of the block storage layout.

X	8	7	6	5	4	3	2	1	Zone
9	8	7	6	5	4	3	2	1	Zone 1
10									
11									
12									
13									
14									
15									
16	8	7	6	5	4	3	2	1	
17	22-23	20-21	18-19	16-17	14-15	12-13	10-11	8-9	Zone 2
18									
19									
20									
21									
22									
23									
24	22-23	20-21	18-19	16-17	14-15	12-13	10-11	8-9	
25	49-53	45-48	41-44	37-40	33-36	29-32	25-28	21-24	Zone 3
26									
27									
28									
29									
30									
31									
32	49-53	45-48	41-44	37-40	33-36	29-32	25-28	21-24	

Y: 8 7 6 5 4 3 2 1

Figure 4: Illustration of the due-date storage layout.

and routing. We will distinguish between off-line problems where all information is available before the planning starts and online where decisions have to be done before all information is available. Available information can be deterministic or stochastic. Real-time online problems are problems where the decisions must be computed within tight time bounds. How tight the time bounds are depend on the specific application. Grötschel et.al. [6] gives a good introduction to the subject specifically on evaluation of online algorithms. Often online algorithms are referred to as dynamic and off-line as static. For some problems the achieved plan are to be executed in a cyclic fashion. Everything must then be deterministic.

A similar problem to the problem we are considering is the well known *travelling salesman problem* (TSP) where a number of customers must be visited exactly once minimizing the travelled distance by the salesman. A variant of this is *the stacker crane problem* first introduced by Fredrickson et. al. [3] where a crane must move given items from their origin to some given destination. With multiple salesmen or cranes we get *the multiple travelling salesman problem* or *the vehicle routing problem* (VRP). In our case we have additional precedence constraints representing the order in which plates can be lifted from the stacks. Much research has been done on these problems and recently more attention has been put on the dynamic versions as in Larsen [11].

Avoiding conflicts between cranes or other machinery is often an issue in production and logistic systems. Lee et. al. [12] gives a state-of-the-art overview on deterministic scheduling including robotic cells, automated guided vehicles (AGVs) and cyclic hoist scheduling. AGVs are computer-controlled vehicles without drivers and can hence be considered as a generalization of the above routing problems where congestion, conflicts and deadlocks must be avoided. Refer to Qiu and Hsu [14] for an in depth survey on AGVs.

In the *hoist scheduling problem* one or several hoists are to perform movements of items with associated time windows and precedence relations. Bloch et.al. [2] gives an overview of work done on this and related problems. Of specific interest to our case are Lamothe et.al. [10] that solves a real time multi-hoist problem.

Some of the earliest work we have found on similar “real life” applications are Fujita et. al. [4]. They are scheduling multiple overhead cranes moving around coils. They have implemented a simulator and a very simple control system. Another interesting area are container terminals where combinations of cranes and AGVs are extensively used. Alicke [1] uses constraint programming for solving the scheduling problem at a container terminal. Steenken

et.al. [16] solves a problem composed of planning stowage of containers on ship and transport of the containers to the quay by AGVs. The quality of the stowage determines how many unproductive container movements will be necessary in the future to unload the ship. Gantry cranes stacking containers at terminals also resembles our planning problem, even though fewer containers are stacked than steel plates.

4 Model

A Java program has been developed to test the control and planning systems in different scenarios. To perform these tests the program includes an object-oriented model of the plant to be used in simulations. The model is parameterized such that different scenarios can be tested. All data in the model are generated by methods based on specifications and on statistical information. Process times are varied by probability distributions. Furthermore the due date of the plates are perturbed by methods based on rules that emulate the phenomena in the storage. These changes are caused by the dynamical adjustment of the production plans. The simulation model is also described in Range and Yde [15] who have participated in the development of the simulator. The simulation model is a discrete event simulation model as described in Pidd [13].

4.1 Facilities

The model consists of the following facilities: Two cranes, 8 stacks for new plates delivered at the quay, exit-belt and 191 stacks.

4.1.1 Stacks of plates

The number of stacks is reduced to 191 (8×24-exitbelt). The stacks excluded are in columns 1-8 containing plates that are not assigned to a specific block; therefore these stacks are not taken into account when simulating. 8 additional stacks are dedicated to contain new plates delivered to the shipyard.

4.1.2 Gantry cranes

The acceleration/deceleration time of the cranes is not taken into account. The lift/drop time is set to 15 seconds but varied by an exponential distribution. This variation emulates the variations that occur when lifting a plate. As described in section 2, it sometimes occur that two plates are lifted at the

same time because the magnetism "runs through" the upper plate. This is one of the reasons for the variation in the lift/drop time.

4.1.3 Exit-belt

The exit-belt is modelled with a maximum capacity of 8 plates. The average time interval between two plates being removed from the queue is set to 192 seconds. This was found from statistical data. The interval time is varied by an exponential distribution. This variation is emulates the variations that occur on the production line.

4.2 Plate generation

The plates in the model are generated by the program instead of retrieving real data from the shipyards Production Management System (PMS). The method is a simplified model of the flow of plates through the plate storage, but the characteristics of the real plate storage is retained. The generation method is used for two purposes: When the simulation is started it is used to generate plates initially in the storage, and when the simulation is running it is used for generating delivered plates. In the model the plates only have three attributes, due date, block and ID. The plate generator method uses 4 parameters, which are set by the user of the system. We will first describe the generation of delivered plates and afterwards generation of the plates initially on the storage.

4.2.1 Generation of new plates

The new plates are generated before the start of a new day of the simulation. The plates are placed in the arrival stacks. This is partly a simplification of the real life situation described in section 2.4. When the plates are identified and marked we assume that they are moved to designated arrival stacks and the decision where to move the plates into the storage is planned for the next day and not done online the same day.

We have chosen what could be called a *triangular distribution* of arriving plates. In figure 5 on the next page is shown an example for the parameter value $SRP = 3$. SRP are the number of new plates with the median due date of the arriving plates. By construction the number of arriving plates is exactly SRP^2. This explains the acronym SRP as **S**quare**R**ootof**P**lates.

The number of days between two plate deliveries is determined by the parameter $DBSA$ (**D**ays**B**etween**S**hip**A**rrival). In case the number of days

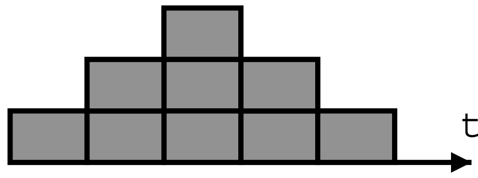

Figure 5: Distribution of plate due dates for $SRP = 3$

between two deliveries of new plates is set to $DBSA$ days, then the number of new plates is $SRP^2 \times DBSA$, but only every $DBSA$th day. The white rectangles in figure 6 represent the plates already on the storage. The parameters BD and ACC are explained in section 4.2.2. The leftmost column are the plates due today and for each column going right the due date is one day into the future. The grey rectangles represent arriving plates with $DBSA = 1$. In figure 7 on the facing page the change in generation of plates are shown when $DBSA = 3$. New due dates are introduced on the storage every time new plates are generated. The number of new due dates is equal to the value of $DBSA$.

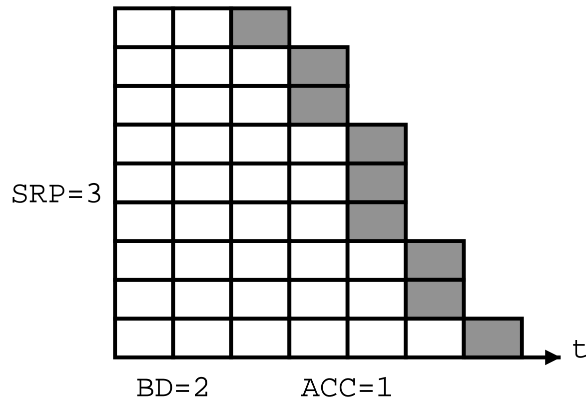

Figure 6: Example of generation of plates with $DBSA = 1$.

4.2.2 Generation of initial plates

The user of the system can control the size of the problem to be solved, using the parameters mentioned above and in the following. The user can hence

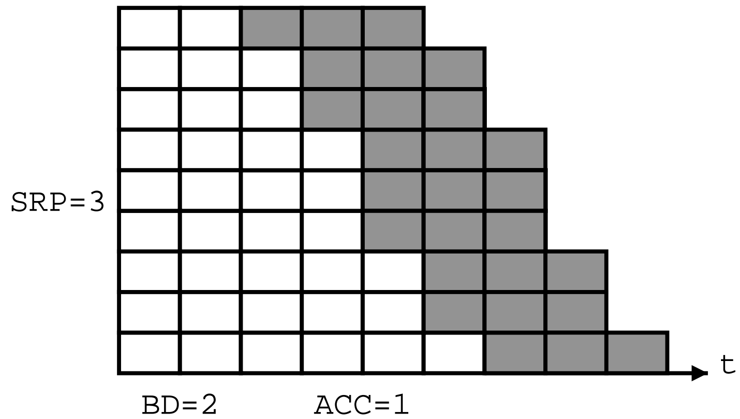

Figure 7: Example of generation of plates with $DBSA = 3$.

test different scenarios by creating different due date profiles of the plates.

In our model, the number of plates removed from the storage each day is constant. This is justified by the fact, that the production planner selects a more or less constant number each day in order to avoid large fluctuations in the production level. The number of plates is specified by the variable SRP. The square of this parameter determines the number of plates removed from the storage. For instance if the value of the parameter is set to 3 then the number of plates removed will be 9 as in figure 6. In this way the number of outgoing plates will match the number of incoming plates. The number of plates removed can then assume the following values: 1, 2, 4, 9, 16 etc.

As described in section 2.4 the company policy regarding the storage says that plates should be delivered 18 working days before they are required in production. The length of this period can be controlled by the parameter BD (**B**uffer**D**ays). In figure 7 for instance all plates have arrived for the next two days, $BD = 2$.

The time beyond the BD days is the period where plates are arriving to the storage – or what we have called *accumulating*. This period can be adjusted by another parameter, ACC (**ACC**umulating). Increasing the parameter increases the period of days where plates arrive to the storage. Two examples are shown in figure 6 with $ACC = 1$ and figure 8 on the next page with $ACC = 3$. Again the grey boxes are arriving plates. With $ACC = 3$ the arrival of plates are over a longer period of days resulting in more plates on the storage. In figure 8 we see that the due dates of the arrival plates are

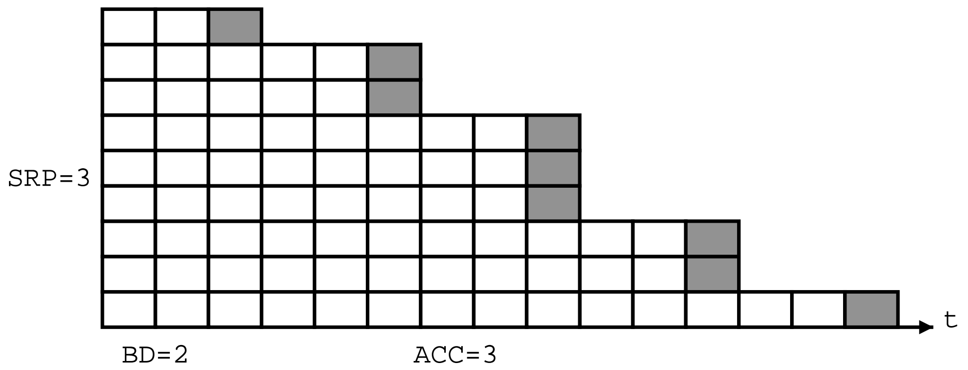

Figure 8: Example of generation of plates with $ACC = 3$.

exactly 3 days apart.

The number of plates in the initial storage is $((SRP-1)ACC+BD)SRP^2$.

4.3 Storage Types

As described in section 2.6 three different storage-types are considered: Self-regulating, block and due- date. The selected storage type can be set by a parameter.

4.3.1 Initialization of the storage

An available parameter defines how the plates are placed initially on the storage. Selecting Block or due-date storage places the plates as expected in the stacks corresponding to the characteristics of the plates. If more than one stack corresponds to the characteristics of the plate, the particular stack is chosen randomly. For the block storage, a plate fitting the sort stack due-date is placed on such a stack. Otherwise it is placed in a random stack according to the block number.

When choosing the self-regulating principle the plates are placed randomly on the stacks. The stack- height is checked as well as any upper limit on the number of plates in a stack.

4.3.2 Self-regulating storage

Defining the storage layout or organization of the self-regulating principle is straightforward: All stacks are identical and not dedicated to specific pur-

poses or dates. The user can only adjust the maximum stack height and the maximum number of plates in the stacks. The storage is organized by use of objective functions that express how well the storage is organized. These functions are discussed in detail in Hansen and Kristensen [9].

4.3.3 Due-date storage

The due-date storage is divided into three zones as seen in figure 9. Each stack, s, has a dedicated due-date interval, $I_s = [b_s, e_s]$, where $b_s \leq e_s$. When only one value is depicted in the figure, $b_s = e_s$. In the ideal situation the stack only contains plates with a due-date $d_p \in I_s$. Generally the size of the interval varies from one and up to many days depending on the zone.

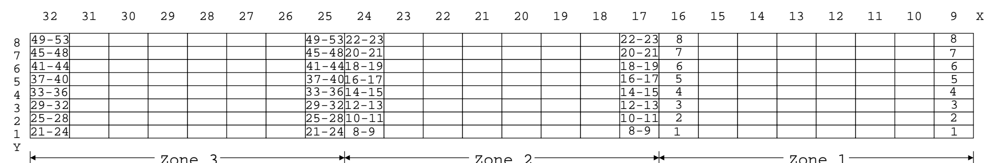

Figure 9: Setting the due-date storage layout.

The due-date intervals for the stacks in the storage can be controlled by 3 parameters for each zone, $z \in Zones$:

Number of days in each interval defines how many due dates are included in each stack. The interval for the stacks in zone 1 is typically 1. The interval for zone 2 is chosen to be between 2 - 5 days and for zone 3 the interval is typically 4 - 20 days. In figure 9 the chosen values are 1, 2 and 4 for zones 1, 2 and 3. Let $Days_z$ be the associated parameter.

Number of stacks per interval specifies the number of stacks included in each interval in the zone. For example if the number of plates requested in production each day is 100 and the number of stacks per interval in zone 1 is equal to 5 and the number of days in each interval is 1, then the number of plates in each of the 5 stacks will be 20 on average. In figure 9 the chosen values are 8 stacks per interval for all 3 zones. Let $Stacks_z$ be the mentioned parameter.

Number of due-date intervals specifies the number of different due-date intervals in each zone. Consider the case of zone 2 where the number of

days per interval could be set to 2 days and the number of intervals is set to 5, then 10 due dates are included in zone 2. Typical values are 8 since the storage has 8 rows, which is also the choice in figure 9. Let $Intervals_z$ be the mentioned parameter.

Overlap between zones are introduced in order to avoid large fluctuations in plate movements from day to day when plates are moved between zones. Overlap between zone 1 and 2 defines how many due-dates are included in both zone 1 and zone 2 and similarly for zone 2 and 3. In figure 9 we have intervals corresponding to days 1 up to 8 in zone 1 and an interval in zone 2 with the days 8 and 9. Plates with due date 8 can hence be moved from zone 2 to 1, but plates with due date 9 cannot. Let $Overlap_z$ be the overlap between zones z and $z + 1$.

The optimal setting of the parameters is depending on the number of plates on the storage and the distribution of due-dates, since generally the state of the storage is better if the plates are distributed as evenly as possible on the stacks.

When assigning due-date intervals to the stacks this is done with the procedure shown in algorithm 4 on the next page.

Algorithm 4: Assignment of due dates

```
b = today, e = today
Put stacks in a list, L
Sort in increasing distance from the exit-belt in the Y direction
Sort in increasing distance from the exit-belt in the X direction
for all z ∈ Zones do
    for i = 1 to Intervals_z do
        Remove first stack s from L
        e = e + Days_z − 1
        b_s = b, e_s = e
        x = x_s, y = y_s
        for j = 1 to Stacks_z − 1 do
            x = x + 1
            if ∃s' ∈ L where x_s' = x and y_s' = y then
                Remove s' from L
            Remove first stack s' from L
            x = x_s', y = y_s'
            b_s = b, e_s = e
        b = e + 1, e = b
    b = b − Overlap_z, e = b
```

First the stacks are placed in a list and sorted primarily in the horizontal direction (X) and secondarily in the vertical direction (Y). Next, in a nested loop is iterated over zones, number of intervals per zone and number of stacks per interval. Let us consider zone 1. When assigning due dates to stacks for an interval we pick the first stack in the list starting with stack $(1,1)$ and assign due dates to stacks according to the number of stacks in the interval. In figure 9 8 stacks are assigned to each interval, which results in the stack $(1,1),(2,1),...,(8,1)$ being assigned the due date 1. A stack is removed from the list when assigned a due date. For the next interval we again pick the first stack in the list, which now is $(1,2)$, etc. until the last interval in zone 1 starting at $(1,8)$. After zone follows the same procedure for zone 2 and 3.

If the assignment of stacks were performed in the opposite direction from $(1,1)$, to $(1,2)$ up to $(1,8)$ instead, it would be difficult for the cranes to work together to empty and fill up stacks in the same interval. The distance to the exit-belt from stacks in zone 1, between zones and from the quay and to zone 3 will vary more with this layout as well.

4.3.4 Updating the due-date storage

Since the due date of stacks reflect time, due dates for stacks change over time. Each day we need to update the due dates of several stacks. Plates due today are being removed from their stacks in zone 1. These stacks will then get a new due-date interval equal to the stacks with the latest due-date in zone 1 plus the number of days per interval in zone 1.

Plates in zone 2 with a due date in that particular interval can now be moved from zone 2 to 1. The same holds for a given interval in zone 2. If all due dates in an interval in zone 2 are present in zone 1 then the due date interval is updated as well. Plates can now be moved from zone 3 in the same way.

For example, in figure 9 plates with due date 1 will be removed from the storage and since the latest due date in zone 1 is 8, these stacks will get the new due date 9. It is now possible to move plates with due-dates 8 and 9 from zone 2 to 1. Stacks in zone 2 with due dates 8 to 9 will hence get the new interval 24-25. In zone 3 stacks with interval 21-24 can now get the new interval 54-58.

The update of due dates is done automatically. The user only specifies the 3 parameters for each zone and the two overlap parameters discussed earlier.

4.3.5 Block storage

In the model of the block storage we ignore waiting stacks and only have block stacks and sort stacks. The number of sort stacks is defined in the same way as zone 1 for the due-date storage. The rest of the stacks are used for block stacks. It is assumed that each block is stored in exactly two stacks, but this could easily be generalized. When generating plates a random block is assigned to each plate. Assigning blocks to stacks are done first in the Y-direction and secondly in the X-direction for example like the following (ignoring sort stacks): Block 1 is $(1,1)$ and $(1,2)$, block 2 is $(1,3)$ and $(1,5)$ since $(1,4)$ is the exit-belt. Block 3 is $(1,6)$ and $(1,7)$, block 4 is $(1,8)$ and $(2,8)$, block 5 is $(2,7)$ and $(2,6)$, etc. Note that this is a simplification of the current operation where more than two stacks can be assigned to blocks with many plates. In other cases blocks with few plates are pooled together sharing stacks. We have instead chosen to distribute the necessary number of plates and hence stacks evenly between block.

4.4 Generating Movements

All movements of plates are generated before the simulation of the current day is started. Some movements are generated to deliver the plates ordered by the production for the current day while others are generated to improve the organization of the storage. A strong characteristic of the system is that all plates to move can be identified before the execution starts. Note however that a plate may be moved more than once if it is put on a stack from which plates are going to be moved later on the same day. This will dynamically introduce extra movements to schedule. A movement, m, is defined by the plate to be moved, p_m, the source stack, s_m and the destination stack, d_m. The movement types are defined as follows.

Exit and Dig-up movements

An exit movement is a movement of a plate, which has a due-date that matches the current day. In real life the user of the system selects plates to be removed from the storage. For example in figure 10, today is 1 November 01 and exit-movements of the plates p_2 and p_7 are therefore generated.

A dig-up movement is a movement of a plate, that has to be moved because an exit movement is generated for a plate below it in the stack, or that has to be emptied the current day because the due date interval of the stack is changed. Again in figure 10, a dig-up movement of plate p_1 is necessary since p_2 is going to be moved.

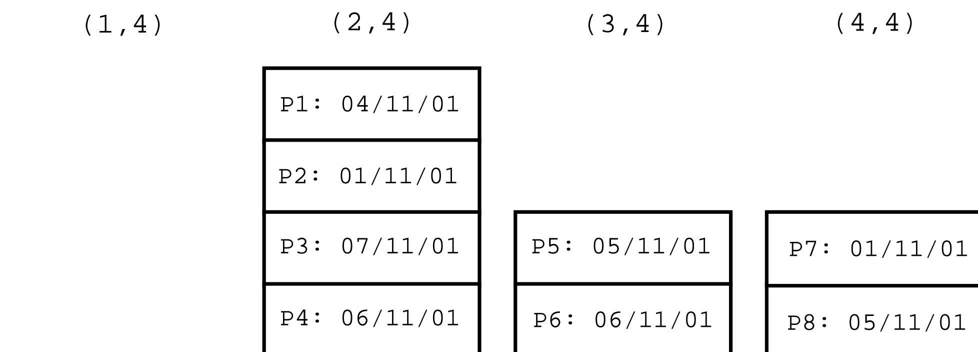

Figure 10: Example of generating movements

Sort movement

In order to improve the state of the storage some sorting of the stacks are performed. The reason for doing sorting is that in the future less dig-ups will be necessary, and less time will be spent removing plates from the storage. However if the due-date of the plates in the storage change frequently there is a risk of doing a lot of sorting that later turn out to be wasted.

The chosen sorting strategy is depending on the storage principle. For the block-principle we identify plates that are due the next day. These plates are moved to special due-date stacks near the exit-belt.

For the due-date principle sort movements can be split into three main categories:

1. A movement of a plate in one of the stacks that contain new plates delivered to the quay area.

2. Movements of plates that are immediately accessible and are located in a stack where the due date of the stack does not include the due date of the plate.

3. if a given percentage, e.g. 75%, of the plates in a stack do not have a due date that match the due date of the stack, then a sort movement for each of the plates in the stack is generated. To avoid generating too many movements the fraction should be set close or equal to 100%.

Optionally the user can choose to extend the sorting in zone 2 and 3: Plates that are to leave the storage within b days from today are to be moved. b is usually set to 1. The above strategy seems a bit complicated, but basically makes sure that plates are moved from zone 3 to 2 to 1 when the due date of the stacks changes. The rest is just trying to move plates as soon as possible before the due date changes if it does not create much extra work.

For the self-regulating principle experiments show that sorting in the following way is appropriate: Move plates away from stacks such that plates that are due tomorrow are on top of the stacks. In that way no dig-ups will be necessary the day after, if the due dates have not changed. Note that the plates that are due tomorrow are not collected in stacks near the exit-belt as for the block storage. This should in principle save one movement per plate.

4.5 Disturbances

A lot of decisions are made on the basis of expectation, but often these expectations are not fulfilled and may lead to poor or even infeasible decisions. This

is also the case at the plate storage. The processes performed at the plant are somewhat stochastic and as mentioned above these stochastic disturbances are included in the model. We have earlier discussed disturbances concerning the time of the lift/drop process of the cranes and the interval between two plates being removed from the exit-belt. Other stochastic phenomena concerning the due dates of the plates are included in the model as well. As mentioned in section 2.2 the due date on the plates are changed during their stay at the storage. The due dates are changed because of adjustments to the production plan. The adjustments are caused by different phenomena. In the simulation model three of these phenomena are emulated. In the following the disturbances on the facilities and due dates are described.

4.5.1 Disturbances on facilities

The processes performed by the two cranes and the exit-belt are influenced by stochastics. The process times for both processes are modelled by an exponential function. The process times can never drop below a certain level, i.e. they have a minimum process time. Theoretically the process times have no maximum.

Cranes

As described earlier, lift and drop times vary because of difficulties in lifting the plates. The process time for a lift/drop has a lower limit. From statistical data a mean value of the time has been established. The time used for the simulations is 15 seconds. The process time has a lower limit of 5 seconds.

Exit-belt

The bottleneck on the production line sets the speed by which plates can be removed from the queue at the exit-belt. The bottleneck on the line is the painting station. From statistical data a mean value of the time has been established. The time used for the simulations is 192 seconds. The process time has a lower limit, which is found to be 64 seconds.

4.5.2 Disturbances on due dates

The ideal situation would be a situation where the plates are given only one due-date through their "lifetime". But as explained earlier this is not the case at the plate storage. Plates can have several due-dates during their stay

at the storage. Sometimes the change is only a single day, but can also be several days or weeks. Some of the changes are performed to smoothen out production workload. Other changes are performed because plates have been used for other purposes than planned, i.e. they have been used for substitution of other plates. In the simulation model three methods are implemented to change the due date of the plates, they are described in the following. The user of the system can control the methods by parameters.

Rush jobs

In the model plates that have a due date that matches the current day, are supposed to be delivered at the exit-belt the current day. In the real system an employee prepares the movement-list for each day. Not only does he select the plates with a due date matching the current day, furthermore he selects plates, which are supposed to be delivered on other days. He determines the need for plates at the machines where the first operation will be performed on the plates. This behaviour should actually result in a change of the due date on the plates. This change in due dates is not done because it does not affect the real plant, but to emulate this behaviour in the model a random number of plates swap due dates. The swaps are done before the workday starts.

The method uses two sets of plates. The first set contains plates with due date matching the current day and the second set contains plates with a due date later than the current day, but not later than a date specified by the user. In the method the changing is done by swapping the due date of a plate from the first set with a plate from the second set. Both plates are randomly chosen from both sets. Stacks that before the swap were well sorted can afterwards incur additional dig-up or sort movements. An example of the swap procedure can be seen in figure 11 on the next page where the grey boxes indicate plates with swapped due-dates.

In the example the current date is 1/3. Plate $P01$ is swapped with plate $P09$. Before the swap 4 dig-ups were going to be performed to deliver the 4 plates with due-date 1/3. After the swap this number is increased to 9 moves. As can be seen from this example the swapping of plates is a very "effective" way of increasing the complexity of the problem to solve.

An extension of the above rush jobs has been included in the system, which also allows plates due the day after to be swapped.

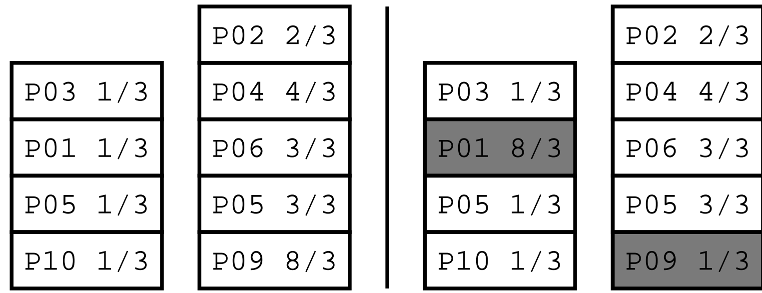

Figure 11: Two stacks of plates before and after swapping due-dates.

Weekly change of due-dates

Each week a re-planning is run on the company's production management system, resulting in major changes in the due dates of the plates. Blocks being hurried or postponed cause the changes. When blocks are postponed other blocks are hurried and the other way around because of the limited capacity in the production. The due-date is changed on approximately 500 plates. This means that the sorting of the plates in the stacks will be spoiled and on the following days a lot of dig-up movements will be necessary. This phenomenon is also emulated in the model.

The number of days between the rescheduling can be set by the user. Usually the parameter is set to 5, which is equivalent to once a week.

A random number of plates between a lower and upper bound change due-date by the following procedure, which is essentially a cyclic shift (cf. figure 12 on the following page):

- A randomly chosen plate, p_1, is chosen from plates with a due-date in the interval from some minimum change horizon from the current day to some maximum change horizon from the current day. From statistical data these values have been set to 10 and 20 days from the current day.

- The value corresponding to the change in due-date is sampled from a normal distribution with mean and variance specified by the user. After the change the due-date, d^{new}, must still be in the above due-date interval. The spread was found to be 6.4 days from statistical data.

- A randomly chosen plate p_2 with due-date d^{new} is selected. p_2 now gets a due-date one day closer the original due-date of plate p_1. A new plate

is randomly found with that due-date, etc. This procedure is continued until a plate is assigned the original due-date of plate p_1.

- The above procedure is repeated until a random number of changed due-dates has been reached.

In figure 12 is shown an example of the above procedure. Plate p_1 changes due-date from 1/3 to 4/3. A plate p_2 is chosen with due-date 4/3 which is then changed to 3/3, a plate p_3 is chosen with due-date 3/3, which is changed to 2/3 and finally the due-date of p_4 is changed to the original due-date of p_1.

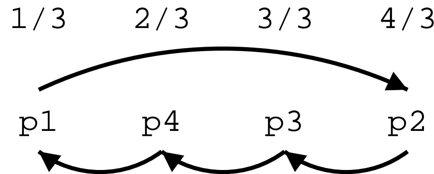

Figure 12: Example of weekly change in due-dates.

The changes of the due dates described above are done before the start of a new day in the simulation. Therefore these changes can be included in the optimized schedule. The disturbances on the facilities are obviously generated during execution and can therefore not be included in the optimization of the schedule.

5 Conclusion

In this paper we have described the physical environment on the plate storage and how it fits into the overall process of building steel plate ships. Based on the problem description a model has been developed including both the physical entities of the system and the planning and processing aspects. Specially different storage principles have been introduced as well as different ways of designing the layout of the storage.

Methods for solving the problem are described in Hansen and Kristensen [9] and the validity of the model is shown by simulation experiments reported in both Hansen and Clausen [7] and Hansen and Kristensen [8].

References

[1] K. Alicke, *Modeling and optimization of the intermodal terminal mega hub*, OR Spectrum, 24 (2002), pp. 1–17.

[2] C. Bloch, A. Bachelu, C. Varnier, and P. Baptiste, *Hoist scheduling problem: State-of-the-art*, in IFAC Intelligent Manufacturing Systems, 1997, pp. 127–133.

[3] G. N. Fredrickson, M. S. Hecht, and C. E. Kim, *Approximation algorithms for some routing problems*, SIAMM Journal of Computing, 7 (1978), pp. 178–193.

[4] F. Fujita, M. Okada, and T. Kaita, *Automatic scheduling system for an overhead multi-crane yard*, in IFAC Control Science and Technology 8th Triennial World Congress, 1981.

[5] M. Grötschel, S. O. Krumke, and J. Rambau, eds., *Online Optimization of Large Scale Systems*, Springer Verlag, 2001.

[6] M. Grötschel, S. O. Krumke, J. Rambau, T. Winter, and U. Zimmermann, *Combinatorial Online Optimization in Real Time*, in Grötschel et al. [5], 2001, pp. 679–704.

[7] J. Hansen and J. Clausen, *Crane scheduling for a plate storage*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-1 (2002).

[8] J. Hansen and T. F. H. Kristensen, *Crane scheduling for a plate storage in a shipyard: Experiments and results*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-12 (2003).

[9] ——, *Crane scheduling for a plate storage in a shipyard: Solving the problem*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-11 (2003).

[10] J. Lamothe, C. Thierry, and J. Delmas, *A multihoist model for the real time hoist scheduling problem*, in Symposium on Discrete Events and Manufacturing Systems. CESA'96 IMACS Multiconference. Computational Engineering in Systems Applications, 1996.

[11] A. Larsen, *The Dynamic Vehicle Routing Problem*, PhD thesis, Technical University of Denmark, 2000.

[12] C.-Y. Lee, L. Lei, and M. Pinedo, *Current trends in deterministic scheduling*, Annals of Operations Research, 70 (1997), pp. 1–41.

[13] M. Pidd, *Computer Simulation in Management Science*, John Wiley & Son, 4. ed., 1998.

[14] L. Qiu and W. Hsu, *Scheduling and routing algorithms for AGVs: a survey*, Tech. Rep. CAIS-TR-99-26, "Centre for Advanced Information Systems, School of Applied Science, Nanyang Technological University, Singapore", 1999.

[15] T. M. Range and S. Yde, *Storage management at odense steel shipyard. simulation, product placement and control.*, Master's thesis, University of Southern Denmark, 2002. (In Danish).

[16] D. Steenken, T. Winter, and U. Zimmermann, *Stowage and transport optimization in ship planning*, in Grötschel et al. [5], 2001, pp. 731–745.

[17] T. K. Stidsen, J. Hansen, and J. Clausen, *Large steel plate storage optimization*, in The 1st International EuroConference on Computer Application and Information Technology in the Marine Industries 2000, COMPIT'2000, 2000, pp. 449–462.

Paper II

Crane scheduling for a Plate Storage in a Shipyard: Solving the Problem

Available as IMM Technical Report 2003-11.

Crane scheduling for a Plate Storage in a Shipyard: Solving the Problem

Jesper Hansen[1] **and Torben F. H. Kristensen**[2]

Abstract

This document is the second in a series of three describing an investigation of possible improvements in methods for handling the storage of steel plates at Odense Steel Shipyard (OSS). Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce a ship.
Two gantry cranes move the plates into, around and out of the storage when needed in production. Different principles for organizing the storage and also different approaches for solving the problem are compared. Our results indicate a potential reduction in movements by 67% and reduction in time by 39% compared to current practices. This leads to an estimated cost saving by approx. 1.0 mill. dkr. per year.
This paper describes aspects of solving the model developed and described in Hansen and Kristensen [8]. Conducted experiments and achieved results are reported in Hansen and Kristensen [7].

Keywords: Crane scheduling, plate storage

1 Introduction

This document describes an investigation of possible improvements in methods for handling the storage of steel plates at Odense Steel Shipyard (OSS). Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce a ship.

In Hansen and Kristensen [8] we described the physical environment on the plate storage and how it fits into the overall process of building steel plate ships. Based on the problem description a model was developed including both

[1]Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark

[2]Department of Production, Aalborg University, 9220 Aalborg Ø, Denmark

the physical entities of the system and the planning and processing aspects. In this paper we will discuss the aspects of solving the problem, but we will first briefly repeat the main modelling concepts from [8].

Our model of the storage consists of 8×24 plate stacks, and two gantry cranes can move plates between the stacks. 8 additional stacks are used for arriving plates. Each plate has a due date specifying at which date it must leave the storage and enter the production line. When a plate leaves the storage, it is placed on a conveyer belt referred to as the exit-belt. A maximum of 8 plates can be at the belt at the same time and a plate is drawn from the exit-belt at certain intervals. The two gantry cranes share tracks and collision conflicts must hence be avoided.

The problem adressed is to develop approaches for scheduling the crane operations better than current practices. Three different storage principles are considered:

The block storage is the current storage principle. Two stacks are associated to a specific block of the ship and all plates to be used to produce that block are placed in these designated stacks.

The due date storage was suggested by the shipyard management as an alternative to the block storage. Here each stack is assigned a due date interval and a plate with a due date in that interval can be placed on that stack. The storage is divided into zones and each zone is divided into a number of due date intervals. Several stacks are assigned each due date interval. The user can determine the layout of the storage by specifying the different parameters regarding size of the intervals, stacks per interval, number of intervals and overlap in due dates between zones. Plates are initially placed in zone 3 and are afterwards moved through the zones closer to the exit-belt. Refer to [8] for further details.

The self-regulating storage principle is the last alternative. No specific purpose or due date is assigned to the stacks. The organization of the storage is determined by the planning procedure.

Two approaches have been investigated to solve the scheduling problem. The two approaches are based on different principles. The first approach is an *on-line algorithm* or more precisely a *heuristic discrete event feedback control system* where a movement is chosen in real-time. The other approach uses the on-line algorithm off-line as a greedy construction heuristic to get a good initial solution, which is then improved by local search heuristics. The off-line algorithm makes a schedule for the controller to follow. The schedule

specifies the order in which the movements are to be executed. A control module dispatches the movements in the sequences and adjusts the schedule if the initial schedule becomes infeasible because of the underlying stochastic nature of the system.

In section 2 we describe the different local search heuristics implemented, and in section 3 the control system is described. Experiments and results are reported in Hansen and Kristensen [7].

2 Planning Procedure

Given the model of the plate storage, the planning task is now to create a schedule of movements for the 2 cranes without collision, that delivers the plates needed for the given day and minimizes costs. In order to avoid collisions of the two cranes extra positioning and wait movements may have to be inserted in the sequence of movements.

To give a flavour of the size of the problem; consider the scheduling of N movements for 1 crane and M possible stacks. All permutations of N movements are potential, but not necessarily feasible sequences, resulting in $N!$. For each non-exit-movement we have the choice of M possible destination stacks. In total $M \Delta N!$ potential solutions. For M and N only 100 we get a 157-digit number! If we consider the case where we only get plates from 10 stacks of 10 plates each, we get at least 10^{10} (10 billion) number of solutions.

We have implemented a construction heuristic (or control method) described in section 3 and local search heuristics for improving the initial plan. After improving the plan, it is returned to the control module to be executed in the simulator with disturbances also described in 3.

A local search heuristic tries to improve the solution by making local or small changes to the plan. A solution achieved by such a local change is called a *neighbour solution*. How we define the changes on a plan is called the *neighbourhood structure*. For a given solution S the set of neighbour solutions reachable with the given neighbourhood structure is simply called the *neighbourhood of* S, $N(S)$. Refer also to Pirlot [12] for an introduction to local search heuristics.

The components in a local search procedure are then the following:

- The *representation* of a *solution*. In our case the initial storage and the sequence movements with movement times for each crane exactly determine the solution.

- *The objective function* is described in section 2.1. The evaluation of the function is in our case done by simulation. This is described in section 2.2. One of the most important points in a successful local search approach is the fast evaluation of the cost function. As we see later this is not trivial in our case. An alternative is trying to estimate the costs, which is described in section 2.6.
- *The neighbourhood structure* is described in section 2.4.
- *Search procedures* are described in section 2.5.

Some of the components can be considered problem *independent* while others are *specific* for the given problem. Generally, the objective function, the representation of a solution and the neighbourhood structure are problem specific.

We have implemented a local search object-oriented framework similar to Andreatta et.al [2] where the problem independent parts of the algorithm is separated from the problem specific. This allows easy reuse of code for other problems. Other local search frameworks and class libraries can be found in Voss and Woodruff [15].

2.1 Objectives

Costs include salary to the crane operators, power for the cranes to move and activation of the magnetic lift as well as maintenance on the cranes. These costs can be mapped to the make-span or finish time of the schedule, the total travelled distance and total number of movements. Minimization of the time when the last plate is put on the exit-belt is also important to be able to start the following processes as early as possible.

One day of production is planned at a time, so in order to minimize the long-term costs, we measure the quality of the state of the storage after each day. The state is evaluated by 3 different criteria explained in the following.

How well are the stacks sorted by plate due date?

More specifically, how many dig-ups have to be done for each stack with the given plate due-dates. For stack (2,4) in figure 1 on the facing page plate p_2 is the first to be removed from the storage on November 1. This requires the dig-up of p_1. The next plate to be removed from the stack is p_4, which causes the dig-up of p_3. The result is 2 dig-ups. We have defined the *cost of a dig-up* to be the sum of the following terms:

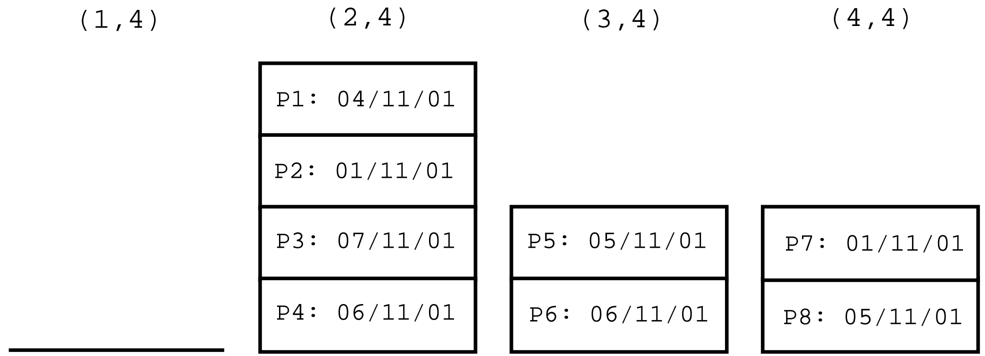

Figure 1: Objective function example.

- Estimated cost of power and maintenance used for lifting and dropping a plate.
- The cost of power for moving the crane to the nearest stack and back.
- The cost in salary for operating the crane in the amount of time according to the dig-up movement.

It is perhaps a bit surprising that according to the management of the yard, the main part of the cost comes from maintenance on the cranes.

How close are the plates to the exit-belt compared to the due date of the plates?

When minimizing travelling distance it is better always to move plates in direction of the exit-belt and when the due-date of a plate is close to the current date, we want it to be located close to the exit-belt. We have chosen to model the cost for each plate p as:

$$\frac{\mathrm{dist}(c_p, c_e)}{\mathrm{diff}(d_p, today) + 1} \tag{1}$$

where $\mathrm{dist}(c_p, c_e)$ returns the distance from the location, c_p, of plate p to the exit-belt, c_e and $\mathrm{diff}(d_p, today)$ returns the difference in days between the due-date d_p of plate p and the current date.

Let us assume that the exit-belt is located at the coordinate (1,4) in figure 1, then stack (2,4) is 1 closer than (3,4). If for instance today is 01/11/2001

then the cost for plate p_1 is 1/4, for p_4: 1/6 and p_6 it is 1/3. The above exit distance can be converted to time and the cost is then achieved by multiplying with the salary of the operators.

How many plates are there in each stack?

We want the plates to be as evenly distributed on the stacks as possible, which will result in less dig-ups when changes in due dates occur. The cost for a stack s is defined to be:

$$\frac{size(s)^2}{maxsize} \tag{2}$$

where $maxsize$ is a user supplied upper limit on the number of plates in the stacks. We observe that the cost per stack is in the interval from 0 to the maximum number of plates in the stack.

If we have an upper limit of 10 plates, then the stacks in figure 1 on the page before will have a cost of $\frac{16+4+4}{10} = 2.4$ while if p_1 was moved to stack (3,4), the cost would be $\frac{9+9+4}{10} = 2.2$, and hence better.

What we are minimizing is in some sense the worst case number of dig-ups if a plate in the bottom of a stack is requested. The cost of a dig-up is therefore used here as well.

Due-date and block principles

For the due date and block principles all criteria are used except for the stack sorting criteria, since that criteria is not relevant for these principles.

2.1.1 Weighting of criteria

We have chosen to convert all the criteria into a common unit, which is then minimized. The user has to supply the operating costs and the wage of the operators. The operating costs of the crane are dividided into costs per lift and per moving second. The majority of the costs are due to maintenance especially on the electromagnetics. The wage of the operators are divided into operators of the cranes and operators of the following production processes and again with the possibility of different wages per shift or for working overtime. The user specifies the length of the three shifts in seconds. The wage is calculated per second, which of course is a simplification of the real world. It is however possible to model a fixed cost of one shift with overtime pay in second shift: Set the wage in the first shift to zero and to the wage plus

overtime in second shift. If it is desired to avoid work for instance in shift h a penalty in form of the power, p, can be set:

$$\max(0, tt - t_h)^p * w_h$$

where tt is the total time, t_{h-1} is the user specified time length of shift $h-1$ and w_h is the wage in shift h. Larger $p > 1$ makes it much more expensive to violate the soft constraint. This flexible setting makes it very easy to model different wage structures. This is similar to Lagrangian Relaxation in Integer Programming (IP) where constraints are relaxed and added to the objective function. In IP we usually only allow linear constraints, but in this setting any non-linear constraint can be modelled.

We will later see that the objectives are used both for evaluating solutions and to guide the search for good solutions. In the next section we describe how to calculate the objective value.

2.2 Simulation

Assume that we have given a sequence of movements for each crane. To completely determine the plan we need to find the start time for each movement, determine destination stacks for dig-up and sort movements and perhaps insert necessary positioning and wait movements in the two sequences. Simulating the crane movements for the given sequences of plate movements does this. Note that we in the system have 2 simulators one for the planning module and one for the control module. The one described here is the planning simulator.

The times for lifting and dropping plates, moving the crane and speed of the exit-belt are not deterministic, but for the purpose of planning they are considered to be. During execution of the plan, changes in times may cause the plan to be infeasible. In that case the plan must be revised. One approach is to adjust the plan according to the changes (control with a plan) and the other approach is to use the construction heuristic as an on-line algorithm (control without a plan). We will consider these issues in detail in section 3.

The complex part of the simulation procedure is handling each type of occuring event:

- Before moving a crane, a potential conflict between the cranes must be identified and taken care of.
- The exit-belt is not ready to receive plates.
- A plate is for some reason not ready to be moved to or from stacks.

In the following we describe in detail how these events are handled in the simulator. Figure 2 on the facing page shows an activity diagram of the process of simulating a movement of a plate by a crane.

First the crane must move to the source stack of the plate. Then the plate is lifted, moved to the destination and dropped. Before actually moving the crane, it must be checked whether a collision conflict between the cranes will occur. We will later return to the handling of conflicts.

The cranes work in parallel. The simulator must therefore keep track of which crane first finishes its current operation and then determine the next operation for that particular crane. When entering the box in figure 2 the next activity for the crane to perform depends on the previous operation done by the crane. If for example the last operation was lift, then the next is to go to the destination of the movement.

Figure 3 on page 126 elaborates the choices made before moving to the source stack. In situations of choice, arrows with no legend cover the situations not described by any other legend. Text in boxes written in italics are elaborated in other figures. There can be several reasons for the requested plates not being ready to be moved. We return to these issues later. If the plate is ready and no crane conflicts will occur the crane is moved to the source stack.

When the crane arrives at the source stack it is again tested if the requested plate is ready to be moved, since the state of the storage can have changed, while the crane was moving to the source stack. This process is shown in figure 4 on page 126.

After lifting the plate the crane must move to the destination stack as shown in figure 5 on page 127. Again potential crane conflicts must be handled properly.

When the crane arrives at the destination stack, the plate can be dropped immediately. However if the destination is the exit-belt a slot must be available. Otherwise the crane waits until a slot becomes available.

After dropping the plate the crane is ready for a new plate movement as shown in figure 7 on page 128. If a crane has no more movements to execute, it is moved to the end of the tracks in either direction. The tracks end outside the area defined by the stacks and conflicts between the cranes are in that way avoided.

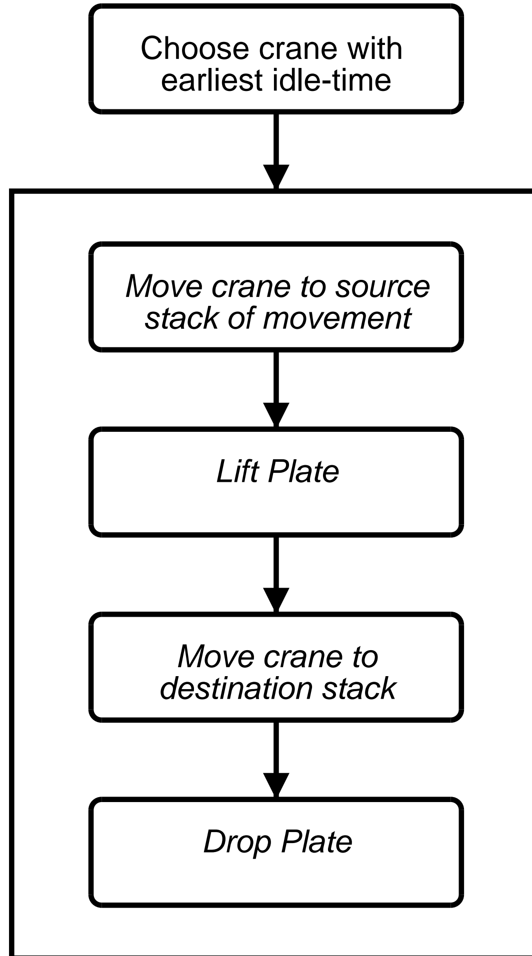

Figure 2: Simulation of a movement.

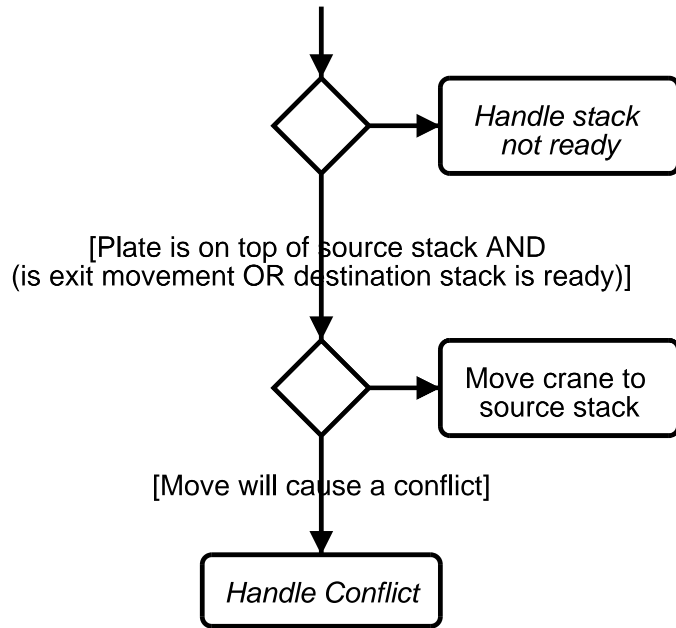

Figure 3: Move crane to source stack of movement.

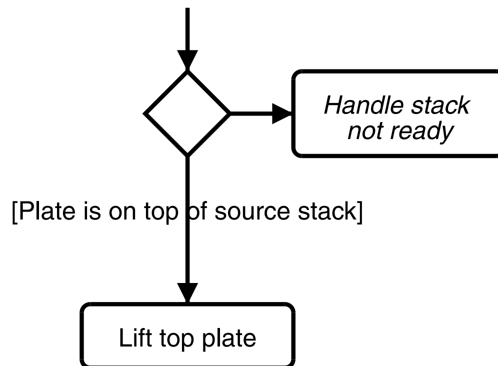

Figure 4: Lift plate.

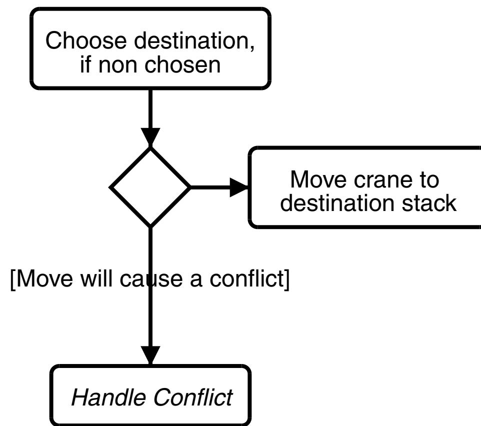

Figure 5: Move crane to destination stack of movement.

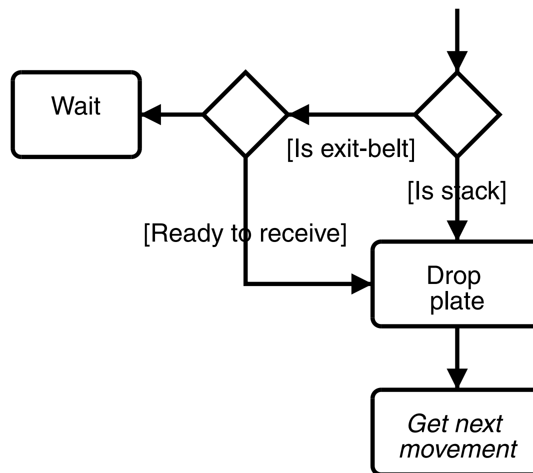

Figure 6: Drop plate.

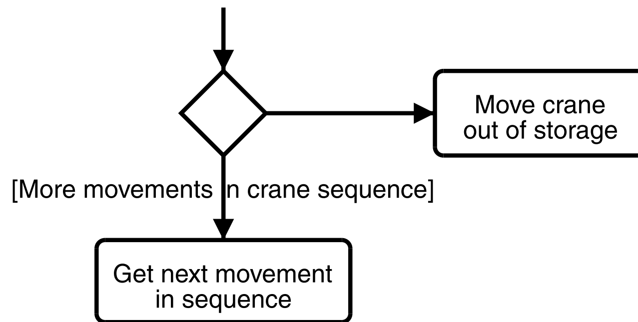

Figure 7: Get next movement in sequence of crane.

2.2.1 Handling of Stack not ready

Figure 8 on page 130 shows the process of handling the case when a requested plate is not ready to be moved. Several cases must be taken into account:

The plate is not in the source stack

A plate will need to be moved more than once, if it is moved to a stack in which there are plates that are going to be moved later on the same day. Generally one or more movements are necessary for a given plate.

If at some stage a plate is not in the expected stack, it means that it has not yet arrived from some other stack. If the current movement of the other crane is to move the plate to the stack, then we wait for the other crane to finish its movement. Otherwise we remove the movement from the sequence of the crane as well as any following movements of that plate if any exists. Note that the movement cannot be an exit-movement, since plates involved in exit movements are moved directly to the exit-belt and are therefore always present in the source stack until it is moved.

The plate is not the top plate of the source stack

If the plate is not the top plate of the source stack, the crane does a dig-up of the top plate, unless the current movement of the other crane is to dig-up the top plate. In that case we wait for the other crane to take the plate.

The destination stack is not ready to receive the plate

If the destination stack is not ready to receive the plate, the reason is that all plates to be removed from the destination stack have not yet been removed. To manage that we first consider moving the plate to another stack, which is ready, and alternatively removing the top plate of the destination stack before the current plate movement. In that way we will in many cases avoid moving a plate more than once on the same day.

This is especially a problem for the block storage where only two stacks are designated to each block, and hence a large number of dig-ups can occur. In the worst case the simulation will enter an infinite loop of dig-ups and positioning movements of the cranes: Assume that crane 1 is going to move several plates from the first block stack and crane 2 from the second stack. First crane 1 lifts a plate from the first block stack and puts it on the second. Meanwhile the crane 2 is requesting the top plate of the second, but has to wait for crane 1 to finish its move. Now another plate is on top and crane 2 needs to do a digup. Crane 2 then moves the plate to the first stack. Again crane 1 has an extra plate to dig-up etc. We deem the solution infeasible if a relatively large number of dig-ups has been done from the same stack. This is actually the only case not taken directly into account by the simulation. Infeasible solutions are not a problem when using a local search heuristic, since we can simply try other neighbour solutions until we find a feasible one. We must however make sure that the first solution is guaranteed to be feasible. The strategies of the online heuristic are somewhat different avoiding this cyclic behavior. Experiments have shown that the cycles only occur for the block storage caused by the many digups. Cycles can hence be avoided by only allowing one of the cranes to move plates from stacks dedicated to a particular block.

2.2.2 Collision Handling

It is quite easy to check for collision, when the speed of the cranes is fixed. We only need to check the destination of the crane movement. Assume that the following holds for cranes 1 and 2: $x_1 < x_2$. If crane 1 is moving from x_1 to x'_1, we need to check if crane 2 is moving or not. If it is not moving, we check if $x'_1 < x_2$, or if it is moving the same for the destination $x'_1 < x'_2$.

If a collision of the cranes would occur we handle this as depicted in figure 9 on page 131. If possible the crane moves closer to the other crane, which will always be closer to the destination of the crane. If the crane is already next to the other crane we either wait or move away for the other crane to

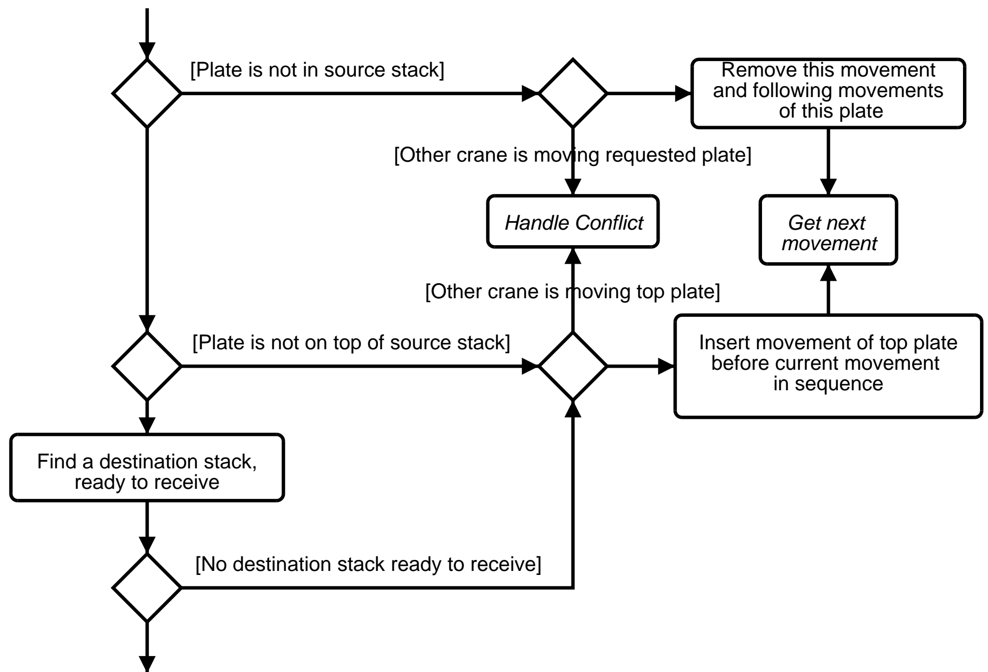

Figure 8: Handle stack not ready.

finish its operation. This procedure will solve any collision conflict between the cranes.

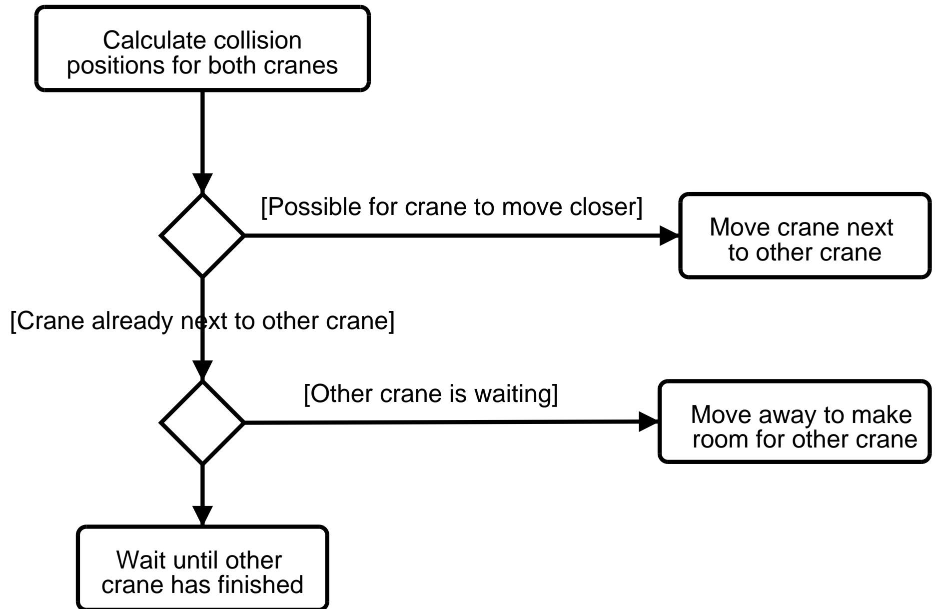

Figure 9: Handle collision conflict between cranes.

Note that this heuristic is not guaranteed to result in the best sequence of positioning and wait movements globally or even locally. We have tried different strategies for handling collisions, and clearly there is a trade off between tight and efficient schedules where the cranes are allowed to move very close to each other and robust but inefficient schedules where the crane movements are more restricted. The most "loose" strategy is when a crane can only perform its plate movement if the other crane is completely out of the area defined by the current position of the crane and the source and destination stack of the plate. Our choice is hence much more tight.

The simulation ends when all movements have been executed. The schedule can now be evaluated according to the criteria mentioned earlier.

2.3 Choosing a destination stack

For movements other than exit-movements we have to decide to which stack to move the plate. This is done during simulation just after lifting the plate.

How the stack is chosen depends on the storage principle.

For the block principle either the plate is going to leave the storage tomorrow and a corresponding set of due-date stacks exist, or another block stack next to the source stack is selected.

For the due-date principle a set of due-date stacks exists where stacks in zone 1 are preferred to stacks in zone 2 and so on. From the set of preferred stacks we always choose at random.

For the self-regulating storage principle all stacks are evaluated with regard to several criteria similar to the objective function. Preferably we want to put the plate on a stack where the plate with the minimum due date has a due date, which is later than or equal the due date of the plate we are moving. In that case the stack sorting will not deteriorate. On the other hand, we want the difference to the minimum due date in the stack to be as small as possible, because this will make it easier to find suitable stacks for other plates.

Consider for example the case in figure 10. If we in our sequence are moving plate p_7 followed by p_8, we have to decide to which stack of (2,4) and (3,4) to move p_7. The plate p_7 will not deteriorate the sorting of either stacks in position (2,4) or (3,4), but it is still better to move it to (2,4): If p_7 is moved to (3,4), then moving p_8 will cause an extra dig-up at a later stage.

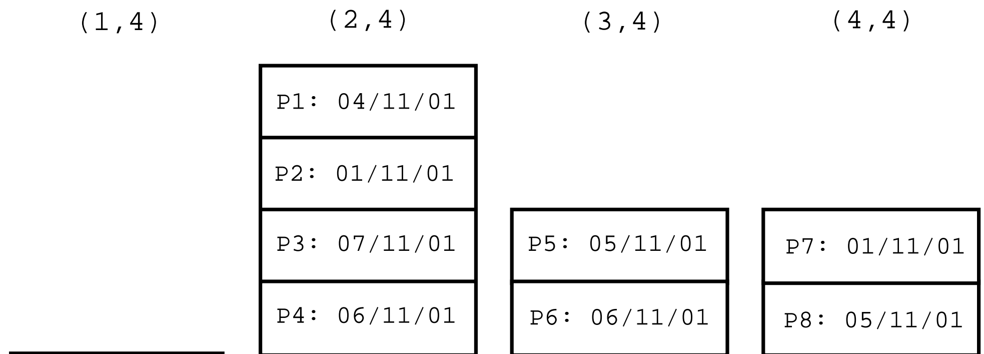

Figure 10: Choosing destination stack example.

We want to minimize travelled distance to the stack and further on to the source stack of the next movement. For this criteria (3,4) is better than (2,4), since the crane afterwards will move back to (4,4) to pickup p_8. When minimizing stack height, again (3,4) is better than (2,4) according to the stack height criteria. For minimizing distance to the exit-belt, (2,4) is better than (3,4) since stack (2,4) is closer to the exit-belt placed at (1,4).

We will now describe in more detail how a new destination is found for a movement m of a plate p_m with due-date dd_p. First stacks different from the source stack and the currently chosen stack (if any) are identified. From that set we select stacks that are ready to receive the plate, i.e. all plates leaving the stacks have been removed. If that set of stacks is empty, the new destination stack is picked randomly. Let us in the following call the set S' and let md_s be the minimum due date of a plate in stack $s \in S'$:

$$md_s = \min_{p \in s}\{dd_p\}, \ \forall s \in S' \tag{3}$$

From the set S', we identify two subsets:

$$S^{\geq} = \{s \in S' | md_s \geq dd_p\}, \ S^{<} = \{s \in S' | md_s < dd_p\}, \tag{4}$$

If $S^{\geq} \neq \emptyset$ we in the following use that set otherwise we use $S^{<}$ and refer to that set as S''. A subset of the best destination stacks are now chosen and the new destination stack for the movement is a random one from this subset. In the following we describe how the destinations are ranked.

For $s \in S''$ we record $md_s - dd_p$ multiplied by the cost of a dig-up. Let dc_s be that value. We will then have $dc_s \geq 0$ when $dd_p \leq md_s$. Minimizing this cost, we want dc_s to be as small as possible. If $dd_p > md_s$, this cost is disregarded, since no matter the cost we will have to do one dig-up. In addition to the stack sort the following cost changes are calculated:

- Change in cost caused by change in source and destination stack size.
- The cost of moving the crane: The change in cost is based on the time to move from the source to the new destination stack, $dist(s_m, d_m)$, and further on to the start of the next movement in the sequence, $dist(d_m, s_{succ(m)})$, where $succ(m)$ is the successor of m in the sequence of movements.
- The new distance to the exit-belt: The distance is weighted, such that the cost of plates with a due date close in time is larger than for plates due further into the future.

The motivation for these criteria is of course to minimize dig-ups, minimize makespan, travelling time and moving plates due for exit in the near future closer to the exit station. The weighting of these criteria is the same as for the objective function.

2.4 Neighbourhood Structures

It has turned out to be difficult to construct neighbourhood structures where local changes can easily be propagated to global changes in the objective value. Therefore the first attempt was to evaluate the change in objective value by "simply" simulating the crane movements from start to end for every new neighbour solution. This is of course computationally expensive and we have therefore investigated the possibility to estimate the change in objective value instead. This is discussed in section 2.6. Three simple operators define the neighbourhood structure:

Destination operator: Delete a specified destination for a movement and any following movements of that plate. A new destination will then be chosen afterwards while simulating the sequences.

TSP operator: Remove a movement from a crane sequence and insert it at another position in the same sequence.

VRP operator: Delete a movement from one crane sequence and insert it in the sequence of the other crane. The insertion is done at a position in the sequence at approximately the same time in the sequence.

Assume that two plates, p_1 and p_2 in a stack are going to be moved. When using the TSP or VRP operator, it must be checked that plate p_1 above p_2, is earlier in the schedule. We make the following checks for the TSP operator:

- If the movements are in the same sequence, we check if p_1 is earlier in the sequence than p_2.
- If the movements are in different sequences, the solution will always be legal, since the crane doing the succeeding movement just waits until the other crane has finished the preceding movement. This can obviously result in poor solutions. Therefore we try to estimate the new start and end times of the movements and if waiting is introduced we reject the neighbour solution. Note that this is also handled in the simulation if necessary.

For the VRP operator the two cases are in principle the same. In order to speed up these checks we set up suitable data-structures, such that it is possible to access both predecessors and successors of a movement in constant time. Note that we have two types of predecessors and successors of a movement given by the order of the plates in the stack and in the case that a plate is moved more than once.

In order to reduce the amount of copying object data we simulate the schedule backward to place the plates in the stacks they were at the beginning of the day.

One could come up with more complex neighbourhood structures. Basically all structures suitable for multiple travelling salesman type problems could be used. Note however that more complex structures will also lead to more difficulties in estimating possible savings and trying to check feasibility. This is the reason for our choice of simple operators.

2.5 Local Search Heuristics

A local search heuristic tries to improve the solution by continuously making local or small changes to the plan. We will in the following describe some of the different local search heuristics, which have been implemented.

2.5.1 Steepest Descent

The most simple heuristic is the *Steepest Descent* algorithm shown in algorithm 5. If the neighbourhood is very large we can search for the best neighbour in a subset of the neighbourhood set.

Algorithm 5: Steepest Descent

Select a solution, $s_0 \in S$ with objective $F(s_0)$. $n = 0$
repeat
  $F(s_n) \geq F(s_{n-1})$
until $n = n + 1$
Find the best neighbour s_n *in the neighbourhood* $N(s_{n-1})$

The algorithm can be modified to a *Descent* algorithm if the first improving neighbour solution is picked when searching the neighbourhood.

One obvious disadvantage of a simple descent algorithm is it's lack of ability to escape local optima. Other heuristics that do not suffer from this lack are *Simulated Annealing* and *Tabu Search* described in the following.

2.5.2 Simulated Annealing

Simulated Annealing originates in thermodynamics and metallurgy. Basically the problem is that of slowly cooling down metal to a state of minimal energy. In the local search version we have a temperature T as well, the state of the metal is our solution and the energy is our objective function value, which

we want to minimize. Kirkpatrick et.al. [10] was the first to "discover" this analogy. The procedure is shown in algorithm 6.

Algorithm 6: Simulated Annealing

Select a solution, $s_0 \in S$ with objective $F(s_0)$. $n = 0$
$F^\star = F(s_0)$. $s^\star = s_0$
$T(0) = T_{init}$
repeat
  Stopping criteria fulfilled
until *Draw a random neighbour s from* $N(s_n)$
if $F(s) \leq F(s_n)$ ***then***
  $s_{n+1} = s$
  if $F(s) < F^\star$ ***then***
    $s^\star = s$
Draw a random number $p \in [0; 1]$
if $p < \exp\left(\frac{F(s_n)-F(s)}{T(n)}\right)$ ***then***
  $s_{n+1} = s$
$n = n + 1$

We see in that it is possible to move to solutions, which are not better than the current solution. If the neighbour is worse, we still accept it with a probability depending on the temperature and how much worse the solution is.

The point is that we start with a relatively large temperature T_{init} where almost every solution is accepted, the temperature is gradually decreased until only improving solutions are accepted. In theory this procedure guarantees convergence to the global optimum with probability 1, but unfortunately in an exponential number of iterations, which is generally not feasible in practice. Instead we accept good solutions in a reasonable amount of time.

A number of decisions have to be taken. Choice of initial temperature, T_{init}, the procedure by of which the temperature is decreased (often referred to as the *cooling schedule*), and the stopping criteria.

Initial Temperature

Generally we want the temperature to be high in the beginning, but if the initial solution is reasonably good, we risk destroying the good features of the solution and waste time in the beginning of the search moving to increasingly

worse solutions. If the temperature on the other hand is too low, the search will behave like a descent algorithm. Note also that the probability of accepting solutions depends on the objective function. The initial temperature should therefore depend on the objective function as well.

We want in the beginning to accept worse solutions with a probability p_0 which is relatively large: Experiments of Johnson et. al. [9] indicate that a value between 0.4 and 0.9 is appropriate, depending on the quality of the initial solution and how much time is available. We have found that much lower values around 0.1 and 0.2 are required in order not to destroy a good initial solution.

Let ΔF be the change in objective function value between solutions. T_{init} should then for neighbours with increasing costs be set such that

$$e^{\frac{\Delta F}{T_{init}}} \approx p_0 \tag{5}$$

The most widely used procedure to find T_{init}, is to run the simulated annealing in an initial phase where the temperature is adjusted to approach the probability p_0 of accepting worse solutions. Johnson et. al. [9] were the first to suggest this procedure. Let $\overline{\Delta F}^+$ be the average increase in objective value for neighbour transitions to worse solutions. The temperature is then adjusted in the following way:

$$T_{n+1} = -\frac{\overline{\Delta F}^+}{\ln(p_0)} \tag{6}$$

After a number of iterations the initial phase is stopped and the decrease of the temperature is started according to the cooling schedule. T_{n+1} will approach the probability p_0 in (5).

The cooling schedule

The most widely used cooling schedule is the *geometric schedule*. Starting with the initial temperature T_0, the temperature is kept constant for L consecutive moves. Then after L moves it is decreased by multiplication with a constant α, $0 < \alpha < 1$. After nL iterations the temperature is

$$T(nL) = \alpha^n T_0 \tag{7}$$

No definite rules can be given on how to choose L and α, but generally α should be close to 1, e.g. $\alpha \approx 0.99$. L is more difficult to determine. When the temperature is low it should in principle be possible to try all possible

neighbour solutions at least once, which suggests a correlation of the size of the neighbourhood and L. For some problems with huge neighbourhood sizes it is unrealistic to have L even close to this value. We decided to set a low L of 10. Assume that a given number of iterations are to be used to get from the initial temperature to the final temperature. Then if L is increased, α must be decreased accordingly and vice versa. In that sense the choice of L and α cannot be separated.

We have implemented a more complex cooling schedule introduced by Aarts and van Laarhoven [1] and also discussed in van Laarhoven and Aarts [14]. Here we will give a brief description of the schedule. It is suggested that the search should be able to reach some *quasi-equilibrium* in the objective values for the given temperature before adjusting it. To ensure a fast convergence to a quasi-equilibrium and hence a small L the decrements of the temperature must be small. During the n'th value of the temperature, we record the objective values and calculate the mean, μ_n, and variance, σ_n, assuming that the values are normally distributed. The decrement rule is then specified as:

$$T_{n+1} = \frac{T_n}{1 + \frac{T_n \ln(1+\delta)}{3\sigma_n}} \tag{8}$$

The *distance* parameter, δ, set by the user, specifies the rate of decrease of T. Larger δ generally result in faster convergence, but worse solutions. After initial experiments the value was set to 0.5. A small value of σ basically means that the search has reached a quasi-equilibrium and a larger decrease in T is justified. Refer to van Laarhoven and Aarts [14] for an overview of other similar cooling schedules.

Several runs with a faster decrease of the temperature can be superior to one run with a very slow decrease. In particular if the initial solution has been generated with a good construction heuristic. Another possibility we have implemented as well, is a *re-heating* procedure, which increases the temperature, if the search has been trapped in a local optima or has not improved the best solution in a number of iterations. Rules for when, how much and how often to re-heat proved difficult to determine. Since complete restart of the search was just as good, we decided to recommend multiple runs instead.

Stopping criteria

We want the algorithm to stop when future improvements are expected to be small. Usual criteria are therefore maximum number of iterations without

improvement or temperature less than a threshold value. In other situations the search has to be stopped because of limited time: maximum running time or maximum number of iterations.

2.5.3 Tabu Search

Tabu Search is like Simulated Annealing a general local search heuristic constructed to be able to escape local optima. Glover et. al. [6] gives an extensive guide to the use of Tabu Search.

The basic idea of Tabu Search is to move to the best solution in the neighbourhood as in Steepest Descent, but where moving to worse solutions is possible in order to escape a local optimum. In case of a symmetric neighbourhood the following cyclic behaviour will occur: When reaching a local optimum there is an immediate risk that moving away from the optimum will result in a move back to the local optimum. To avoid the cyclic behaviour a tabu list, TL, is introduced: We want it to be tabu to move back to the solution we came from, so in principle we could save the entire solution and check whether or not we had visited this solution earlier in the search. It can be both time and memory consuming to do this. Instead we store certain ***attributes***, $a(s)$, in the list representing the movement from one solution to another solution. In the following $|TL|$ iterations, it is then forbidden to make any changes to the solution according to the attribute preventing a return to previously visited solutions.

Let us review the tabu list issue for our crane scheduling problem. For example when changing the destination of a plate movement m from stack (x_1, y_1) to (x_2, y_2), the following different attributes are possible to save in the tabu list:

1. Save the combination move, m, and stack (x_1, y_1), which means that in a certain number of iterations we cannot change the destination of movement m back to (x_1, y_1).

2. Make it tabu to change the destination to *any* other stack.

3. Make it tabu to change m at all, for instance by assigning it to the sequence of the other crane.

We see that the first is less restricting than the second one and again less restricting compared to the third. Later we discuss different ways of representing the tabu list. The pseudo-code of the algorithm is shown in algorithm 7 on the following page.

Algorithm 7: Tabu Search

```
Select a solution, s_0 ∈ S with objective F(s_0)
F* = F(s_0), s* = s_0, n = 0
TL = ∅
repeat
 | Stopping criteria fulfilled
until F̄ = ∞
for all s ∈ N(s_n) do
 | if (F(s) < F̄ and a(s) ∉ TL) or F(s) < F* then
 |  └ s̄ = s. F̄ = F(s)
 └
s_{n+1} = s̄
TL = TL ∪ a(s_{n+1})
if TL is filled up then
 └ remove oldest element from TL
if F̄ < F* then
 └ s* = s̄. F* = F̄
n = n + 1
```

Note that we accept a solution if it is better than the best solution found so far even if it is tabu. This is often called an *aspiration criterion.*

Defining a move to be tabu in the above algorithm is done by adding the attribute to a FIFO list. Alternatively we can define an attribute to be tabu in a number of future iterations. In our application we have an array for each neighbourhood operator and an entry in the array for each plate movement. When a neighbour solution is selected, we record in the appropriate array-entry the iteration number when the plate movement can be changed again by the given operator. We are in other words using the second attribute alternative discussed earlier on the page before.

The most difficult aspects of implementing good Tabu Search heuristics are the questions of which attributes to set tabu and how long the attributes should be tabu. The first is very much depending on the chosen neighbourhood structure and how much cyclic behaviour is observed, but this is however not known in advance.

2.5.4 Reactive Tabu Search

Concerning how long attributes should be tabu, a good choice is to implement a *reactive tabu search* introduced by Battiti and Tecchiolli [4, 3]. The tabu

length is changing dynamically in order to intensify the search in promising regions and to diversify the search to investigate other regions of the solution space. In this way the issue of tuning parameters is partially avoided, since parameters controlling the dynamic algorithm still have to be tuned. These parameters are however in our experience easier to adjust and more robust.

In our implementation we record the solution value, iteration number and the number of occurrences of the solution value in a map-type data structure. In that way we can check if the search repeatedly return to the same solution. The chain of neighbour solutions leading back to an already visited solution is often refered to as a *cycle*. Different solutions can of course result in the same objective function value, but that is not a major problem. For a more detailed discussion on that and other ways to store solution configurations, we refer to [4, 3]. If, during the search, the current solution has been visited earlier or rather the objective value has occured earlier, the tabu length is increased. If solutions are chosen which have not been visited earlier, the tabu length is slowly decreased. The reactive search scheme includes a final *escape mechanism*, when revisiting solutions repeatedly. In that case a random number of random neighbour movements are performed. The number of moves is depending on the length of the cycle.

2.6 Estimation

Calculating start and end times of the movements and evaluating a solution by simulation has shown to be very time consuming. This is especially a problem for heuristics that investigate the entire neighbourhood like Tabu Search. For some real-life-size instances with approximately 800 movements on a day every solution has up to 1/4 million potential neighbour solutions. For this reason we have investigated the possibility of estimating the change in cost of a local change to the solution. Since the cost is a weighted sum of different criteria, we must estimate the change of all these different criteria. In the following we will describe the calculation of the estimates for the different neighbourhood operators.

2.6.1 Destination Stack Neighbourhood

For the destination stack neighbourhood we evaluate changing the current destination stack of a movement to all other stacks. The evaluation is divided into two parts: The saving in cost of not moving the plate to the current destination stack and the cost of going to another stack instead.

The cost saving is a sum of the following terms. Saving in moved distance by the crane from the source stack to the destination and further on to the source stack of the next movement. This saving can influence several cost criteria: Cost of moving the crane, change in makespan for all movements, change in makespan of exit movements and stack costs.

The potential saving in makespan is not at all guaranteed for several reasons. The movement might not be in the sequence defining the makespan and the fact that the two cranes cannot cross each other may later reduce the saving to nothing or even increase the makespan. To reduce the error we only count the saving in makespan, if the sequence is actually defining the makespan.

The saving in makespan of exit movements can only be decreased if the movement is before the last exit movement and it is actually possible to decrease the exit time. We say that it is only possible to reduce the makespan, if the following is fulfilled:

- The crane at the exit belt does the movement.
- No crane has been waiting at the exit belt to deliver a plate.
- The exit time of the current solution is larger than a bound on the best possible exit time. The bound is the time to get all exit-plates through the exit-belt assuming that the belt is never empty.

Note that the issues above for the total makespan also apply for the exit makespan.

Finally we calculate the saving in stack sort, size and exit distance. These costs are estimated by removing the plate from the stack. For the size and exit distance this estimate is correct. For the stack sorting this is however not necessarily the case. Removing a plate and recalculating the cost is of course correct, but the change in the plan can actually change the order of plates being put on the stack later on the same day and hereby result in an error.

Now the plate is to be moved to another stack. The estimated cost of a new destination stack follows the same procedure as for the saving above, but is a bit different for the stack sorting. Now we are to find the correct position in the stack to put in the plate. This is done by estimating the arrival time at the stack and other plates removal times from and arrival times to the stack and by this estimating the position in the stack. Again this estimate may be crude.

2.6.2 TSP and VRP operators

Now we consider the neighbourhood defined by re-insertion of a movement at another position in the same sequence. When removing a movement from a sequence the saving is the start time of the following movement minus the end time of the previous movement plus the time of moving the crane from the end of the previous movement directly to the following movement. When inserting the movement between movement i and j the opposite applies. Instead of going directly from i to j we go via the inserted movement. Whether or not the saving is effective again depends on the movements of the other crane and the exit-belt. If the movement was inserted earlier in the sequence, the plate might have a lower position in the destination stack, and if the movement is inserted later a higher position may result. This is estimated in the same way as for the destination stack neighbourhood.

The estimate for the VRP operator is calculated basically in the same way as for the TSP operator.

2.6.3 Comparison of estimation and correct evaluation

When comparing estimation and exact evaluation two factors are important, speedup and quality of the evaluation. On an example instance with 1856 movements on a day by two cranes it took 67 seconds to evaluate 1000 solutions exactly while estimating the same number of solutions took only 0.19 seconds. A significant speedup of 352.

Figure 11 on the following page shows an indication of the quality of the estimation compared to the correct objective function evaluation. The quality is measured in the following way. If the estimated change in objective value and the correct change are of the same sign, then we say that the estimate is "correct". If on the other hand the estimate indicates an improvement and the change is actually the opposite, the estimate was "optimistic". The last possibility is the "pessimistic" estimate compared to an improvement of the solution. The stars ($*$) in figure 11 indicate the percentage of neighbour solutions that were estimated correctly. On average only about 50% of the solutions were estimated correctly and it is down to 40%. About 30-40% of the solutions were wrongly estimated to improve the solution, which is too high to be useful. The figure also indicate that after around 500 iterations the quality significantly deteriorate. This is partly caused by the way we measure the quality. At that stage the best changes in objective value are close to zero and hence much more likely to be estimated with an opposite sign.

Figure 12 on page 145 shows a plot with the first 100 iterations comparing

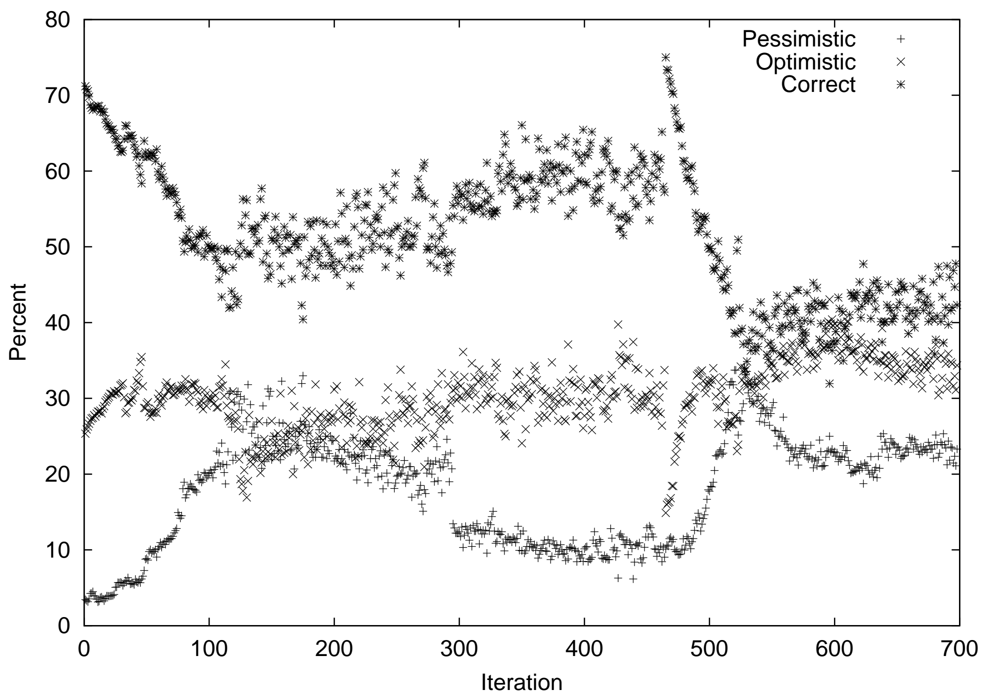

Figure 11: Quality of estimation.

the *best estimated* change and the corresponding *correct* value as well as the neighbour solution with the *best correct* value and the corresponding *estimate*. After each iteration the best estimated neighbour solution is picked. In all 100 iterations the best estimate is too optimistic, but the correct value is still improving the solution.

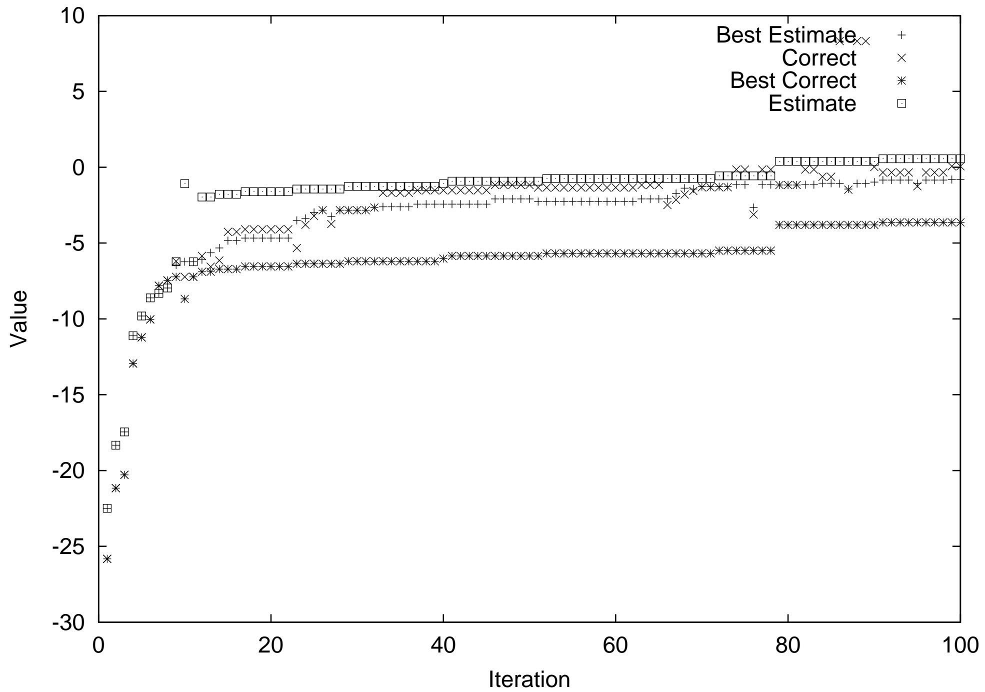

Figure 12: Estimates compared to correct values for iteration 0-100.

In figure 13 on the following page the following 100 iterations are shown. As in the previous figure, the *best correct* value is disregarded since the *estimate* is too pessimistic – now above zero. In a lot of cases the *best estimate* is completely wrong leading to very poor solutions.

To avoid this behaviour we sort the neighbour solutions in increasing order of the change in objective value. Before picking a neighbour as a new solution, the correct objective value is found. If the value is improving the current solution we pick the solution otherwise the next best estimate is evaluated. This is done until all neighbours are evalutated. At that stage we select the neighbour with the best correct value which is then worse than the current.

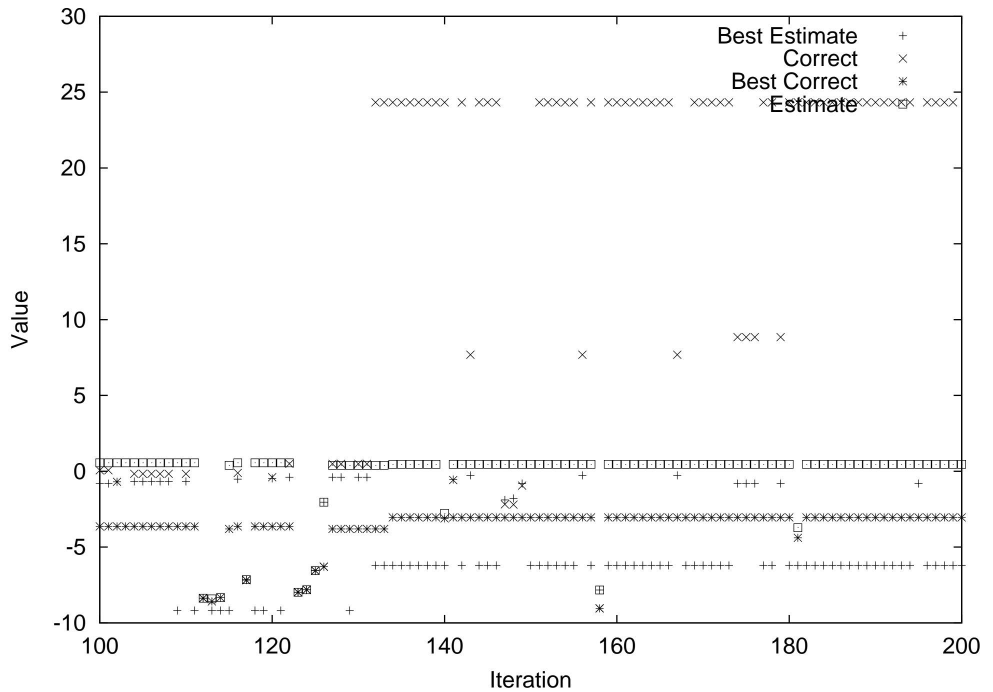

Figure 13: Estimates compared to correct values for iteration 100-200.

In a trial run of 3000 iterations, the first solution was picked in 1997 cases, one of the first 4 solutions were picked in more than 90% of the iterations and the maximum number of solutions evaluated was 14. This simple procedure significantly improves performance.

2.7 Hybrid Search

The entire neighbourhood size was 458.012 for the example in the beginning of section 2.6.3. It takes 86 seconds to evaluate all neighbours even with estimation and hence it is too slow to be useful in practice. Instead of evaluating the entire neighbourhood a random subset is evaluated. We have experimented with an initial subset size of 100 ($subsize = 100$), which is slowly increased according to different strategies in order to allow a more thorough investigation of the neighbourhood when it becomes more difficult to find improving neighbour solutions. Specially we increase the subset by one, if the picked

neighbour was not the best estimated or if the best estimated was worse than the current solution. In addition to that is a more radical change of the size: If no improvement of the best solution so far has occurred in the last *subsize* iterations, the *subsize* is doubled. Trial runs were also done with the descent algorithm. Here a neighbour were only picked if the solution was strictly better than the current. The subset size was incremented in the same way.

Voudouris and Tsang [16] introduces a more intelligent way of restricting the size of the neighbourhood called *Fast Local Search*. Another possibility is *Variable Neighbourhood Search* by Mladenovic and Hansen [11] where the algorithm shifts from one neighbourhood structures to the next at certain intervals or each time a local optima is found.

In Reactive Tabu Search the tabu length determines the degree of intensification and diversification during the search. In Simulated Annealing the temperature and randomness take this role. As we have seen above, randomization can be successfully combined with Tabu Search as well and more intensive neighbour search can be used in Simulated Annealing.

In the literature a lot of other interesting ideas have been suggested to diversify and intensify the search. We will briefly comment on some of them in the following, although we have not implemented them in our system.

Frequency functions are widely used in combination with Tabu Search [6]. The frequency of specific movement attributes are recorded during the search. A high frequency is perhaps an indication of long cycles, which could then be avoided by penalizing the move attribute in the objective function and hence diversify the search. Another strategy is to intensify the search by recording solution features occurring in good solutions and fixing them in a number of iterations.

Augmenting the cost function with penalty functions is also the main idea in *Guided Local Search* by Voudouris and Tsang [16]. The method works on top of another local search heuristic guiding the search to unexplored regions by penalizing solution features occurring in local optima. Solution features in the plate storage case could for instance be of the following types:

- Destination stack d for a movement m.
- Movement m_i followed directly by m_j.
- Movement m assigned to crane c.

The question is how to represent the features in the objective function: One way is directly, by setting up penalty matrices with elements, p^1_{md}, p^2_{ij} and p^3_{mc}. We get the total penalty by multiplying the penalty matrices with matrices

indicating occurring features. Another possibility is to adjust the distance between stacks, which is similar to [16] for the Travelling Salesman Problem. Note that this should only influence the cost – not feasibility.

Our problem is a multi criteria problem where the criteria have been weighted to form one single objective. If we were interested in a *pareto efficient* set of solutions, forcing the search to visit different parts of the solution landscape by changing the weights in objective function could be applied. Ehrgott and Gandibleux [5] gives a good overview of multi-objective combinatorial optimization related papers.

3 The Control System

The control modules are developed in order to handle the stochastic phenomena at the storage and deadlocks or blocking situations.

Two different types of control modules are developed. Both are heuristic discrete event feedback control methods. One of the control modules simply execute the plan received from the planning module. The method is described in section 3.6.

The other module discussed in the following choose the "optimal" job online for the crane that "asks" for a new job. Basically the state of the storage plant is fed back to the control module and it chooses the optimal job from the list of movement jobs and the state of the storage plant in order to fulfil the control-rules and respect the constraints. The concept of the module is depicted in figure 14.

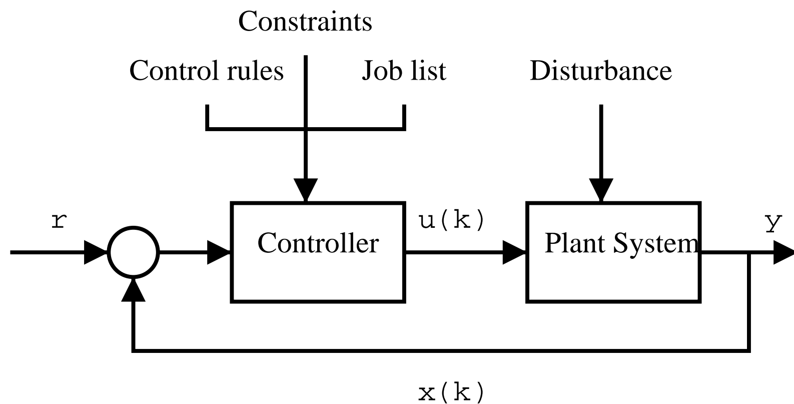

Figure 14: The concept of the controller.

Before the control module is called the movements are generated that are to be executed the current day. These movements are kept in an unsorted movement list. When the control module is called it splits the movement list into two lists. One of the lists contains all executable movements and the rest of the movements are contained in the other. This is done to reduce the computation time when the movement to be executed next is found, because only the executable movement list has to be searched. A movement is put into the executable movement list when the plate for the movement is located at the top of the stack and the destination stack for the plate is ready to receive the plate. The destination stack is ready to receive a plate when all plates that are supposed to leave the stack the current day have been removed. If the destination is the exit-belt then at least one slot has to be empty.

3.1 Performance

The control systems developed are making decisions by use of information of the current state of the storage. The decision is made without considering how the decision will affect the performance in the future, but when measuring the performance of the system, it is done over a period of time. This means that the rules used by the control system should be formulated to fulfill the goals over a period of time. Performance and robustness are strongly related. It is important that the control system can handle situations where the number of resources is reduced or the capacity of the plant is reduced because of a break down or other reasons. Furthermore it is important that disturbances that result in variations in process times do not cause the system to break down.

If a control system is designed to support good performance it often uses simulation into the future based on estimates for the process times.

3.2 Robustness

It is important to remember that humans operate the cranes, so the control system is a so-called *Human-in-the-loop feedback control system*. A control system can control both humans and fully automated systems, e.g. robot welding cells. The main difference between controlling humans and robots is that robots must have a control system while the workers use the control system as a guide to make better decisions and thereby increase the utilization of the production equipment and the attached resources. The difference means that control systems developed for automated systems are more detailed and the constraints are not to be violated, whereas control systems developed to control humans can profit by the flexibility of the human brain. With respect

to rules and constraints, humans can handle some special or extreme situations. In other words automated systems have a higher need for robustness.

A couple of examples on this for the plate storage:

- One of the cranes breaks down. If the crane can be moved to the safe-position, then the other crane just executes all the remaining movements.
- The exit-belt breaks down. The two cranes execute all movements except from the exit-movements and the movements that are blocked by the exit-movements.

3.3 Collision check

Critical situations never occur when the cranes are moving in the same direction. The cranes move linearly between two points and therefore it is only necessary to check these end-points. The check is performed by simulating one

Position	**Point of time**
Current position	Current time; (time = 0)
Current position of the plate	Arrival time for the crane
Current position of the plate	Departure time for the crane
Destination for the plate	Arrival time for the crane
Destination for the plate	Ready to departure time for the crane

Table 1: Collision check points.

movement forward. Critical positions and respective times for the idle crane are calculated according to table 1. These positions and times are compared with the position of the other crane at the respective times. To make a robust check the check is performed by calculating the combinations of worst/best case scenarios with respect to the lift/drop times. In extreme situations when there is no job to execute and the job list is not empty then new destinations are found for these plates and the control module tries to find a job to execute. This is just done for a limited number of times and if an executable job is still not found then the idle crane is moved to a *safe position.*

3.4 Deadlocks

A deadlock is a situation in which resources used for the same process are effectively preventing each other from executing any task, resulting in both

resources ceasing to fulfill their task. Basically there are two ways of dealing with deadlocks. The control module can either be designed to avoid them or they can be designed to deal with the problem when they occur. One may argue that it is obviously better to avoid the problem than deal with deadlocks when they occur. Avoiding them may be cheapest, but it is not necessarily the case, because it also cost to avoid deadlocks. Furthermore it is not guaranteed that all deadlock situations can be avoided, because of disturbances. Dealing with deadlocks dynamically when they occur also costs. If a deadlock occurs a strategy is needed to decide which of the resources that has "the right of way". The other resource should then allow this resource to start executing a task. The strategy can be more or less simple. A simple strategy can be more expensive, but more robust, while a more complex strategy can be better, but less robust.

At the plate storage a deadlock occur when the only executable movements are on the other side of the other crane or if none of the remaining movements are executable, e.g. when the top plates are exit-moves and the exit-belt is full. In these situations when no movement can be dispatched to the idle crane a movement is generated for the crane ordering it to move out of the storage. These positions are denoted as *safe-positions*. When one of the cranes is in its safe-position the other crane can operate freely in the entire storage area. When the idle crane arrives to the safe-position it naturally asks for a new movement to execute. If there is still no movement to execute then the crane is told to stay at the safe-position until a movement is executable.

The exit-belt cannot result in a deadlock because the cranes are not allowed to start executing a job that is not possible to finish.

3.5 Priorities

If a set of tasks is possible to execute, a choice have to be made as to which of the tasks to execute first. The tasks can often be divided into types with different priorities. The priorities can indicate the necessity of executing a task. The priority can be static or dynamic. If the priority is dynamic the state of the plant and the available resources determines the priorities. The purpose of having different priorities is to increase the performance of the resources and plant.

Generally exit and dig-up movements have higher priority than doing sort movements or moving arriving plates into the storage. These priorities change dynamically for instance if the exit-belt is filled up.

Other rules ensure that one crane is operating in the area near the quay

and the other crane is operating near the exit-belt. Thereby the cranes are separated as much as possible and more movements are possible to execute.

As an alternative one could choose to use just one crane instead of two cranes and hence the coordination problem would be eliminated. Naturally the collision check is then not performed, the priorities are different and the crane is allowed to operate in the entire storage area.

3.6 Control with a reference Sequence

A simple control system has been implemented to execute the optimized sequences of movements returned from the planning module. This control module uses the same method as described above for the collision check. If there is no collision the execution continues. Otherwise the controller adjusts the progress of the execution of the sequences for the two cranes. If a crane is delayed the other crane waits. Alternatively if a crane is ahead in time compared to the plan it is ordered to wait for the other crane.

More advanced control systems include the possibility to change the reference plan or let the planning module dynamically improve the plan to take disturbances into account.

Range and Yde [13] suggest another decomposition where the planning module determines a partial order of the movements and the control module afterwards schedules the movements with respect to the partial order. A partial order is found in the following way. For plates in the same original stack a partial order is naturally defined. For all the movements the destination stacks are determined and an order in which the plates are to arrive at the destinations is found. Two movements can then be scheduled independently if all source and destination stacks are different. Given the partial order of movement a second planning phase can either on-line or off-line fully determine the order in which the movements are to executed by the cranes. The suggested approach have not been implemented and will hence not be discussed further.

4 Conclusion

In this paper we have described different approaches to solving the problem of scheduling cranes at a plate storage. The problem is hard to solve since it is both a question of placing the plates on stacks in order to minimize future movements, but also scheduling the crane movements to minimize moved distance and at the same avoiding collisions.

Two different approaches were suggested to solve the problem. An on-line approach and a control approach executing sequences of movements achieved by meta-heuristics.

The complexity of the problem makes it impossible to evaluate exactly feasibility and change in cost for neighbour solutions without simulating the entire sequence. Experiments on estimating the change instead were reported, showing the difficulty of estimating the cost function.

References

[1] E. Aarts and P. van Laarhoven, *Statistical cooling: A general approach to combinatorial optimization problems*, Philips Journal of Research, 40 (1985), pp. 193–226.

[2] A. A. Andreatta, S. Carvalho, and C. Ribeiro, *A framework for local search heuristics for combinatorial optimization problems*, in Voss and Woodruff [15], 2002, ch. 3.

[3] R. Battiti, *Reactive Search: Toward Self-Tuning Heuristics*, Modern Heuristic Search Methods, John Wiley and Sons Ltd., 1996, ch. 4, pp. 61–83.

[4] R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing, 6 (1994), pp. 128–140.

[5] M. Ehrgott and X. Gandibleux, *A survey and annotated bibliography of multiobjective combinatorial optimization*, OR Spectrum, 22 (2000), pp. 425–460.

[6] F. Glover, E. Taillard, and D. de Werra, *A user's guide to tabu search*, Annals of Operations Research, 41 (1993), pp. 3–28.

[7] J. Hansen and T. F. H. Kristensen, *Crane scheduling for a plate storage in a shipyard: Experiments and results*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-12 (2003).

[8] ——, *Crane scheduling for a plate storage in a shipyard: Modelling the problem*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-4 (2003).

[9] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon, *Optimisation by simulated annealing: an experimental evaluation; part i, graph partitioning*, Operation Research, 37 (1989), pp. 865–892.

[10] S. Kirkpatrick, C. Gelatt, and M. Vecchi, *Optimization by simulated annealing*, IBM Research Report, RC9355 (1992).

[11] M. Mladenovic and P. Hansen, *Variable neighbourhood search*, Computers and Operations Research, 24 (1997), pp. 1097–1100.

[12] M. Pirlot, *General local search heuristics in combinatorial optimization : a tutorial*, Belgian Journal of Operations Research, Statistics and Computer Science, 32 (1992).

[13] T. M. Range and S. Yde, *Storage management at odense steel shipyard. simulation, product placement and control.*, Master's thesis, University of Southern Denmark, 2002. (In Danish).

[14] P. van Laarhoven and E. Aarts, *Simulated Annealing: Theory and Applications*, Mathematics and its applications, D. Reidel Publishing Company, 1987.

[15] S. Voss and D. Woodruff, eds., *Optimization Software Class Libraries*, Kluwer Academic Publishers, 2002.

[16] C. Voudouris and E. Tsang, *Guided local search and its application to the travelling salesman problem*, European Journal of Operational Research, 113 (1999), pp. 469–499.

Paper III

Crane scheduling for a Plate Storage in a Shipyard: Experiments and Results

Available as IMM Technical Report 2003-12.

Crane scheduling for a Plate Storage in a Shipyard: Experiments and Results

Jesper Hansen[1] **and Torben F. H. Kristensen**[2]

Abstract

This document is the third in a series of three describing an investigation of possible improvements in methods for handling the storage of steel plates at Odense Steel Shipyard (OSS). Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce a ship.
Two gantry cranes move the plates into, around and out of the storage when needed in production. Different principles for organizing the storage and also different approaches for solving the problem are compared. Our results indicate a potential reduction in movements by 67% and reduction in time by 39% compared to current practices. This leads to an estimated cost saving by approx. 1.0 mill. dkr. per year.
This paper describes the experiments and achieved results based on the model and the approaches to solve the problem described in Hansen and Kristensen [2] and [3].

Keywords: Crane scheduling, plate storage

1 Introduction

This paper is the third in a series of three papers describing an investigation of possible improvements in methods for handling the storage of steel plates at Odense Steel Shipyard (OSS). Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce a ship.

In Hansen and Kristensen [2] we described the physical environment on the plate storage and how it fits into the overall process of building steel plate ships. Based on the problem description, a model was developed including

[1] Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark

[2] Department of Production, Aalborg University, 9220 Aalborg Ø, Denmark

both the physical entities of the system and the planning and processing aspects. In this paper we will discuss experiments and achieved results, but we will first briefly repeat the main modelling concepts from [2].

Our model of the storage consists of 8×24 plate stacks and two gantry cranes can move plates between the stacks. 8 additional stacks are used for arriving plates. Each plate has a due date specifying at which date it must leave the storage and enter the production line. When a plate leave the storage, it is placed on a conveyer belt referred to as the exit-belt. A maximum of 8 plates can be at the belt at the same time and a plate is drawn from the exit-belt at certain intervals. The two gantry cranes share tracks and can hence not cross each other. The adressed problem is to develop approaches for scheduling the crane operations better than the current practices. Three different storage principles are considered:

The block storage is the current storage principle. Two stacks are associated to a specific block of the ship and all plates to be used to produce that block are placed in these designated stacks.

The due date storage was suggested by the shipyard management as an alternative to the block storage. Here each stack is assigned a due date interval and a plate with a due date in that interval can be placed on that stack. The storage is divided into zones and each zone is divided into a number of due date intervals. Several stacks are assigned each due date interval. The user can determine the layout of the storage by specifying the different parameters regarding size of the intervals, stacks per interval, number of intervals and overlap in due dates between zones. Refer to [2] for more details.

The self-regulating storage principle is the last alternative. No specific purpose or due date is assigned to the stacks. The organization of the storage is determined by the planning procedure.

In Hansen and Kristensen [3] different approaches to solving the problem of scheduling cranes at plate storage was discussed. The problem is hard to solve since it is both a question of placing the plates on stacks in order to minimize future movements, but also scheduling the crane movements to minimize moved distance and at the same avoiding collisions. Two approaches are investigated. The first approach is an *on-line algorithm* or more precisely a *heuristic discrete event feedback control system* where an operation is chosen in on-line. The other approach uses the on-line algorithm off-line as a greedy construction heuristic to get a good initial solution, which is then improved by

local search heuristics. The off-line algorithm makes a schedule for the controller to follow. The schedule specifies the order in which the movements are to be executed. A control module dispatches the operations in the sequences and adjusts the schedule if the initial schedule becomes infeasible caused by the underlying stochastic nature of the system.

The major goal of the investigation reported in this and earlier papers is to compare the three proposed ways of managing the steel plate storage: The block, due date and self-regulating storage. Experiments reported in Hansen and Clausen [1] shows that a change from the block principle to any of the two others would result in a saving of approximately 50% in number of movements and 40% in total makespan. No further experiments have been conducted for the block storage since it is clearly inferior. Instead we will focus on a comparison of the due date and the self-regulating principles.

The paper is organized as follows: In section 2 the different local search methods described in Hansen and Kristensen [3] are compared. In section 3 we describe the experimental setup and report on the results comparing the different storage principles and solution approaches. Finally in section 4 we conclude on the findings.

2 Comparison of Search Methods

The complexity of the problem makes it impossible to evaluate exactly feasibility and change in cost for neighbour solutions without simulating the entire sequence. Experiments on estimating the change instead were reported in [3] showing the difficulty of estimating the cost function. Before picking a neighbour as a new solution when using estimation, the correct objective value is found. If the value is improving the current solution, we pick the neighbour solution and otherwise the next best estimate is evaluated. This is repeated until all neighbours are evalutated. At that stage we select the neighbour with the best correct value which is then worse than the current solution. For the Steepest Descent algorithm and Tabu Search, estimation was still better than a complete evaluation since complete evaluation was too time consuming.

Here we compare the performance of the different local search methods that we have implemented. More precisely a Simulated Annealing, a Steepest Descent algorithm and a Reactive Tabu Search. The last two with estimation and evaluation of only a subset of the neighbours. Unless a subset is chosen, the search will be too time consuming. Initially the subset has size 100, but it is gradually increased when the estimates deteriorate as explained in [3].

The experiments are divided into two parts. First we discuss the comparison for the due date principle only. The results are shown in section 2.1. Following that, the conclusion is tested on the self-regulating storage principle. This experiment is described in section 2.2.

2.1 Tests for the Due Date Storage Principle

The test instance has 1856 movements on one day performed by one crane. The potential number of solutions are 1856! not including destination stack decisions. All three algorithms are run for 2 hours on the same machine.

In figure 1 on the next page the runs of the different search methods are depicted. For the Simulated Annealing the objective value seems to converge nicely to 65402. Looking at the Tabu Search it converges much faster and reaches the same value after just 443 seconds ending at 63434. Finally the simple descent algorithm rather surprisingly reaches a similar value and at that stage still finds improving solutions i.e. a local optimum is not found within two hours. The Descent and Tabu Search only iterates through approximately 1800 solutions.

In some cases, in the end over 500 neighbour solutions are evaluated. This is an indication of the difficulty of estimating the change in objective function. Apparently 499 solutions were estimated better than the chosen one, but all of them were non-improving.

Further investigating the supposedly good performance of the descent algorithm, we exactly evaluate all 100 neighbours in the subset before picking the best of them – a form of steepest descent. The search reached a local optimum of the sub neighbourhood after only 70 iterations with an objective value of 67621. The "more gentle" descent, estimating the objective instead of exact evaluation gives a better convergence, since the estimation error allows the search to avoid local optima.

The descent algorithm is then run for more than two hours. After almost 5 hours and 2256 iterations a solution with value 62916 is achieved. In the final iteration 1560 random neighbour solutions were evaluated taking 1.5 minutes without finding an improving solution. There is of course no guarantee that this is actually a local optimum, but it is still surprising how much better the performance is compared to the Simulated Annealing.

To check the possible variation of quality over different runs, 5 runs are performed and the results plotted in figure 2 on page 162. The longer running times are to compensate for a slower machine. Note that the time actually runs out without a local optimum being reached. The variation is quite small

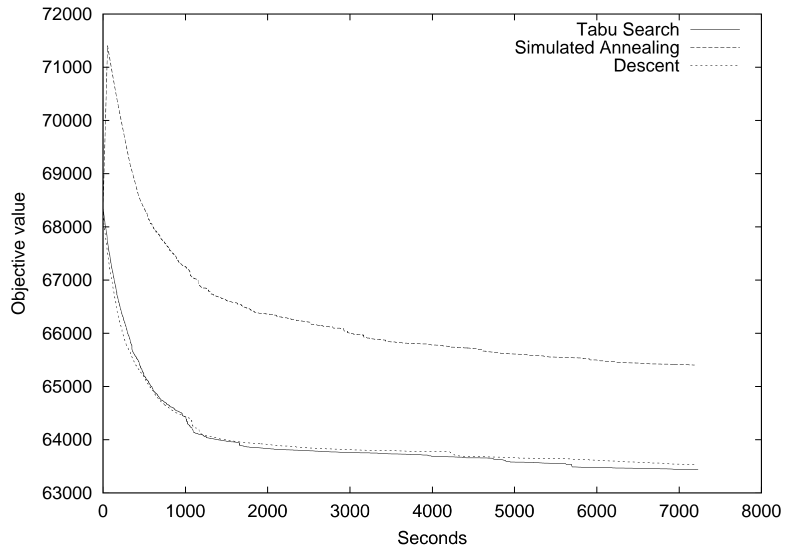

Figure 1: Comparison of different search methods.

with a spread of only 42 indicating a robust search method.

We now return to the Simulated Annealing. In the beginning the temperature may be too high, since the search is not able to recover the large increase in objective value. We decrease the initial probability of accept to 0.01 even though in figure 1 it is only 0.1. This results in a lower start temperature and therefore be closer to the descent behavior. In figure 3 on the next page we have plotted the original run and 4 new runs with $p_0 = 0.01$ together with the Descent and Tabu Search runs. The Simulated Annealing is now clearly competitive within the given amount of runtime.

The runs are performed for a quite large instance measured in number of movements. To verify the results similar tests are carried out on a smaller instance with 753 movements. The initial solution will most likely be closer to a local optimum. The hypothesis is that the Descent Algorithm will sooner get caught in a local optimum, while the Simulated Annealing and Tabu Search will be able to escape and end up with better solutions. 5 runs are executed with a 18000 CPU seconds available for all runs in total. Figure 4 on page 163 shows the descent runs where all the runs reach a local optimum using 9538 CPU seconds in total. The average objective value is 23212. Note that the local optima are for the limited subset size mentioned earlier.

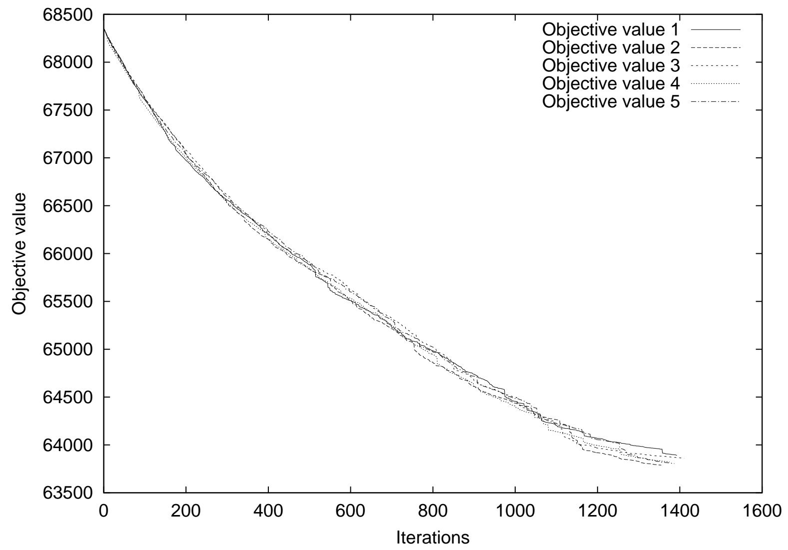

Figure 2: Steepest Descent with neighbourhood subset.

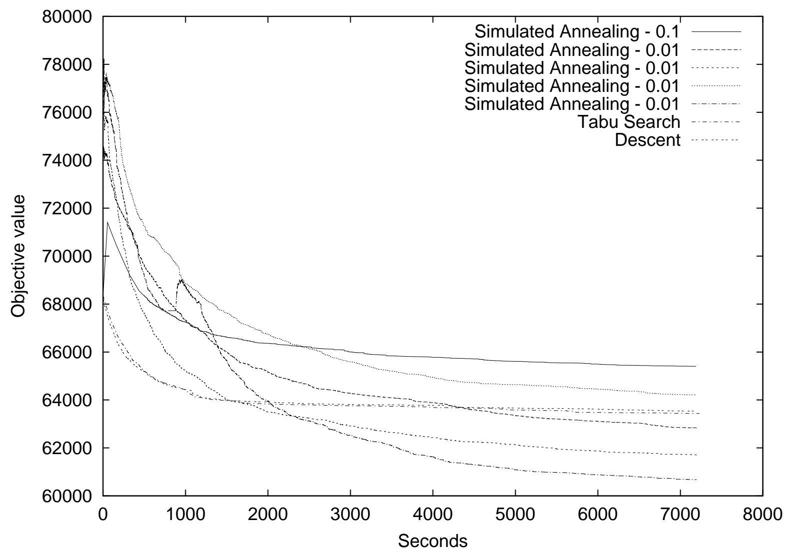

Figure 3: Simulated Annealing with $p_0 = 0.01$ compared to runs in figure 1.

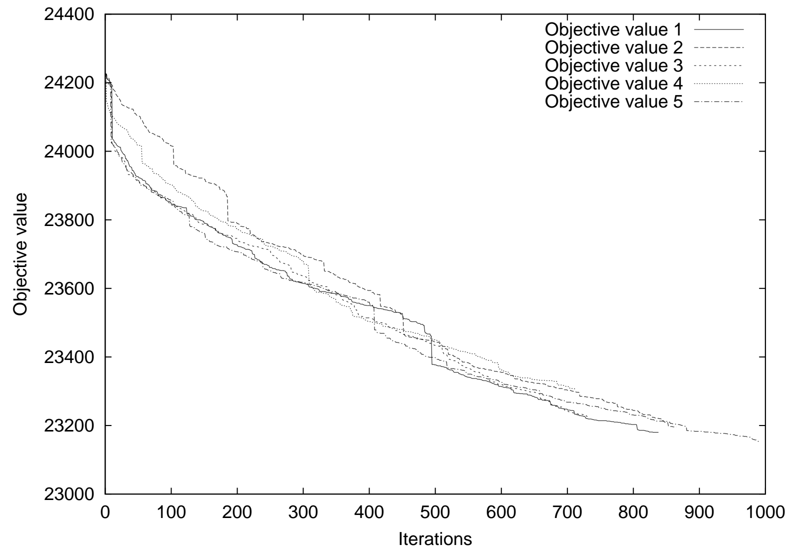

Figure 4: Descent on a smaller instance.

For the Tabu Search in figure 5 on the next page the average is 23152 using all 18000 seconds.

For the Simulated Annealing in figure 6 on the following page the average is 23826 with initial probability of accept $p_0 = 0.1$. In some cases the search gives solutions not far from the Tabu Search, but in other cases it gets completely off track. We again tried to set the accept probability to 0.01 resulting in the plot in figure 7 on page 165. The average is now down to 23653, but we still observe runs with very poor performance.

Using Simulated Annealing, all considered neighbours are evaluated exactly, so estimation can not be blamed for the (in some cases) bad performance indicated in figures 6 and 7. The plot in figure 8 on page 165 perhaps gives another hint. Here the objective values for all tried neighbours are depicted. It is clear that the "small" and "local" changes in the solution resulting in neighbour solutions have a significant and not "small" impact on the solution. This is not surprising, since we have earlier discussed the difficulty of estimating these local changes. They *do* have a global impact. Together with the notoriously slow convergence, Simulated Annealing seems inferior on this problem type.

In conclusion Tabu Search seems to have the most robust behaviour, since

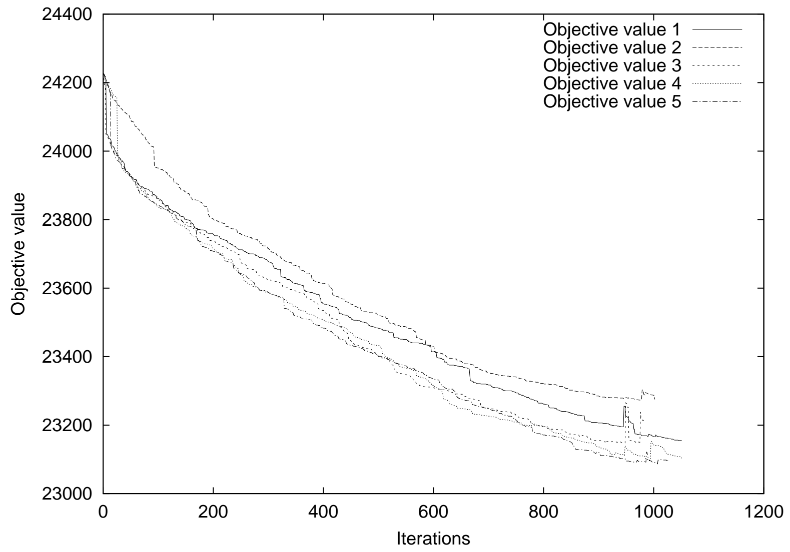

Figure 5: Tabu Search on a smaller instance.

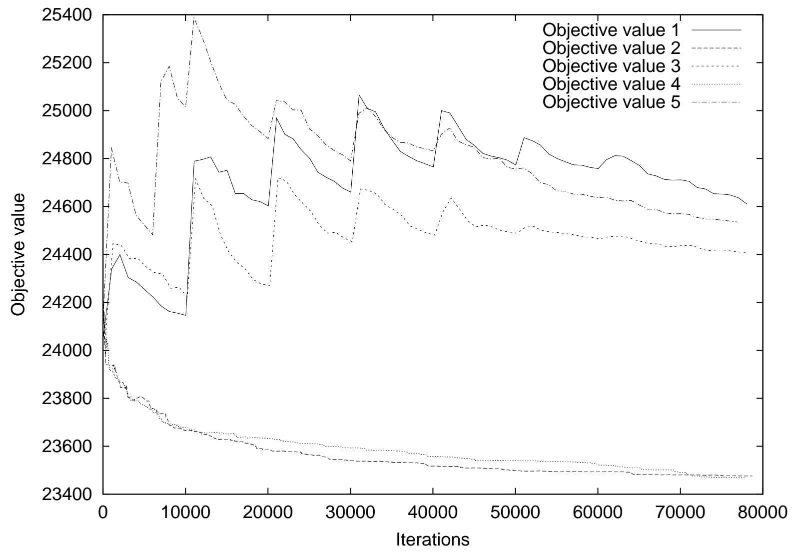

Figure 6: Simulated Annealing on a smaller instance with $p_0 = 0.1$.

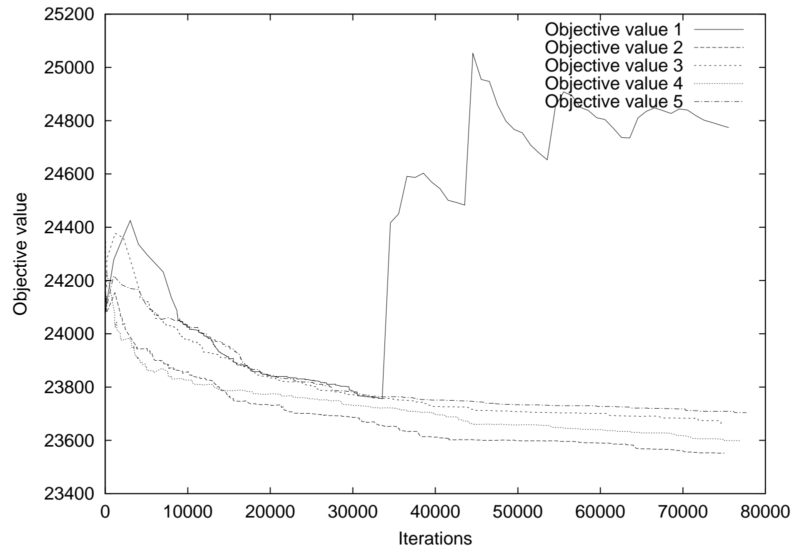

Figure 7: Simulated Annealing on a smaller instance with $p_0 = 0.01$.

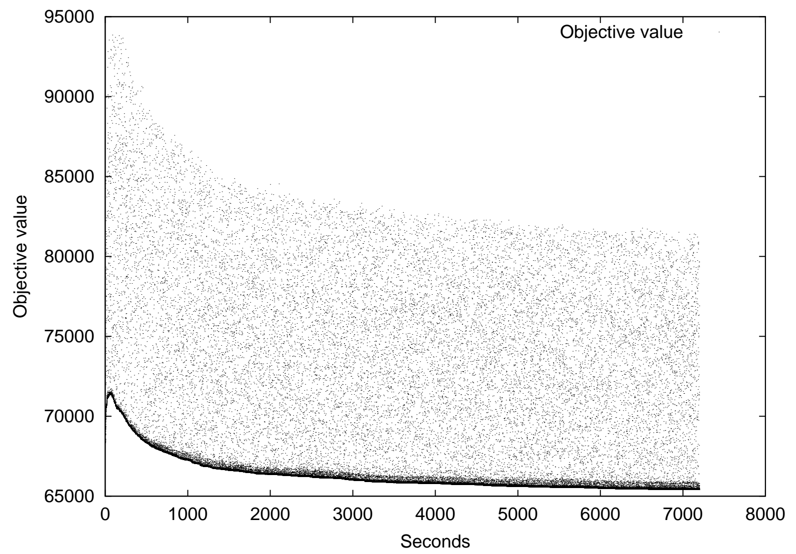

Figure 8: Simulated Annealing.

we do not observe runs with very poor performance as is the case with Simulated Annealing. If the planning time is very limited Tabu Search and the Descent Algorithm are superior, since they are much more intensive in the beginning of the search while Simulated Annealing is much less aggressive. Tabu Search is generally better than the Descent Algorithm if more time is available.

2.2 Tests for the Self-Regulating Storage Principle

We now investigate whether the results for the due date principle also applies for the self-regulating storage principle. We have limited ourselves to a comparison between Simulated Annealing and Tabu Search. For the self-regulating principle more degrees of freedom exist since all stacks are potential destinations for all but the exit-movements. Hence more iterations are needed for finding good quality solutions. 6 hours of computation time is available for both methods. Figure 9 shows the objective values for the two methods and Tabu Search is clearly superior.

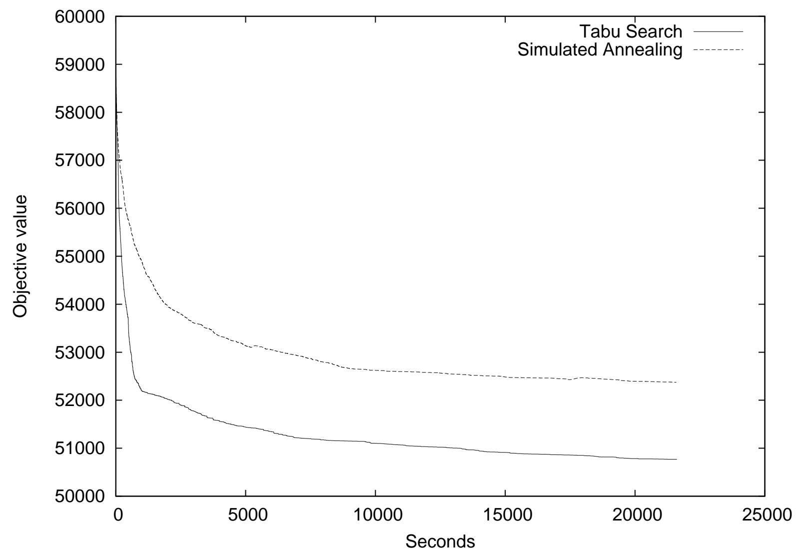

Figure 9: Objective value for the self-regulating principle for 1 day.

Running the Simulated Annealing and Tabu Search over 50 working days on the storage gives the picture shown in figure 10 on the facing page. Only the first day a better solution is achieved with Tabu Search. The remaining

days are significantly better with Simulated Annealing. This result is quite surprising since supposedly Tabu Search is better.

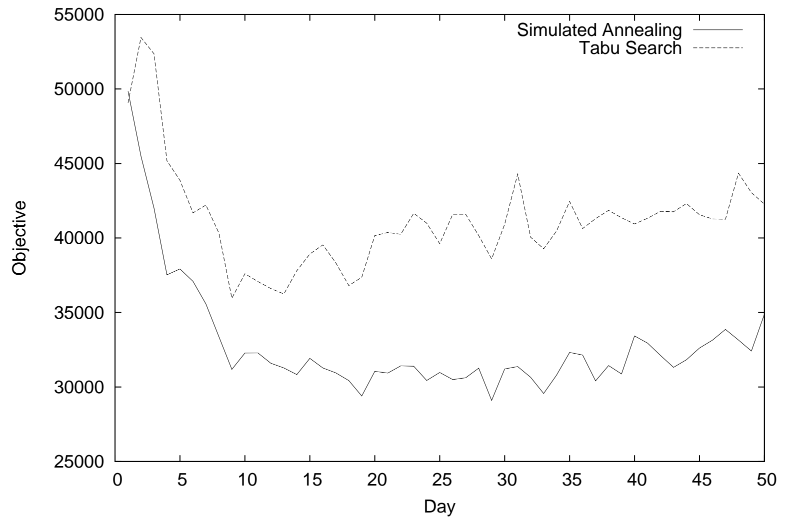

Figure 10: Objective value for the self-regulating principle over 50 days.

Recall that the objective value was better the first day with Tabu Search, we observe that Tabu Search actually creates better solutions with regard to total time (makespan) and movement time (total moved distance for both cranes in time) as indicated in figure 11 on the next page.

Investigating matters further reveals that the difference lies in the ability to leave a well-sorted storage. In figure 12 on the following page, the stack sorting is plotted for the two methods. Clearly Simulated Annealing reaches much better solutions with regard to stack sorting than Tabu Search. This is the reason that Simulated Annealing the following 50 days consistently performs better than Tabu Search. The objective function seems to be too short-sighted focusing on time spent today rather than in the future. Another possible reason is that Tabu Search is using cost estimation, which might favour time improvements over stack sorting. We performed an experiment with the stack sort cost multiplied by 10. The stack sort was naturally improved, but still worse than Simulated Annealing with the original cost and more time was spent as well. In the following experiments, we will hence use Simulated Annealing only for the self-regulating storage principle.

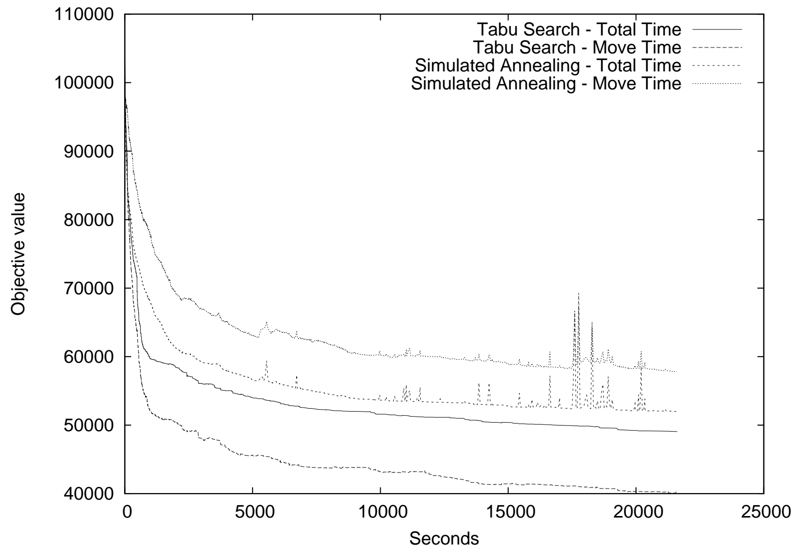

Figure 11: Time for the self-regulating principle for 1 day.

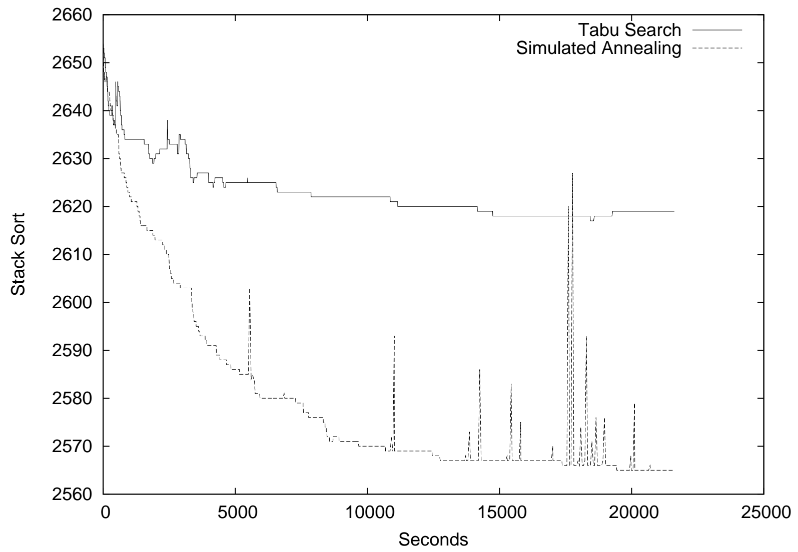

Figure 12: Stack sort for self-regulating principle for 1 day.

3 Experiments and Results

In section 2 the performance of the different local search methods were investigated. The best approach was shown to be Tabu Search for the due date principle and Simulated Annealing for the self-regulating principle. In the following we will therefore only use these combinations of methods and principles. The focus will be on comparing the on-line control algorithm and the approach of combining planning and control.

The layout of zones for the due date principle has an impact on the performance. Another issue is the periodically large peaks of work when using the due date principle. In section 3.1 we show the gain in performance when smoothing out peaks of work. The performance of the self-regulating storage is adjusted by the objective function, which is discussed in section 3.2. Finally in section 3.4 we compare the different solution approaches on the different storage principles to reach a final conclusion.

In all the experiments 121 plates arrive to the storage and leave the storage every day over 50 working days. The number of days where all plates have arrived is 10 days and the period where plates are accumulating on the storage is adjusted by the accumulation parameter, which is set to 2 (See Hansen and Kristensen [2]). This leads to a total of 3630 plates placed in 199 stacks – 18.24 on average in each stack. Every 5 days 500 plates change due date; all 500 plates have a due date from 10 to 20 days into the future. A uniformly random number of plates between 0 and 20 with due date today and tomorrow swap due date with other plates having due date no more than 20 days into the future. Disturbances on lift/drop times and exit-belt are only enabled for the experiments in section 3.4, when comparing the different control systems.

For the due date storage the plates are initially placed according to that principle, while for the self-regulating principle the plates are placed according to the block storage. This initially gives an advantage to the due date principle, but by measuring the performance as the average of the days from 20 to 50 for both storage types, the unfairness is removed. Figure 10 on page 167 confirms our hypothesis for the self-regulating storage.

For the planning module 1200 CPU seconds (20 minutes) are available for each day for the due date storage and 10800 seconds (3 hours) for the self-regulating storage. The search space for the self-regulating storage is much larger than the due date storage, which justifies the increased run times. When comparing solutions we consider only the 3 most important cost factors: The exit and total makespan and number of movements.

3.1 Due Date Storage Layouts

Finding the best storage layout for the due date principle is a cumbersome task. An obvious idea is to construct an automatic procedure to search the space of different layouts and pick the best of the visited – a meta problem of the crane scheduling problem. Considering the required runtime for such a procedure makes it next to impossible. Instead we make a less ambitious attempt simply trying a few different layouts using our intuition. Doing this will most likely not result in the best layout, but a good one is sufficient.

The first layout is a slight modification of the one originally suggested by the yard management. The parameters are shown in table 1. The yard suggested that zone 1 consists of one week of one day stacks closest to the current day. We have changed it to 8 days, since the storage has 8 rows. Zone 2 would consist of several weeks of stacks with 5 day intervals. We have again chosen 8 intervals of 5 days. The last zone would have 2 intervals of 1 month each, which in our version is 10 days, since no more due dates into the future are required to place all plates on the storage.

Parameter	Zone 1	Zone 2	Zone 3
Number of days in each interval	1	5	10
Number of due-date intervals	8	8	2
Number of stacks per interval	6	13	20
Overlap between zones	2	5	

Table 1: Originally suggested Due Date storage layout.

The original suggestion leads to huge fluctuations in the number of movements over time. This is mainly because of the quite large intervals in zone 3, but also in zone 2. At given intervals the due dates of the stacks are changed and the stacks must be emptied. The plates are typically moved to several different intervals in the next zone. If the intervals were smaller resulting in fewer plates per interval, then the due dates would change more often, but less plates had to be moved per change. This is the motivation for the alternative suggestion where the intervals in zone 2 and 3 have been reduced to 2 and 4 days and the number of intervals in zone 3 has increased to 8, shown in table 2 on the next page.

In a perfect world, 4 moves are necessary to move a plate from arrival through the 3 zones and further on to the exit belt. Clearly by reducing the number of zones the number of necessary movements would decrease. Using

Parameter	Zone 1	Zone 2	Zone 3
Number of days in each interval	1	2	4
Number of due-date intervals	8	8	8
Number of stacks per interval	6	10	8
Overlap between zones	1	2	

Table 2: Alternative Due Date storage layout

only one zone with 1 day intervals would lead to too long travel distances and crane conflicts on days where a stack far from the exit belt is emptied. Using two zones makes more sense. In our suggestion zone 2 has 16 intervals of 3 days, shown in table 3.

Parameter	Zone 1	Zone 2	Zone 3
Number of days in each interval	1	3	0
Number of due-date intervals	8	16	0
Number of stacks per interval	6	9	0
Overlap between zones	1	0	

Table 3: Due Date storage layout with 2 zones.

The result of the 3 suggested layouts are shown in table 4 where "Exit" is the makespan of exit movements and "Total" is the makespan of all movements both in hours. "MovesD" and "MovesP" are the average number of movements per day and per plate respectively. On average the original suggestion by the yard is better than our alternative. The run with two zones has a somewhat smaller number of movements per day, but needs more time on average to perform the moves. More time is spent on unproductive waiting and moving away from the other crane. Basically the cranes are more in conflict. In the following we use only the two-zone layout, since the primary objective is number of movements.

In the above runs, there were a maximum of 14.4 hours available each day for both cranes and the fluctuations in work were reduced by changing the layout. Another way of reducing the fluctuations or smoothing out the workload is simply to reduce the available amount of work time for each day. Any remaining movements, after the available time has been used, are not executed, but delayed until the next day. Figure 13 on the next page illustrates the result when smoothing out the movements by reducing the available time.

Due Date Layout	Exit	Total	MovesD	MovesP
Original	6.0	9.0	552.0	4.6
Alternative	6.0	9.3	595.7	4.9
Two zones	6.1	9.6	524.4	4.3

Table 4: Comparing different due date layouts.

The fluctuations with 10 hours are much smaller than with 14.4 hours.

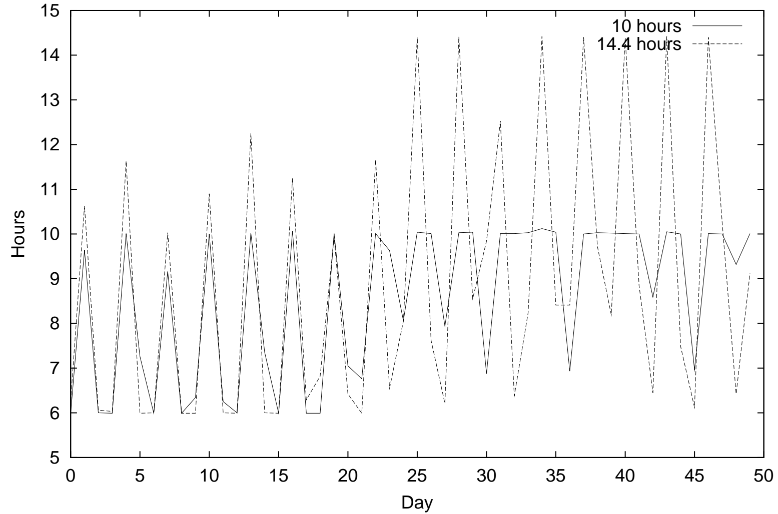

Figure 13: Total Makespan with 10 and 14.4 hours available time.

For 2 cranes we made two runs with 9, 10 and 14.4 hours and for 1 crane 14.4, 16 and 21.4 hours. 14.4 and 21.4 are 2 and 3 work shifts. The results are shown in table 5 on the facing page.

When lowering the available time for the two cranes, the average time decreases. The average number of movements is reduced as well, because movements of some plates are moved directly from the arrival stacks to zone 1 or from zone 2 directly to the exit-belt. When comparing 10 and 9 hours, the average total time still reduces, but the number of movements and exit time increases. The reason for this is illustrated in figure 14 on page 174. The remaining movements to be executed at the end of the day is plotted over

Max Time of Day	Exit	Total	MovesD	MovesP
2 cranes, 14.4 hours	6.1	9.6	524.4	4.3
2 cranes, 10.0 hours	6.1	9.3	516.2	4.3
2 cranes, 9.0 hours	6.4	8.8	530.4	4.4
1 crane, 21.4 hours	6.4	16.5	529.8	4.4
1 crane, 18.0 hours	6.3	16.9	559.0	4.6
1 crane, 16.0 hours	6.7	15.5	508.0	4.2
2 cranes, 10.0 hours no due date disturbance	6.0	8.5	363.3	3.0

Table 5: Smoothing out movements.

the 50 days. In the first half of the period there are peaks, but the remaining number of movements are regularly reduced to zero. This is not the case in the second half with only 9 hours, resulting in increasing numbers of remaining movements.

We conclude that lowering the available time can be useful for reducing the average working time, but only to a certain degree. It must be possible regularly to finish more or less all movements. In practice for instance once a week. The conclusions are however quite fragile since the due date disturbances introduce noise in the comparison. The results for 1 crane in table 5 illustrate this fact. By decreasing the available time from 21.4 to 18 hours the average time actually increase caused by additional movements, however reducing the exit time. Decreasing again to 16 hours reduces the amount of possible movements, significantly aggravating the exit time.

Table 5 also shows the result of a run without due date disturbances. We see that the theoretical limit of 3 movements per plate is actually achievable. Another way of reducing the number of movements is hence to reduce the magnitude of changes on due dates, i.e. make better or more robust plans for the entire yard.

Now let us compare the above results with the results in table 6 on the following page achieved with the control module alone. For two cranes with 10 hours, the results are approximately the same while for 14.4 hours the time is reduced by 8.5% when using Tabu Search. Generally the exit time is improved with approximately 5% as well. The number of movements are also the same, which is expected. In the case of one crane the planning module however is significantly better measured both in time and movements.

In conclusion, not much is gained for the due date storage by using more

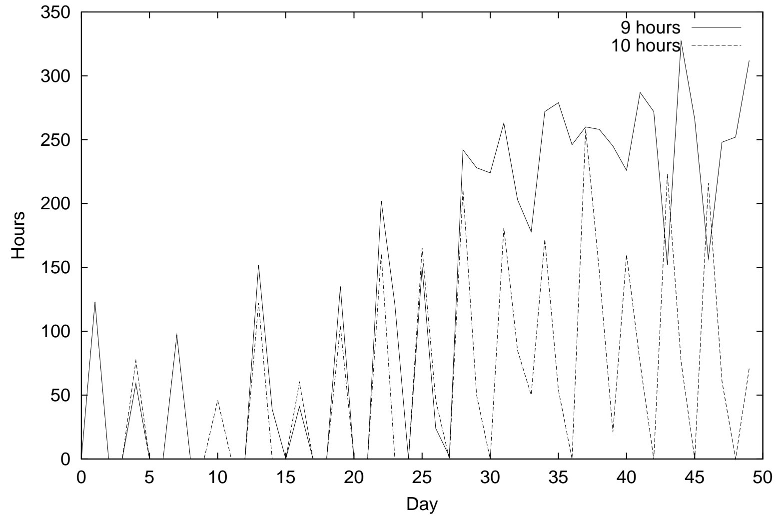

Figure 14: Remaining moves each day after 9 and 10 hours available time.

Parameters	Exit	Total	MovesD	MovesP
Control module				
2 cranes, 14.4 hours	6.3	10.5	534.3	4.4
2 cranes, 10.0 hours	6.4	9.3	518.3	4.3
1 crane, 21.4 hours	7.1	16.6	583.0	4.8
Planning module				
2 cranes, 14.4 hours	6.1	9.6	524.4	4.3
2 cranes, 10.0 hours	6.1	9.3	516.2	4.3
1 crane, 21.4 hours	6.4	16.5	529.8	4.4

Table 6: Control vs. Planning module.

complex approaches such as Tabu Search in case of two cranes. In case of one crane the conclusion is the opposite.

3.2 Adjusting the Self-Regulating Storage

The performance of the self-regulating storage is adjusted by the weights in the objective function. There are four main weights to set. The weights on exit and total time reflect the hourly wages for the crane operators and operators on the machines following directly after the storage. The machines are manned by 6 people and the exit weight is hence significantly more important than the weight on total time. The costs on lifting and dropping plates are quite high and perhaps the most important cost factor. The last cost factor is the cost of moving the cranes. It is insignificant compared to the other three factors. In the following experiment we have kept the costs on time and crane movement constant and only changed the cost on movements. In fact the parameter we are adjusting in the objective function is not directly corresponding to the number of movements, but the number of future dig-up movements. We refer to the parameter as the stack sort parameter. The default value is 7.2 and trials with 0.72 and 72.0 have been made as shown in figure 7. Only one trial with 7.2 have been made with only one crane. The experiments clearly show a dependency between the sort parameter and the average number of movements. Increasing the weight on sorting significantly reduces the number of movements, but the cranes travel longer distances and spend more time to place the plates on "good" stacks. Reducing the weight instead increases the number of movements, since the objective function favours reducing the time spend by the cranes today. Unfortunately the shortsighted focus increases the average time spend in the future since more movements are necessary.

Stack Sort		Exit	Total	MovesD	MovesP
2 cranes,	7.20	6.0	8.0	540.6	4.5
2 cranes,	0.72	6.0	8.5	675.9	5.6
2 cranes,	72.00	6.1	8.2	488.1	4.0
1 crane,	7.20	6.9	15.5	536.4	4.4

Table 7: Adjusting the self-regulating storage.

We have illustrated that it is possible for the management to prioritize between cost factors by adjusting the weights in the objective function. Note that the 1 crane solution might also be interesting since it is competitive both

in total time and number of movements. Specially of the exit time is not critical.

Comparing these results with the results so far for the due date principle, we can conclude that it is possible to reduce the total time by approx. 12% and at the same time reduce the number of movements by approx. 5%. The results are repeated in table 8.

Storage principle	Exit	Total	MovesD	MovesP
Self-regulating	6.1	8.2	488.1	4.0
Due-date	6.1	9.3	516.2	4.3

Table 8: Comparison of self-regulating and due date storage.

3.3 Comparison without Due Date Disturbances

In this section we discuss the results achievable in case of no due date disturbances. Even though this is an unrealistic case, it is still interesting also from a practical point of view. It gives an indication of the potential gains from the different storage principles. 3 experiments have been made: The due date storage with 2 cranes and the self-regulating storage with both 1 and 2 cranes. The results are shown in table 9 on the next page. In the table the “Construction” rows are the results achieved with the control module constructing the initial solution and the following row indicates the result achieved with Tabu Search or Simulated Annealing. The plates moved are in all cases only moved once, hence we see no change in number of movements. Clearly, the control module performs reasonably well on the due date storage, which we have seen earlier as well, but on the self-regulating principle significant improvements in both exit and total time is possible. In fact we do not even consider the improvement in stack sort. The last row of the table is the result when using only the control module. Hence when using Simulated Annealing, the number of movements is decreased by 35%, which is caused by the better sorting of the stacks.

Comparing the due date and self-regulating principles with 2 cranes, gives a potential gain from using the self-regulating principle with 21% in total time, but with an increase in movements of 5%.

Methods	Exit	Total	MovesD	MovesP
Due date, 2 cranes				
Construction	6.0	9.8	363.3	3.0
Tabu Search	6.0	8.5		
Self-regulating, 2 cranes				
Construction	6.4	10.5	382.0	3.2
Simulated Annealing	6.0	6.7		
Self-regulating, 1 crane				
Construction	7.1	16.2	370.0	3.1
Simulated Annealing	6.2	13.3		
Self-regulating, 2 cranes				
Control	6.7	10.3	590.6	4.9

Table 9: Without due date disturbances.

3.4 Comparison with Time Disturbances

Finally comparisons are made when both due date and time disturbances are included. In table 10 on the following page the results are shown. First the on-line control approach is used for the due date principle. Note that the time disturbances have no impact on the performance of the on-line control module. The reason is that the next decision taken by the algorithm is done after the actual time of the previous operation has been observed. This is confirmed by looking at table 6 on page 174.

The control module working on the basis of a plan is inferior to the on-line control approach for the due date storage. This is not surprising when considering the very simple control procedure deployed together with the fact that insignificant improvements are possible by using off-line planning.

Finally we consider control on the basis of a plan for the self-regulating storage. The total time is increased by 10% and the exit time by 8.3% compared to the case without time disturbances for two cranes. Even with this increase the time is still comparable with the due date principle. The exit time is similar while the total time is approximately 5% better, but with 3% more movements.

For one crane the total time and exit times are only increased by 4.5% and 1.5% respectively compared to the case without time disturbances. The reason is that the plan is not disrupted because of occurring crane collisions, but only because of time disturbances on the exit belt. Plans with only one

Method & Principle	Exit	Total	MovesD	MovesP
On-Line control				
Due date storage				
2 cranes, 14.4 hours	6.3	9.7	523.5	4.3
2 cranes, 10.0 hours	6.4	9.3	520.9	4.3
1 crane, 21.4 hours	7.3	16.2	583.2	4.8
Control with a plan				
Due date storage				
2 cranes, 14.4 hours	6.7	10.6	531.6	4.4
Control with a plan				
Self-regulating storage				
2 cranes	6.5	8.8	537.2	4.4
1 crane	7.0	16.2	535.1	4.4

Table 10: On-Line control and control with a plan.

crane is hence more robust than with two cranes. If the increase in exit time when using one crane is not critical, it could be beneficial to use only one crane operator, since two crane operators in 8.8 hours is in total 17.6 man hours, while one crane operator alone could do the work in 16.2 hours.

4 Conclusion

We have in this paper reported on a number of experiments conducted to compare different approaches and principles for improving the current practices on a steel plate storage. First Simulated Annealing and Tabu Search were compared leading to a conclusion depending on the storage principle at hand. Tabu Search was seemingly superior to Simulated Annealing, but the Tabu Search used an estimation of the cost function, which was not sufficiently precise for the self-regulating principle. It was hence optimizing in the "wrong" direction leading to solutions, which were inferior in the long run compared to Simulated Annealing.

Then we turned our attention to comparing the different storage principles, storage layouts and objective weights. The experiments indicate that significant improvements can be achieved. No final conclusion was reached on best layouts or weights. These decisions should be taken by the management and users of the system reflecting the overall goals of the organization.

To conclude, the on-line control approach is more robust to time distur-

bances and is performance-wise superior to control based on a plan for the due date principle.

The self-regulating principle on the other hand is dependent on improving the plan initially achieved with the on-line control approach. Better solutions time-wise can be achieved with the self-regulating storage compared to the due date storage, but the saving of 12% is reduced to 5% when introducing time uncertainties. Approximately the same number of movements is required. The user of the software can however adjust the behavior of the self-regulating storage to focus more on reducing the number of movements, reducing the number of movements down to 4 per plate. Compared to current practices this is a reduction by 67% and at the same time a reduction in time by 39% leading to a saving of approx. 1.0 mill. dkr. per year.

The development of the system and these investigations should not end here. Significant improvements can be made to all modules: The on-line control module, control module based on a plan and the planning module itself. One must however take into consideration whether the more complex approach of control based on a plan is sufficiently more promising than the more simple on-line control approach, which might be sufficient for the purposes of the yard.

References

[1] J. Hansen and J. Clausen, *Crane scheduling for a plate storage*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-1 (2002).

[2] J. Hansen and T. F. H. Kristensen, *Crane scheduling for a plate storage in a shipyard: Modelling the problem*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-4 (2003).

[3] ——, *Crane scheduling for a plate storage in a shipyard: Solving the problem*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-11 (2003).

Paper IV

Crane scheduling for a Plate Storage

Available as IMM Technical Report 2002-1.

Crane scheduling for a Plate Storage

Jesper Hansen[1] and Jens Clausen[1]

Abstract

Odense Steel Shipyard produces the worlds largest container ships. The first process of producing the steel ships is handling arrival and storage of steel plates until they are needed in production.
This paper considers the problem of scheduling two cranes that carry out the movements of plates into, around and out of the storage. The system is required to create a daily schedule for the cranes, but also handle possible disruptions during the execution of the plan. The problem is solved with a Simulated Annealing algorithm.

Keywords: Crane scheduling, plate storage

1 Introduction

We first describe the plate storage at Odense Steel Shipyard and the resulting planning problem in section 2. This is compared to work on related problems in section 3. Afterwards we introduce the complex objective function of this problem in section 4 and our heuristic for solving the problem will be explained in section 5. We end with a section on achieved results and a conclusion.

2 The Problem

The plate storage is organized in 8 times 32 stacks. Approximately 20 plates are stored in each stack on average. $\frac{1}{4}$ of the stacks are used for special purpose plates for instance stacks with identical plates. The rest of the storage are used for plates which for the main part have different sizes. Each plate is ordered for a specific purpose and it is known at which date it is needed in production. Changes in the overall plan for the yard, can however influence the due dates of the plates on the storage, which means that the due date of a plate can change several times while it is in storage.

Two gantry cranes carry out the movements of plates. The cranes share tracks and hence can not pass each other. For each of the 32 rows the cranes

[1]Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark

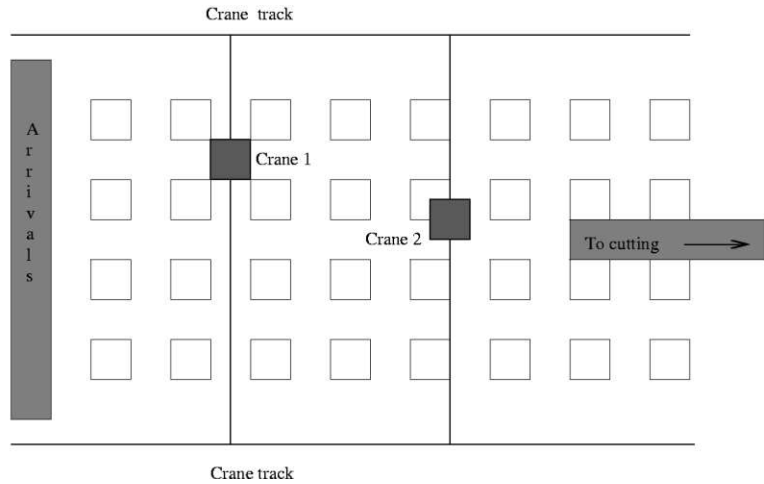

Figure 1: Illustration of the storage layout.

can reach each of the 8 stacks. The cranes are equipped with an electromagnet which can lift only the top plate from a stack and move it to some other stack or remove it from the storage. Since the cranes are gantry cranes there is a limit on the height of the stacks.

The plates arrive by ship in bulks and are put in 8 arrival stacks by a dedicated crane and are then moved into the storage by the gantry cranes.

Each day a set of plates must leave the storage to be cut up. The order in which the plates are leaving is of no importance. The plates are put on a conveyer belt with a capacity of 8 plates, which is called the exit-belt. The exit-belt works as a queue and plates are drawn from the queue with time interval λ. We assume in this paper that λ is a constant, but in reality it is a function of the dimensions and order of the plates put on the exit-belt. An illustration of the storage can be seen in figure 1.

If a specific plate is requested, which is not on the top of a stack, the plates on top must be moved to other stacks before the requested plate can be removed from the storage. This procedure introduces a number of *unproductive* movements, which we call *dig-up movements*.

The planning task is now to create a schedule of movements for the 2 cranes without collision, that delivers the plates needed for the given day and minimizes costs.

3 Related Research

A lot of different problem classes within scheduling and routing have been studied in the litterature. Many of them have similarities with the problem considered in this paper. One problem is the Travelling Salesman Problem (TSP) where a salesman must visit a number customers or rather nodes in a graph. For the Stacker-Crane Problem (SCP) a crane must move items from a node to another node in other words visit a subset of arcs in a graph. In our problem the items are steel plates. If the problem includes M cranes or salesman we call it M-SCP or M-TSP. The SCP was introduced by Frederickson et. al. in [1].

If k plates is to be moved from a stack s_o to other stacks it means that k arcs from s_o must be visited, but in an order such that the arc corresponding to the top plate must be visited before the arc corresponding to the next plate and so on. I.e. We have precedence relations between the arcs leaving s_o. Many scheduling and routing applications have precedence relations for instance pickup and delivery of items. If more items can be picked up before delivering, the problem is in the category of Vehicle Routing Problems (VRP). For instance if customers in the M-TSP problems have a specified demand and the salesmen or vehicles have a limited capacity then the problem is called the VRP.

In our application a plate may be moved more than once if it is put on a stack from which plates are going to be moved later on. This means that extra arcs that must be visited by a crane are dynamically inserted in the graph. Insertion of extra arcs are depending on the order in which the arcs are traversed and we have precedence relations between arcs corresponding to movements of the same plate.

Another complicating factor is the the destinations for the dig-up movements are decision variables. In the graph setting it means that the head of an arc from node s corresponding to a dig-up movement from stack s is not given, but is part of the optimization problem.

In our application the two gantry cranes share tracks and in our schedule we must avoid collisions of the two cranes. Problems of scheduling cranes and robots have many applications and some work have been done in this area. Fujita et. al. [2] are solving a very similar problem of scheduling overhead cranes transporting coils. Differences in the models are mainly how the robots or cranes move. For instance in the Hoist Scheduling Problem (HSP) the hoists are moving on a line and cannot cross each other on that line. Gantry cranes share tracks and cannot cross each other as well and the gantry normally only

have one hoist, but could have more that as well cannot cross each other. In order to avoid collision of the cranes extra *positioning* and *wait movements* may have to be inserted in the sequence of movements. In other cases crane or robot arms can only move around its centre where they share some common space where collision of the arms must be avoided. See Chung-Yee et. al. [3] for an overview of current trends in scheduling.

4 Objective Function

Costs include salary to the crane operators, power for the cranes to move and activation of the electro-magnetics as well as maintenance on the cranes. In other words, make-span of the schedule, total traveled distance and number of movements. Minimization of the time when the last plate is put on the exit-belt is also important to be able to start the following processes as early as possible. One days production is planned at a time, so in order to minimize the long term costs, we measure the quality of the *state* of the storage after each day. The state is evaluated by 3 different criteria:

1. How well are the stacks sorted by plate due date? More specifically, how many dig-ups have to be done for each stack with the given plate due dates. For stack (2,4) in figure 2 on the next page plate p_2 is the first to be removed from the storage on the November 1. This requires the dig-up of p_1. The next plate to be removed from the stack is p_4, which causes the dig-up of p_3. The result is 2 dig-ups.

   We have defined the cost of a digup to be the estimated cost of power used for lifting and dropping a plate plus the cost of power for moving the crane to the nearest stack and back plus the cost in salary for operating the crane in the amount of time according to the digup movement.

2. How close are the plates to the exit-belt compared to the due date of the plates? When minimizing traveling distance it is better always to move plates in direction of the exit-belt and when the due date of a plate is close in time, we want it to be close to the exit-belt. More formally the cost for a given plate is:

   $$\frac{\textit{distance to exitbelt}}{\mathrm{diff}(\textit{plate due date},\ \textit{today}) +\ 1}$$

   where $\mathrm{diff}(\textit{date}_1, \textit{date}_2)$ returns the difference in days between two dates. Let us assume that the exit-belt is located at the coordinates (1,4), then

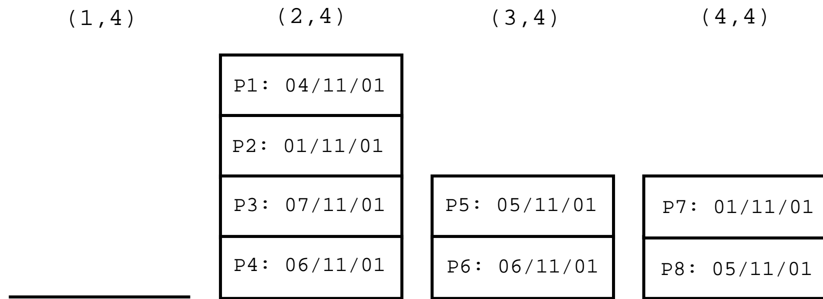

Figure 2: Example of plates in stacks.

stack (2,4) is 1 closer than (3,4). If for instance today is 01/11/2001 then the cost for plate p_1 is $\frac{1}{4}$, for p_4: $\frac{1}{6}$ and p_6 it is $\frac{1}{3}$.

The above exit distance can be converted to time and the cost is then achieved by multiplying with the salary of the operators.

3. How many plates are there in each stack ? We want the plates to be as evenly spread out on the stacks as possible, which will result in less dig-ups when changes in due dates occur. Generally the cost for a stack is:

$$\frac{(no.\ of\ plates\ in\ stack)^2}{max.\ no.\ of\ plates\ in\ stack}$$

where the maximum no. of plates in a stack is a user supplied upper limit on the number of plates common for all stacks. We observe that the cost per stack will be in the interval from 0 to the maximum no. of plates in the stacks. If we have an upper limit of 10 plates, then the stacks in figure 2 will have a cost of $\frac{16+4+4}{10} = 2.4$ while if p_1 was moved to stack (3,4), the cost would be $\frac{9+9+4}{10} = 2.2$, and hence better.

What we are minimizing is in some sence the worst case number of digups if a plate in the bottom of a stack is requested. The cost of a digup in 1 is therefore used here as well.

We have managed to convert all criteria functions into a common unit, which is then minimized. All the user has to supply are the operating costs of the crane and the salary of the operators.

5 Solution Procedure

Each day we know which plates must leave the storage. These movements of plates out of the storage are called *exit movements*. Given these movements we can identify plates that have to be dug up during that day. Note that these plates can be identified before scheduling any movements.

When plates arrive by ship they must be moved into the storage. Given a day with arrival of plates we assume that all plates are put in 8 stacks outside the storage. This means that we can generate the *arrival movements* before scheduling just as for exit movements. Note that it has not yet been decided to which stacks the plates will be moved, but only that they *will* be moved.

In order to improve the state of the storage, we may perform some sorting of the stacks. Several different sorting strategies may be applied, for instance:

1. Generate movements for all plates in the stack, if it is not sorted. This will result in movement of all the plates in stack (2,4) in figure 2 and none from stack (3,4) and (4,4).

2. Generate movements for plates from the top of the stack until the stack is sorted: The plates p_1, p_2 and p_3 will be moved.

3. Find the plate with minimum due date - in case of ties choose that closest to the bottom . Generate movements for all plates above this plate with later due date: p_1 will be moved.

4. As 3., but now we only generate movements, if the minimum due date is no later than δ days from today. If for instance δ is 1 and today is the 31/10/2001 then p_1 will be moved. If today is before the 31. October then none will be moved.

5. Generate movements for all plates in the stack, if the stack sort cost incurred by the state of the stack is larger than some given value.

We call these movements for *sort movements*. The reason for doing sorting is that in the future less dig-ups will be necessary, and less time will be spent removing plates from the storage. If the due dates of the plates change frequently there is a risk of doing a lot of sorting that later turn out to be wasted. Experiments indicate that strategy 4 with $\delta = 1$ is a good choice, since no dig-ups will be necessary the day after if the due dates are not changed.

We need to be a bit more specific about the term *movement*: A movement is specified by a plate, origin and destination stacks and a crane. In other

words it is a specific crane moving from its current position to the origin stack to pick up the plate, afterwards moving to the destination stack, where the plate is dropped. Below we see an example of a movement of plate p from (x, y) to (x', y') as a sequence of *operations* for a crane:

1. Move crane from current position to (x, y).
2. Lift plate at current position.
3. Move crane from current position to (x', y').
4. Drop plate at current position.

In some cases it will be necessary for a crane to move away from the other crane to avoid collision or wait for the other crane to finish some operation. In that case the above described movement will be split up into several distinct movements and special types of movements called *position* and *wait* movements will be inserted. Below, we give an example:

1. Move crane from current position to (x, y).
2. Lift plate at current position.
3. Move crane from current position to (x'', y'') to avoid collision.
4. Wait t seconds.
5. Move crane from current position to (x', y').
6. Drop plate at current position.

The above sequence is actually divided into 3 movements:

1. Move plate p from (x, y) to (x'', y'').
2. Wait t' seconds.
3. Move plate p from (x'', y'') to (x', y').

Here the crane should not drop and again lift the plate at position (x'', y'').

Earlier we described how to identify plates that have to be moved on a given day. Given the precedence relations that a plate on top of another has to be moved first, we have a partially ordered set of movements. Assume that we in some way have divided the movements into two sets (one for each crane),

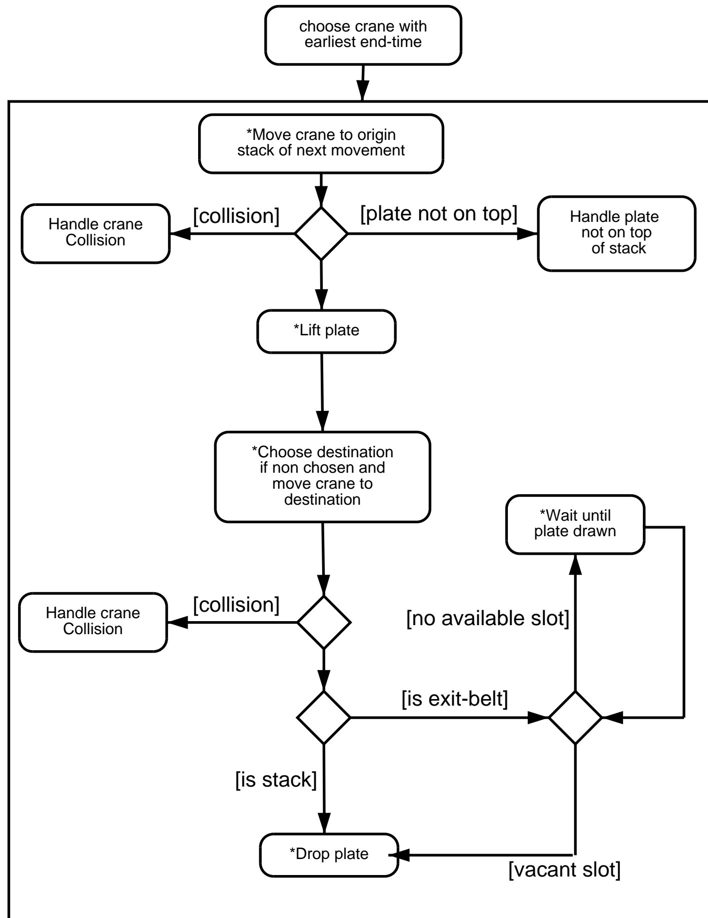

Figure 3: Simulation of a plate movement.

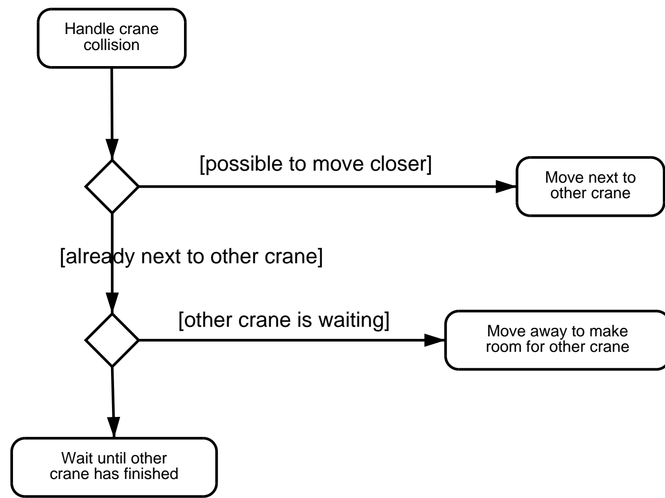

Figure 4: Simulation of a plate movement.

and found an ordering of the movements in each set which fulfills the precedence relations. In order to determine start- and end-times of the movements, we simply simulate the movements while inserting wait, positioning and extra dig-up movements as necessary.

In figure 3 on the facing page we see an activity diagram of the process of simulating a movement for a crane. The cranes work in parallel, so the simulator must keep track of which crane first finishes its operation and then determine the next operation for that particular crane. When entering the box the first activity can be any activity with a star (*) depending on the previous operation done by the crane. If for example the last operation was lift then the next is to go to the destination of the movement. In choice situations, arrows with no legend cover those situations not described by any other legend.

If a collision of the cranes would occur we handle this as depicted in figure 4. If possible the crane moves closer to the other crane, which will always be closer to the destination of the crane. If the crane is already next to the other crane we either wait or move away for the other crane to finish its operation.

The case when the requested plate is not on top of the stack is shown in figure 5 on the following page. A dig-up of the top plate is done by the crane, unless the current movement of the other crane is to dig-up the top plate. In that case we wait for the other crane to take the plate.

If the plate is not in the stack it means that it has not yet arrived from

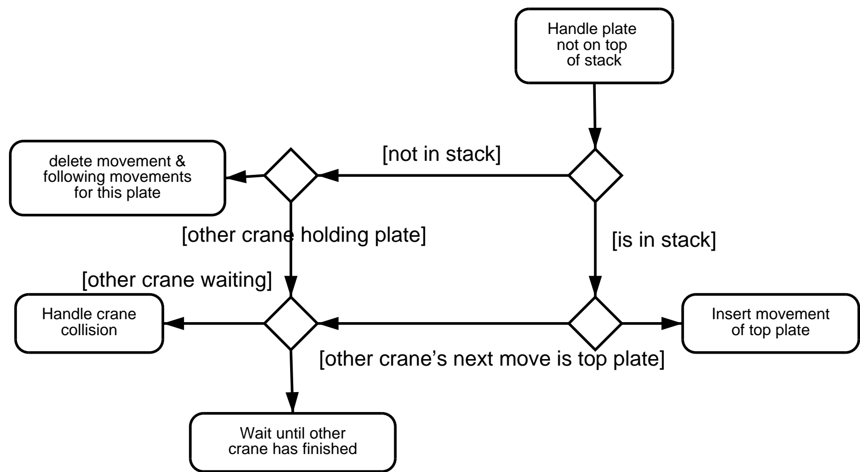

Figure 5: Simulation of a plate movement.

some other stack. If the current movement of the other crane it to move the plate to the stack, then we wait for the other crane to finish. Otherwise we remove the movement from the sequence as well as any following movements of that plate if any exists. Note that the movement cannot be an exit-movement, since plates involved in exit movements are always present in the origin stack of the movement.

For movements other than exit-movements we have to decide to which stack to move the plate. This is done just after lifting the plate. All stacks are evaluated with regard to several criteria:

1. Preferably we want to put the plate on a stack where the plate with the minimum due date has a due date that is later than or equal the due date of the plate we are moving. In that case the stack sorting will not deteriorate. On the other hand we want the difference to the minimum due date in the stack to be as small as possible, because this will make it easier to find suitable stacks for other plates.

   For example if we in our sequence are moving plate p_7 followed by p_8 in figure 2 on page 187, we have to decide to which stack of (2,4) and (3,4) to move p_7. The plate p_7 will not deteriorate the sorting of either stacks in (2,4) or (3,4), but it is still better to move it to (2,4): If p_7 is moved to (3,4), then moving p_8 will cause an extra dig-up at a later stage.

2. Minimizing traveled distance to the stack and further on to the source

stack of the next movement. For this criteria (3,4) is better than (2,4), since the crane afterwards will move back to (4,4) to pickup p_8.

3. Minimizing stack height. Again (3,4) is better than (2,4) according to the stack height criteria on page 3.

4. Minimizing distance to the exit-belt. Here (2,4) is better than (3,4) since stack (2,4) is closer to the exit-belt placed in (1,4).

The weighting of these criteria is the same as for the objective function. A random stack is chosen from the best set of stacks.

The simulation ends when all movements have been executed. The schedule can now be evaluated according to the criteria mentioned earlier.

As earlier noted the problem is solved over a period of time, but one day at a time. We have used a standard Simulated Annealing (SA) algorithm as in [4]. The reason for using a SA compared to for instance a Genetic Algorithm or a Tabu Search is mainly that evaluating neighbour solutions is quite expensive, since it involves simulating the crane sequences every time.

Given our solution of two sequences of movements, we only need to specify a way to move to a new neighbourhood solution. Our neighbourhood structure is given by 3 simple *operators*:

- Delete a specified destination for a movement and any following movements of that plate. A new destination will then be picked when simulating.

- Reinsert a movement from a crane sequence at another position in the same sequence.

- Delete a movement from one crane sequence and insert it in the sequence of the other crane.

We only allow neighbour solutions where the precedence relations are fulfilled. For operator 2 we have two cases:

1. If the predecessor/successor are in the same sequence, we just check if the predecessor is earlier in the sequence than the successor.

2. If the predecessor/successor are in separate sequences, the solution will always be legal, since the crane doing the succseeding movement just waits until the other crane has finished the preceeding movement. This can obviously result in poor solutions. Therefore we try to estimate the new start and end times of the movements and if waiting is introduced we reject the neighbour solution.

For operator 3 the two cases are the same, but turned up side down.

6 Results

At present the storage at Odense Steel Shipyard is managed in a block-oriented fashion. A block is a large section of the ship and all the plates that are needed for producing this section is placed in the same stacks on the storage. A block is produced over a period of time, and the plates are therefore also due over a period of time. Hence searching for the plates involves a huge number of dig-ups every day. For comparison we have implemented the block-oriented approach as well as the approach presented in this paper.

For the test scenario presented in this paper the storage is 24×8 stacks with approximately 3600 plates of which 121 are due each day. Plates arrive every 3 days. The scenario runs over 30 days and initially the plates are placed according to the block-oriented approach. Several other test scenarios have been tested, but the results are similar to the one shown here. In figure 6 on the facing page "block" indicates the block approach and "selfreg" the approach described in this paper. "dist" indicates that *disturbances* or disruptions in the plate due dates has been introduced. In our scenario 0-10 exit plates swap due dates with other plates on the storage each day and every 5 days 500 plates shift due dates. Generally the new approach is much better than the old. Looking at number of movements the conclusion is even more clear as shown in figure 7 on page 196. On average the improvement in number of movements is around 50% and in make-span around 40%.

The time to plan one days production is approximately two hours in the prototype implementation of the method.

In table 1 on the facing page results are shown when increasing the number of exit-movements to leave the storage each day from 121 to 169 and 225. The times are in hours per day in average over 30 days. With the size of the storage not changing you would expect the amount of time spend and total number of movements to increase faster than linear since more plates must be placed in each stack. This is also the case since a plate must be moved 4.57 times on avarage for 121 plates leaving the storage, 5.29 times for 169 plates and 5.88 for 225 plates. This must be compared to the 11.12 movements on average for the block principle. All numbers are found for the case with due date disturbances.

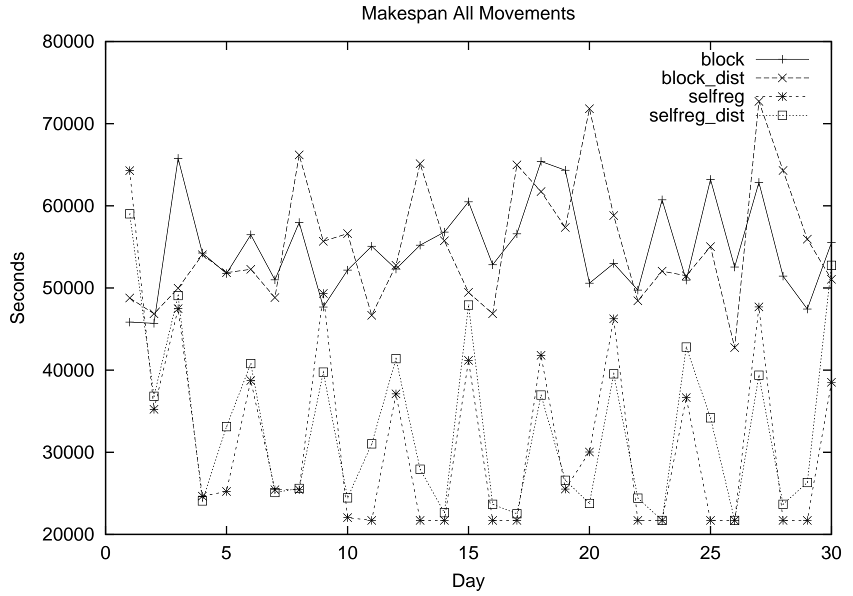

Figure 6: Makespan for block and new appoach.

Principle	Exit	Dist.	Exit Time	Total Time	Movements
Block	121	No	6.03	15.24	1332
Block	121	Yes	6.56	15.34	1346
Selfreg	121	No	6.06	8.72	441
Selfreg	121	Yes	6.24	9.15	553
Selfreg	169	No	8.61	13.97	748
Selfreg	169	Yes	8.92	14.41	894
Selfreg	225	No	12.13	20.87	1250
Selfreg	225	Yes	12.47	20.27	1322

Table 1: Results when increasing number of exit-movements.

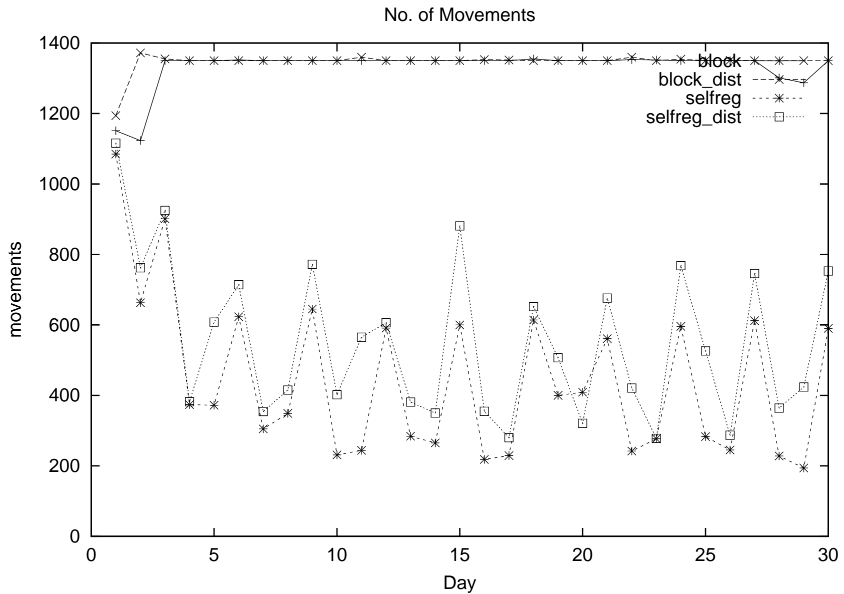

Figure 7: No. of movements for block and new appoach.

7 Conclusion

We have considered a hard real life application. Our pilot results are extremely promising, but we believe that even better solutions can be achieved by refining the heuristics.

Currently research focuses on enhancing the planning software with facilities to handle disruptions dynamically for due dates, break down or changes in operation times of cranes and exit-belt. Here efficiency of the computations has crucial importance, and hence another important line of work is to reduce the running time of the method.

References

[1] G. N. Fredrickson, M. S. Hecht, and C. E. Kim, *Approximation algorithms for some routing problems*, SIAMM Journal of Computing, 7 (1978), pp. 178–193.

[2] F. Fujita, M. Okada, and T. Kaita, *Automatic scheduling system for an overhead multi-crane yard*, in IFAC Control Science and Technology 8th Triennial World Congress, 1981.

[3] C.-Y. Lee, L. Lei, and M. Pinedo, *Current trends in deterministic scheduling*, Annals of Operations Research, 70 (1997), pp. 1–41.

[4] P. J. M. van Laarhoven, *Theoretical and computational aspects of simulated annealing*, tech. rep., Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1988.

Paper V

Solving the non-oriented three-dimensional bin packing problem with stability and load bearing constraints

Solving the non-oriented three-dimensional bin packing problem with stability and load bearing constraints

Jesper Hansen[1]

Abstract

The three-dimensional bin packing problem is concerned with packing a given set of rectangular items into rectangular bins. We are interested in solving real-life problems where rotations of items are allowed and the packings must be packable and stable. Load bearing of items is taken into account as well. An on-line heuristic and an exact method have been developed and compared on real-life instances and as well on some benchmark instances. The on-line algorithm consistently reaches good solutions within a few seconds. The exact method is able to improve the solutions, but a significant amount of computation time is required.

Keywords: 3-dimensional packing, bin packing

1 Introduction

The three-dimensional bin packing problem (3D-BPP) is concerned with orthogonally packing a given set of rectangular items in rectangular bins. The bins can be identical or of different sizes. If they are identical we want to minimize the number of bins used; in case we can choose from different sizes, we want to minimize the cost of the used set of bins. We assume that it is possible to fit each item in at least one of the bin types.

We are interested in solving real-life problems, and hence additional constraints must be taken into account. It is generally allowed to rotate the items in the 6 different possible orientations, but restrictions on some items can occur, allowing only a subset of the orientations. The packing of items must be stable and possible to pack by a person or a packing robot. In order to guarantee this we will restrict ourselves to so-called robot-packable packings a concept introduced in den Boef et.al. [5] and Martello et. al. [11]. Basically items are placed starting from the bottom-left-back corner of the bin and successively placing items in front, on top or right of already placed items. The robot arm will in that way avoid collision with already placed items.

[1]Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark

Dyckhoff [6] introduced a typology of cutting and packing problems denoting the 3D-BPP as 3/V/D/M. V and D denote problems where all items must be packed in possibly different bins and M that we consider instances of many items of different sizes. In principles of optimization there is no difference between cutting, nesting and packing problems. Note however that usually all items are considered different in Bin Packing, but we also consider the case of many items of a few different types denoted 3/V/D/R.

Lodi et. al. introduce a more detailed classification scheme focused on bin packing problems. They classify problems in the following dimensions:

- Dimension of the problem: 1,2,3,..
- Orientation of items: fixed orientation (O) or rotation allowed (R).
- Cutting restrictions on packings: guillotine cutting (G) or free cutting (F)

In our version of the problem we only allow robot-packable packings, which we denote (R) and the problem class is then 3BP|R|R.

The first attempt on solving multi-dimensional bin packing or cutting stock problems dates back to Gilmore and Gomory [8]. They were using column generation, but they restricted themselves to guillotine cuts. Chen et.al. [3] formulates an Integer Programming Model, but it is only solvable for very small instances. The first exact algorithm solving the 3D-BPP was proposed by Martello et.al. [11], but they were not considering rotation, stability and load bearing constraints. They used what could be considered a direct Branch & Bound approach branching on items in bins and afterwards position of the items in the bins.

A large number of papers have been published on heuristic approaches for the 3D-BPP. Ivancic et.al. [10] propose an integer programming based heuristic approach, which basically is a column generation approach without branching. Feasible solutions are merely achieved by solving the integer programming problem over the generated packings. The packings are generated with a greedy construction heuristic taking the dual prices into consideration. More recent is the use of Guided Local Search by Faroe et.al. [7].

We are interested in solving another variant of the problem introduced earlier – an on-line version where the order of the items arriving at the packing site is unknown. The items are in a queue, where we can observe and pick an item to pack from the first Q items. At the packing site S bins are available to pack at a time. When no more items can be packed in the bins one or more

of the bins must be shipped off and replaced by one or more empty bin(s). To solve the on-line version we use the greedy heuristic described in section 6.

We use another approach for the problem where all data is known a priori. We also adopt the column generation approach. The pricing problem is a 3D Knapsack Problem, which we decompose as suggested by Pisinger and Sigurd [12] for solving the 2D Bin Packing Problem. The 3D Knapsack Problem is decomposed into a 1D Knapsack Optimization Problem and a 3D Knapsack Feasibility Problem. The 1D Knapsack Problem is a relaxation of the 3D Knapsack Problem. Pisinger and Sigurd use Constraint Programming to solve their 2D feasibility problem and at failures they add cuts to the 1D Knapsack Problem in a Branch & Cut fashion. We use an algorithm similar to the ONEBIN packing algorithm of Martello et.al [11] to solve the feasibility problem.

Figure 1 gives an overview of the entire column generation scheme. Duals from the Restricted Master Problem is sent to the 1D Knapsack relaxation. The optimal solutions are checked for 3D Knapsack feasibility. Either a cut is sent to the 1D Knapsack or a feasible packing is sent to the Restricted Master Problem. This procedure continues until no feasible packing exists, which will improve the LP solution of the Restricted Master Problem.

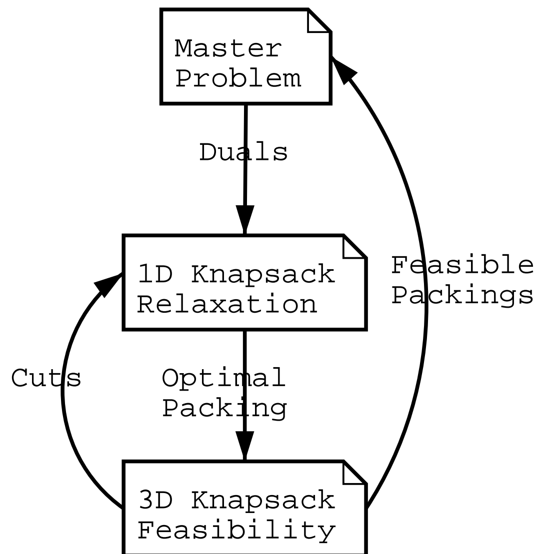

Figure 1: Overview of the solution approach.

The rest of the paper is organized as follows: In section 2 the column generation approach is described. The pricing routine is the topic of section 3. In section 4 we discuss branching schemes for column generation and in section 5 we give some lower bounds useful for pruning the Branch & Bound tree. In sections 7 and 8 we consider stability issues and load bearing constraints on the packed items. Finally in section 9 we compare the on-line algorithm described in section 6 with the exact approach on both real-life instances and different benchmark instances of varying difficulty.

2 Column Generation

Assume that we have generated all feasible packings of a single bin. The problem of selecting the best subset of packings covering all items can then be formulated as an integer programming problem in the following form where $\mathcal{P}$ is the set of generated packings:

$$\min \sum_{p \in \mathcal{P}} c_p q_p$$

$$\sum_{p \in \mathcal{P}} a_p^t q_p = d_t, \quad t \in \mathcal{T} \tag{1}$$

$$q_p \geq 0 \text{ and integer}, \quad p \in \mathcal{P}$$

c_p is the cost of packing p depending on the cost of the used bin type, a_p^t is the number of items of type t in packing p and d_t is the number of items of type t to be packed. The problem can also be formulated with $a_p^t \in \{0,1\}$ and $d_t = 1$ by regarding identical items as different items. The size of $\mathcal{P}$ is exponential in the number of items, which is why we solve the problem with column generation. The integrality constraint is relaxed in (1) and we solve over a smaller set $\mathcal{P}' \subset \mathcal{P}$ of packings. The result is a lower bound for (1) and if an integral solution is found it is optimal. This problem is called the Restricted Master Problem. Initially we use a small set of packings for instance generated from a greedy construction heuristic ensuring a feasible solution. In following iterations we add columns improving the current solution. Let π_t be the dual variables from (1). An improving packing will now have a reduced cost, $\overline{c}_p = c_p - \sum_{t \in \mathcal{T}} a_p^t \pi_t < 0$. If for all $p \in \mathcal{P}$ $\overline{c}_p \geq 0$, then no improving packing exist and the current solution is optimal. After adding columns the problem is resolved giving new dual variables etc. The problem of finding columns with negative reduced costs is called the subproblem or pricing problem.

3 Pricing Problem

The pricing problem is a 3D Knapsack Problem. As Pisinger and Sigurd [12], we have chosen to decompose the problem into a 1D Knapsack Optimization Problem and a 3D Knapsack Feasibility Problem. The 1D problem is a relaxation of the 3D problem, which means that the solution value will be a lower bound on the optimal value.

There is a pricing problem for each bin type. Bin types are characterized by a length L_b, width W_b, height H_b, volume $V_b = L_b W_b H_b$, weight limit E_b and cost c_b for each b of the types in $\mathcal{B}$. A set of different item types $t \in \mathcal{T}$ are to be packed in bins. Each type is defined by the length l_t, width w_t, height h_t, volume $v_t = l_t w_t h_t$, possible rotations $R_t \subseteq \{1,2,3,4,5,6\}$, weight e_t and the number of items of this type to be packed d_t.

We can now formulate the problem as a Knapsack Problem for a bin type $b \in \mathcal{B}$, where the variable u_t denote the number of items of type t in the knapsack:

$$\eta_b = \min \; c_b - \sum_{t \in \mathcal{T}} \pi_t u_t$$

$$\sum_{t \in \mathcal{T}} v_t u_t \leq V_b, \tag{2}$$

$$\sum_{t \in \mathcal{T}} e_t u_t \leq E_b,$$

$$0 \leq u_t \leq d_t \text{ and integer}, \quad t \in \mathcal{T}$$

If $\eta_b < 0$ then the packing will improve the solution to the restricted master problem. Note that the problem has two knapsack constraints while the Knapsack Problem only have one. The two constraints do however not increase the dimension number of the problem.

3.1 3D Knapsack Feasibility Problem

After achieving the solution $u^\star$ to (2), it must be checked if the items can actually be packed in the bin. $u_t^\star$ defines how many items of type t must be packed in the bin; let $\mathcal{J}_t$ us denote the set of items. Each item $j \in \mathcal{J}_t$ is then of a specific type $t \in \mathcal{T}$ and the number of items is $|\mathcal{J}_t| = u_t^\star$.

The variables x_{jt}, y_{jt}, z_{jt} specify the lower-left-back position of the item and $r_{jt} \in \mathcal{R}_t$ specify the orientation of the item given the possible rotations

of the type. Given the orientation r_{jt} of the item we can determine the length l_{jt}, width w_{jt} and height h_{jt} by using the element constraint:

$$\begin{array}{ll} \text{element}(r_{jt}, [l_t, l_t, w_t, w_t, h_t, h_t], l_{jt}), & j \in \mathcal{J}_t, t \in \mathcal{T} \\ \text{element}(r_{jt}, [w_t, h_t, l_t, h_t, l_t, w_t], w_{jt}), & j \in \mathcal{J}_t, t \in \mathcal{T} \\ \text{element}(r_{jt}, [h_t, w_t, h_t, l_t, w_t, l_t], h_{jt}), & j \in \mathcal{J}_t, t \in \mathcal{T} \end{array} \tag{3}$$

Each item must be inside the considered bin of type $b \in \mathcal{B}$:

$$\begin{array}{ll} 0 \leq x_{jt} \leq L_b - l_{jt}, & j \in \mathcal{J}, t \in \mathcal{T} \\ 0 \leq y_{jt} \leq W_b - w_{jt}, & j \in \mathcal{J}, t \in \mathcal{T} \\ 0 \leq z_{jt} \leq H_b - h_{jt}, & j \in \mathcal{J}, t \in \mathcal{T} \end{array} \tag{4}$$

No pair of items $j, k \in \mathcal{J}$ may overlap:

$$\begin{array}{ll} x_{jt} + l_{jt} \leq x_{kt'} \vee x_{kt'} + l_{kt'} \leq x_{jt} \quad \vee & \\ y_{jt} + w_{jt} \leq y_{kt'} \vee y_{kt'} + w_{kt'} \leq y_{jt} \quad \vee & \\ z_{jt} + h_{jt} \leq z_{kt'} \vee z_{kt'} + h_{kt'} \leq z_{jt}, & jt \neq kt',\ j \in \mathcal{J}_t,\ k \in \mathcal{J}_{t'}, \\ & t, t' \in \mathcal{T} \end{array}$$

The disjunctive non-overlap constraints are exactly the constraints making the problem extremely difficult to solve with MIP solvers. There is no guarantee that the solution will actually be packable. Instead we use a variant of the ONEBIN procedure by Martello et. al. [11]. The items are packed in so-called corner points. Initially the only corner point is the lower-left-back corner of the bin. At any following stage, items can only be placed to the right, above or in front of already placed items. Figure 2 on the facing page is an illustration of available corner points for a given set of packed items.

The procedure is a depth first search where we at a given node consider placing all item types in all possible corner points and with each feasible item rotation. Each time an item is placed, the corner points are updated. The search is stopped when a feasible solution is achieved. As in Martello et. al. [11] we prune a node in the search tree if the volume of the remaining items is larger than the remaining volume of the bin.

Any packing found during the search phase with negative reduced cost is added to the master problem. If it was not possible to find such a packing, we want to resolve the 1D Knapsack Problem, but with the infeasible solution cut off the solution space. The procedure is then to add a cut removing the infeasible solution, resolve the 1D Knapsack Problem, check 3D Knapsack feasibility etc. until we achieve a feasible 3D packing. This approach was first suggested by Pisinger and Sigurd [12] for solving the 2DBPP.

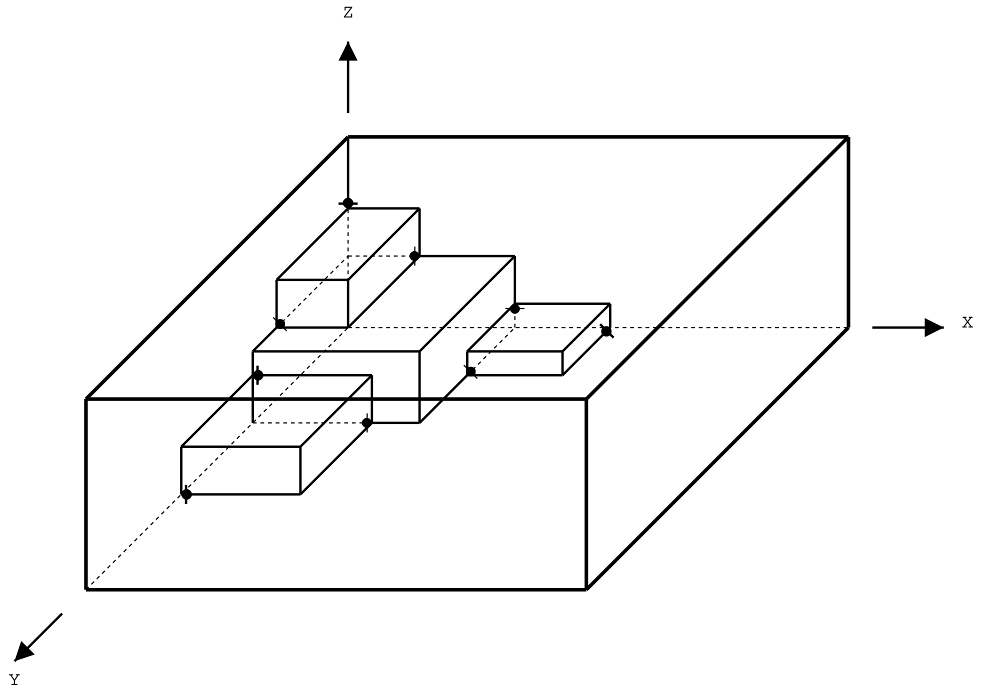

Figure 2: Illustration of corner points for a given packing.

3.2 Adding Infeasibility Cuts

Let U be the set of item types picked by the 1D Knapsack Problem. Now consider the following inequality:

$$\sum_{i \in U} u_i \leq \sum_{i \in U} u_i^{\star} - 1 \tag{5}$$

At first sight this may seem like a valid inequality to cut off the solution $u^{\star}$ with item types U. Assume that $U = \{1, 2\}$, $u_1^{\star} = 2$ and $u_2^{\star} = 1$ resulting in the cut $u_1 + u_2 \leq 2$. This cut will however remove for instance the solution $u_1 = 0$ and $u_2 = 3$ as well, which is not what we want. Instead we reformulate the 1D Knapsack in the following equivalent way where the variables u_{jt} is 1 if the j'th item of type t is placed in the knapsack:

$$\eta_b = \min \; c_b - \sum_{t \in \mathcal{T}} \sum_{j \in \mathcal{J}_t} \pi_t u_{jt}$$

$$\sum_{t\in\mathcal{T}}\sum_{j\in\mathcal{J}_t} v_t u_{jt} \le V_b,$$

$$\sum_{t\in\mathcal{T}}\sum_{j\in\mathcal{J}_t} e_t u_{jt} \le E_b,$$

$$u_{jt} \ge u_{it}, \qquad j < i,\ i,j \in \mathcal{J}_t,\ t\in\mathcal{T} \tag{6}$$

$$u_{jt} \in \{0,1\}, \quad j\in\mathcal{J}_t,\ t\in\mathcal{T}$$

The constraints (6) are included to avoid symmetries and are actually necessary for the introduced 3D infeasibility cuts: Consider again our example $U = \{1,2\}$, $u_1^\star = 2$ and $u_2^\star = 1$. In the new variables, we will have $u_{11}^\star = 1, u_{21}^\star = 1$, $u_{12}^\star = 1$ and all other variables equal to zero. The cut will then be $u_{11} + u_{21} + u_{12} \le 2$ allowing the solution $u_{12}^\star = u_{22}^\star = u_{32}^\star = 1$. Without constraint (6) the solution $u_{11} = 1, u_{21} = 1, u_{12} = 0$ and $u_{22} = 1$ would be feasible, which is obviously an error.

Let U' be the set of pairs consisting of item type and item number in the optimal solution for the new formulation. The cut inequality then becomes:

$$\sum_{(j,t)\in U'} u_{jt} \le |U'| - 1 \tag{7}$$

When we add a cut, we check if one or more of the items can be exchanged by other dominating items and hereby add more cuts. An item of type t is dominated by item t', if $l_{t'} \ge l_t$, $w_{t'} \ge w_t$, $h_{t'} \ge h_t$ and $R_{t'} \subseteq R_t$. All combinations of dominating items are generated. If an item t' is dominating all the items in the constraint, we instead add only one stronger constraint:

$$\sum_{(j,t)\in U'} u_{jt} + u_{1t'} \le |U'| - 1 \tag{8}$$

Thousands of cuts can be generated even on relatively small instances. This will make it quite time consuming to solve the 1D Knapsack Problem. Many of the cuts will fortunately dominate or be dominated by other cuts and can hence be removed speeding up the time to solve the problem. Another trick from Pisinger and Sigurd [12] is to identify unpackable subsets of unpackable sets and hereby strengthening and reducing the number of cuts. Basically we remove the smallest item from the set until the subset is packable and then the last unpackable subset defines the cut.

3.3 Heuristic Packing Generation

The 1D/3D Knapsack phase can be very time consuming. To reduce that time we apply a similar search, as described earlier, before solving the 1D Knapsack Problem. The heuristic packing generation procedure is also a depth first search, but the size of the search tree is reduced. At a given node in the tree we consider placing only a subset of the item types with positive duals and only in a subset of all the possible corner points. For each level from the root of the tree the size of the subsets are halved, although always one possibility is tried. Any packing found during the search with a negative reduced cost is added to the master problem.

For the heuristic packing generation we apply another pruning strategy. During the search we save the volume of the packed items in a given node, if it is the largest volume packed so far. We then prune the tree at a node, if it is not possible to improve the best volume found so far, given the volume of the packed items so far and the remaining volume of the bin. The search is heuristic and if no interesting packings are found, we switch to the 1D/3D Knapsack phase.

4 Branch & Price

There is no guarantee that the optimal solution to the restricted master problem is integral. To achieve that we must embed the column generation in a Branch & Bound scheme resulting in a Branch & Price algorithm. The obvious branching strategy would be to do normal branching directly on the variables in the master problem – basically removing columns from the solution or forcing columns into the solution. Forcing columns into the solution works fine, but when removing a column there is a significant probability that exactly that column will be generated in the next pricing iteration. The branching scheme must work in the pricing problem as well. This is relatively easy to do for Set Partitioning and Covering type of problems, which we will see later, but we are solving general IP problems, which make things more complicated. Vanderbeck and Wolsey [15] developed a branching scheme also discussed in Barnhart et.al. [1], which we will use.

4.1 The Master Problem

Assume that the optimal restricted master solution, $q^\star$, of (1), is fractional. We may find a row t and an integer α_t, such that δ in

$$\sum_{p \in \{a_p^t \geq \alpha_t\}} q_p^\star = \delta$$

is fractional. Basically, we sum the $q_p^\star$ of columns p covering row t resulting in a fractional sum, δ. We can then branch on the following constraints:

$$\sum_{p \in \{a_p^t \geq \alpha_t\}} q_p \leq \lfloor \delta \rfloor \quad \text{and} \quad \sum_{p \in \{a_p^t \geq \alpha_t\}} q_p \geq \lceil \delta \rceil$$

These constraints are upper and lower bounds on the number of bins in the solution with $a_p^t \geq \alpha_t$. Note, that we are not able to remove or force columns into the solution by adding these constraints – the constraints must be added to the restricted master problem explicitly. This leads to an extra dual value in the reduced cost of any new column with $a_p^t \geq \alpha_t$.

In some cases it is not possible to achieve a fractional δ when considering one row only and we will have to consider multiple rows. For instance for the Set Partitioning Problem α_t is always one, since $a_p^t \in \{0, 1\}$ and $\sum_{p \in \{a_p^t \geq 1\}} q_p^\star = 1$ for every row. To overcome this in the general case we search for two rows t and t' with α_t and $\alpha_{t'}$ such that δ is fractional in

$$\sum_{p \in \{a_p^t \geq \alpha_t \wedge a_p^{t'} \geq \alpha_{t'}\}} q_p^\star = \delta$$

We continue increasing the number of rows until δ is fractional. Let α be a vector with an entry for each constraint row and $P(\alpha) = \{p \in \mathcal{P}' | a_p^t \geq \alpha_t, t \in \mathcal{T}\}$. Then generally we search for a vector α where δ is fractional in

$$\sum_{p \in P(\alpha)} q_p^\star = \delta. \tag{9}$$

The restricted master problem at node n now becomes

$$\min \sum_{p \in \mathcal{P}'} c_p q_p$$

$$\sum_{p \in \mathcal{P}'} a_p^t q_p = d_t, \quad t \in \mathcal{T}$$

$$\sum_{p \in P(\alpha^j)} q_p \leq \lfloor \delta^j \rfloor, \quad j \in F^n \tag{10}$$

$$\sum_{p \in P(\alpha^j)} q_p \geq \lceil \delta^j \rceil, \quad j \in G^n \tag{11}$$

$$q_p \geq 0 \text{ and integer}, \quad p \in \mathcal{P}'$$

F^n and G^n in (10) and (11) are index sets over branching constraints defined by the pairs (α^j, δ^j) where δ^j is fractional in (9). The constraints are added from the root of the tree and down to the present node n. At a node a constraint of type (10) is added in one branch and of type (11) in the other branch.

4.2 The Pricing Problem

Now let us consider the changes to the pricing problem, (6). Let $R(\alpha)$ denote the chosen set of branching rows for a given α: $R(\alpha) = \{t \in \mathcal{T} | \alpha_t > 0\}$. Further, let λ be the dual variables corresponding to the added upper bound branching constraints and κ for the lower. The values of the dual variables will be $\lambda \leq 0$ and $\kappa \geq 0$. Finally, let s be a binary variable, where $s = 1$, if $\sum_{j \in \mathcal{J}_t} u_{jt} \geq \alpha_t$, $\forall t \in R(\alpha)$ and $s = 0$, otherwise. The pricing problem at node n of the tree now becomes

$$\eta_b = \min \ c_b - \sum_{t \in \mathcal{T}} \sum_{j \in \mathcal{J}_t} \pi_t u_{jt} - \sum_{j \in F^n} \lambda_j s_j - \sum_{j \in G^n} \kappa_j s_j \tag{12}$$

$$\sum_{t \in \mathcal{T}} \sum_{j \in \mathcal{J}_t} v_t u_{jt} \leq V_b,$$

$$\sum_{t \in \mathcal{T}} \sum_{j \in \mathcal{J}_t} e_t u_{jt} \leq E_b,$$

$$u_{jt} \geq u_{it}, \qquad j < i, \ i, j \in \mathcal{J}_t, \ t \in \mathcal{T}$$

$$s_j \leq u_{\alpha_t^j t}, \qquad j \in G^n, \ t \in R(\alpha^j) \tag{13}$$

$$s_j \geq 1 - \sum_{t \in R(\alpha^j)} (1 - u_{\alpha_t^j t}), \qquad j \in F^n, \tag{14}$$

$$u_{jt} \in \{0, 1\}, \quad j \in \mathcal{J}_t, \ t \in \mathcal{T}$$

$$s_j \in \{0, 1\}, \quad j \in F^n \cup G^n$$

For each node n in the search tree extra constraints are added. Type (13) is added in the case that an upper bound has been added to the master problem and (14) in case of a lower bound.

There is a constraint of type (13) for each row $t \in R(\alpha^j)$ for $j \in G^n$. For $\kappa_j \geq 0$ implies that $s_j, j \in G^n$ should be 1 to minimize the objective. $s_j, j \in G^n$ can only be 1, if for all types $t \in R(\alpha^j)$ the α_t^j'th item is chosen, which basically is $u_{\alpha_t^j t} = 1$.

In case of a type (14) only one constraint is added in the given node. For $\lambda_j \leq 0$ implies that $s_j, j \in G^n$ should be 0 to minimize the objective. Here the variable s_j is forced to 1, if for at least one of the item types $t \in R(\alpha^j)$, the α_t^j'th item is chosen.

4.3 Searching for Branching Constraints

Clearly we want $|R(\alpha)|$ and the number of Branch & Bound nodes n to be as small as possible. Different strategies can be applied to find $R(\alpha)$ or more specifically α, such that $|R(\alpha)|$ is small or minimal. Other possibilities is searching for $\delta - \lfloor \delta \rfloor$ fractional, but close to one in order to quickly find an integral solution or $\delta - \lfloor \delta \rfloor$ close to 1/2 in order for the duals in the pricing problem to be as large as possible. We decided to consider only the last two. We show on the current page the algorithm implemented in the procedures `findAlphaAndDelta` and `searchForOtherAlphas`. We assume that we have a compact representation of each column, $p \in \mathcal{P}'$, consisting of two arrays, A^p and T^p with entries for each $a_t^p > 0$ in A^p and corresponding $t \in \mathcal{T}$ in T^p. The arrays have length m^p. The results are stored in a global storage object, *Save*, which is accessibly by both `findAlphaAndDelta` and `searchForOtherAlphas`.

First the algorithm iterates through all fractional variables. For the corresponding column, f, we iterate through the positive entries and save the row number from T^f in the set R. Then `searchForOtherAlphas` is called.

Procedure `findAlphaAndDelta`

Data : Given $q^\star$ and $A^p, T^p, m^p,\ p \in \mathcal{P}'$, global object *Save*
Result : Return the best α and δ from *Save*
for *all fractional variables,* $q_f^\star$ **do**
  for $i = 1, ..., m^f$ **do**
    $R = \{T^f[i]\}$
    searchForOtherAlphas($A^f, T^f, m^f, i, R, Save$)

In `searchForOtherAlphas` iteration is done over the remaining positive entries in column f. The procedure is called recursively to consider all combinations of entries. For each combination we, in the procedure `saveAlphaDelta`, save the corresponding α and δ, if δ is fractional in the corresponding sum $\sum_{p \in \{a_p \geq \alpha\}} q_p^\star = \delta$. The values are stored in $Save$.

Procedure `searchForOtherAlphas`$(A^f, T^p, m^f, i, R, Save)$

```
Data   : Given q* and global object Save
Result : The sets of potential R's are stored in Save
for j = i+1, ..., m^f do
    R = R ∪ {T[j]}
    if fractionalSum(q*, R) then
        saveAlphaDelta(A^f, R, Save)
    searchForOtherAlphas(A^f, m^f, j, R, Save)
    R = R \ {T[j]}
```

If several rows have $d_t = 1$, then we can in some cases simplify the branching. For fractional $q^\star$, if there exists a pair of rows t, t' and $\alpha_t = \alpha_{t'} = 1$, such that $0 < \sum_{p \in \{a_p^t \geq 1\}} q_p^\star < 1$, then we avoid explicitly adding branching constraints. In one branch we simply remove all columns with $a_p^t = a_p^{t'} = 1$ and add the constraint $u_{1t} + u_{1t'} \leq 1$ to the pricing problem, i.e. items t and t' cannot be packed together. In the other branch we remove all columns with $a_p^t + a_p^{t'} \leq 1$ and add the constraint $u_{1t} = u_{1t'}$ to the pricing problem, i.e. items t and t' must be packed together. This is the branching strategy of Ryan and Foster [14].

5 Lower Bounds

Good lower bounds are always important for pruning the search tree when applying Branch & Bound. When using heuristics lower bounds are of interest for measuring the quality of the heuristic. For the 3D-BPP we can derive lower bounds directly from the problem data, but when applying column generation we can use the LP-solution of the restricted master problem to derive bounds as well.

The most obvious bound to calculate is the continuous lower bound:

$$L_0 = \min_{b \in \mathcal{B}} \max \left\{ \left\lceil \frac{\sum_{t \in \mathcal{T}} d_t v_t}{V_b} \right\rceil, \left\lceil \frac{\sum_{t \in \mathcal{T}} d_t e_t}{E_b} \right\rceil \right\}$$

Martello et. al. [11] show that the asymptotic worst-case performance ratio of L_0 is $\frac{1}{8}$. It is quite simple to realize that it must be close to that value, but harder to prove that it is tight. Here we are satisfied with the first. Consider the case of an instance with T items all of size $l = \frac{L}{2} + \sigma, w = \frac{W}{2} + \sigma$ and $h = \frac{H}{2} + \sigma$. For $\sigma = 0$ exactly $\frac{T}{8}$ bins are needed to pack the items. For $\sigma > 0$, T bins are necessary, but L_0 is still be close to $\frac{T}{8}$ for σ sufficiently close to 0.

The continuous bound is reasonably good when the item sizes are small compared to the bin size, but as we have seen can be very poor for relatively large items. Martello et.al. [11] introduce a new stronger bound, L_2, taking all three dimensions into account. When rotation of items is allowed, we have to use the smallest possible size of the items in all three dimensions, which deteriorates the bound. An alternative is to generalize the bounds of Dell'Amico et.al. [4] for the non-oriented 2D-BPP. All items are cut into smaller cubes, which allows a simplification of the bound calculation. The number of items created on the other hand can be quite large. If rotation is restricted to a subset of items and/or orientations, then the L_2 is definitely preferable. We still calculate the bound using the smallest possible size in each dimension, but these sizes can now be different for each side of the item, since the possible rotations are restricted. This makes the bound stronger than the Dell'Amico bound where the items are cut in cubes.

Note that the bound is not only used to get an initial lower bound on the number of bins. It is also used for detecting infeasible 1D Knapsack solutions in the pricing procedure.

The LP-solution to the restricted master problem is only a valid lower bound when no more columns can improve the solution. Let us formulate the Lagrangian Relaxation of the restricted master problem at node n (shown on page 210):

$$L(\pi, \lambda, \kappa) = \min \sum_{p \in \mathcal{P}} c_p q_p + \sum_{t \in \mathcal{T}} \pi_t \left(d_t - \sum_{p \in \mathcal{P}} a_p^t q_p \right) \tag{15}$$

$$+ \sum_{j \in F^n} \lambda_j \left(\lfloor \delta_j \rfloor - \sum_{p \in P(\alpha_j)} q_p \right) + \sum_{j \in G^n} \kappa_j \left(\lceil \delta_j \rceil - \sum_{p \in P(\alpha_j)} q_p \right)$$

$$q_p \in \mathbf{R}_+,\ p \in \mathcal{P}, \quad \pi_t \in \mathbf{R},\ t \in \mathcal{T}, \quad \lambda_j \leq 0,\ j \in F^n, \quad \kappa_j \geq 0,\ j \in G^n$$

$L(\pi, \lambda, \kappa)$ is a lower bound to the LP-relaxation of the master problem – specifically π, λ, κ taking the values of the dual variables from the restricted

master problem. By rearranging the terms in (15) we reach the following expression:

$$L(\pi,\lambda,\kappa) = \min \sum_{p\in\mathcal{P}} \left(c_p - \sum_{t\in\mathcal{T}} a_p^t \pi_t \right) q_p - \sum_{p\in P(\alpha_j)} \left(\sum_{j\in F^n} \lambda_j + \sum_{j\in G^n} \kappa_j \right) q_p$$

$$+ \sum_{t\in\mathcal{T}} d_t\pi_t + \sum_{j\in F^n} \lambda_j \lfloor \delta_j \rfloor + \sum_{j\in G^n} \kappa_j \lceil \delta_j \rceil \tag{16}$$

$$q_p \in \mathbf{Z}_+,\ p\in\mathcal{P}, \quad \pi_t \in \mathbf{R},\ t\in\mathcal{T}, \quad \lambda_j \le 0,\ j\in F^n, \quad \kappa_j \ge 0,\ j \in G^n$$

Note that the first line of the expression is a sum over the reduced costs, which is basically the pricing problem. Assume that we have a found a feasible solution for each bin type and let U_b be the number of bins of type b for each solution. Assume further that we have the optimal reduced costs for each pricing problem, η_b, from the last iteration. A valid lower bound is then

$$LB^n = \sum_{b\in\mathcal{B}} U_b\eta_b + \sum_{t\in\mathcal{T}} d_t\pi_t + \sum_{j\in F^n} \lambda_j \lfloor \delta_j \rfloor + \sum_{j\in G^n} \kappa_j \lceil \delta_j \rceil$$

For instances with more than one bin type this bound is rather weak. In conclusion we have a valid lower bound for each iteration of the column generation using the reduced costs of the pricing problems. We can at any time stop the column generation, if $\lceil LB^n \rceil \ge LP^n$, where LP^n is the solution value to LP-relaxation of the restricted master problem, since no better bound can be achieved in that node. If $\lceil LB^n \rceil \ge UB$ where UB is any upper bound to the 3D-BPP, then we can prune the node altogether.

6 The On-Line Algorithm

We have developed an on-line greedy algorithm to solve the on-line packing problem. The order of the items arriving at the packing site is unknown. The items are in a queue, where we can observe and pick an item to pack from the first Q items. At the packing site S bins are available to pack at a time. When no more items can be packed in the bins, then one or more of the bins must be shipped off and replaced by one or more empty bin(s).

When considering which item to place, we evaluate the placement of each available item in each available bins, in all available corner points and all possible rotations. We primarily choose that combination of item, bin, corner and rotation where the increase in wasted space is minimal. Wasted space is

basically space, which is not used by an item, but is unavailable for packing after the particular item is placed. Before evaluating all the items we sort them in non-increasing order of the volume of the items. Hence in cases of equal waste, we place the larger item first. When none of the available items can be placed, we replace all available bins with new empty bins. The procedure continues until all items are placed.

The algorithm is not only used for solving the on-line problem, but is also used off-line to generate an initial upper bound and first columns in the Branch & Price approach.

7 Stability Issues

In real-life there is requirements on how items may be packed. Items should be placed on the floor of the bin or be somewhat supported underneath by other items. Support of items from the side of the bin or from the side of other items is not considered. An example of no support from the sides is the case where the bins are pallets.

A common measure of support is to require a certain percentage of an item to be supported – for instance at least 90%. Items supported by more than one item, may result in more stable packings. Generally the presence of guillotine cuts in packings make the packing less stable, but often these cuts are unavoidable, or deeming them infeasible is too restrictive causing less efficient packings.

The above measure of support is however not a sufficient condition for a packing to be stable. Subsets of items in the packing must also be stable. Consider for instance items placed as a stair-case. At some stage, increasing the number of items constructing the stair-case will make the packing unstable, caused by the combination of weight of the items and gravity. Stair-case types of packings are however not likely to be produced when requiring a significant amount of support. Things get even more complicated if the weight of an item is not evenly distributed (which we generally assume is the case). An additional measure of stability is the location of the center of gravity of a packing, which should preferably be located near the center of the bottom face of the bin.

We have in spite of the above considerations only implemented a simple constraint on the amount of support and number of supporting items.

8 Load Bearing Constraints

In practice the load bearing ability of the packed items must be taken into account. We assume that we are given a maximum allowed load per square unit (e.g. mm^2), o_t, for each item type t. This value might depend on which face of the item is up, but we ignore this fact and assume that it is constant or be the lowest value over the different faces. Note that it is straight forward to generalize the method to cope with different values depending on the orientation of the item. We assume that the load bearing ability is the same at any point on the top face, even though it might be stronger close to the edges of the item, also depending on the density of the item. Also we assume that the weight of an item, e_t, is evenly distributed over the contact area of the items below.

The above assumptions are similar to the assumptions of Ratcliff and Bischoff [13]. They, however, allow different load bearing depending on the orientation of the item. In their packing heuristic, they collect a block of identical items and place it on a surface, which completely supports the block of items.

The algorithm `checkLoadBearing` on this page shows the procedure to check if an item j can be placed at a specific location in the bin. Line 1 and the loop starting in line 2 identify items below touching j. The algorithm `overlap` returns the size of overlap between to line segments. Line 3 of `checkLoadBearing` calculates the area of overlap between the two considered items. In this way the total area of support of item j is determined. The weight per square unit is then calculated in line 4 and the load bearing of item j is updated depending on its original load bearing and the maximum extra load items below can bear. If in the end the load bearing of item j is negative, it indicates that some item below is overloaded.

9 Experiments and Results

We have tested the approach described on a number of benchmark instances due to Martello et.al. [11], Ivancic et. al. [10] and as well on some instances that we have generated from real-life data made available by Bang & Olufsen.

The code was implemented in ECL^{i}PS^e [9], which is a Constraint Logic Programming framework based on Prolog. We have used Xpress-MP 13.26 for solving LP and IP problems. The experiments were executed on a SUN Fire 3800, 750 MHz computer.

Early experiments showed that the feasibility problem is extremely time

Procedure `checkLoadBearing`

Data : Item j to be placed. Set of already placed items, PL.

Result : The boolean variable $status$, which is false if constraints are violated.

```
    if z_j = 0 then
        status = true
    else
1       PL^below = {i ∈ PL | z_j = z_i + h_i}
        PL^overlap = ∅
        total = 0
2       foreach i ∈ PL^below do
3           area = overlap(x_j, l_j, x_i, l_i) overlap(y_j, w_j, y_i, w_i)
            if area > 0 then
                PL^overlap = PL^overlap ∪ {i}
                total = total + area_i
        if total > 0 then
            for i ∈ PL^overlap ∧ o_j ≥ 0 do
4               o_j = min{o_i − e_j/total, o_j}
            status = (o_j < 0)
        else
            status = false
```

Procedure `overlap`(x_1, l_1, x_2, l_2)

Data : Location, x, and size, l, of two line segments: (x_1, l_1, x_2, l_2)

Result : The size of the overlap between the segments: l_3.

$x_3 = \max(x_1, x_2)$

$l_3 = \max(0, \min(x_1 + l_1 - x_3, x_2 + l_2 - x_3))$

consuming to solve. To limit the time spent in the feasibility problem, two phases of the packing feasibility problem is applied. We first set a time limit of up to 60 seconds depending on the number of items to be packed. During the search, if no more packings and branches can be applied the algorithm switches to a state with no time limit on the feasibility packing. It is checked at certain time intervals during the search, whether a packing has been added to the master problem. If an item has been added, the search is halted and the master problem resolved. Every 10th iteration we try to improve the upper bound by solving the master problem including the integrality constraints.

When using ECL^{i}PS^e it was found that a "lazy" depth first search Branch & Bound was the easiest approach. This basically means that when branching, we do not find the bound in each child node before choosing the next node to branch on. Furthermore the descendants in one of the branches are fully investigated before considering the second branch. Better lower bounds would be achievable if both children of a node was considered before branching further on one of them. In fact the lower bound of the entire problem can only be improved if the second child in the root node is considered at some stage.

9.1 Bang & Olufsen instances

The Bang & Olufsen instances are generated from real-life data on item sizes, weight, load bearing ability and allowed orientations. Stability in form of 90% support is also required. Data on 21 different item types were used for creating 9 instances. The data and instances are described and listed in appendix A on page 226.

In the following experiment the on-line heuristic could consider 20 items at a time and only place them on one pallet. The packings in the on-line solution was the starting point of the Branch & Price algorithm. Note that the Branch & Price is allowed to consider all packings, i.e. there are no restrictions on the packing order of the items. In case of an on-line problem, we can only use the Branch & Price for comparing the performance of the on-line algorithm. The lower bound of the Branch & Price algorithm is still valid, but the upper bound might not be achievable for the on-line problem.

A significant saving in packing costs by using Branch & Price instead of the on-line heuristic might however allow for a change in the packing procedure of the company, such that the items can be packed in any arbitrary order. For these instances a limit of 3600 seconds of computation time was available.

Table 1 on the following page gives the results of the on-line heuristic followed by the Branch & Price. "LB" and "UB" are lower and upper bounds.

"Cols" and "Cuts" are generated columns and infeasibility cuts, "Nodes" are Branch & Bound nodes and finally "Time" is the computation time in seconds for finding the best upper bound. The initial gap between the lower and upper bounds is 35%, which is reduced to 28%. It is still a quite large gap, but the instances seem to be quite difficult to solve. In one case, over 16.000 cuts are added. This indicates that certain combinations of items are attractive to put together because of volume and dimensions, but the exact shapes of the items make them impossible to pack. These combinations of items takes a very long time to prove infeasible. In fact, 99% of the time is spend in the one bin packing algorithm. The saving by using Branch & Price for these test cases was only 10%. With more computation time it could possibly be increased.

	On-Line		Branch & Price				
Items	LB	UB	UB	Cols	Cuts	Nodes	Time
19	3	5	4	665	2842	11	241
29	4	6	5	1164	5310	13	71
32	5	7	6	591	16056	11	48
33	4	7	6	782	8548	1	49
46	5	8	7	938	5877	11	96
45	7	10	9	728	9972	8	49
46	6	9	9	976	9935	13	0
54	6	9	9	814	8280	7	0
58	7	11	10	454	9209	13	226
Total	47	72	65	7112	76029	88	780

Table 1: B&O instances with "half" branching.

As mentioned earlier, we use a lazy Branch & Bound approach. All the nodes considered for the results in table 1 are in the "left" side of the tree, which means that no improvement of the lower bound was possible.

The branching strategy in the runs reported in table 1 was to branch on fractional sums close 0.5 as discussed in section 4 on page 212 in order for the duals in the pricing problem to be as large as possible. The results in table 2 on the facing page are on the other hand with the strategy of branching on fractions close to 1 to quickly reach integral solutions. The quality of the solutions are exactly the same, but slightly more nodes and time are used in table 2.

Now we turn our attention to the on-line algorithm. We have made several experiments investigating the impact of stability and load bearing requirements as well as how many items are available for packing and finally how

Items	On-Line LB	UB	Branch & Price UB	Cols	Cuts	Nodes	Time
19	3	5	4	610	3598	15	342
29	4	6	5	925	8455	12	51
32	5	7	6	675	14114	12	1137
33	4	7	6	822	4517	11	49
46	5	8	7	710	4808	13	332
45	7	10	9	848	7595	12	73
46	6	9	9	687	8130	14	0
54	6	9	9	873	4950	12	0
58	7	11	10	938	6046	13	130
Total	47	72	65	7088	62213	114	2114

Table 2: B&O instances with "one" branching.

many open pallets that can be packed.

The results are compiled in table 3. The columns are divided into three groups: All the items are visible or packable, the first item is visible and the first 20 items are visible. Each of the groups are again divided into packing on 1 or 2 open pallets at a time. The results are sums over the 9 instances.

In the first row we have no stability constraints (0% support) and no load bearing constraints. In the next row we introduce load bearing constraints. Comparing to the 90% support case, the load bearing constraints accounts for almost half the increase in number of necessary pallets. Going from 90 to 100% support does not introduce a significant increase in pallets.

Two open pallets seem to have a slightly negative effect, which is a bit surprising. More short-sighted decisions must be made in the two pallet case or perhaps the strategy of replacing both pallets leads to lost packing opportunities. The two cases of all or 20 visible items seem to have an insignificant impact. Clearly, the case of only one visible item is much worse.

	All items visible		1 item visible		20 items visible	
Support	1 Pallet	2 Pallets	1 Pallet	2 Pallets	1 Pallet	2 Pallets
no load, 0%	60	61	88	89	60	61
0%	67	69	102	97	68	71
90%	74	74	131	129	72	75
100%	72	73	138	129	73	77

Table 3: On-Line results for B&O instances.

To test the robustness of the on-line algorithm we, for the 58 item case with two pallets, did several runs shuffling the order of item arrivals. The results are shown in table 4. Not that surprisingly the arrival order gets more important when fewer items are visible. The difference between 10 and 20 visible items is insignificant.

Visible items	(no. of runs, bins)	Av. bins
20	(11, 12)	12.0
10	(1, 11), (7, 12), (3, 13)	12.2
1	(1, 19), (6, 21), (2, 22), (2,23)	21.4

Table 4: Results when repeating runs for B&O instances.

By re-running the algorithm shuffling the input order, we have created a randomized algorithm. For instance with 10 visible items, repeating the algorithm the result improves from 13 to 11 bins.

9.2 Martello et. al. [11] instances

In the instances constructed by Martello et. al. [11], we have only one item of each type and the items cannot be rotated. 9 different classes with 10 instances each are given. Class 4 for example has a significant number of very large items making the instances relatively easy to solve. Class 5 on the other hand has mainly small items. Finally we will mention class 9. Here the items are cut out of 3 boxes, which means that the optimal solution is always 3 bins with no waste.

In table 5 on the next page is given the number of instances solved to proven optimality within 1200 seconds for the different classes and instance sizes. A total of 344 instances was solved to proven optimum out of 810 possible.

Clearly, the algorithm does not scale very well for these problem instances. Class 9 seems to be particularly difficult to solve. Class 4 is the exception, since all instances are solved to optimality. Looking at the computational effort, instances with 10 items are solved in 0.8 seconds on average, while for 15 items 109.5 seconds are needed.

This is not particularly impressive compared to the results of Martello et. al. [11]. They solve 698 to proven optimality only leaving 112. There are several reasons for that. They use tricks and heuristics for the single bin packing that we do not. Their algorithm is implemented in C, which is faster than ECL^{i}PS^e – their hardware on the other hand is slower. Finally there is

Items	Class									Total
	1	2	3	4	5	6	7	8	9	
10	10	10	10	10	10	10	10	10	10	90
15	9	10	10	10	10	9	9	10	10	90
20	6	8	8	10	9	10	5	9		65
25	3	2	4	10	8	4	1			32
30	4	1	1	10	2		1	1		20
35	1	1		10	4			1		17
40		1		10						11
45		1		10			1			12
50				10						10
Total	33	34	33	90	43	33	33	31	20	344

Table 5: Instances solved to proven optimality.

the issue of optimization approach: They use a direct Branch & Bound while we use a Branch & Price approach. The exact reasons are of course difficult to establish.

9.3 Ivancic et. al. [10] instances

Ivancic et. al. [10] constructed 17 instances with multiple container types and later Bischoff and Ratcliff [2] made 47 modified versions available electronically. The 47 instances are basically the 17 instances with only one container type per instance.

The instances have relatively few item types, i.e. 2 to 5, but many items are to be packed; in one instance a total of 180 items of 4 different types. The items can be packed in all 6 different orientations. 90% of the bottom face of the items are to be supported, but no load bearing is considered. 1800 seconds was available for each of the 47 instances.

Table 6 on page 225 shows the results achieved on the 47 instances. The first column refers to the numbering scheme in [2] while the second column refers to the one in [10]. The following column gives the number of different item types per instance. The following two columns are the results reported in [10] and [2]. Note that the results of the latter actually is the best over two different methods. The next two columns are lower and upper bounds for our approach without stability constraint and the last two columns are the results with 90% support required. In [2] the items are 100% supported due to the design of their algorithm and in [10] no support is required though a packing

routine similar to ours is applied.

The gap between the total lower and upper bounds without stability considerations is only 3.4% and the solutions are 6.6% better than [10] and 3.6% better than [2]. The instances 25, 26, 32 and 42 to 47 indicate that our approach has difficulties on instances resulting in solutions with relatively few bins. Our method, on the hand, is superior on instances resulting in solutions with more bins: Instances 1 to 5, 11, 12, 16, 21 to 23 and 30.

The picture is the same with stability constraints. The gap is 3.8%, but now comparing to [2] the difference in quality is 2.5% in favour of [2]. Their results are however the minimum over two different methods with totals 763 and 777. Compared to those numbers our method is still competitive.

These instances are clearly much easier to solve than the B&O and Martello et. al. instances. Fewer packing patterns and cuts are necessary to reach the optimum and prove it. Even though the instances are quite large when considering the total number of items, the algorithm takes advantage of the few types of items when packing. In each corner point only one item for each type is tried, but there is still many corner points to consider.

10 Conclusion

In this paper we have described both the off- and on-line versions of the non-oriented 3DBPP with additional constraints concerning the stability of the packing and load bearing of the items.

We have proposed a Branch & Price approach for solving the off-line problem combined with a Branch & Cut procedure for solving the 3D Knapsack Problem. Design and implementation issues have been discussed specially regarding stability and load bearing constraints. For the on-line problem a greedy on-line heuristic was developed and described. In conclusion, our method can solve general problem classes and at the same time take real-life considerations into account.

Experiments were conducted on instances generated from real-life item data. The instances were found to be quite difficult to solve with the proposed approach, but relatively good results were achieved with the on-line heuristic. More work is required for optimally solving the instances and hence exactly evaluate the performance of the on-line heuristic. Our implementation of the Branch & Price approach was not competitive compared to the approach in Martello et. al. designed specifically for the 3DBPP. On the Ivancic et. al. instances however the results were quite promising with a gap between lower and upper bounds on less than 4%.

Ref. [2]	Ref. [10]	Box types	Method of [10]	Method of [2]	LB	UB	Stability LB	Stability UB
1	1a	2	26	27	25	25	25	25
2	1b	2	11	11	9	9	10	10
3	2a	4	20	21	19	19	19	19
4	2b	4	27	27	26	26	26	26
5	2c	4	65	61	51	51	51	51
6	3a	3	10	10	10	10	10	10
7	3b	3	16	16	16	16	16	16
8	3c	3	5	4	4	4	4	5
9	4a	2	19	19	19	19	19	19
10	4b	2	55	55	55	55	55	55
11	4c	2	18	19	16	16	16	16
12	5a	3	55	55	53	53	53	53
13	5b	3	27	25	25	25	25	25
14	5c	3	28	27	27	27	27	27
15	6a	3	11	11	11	11	11	11
16	6b	3	34	28	26	26	26	26
17	6c	3	8	8	8	8	10	10
18	7a	3	3	2	2	2	3	3
19	7b	3	3	3	3	3	4	4
20	7c	3	5	5	5	5	6	6
21	8a	5	24	24	20	20	20	20
22	8b	5	10	11	8	9	8	12
23	8c	5	21	22	19	19	20	20
24	9a	4	6	6	5	6	5	6
25	9b	4	6	5	4	6	4	6
26	9c	4	3	3	3	4	3	5
27	10a	3	5	5	4	5	4	7
28	10b	3	10	11	10	10	10	10
29	11a	4	18	17	17	17	17	17
30	11b	4	24	24	22	22	22	22
31	11c	4	13	13	11	13	13	13
32	12a	3	5	4	4	5	4	5
33	12b	3	5	5	4	5	4	5
34	12c	3	9	9	9	9	9	9
35	13a	2	3	3	3	3	3	3
36	13b	2	18	14	14	14	14	14
37	14a	3	26	23	23	23	23	23
38	14b	3	50	45	45	45	45	45
39	14c	3	16	16	15	15	17	17
40	15a	4	9	10	7	9	7	11
41	15b	4	16	16	15	15	16	16
42	16a	3	4	5	4	5	5	5
43	16b	3	3	3	3	4	3	4
44	16c	3	4	4	3	5	3	6
45	17a	4	3	3	2	4	2	4
46	17b	4	2	2	2	3	2	5
47	17c	4	4	3	3	5	3	4
Total			763	740	689	713	730	759

Table 6: Results for the Ivancic et. al. [10] instances.

A Construction of Bang & Olufsen instances

In the following are the item data and constructed instances listed. Note that the dimensions of the items are measured in mm, weight in kg and load bearing in g/mm^2. The size of the bin is $(L, W, H) = (1200, 800, 2700)$ with unlimited weight capacity (In reality the bins are pallets). The 21 item types are listed in table 7. l_t, w_t and h_t are the length, width and height of item t, R_t is the set of possible orientations, e_t is the weight of item t and o_t is the load bearing capacity per square mm of item t. 9 random instances were

t	l_t	w_t	h_t	R_t	e_t	o_t
1	1200	800	1370	$\{1,3\}$	80	0.16
2	1100	800	1470	$\{1,3\}$	80	0.17
3	800	800	1260	$\{1,3\}$	80	0.23
4	800	600	1020	$\{1,3\}$	80	0.31
5	1100	800	1090	$\{1,3\}$	80	0.17
6	965	590	820	$\{1,3\}$	80	0.26
7	765	595	795	$\{1,3\}$	40	0.00
8	620	495	680	$\{1,3\}$	35	0.00
9	800	600	1175	$\{1,2,3,4,5,6\}$	7	2.08
10	2020	290	320	$\{1,2,3,4,5,6\}$	20	1.71
11	490	490	120	$\{1,2,3,4,5,6\}$	25	4.16
12	455	396	440	$\{1,2,3,4,5,6\}$	15	5.55
13	380	360	450	$\{1,2,3,4,5,6\}$	8	7.31
14	380	360	450	$\{1,2,3,4,5,6\}$	7	7.31
15	1216	170	205	$\{1,2,3,4,5,6\}$	12	4.84
16	1425	360	220	$\{1,2,3,4,5,6\}$	18	1.95
17	1190	280	190	$\{1,2,3,4,5,6\}$	16	3.00
18	650	460	265	$\{1,2,3,4,5,6\}$	12	3.34
19	1060	410	220	$\{1,2,3,4,5,6\}$	22	2.30
20	690	510	270	$\{1,2,3,4,5,6\}$	13	2.84
21	850	435	210	$\{1,2,3,4,5,6\}$	11	2.70

Table 7: Data for the 21 different item types.

generated from these item types as shown in table 8 on the facing page. The best lower and upper bounds found so far are found in the last 2 rows.

Item type	Number of items								
	19	29	32	33	46	45	46	54	58
1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	4	4	3	3
3	1	1	1	1	2	1	1	1	1
4	1	2	2	2	1	2	2	2	4
5	1	1	2	1	2	3	2	1	5
6	1	1	2	2	3	1	1	1	3
7	1	1	2	2	3	1	1	2	4
8	1	2	2	3	3	3	4	4	2
9	1	2	1	1	2	2	1	3	2
10	1	1	2	2	2	1	2	6	5
11	1	2	2	1	2	1	1	1	1
12	1	1	1	1	2	3	4	2	2
13	1	1	1	2	3	3	3	2	5
14	1	2	1	2	2	2	3	4	1
15	1	1	2	2	2	3	2	2	1
16	1	1	1	2	3	2	4	1	4
17	1	2	1	2	2	3	1	4	3
18	1	2	2	1	3	3	3	3	3
19	1	2	1	1	2	3	1	2	4
20		2	2	3	3	3	2	5	4
21			2		3		3	4	
LB	3	4	5	4	5	7	6	6	7
UB	4	5	6	6	8	9	9	8	11

Table 8: Constructed instances and lower bounds.

References

[1] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, *Branch-and-price: column generation for solving huge integer programs*, Operations Research, 46 (1998), pp. 316–329.

[2] E. E. Bischoff and M. S. W. Ratcliff, *Issues in development of approaches to container loading*, Omega, International Journal of Management Science, 23 (1995), pp. 377–390.

[3] C. S. Chen, S. Lee, and Q. Shen, *An analytical model for the container*

loading problem, European Journal of Operational Research, 80 (1995), pp. 68–76.

[4] M. Dell'Amico, S. Martello, and D. Vigo, *A lower bound for the non-oriented two-dimensional bin packing problem*, Discrete Applied Mathematics, 118 (2002), pp. 13–24.

[5] E. den Boef, J. Korst, S. Martello, D. Pisinger, and D. Vigo, *A note on robot-packable and orthogonal variants of the three-dimensional bin packing problem*, DIKU Technical Report 03/02, (2003).

[6] H. Dyckhoff, *A typology of cutting and packing problems*, European Journal of Operational Research, 44 (1990), pp. 145–159.

[7] O. Faroe, D. Pisinger, and M. Zachariasen, *Guided local search for the three-dimensional bin packing problem*, to appear in INFORMS Journal on Computing, (2002). Also available as DIKU Technical Report 99/13.

[8] P. C. Gilmore and R. E. Gomory, *Multistage cutting stock problems of two and more dimensions*, Operations Research, 13 (1965), pp. 94–120.

[9] IC-Parc, ECL^{i}PS^e, Imperial College, London, 5.5 ed., 2002. http://www.icparc.ic.ac.uk/eclipse/.

[10] N. Ivancic, K. Mathur, and B. B. Mohanty, *An integer programming based heuristic approach to the three-dimensional packing problem*, Journal of Manufacturing and Operations Management, 2 (1989), pp. 268–298.

[11] S. Martello, D. Pisinger, and D. Vigo, *The three-dimensional bin packing problem*, Operations Research, 48 (2000), pp. 256–267.

[12] D. Pisinger and M. Sigurd, *On using decomposition techniques and constraint programming for solving the two-dimensional bin packing problem*, DIKU Technical Report 03/01, (2003).

[13] M. S. W. Ratcliff and E. E. Bischoff, *Allowing for weight considerations in container loading*, OR Spektrum, 20 (1998), pp. 65–71.

[14] D. M. Ryan and B. A. Foster, *An integer programming approach to scheduling*, in Computer Scheduling of Public Transport, A. Wren, ed., North-Holland Publishing Company, 1981.

[15] F. Vanderbeck and L. A. Wolsey, *An exact algorithm for ip column generation*, Operations Research Letters, 19 (1996), pp. 151–159.

Paper VI

Constraint Programming versus Mathematical Programming

Published in Orbit Vol. 3 2003.

Constraint Programming versus Mathematical Programming

Jesper Hansen[1]

Keywords: Constraint Logic Programming, ECL^{i}PS^e , Mathematical Programming

1 Introduction

Constraint Logic Programming (CLP) is a relatively new technique from the 80's with origins in Computer Science and Artificial Intelligence. Lately, much research have been focused on ways of using CLP within the paradigm of Operations Research (OR) and vice versa. The purpose of this paper is to discuss the differences and similarities between the two mentioned paradigms and how they can complement each other.

In OR and Mathematical Programming we are modeling and solving hard combinatorial optimization problems with use of Linear Mixed Integer Programming models (MIP). *Programming* in MIP has nothing to do with *computer programming.* Rather, the result is a program or schedule for doing the optimized activities. In CLP on the other hand a constraint program is both a statement of a problem in variables and constraints and a specification of how to solve it. A CLP system contains a programming language component and a *constraint store* component that can store and "reason" about variables and constraints.

In the OR community we are interested in formulating and solving models of the following type:

$$\min z = \sum_{j=1}^{m} c_j x_j \tag{1}$$

$$\sum_{j=1}^{m} a_{ij} x_j = b_j \quad i = 1, ..., n \tag{2}$$

$$x_j \geq 0 \quad j = 1, ..., m \tag{3}$$

$$x_j \text{ integer} \quad j = 1, ..., m \tag{4}$$

A wide range of applications can be modeled and solved within this paradigm,

[1] Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark

but some are not well suited. The objective function and the constraints have to be formulated as linear functions[1], which is not always simple to do.

CLP on the other hand can handle non-linear functions in any type of constraint, for example $x^2 \neq y$. The increase in expressiveness when modeling is the major motivation to take a closer look at CLP from an Operations Researcher's point of view.

The CLP system *ECLiPSe* is a very flexible environment for implementing and solving both MIP, CLP and combined models, which we will use in the paper to illustrate the different possibilities. ECLiPSe is based on Prolog, but we will only describe specific Prolog constructs, which are needed for our purposes. Clocksin and Mellish [2] is the classic, but perhaps a bit out-dated book on Prolog.

Before considering the modeling possibilities in CLP we shall in section 2 recall the components of Branch & Bound for solving MIP problems and compare that to the CLP approach. In section 3 we show an example of modelling and solving a problem with both a MIP and a CLP model in ECLiPSe.

The potential of both paradigms is even more evident when they are combined into *hybrid solvers*, which is however not covered in this article.

2 Comparing Branch & Bound and CLP

Assume we are to minimize the objective function, $f(x)$ given by (1) over the region of feasible solutions, $x \in S$ given by equations (2)-(4). Let z^{opt} be the optimum value.

2.1 Branching and Tree Search

In order to guarantee finding the best solution in Branch & Bound, all solutions need to be enumerated in the worst case. This is basically done by traversing a search tree. In every node of the tree we divide the solution space into smaller subproblems, which are easier to solve. One way is to select a variable and in each subproblem or branch of the tree fix the variable to each of the values in its domain. Another approach is to split the domain in two branches by adding the constraint $x \leq v$ to the left branch and $x \geq v + 1$ to the right for some value $v \in D_x$, where D_x is the domain of the variable x.

For small problem instances as for instance 10 binary variables, we get 1024 potential solutions or leaves in the search tree. Increasing the number of

[1] Almost ignoring quadratic, semi-definite and other convex models solvable with use of interior point methods.

variables to only 100 binary variables, we get a 31 digit number of potential solutions. This exponential growth makes it impossible in practice to do an explicit enumeration of all solutions for problems of an interesting size.

Search by branching is the most obvious similarity between Branch & Bound and CLP.

2.2 Bounding by Relaxation

Branch & Bound is an implicit enumeration as it uses a bounding function to prune the search tree and only consider parts of the tree implicitly. A bounding function is found by relaxing some set of complicating constraints making the relaxed problem easy to solve. Let P be the region of solutions defined by the relaxed problem. Optimizing the objective function over P will give a lower bound z^{LB} on z^{opt}, since we are optimizing over a larger region.

Assume that the best solution found so far has value z^{best}. Now there are three cases where we can prune the search tree:

- $z^{LB} \geq z^{best}$ – no improving solutions in this subproblem.
- No feasible solutions can be found for the relaxed problem, and hence not for the original problem.
- A feasible solution is found to the original problem.

Bounding is not a part of the CLP paradigm. Instead optimization can be done in the following way: Each time a feasible solution has been found with objective value z^{best} a new constraint is imposed, $z < z^{best}$ making only improving solutions feasible.

The most widely used relaxation is *LP-relaxation* where the integrality constraints of the problem are relaxed resulting in a LP problem.

2.3 Strengthening by domain reduction and cuts

A very powerful extension of Branch & Bound is the possibility of adding *cuts* or constraints during the search to strengthen the relaxation. By finding and adding general or problem specific cuts, we want to remove parts of the relaxed solution space and ideally achieve the convex hull of the feasible integer points. This procedure is referred to as *Branch & Cut*.

In Branch & Bound the tree can be pruned, if no feasible solution exists for the relaxed problem in the current node. In CLP infeasibility is detected when any variable gets an empty domain. The variable domains are reduced

by use of *constraint propagation.* Consider the constraint $x_1 \geq x_2 + 1$ for $x_1, x_2 \in \{0, 1, 2\}$. No feasible value for x_1 exist when $x_2 = 2$. The value 2 can hence be removed from the domain of x_2. Similarly is the value 0 in the domain of x_1 inconsistent with any remaining values of x_2. At some stage in the search tree we would branch on x_1 and fix it to 2. Then the value 2 is not consistent in the domain of x_2 and it is removed and x_2 is fixed to 1. This is the most simple example of domain reduction, resulting in removal of the branching on x_2.

The reduced domains of x_1 and x_2 are propagated to other constraints including x_1 and x_2, which can again cause new domain reductions. This procedure continues until no further reductions can be performed and the resulting state is called consistent. Now 3 cases can occur:

1. All variables have exactly one value in its domain and a feasible solution is achieved.
2. One or more variables has no values in its domain – no feasible solutions.
3. At least one variable has a domain size larger than one.

In case of 1 and 2 the search backtracks or the branch is pruned in Branch & Bound terms. In case 3 a variable with domain size larger than one is selected and branching continues.

Generally, a value can be removed from the domain of a variable, if the value is inconsistent with any combination of possible values of the other variables in the considered constraint. If all values in the domains of the variables are consistent, the constraint is called *consistent.*

There are different types of consistency:

Node consistency: The constraint only includes one variable.

Arc consistency: The constraint includes two variables.

Hyper-arc consistency: The constraint has more than 2 variables. In general it is $\mathcal{NP}$-hard to show that a constraint is hyper-arc consistent and hence as hard as solving an arbitrary satisfiability problem [4].

In practice, hyper-arc consistency is not used. Instead the concept of *bounds consistency* is introduced. Consider the constraint $x = y + z$. Let $\min(D_x)$ and $\max(D_x)$ denote the minimum and maximum value in the domain of the variable x. The bounds on x can now be reduced by using the following constraints:

$$x \geq \min(D_y) + \min(D_z), \quad x \leq \max(D_y) + \max(D_z)$$

Isolating the other variables in the forms $y = x - z$ and $z = x - y$ we can reduce the domains of y and z in the same way - here shown for y:

$$y \geq \min(D_x) - \max(D_z), \quad y \leq \max(D_x) - \min(D_z)$$

Algorithms for reducing domains usually only consider one constraint at a time, but in many cases more reduction could be achieved by considering more than one constraint at a time. There is however a clear dependency between the complexity of the algorithm and the ability to reduce the domains. The choice of which algorithm to choose to get the best quality/time compromise can be problem dependent.

Domain reduction is not new in the OR community, since it has been a main part of specially tailored Branch & Bound algorithms. Domain reduction is as well used in presolvers of LP and MIP solvers, but only in the root of the search tree as the name also indicates. Note that from the point of view of Branch & Cut, domain reduction can be seen as adding simple cuts removing parts of the domain.

2.4 Designing Branch & Bound

In a combined framework of Branch & Bound and CLP the following design issues must be considered:

Bounding method: Choice of relaxation and bounding function.

Cuts and propagation: Choice of cuts to add as well as propagation strategies and algorithms.

Selection of next variable to branch on: Often one can do better than to branch on a random variable. In binary branching picking a variable where fixing it to one value gives a low bound and to the other a high bound. Hopefully, this will lead to a good feasible solution in the left subtree and make it possible to prune the right branch. We generally want to branch on "important" variables first. For a given problem it is often quite obvious, which variables are the most important. Later we will se an example on this. Generally, branching on variables with a small domain size first is a good choice for minimizing the number of nodes in the tree.

Branching strategy: When subdividing the solution space, instead of branching on all values in the domain of a variable, which is common in CLP,

the LP-relaxation can be used to split the domain into two. Let $\overline{x}$ be the non-integral value of the LP-relaxation. The problem is then split into two subproblems given by the two constraints: $x \leq \lfloor \overline{x} \rfloor$ and $x \geq \lceil \overline{x} \rceil$.

The following branching example is often used when solving scheduling problems. Here the decision must be taken whether activity i should be executed before j or j before i. This can be embedded in the branching strategy. In one branch the constraint $t_i + p_i \leq t_j$ is added, where t is the start time and p the duration, and in the other branch the constraint $t_j + p_j \leq t_i$. This strategy is not available in a standard MIP solver.

The order in which values are tried in the usual CLP branching is often important. Perhaps it is known in advance that values in the middle of the domain are most likely to lead to good solutions. Incorporation of problem specific knowledge is necessary to make the search efficient both in CLP and MIP.

Search strategy: Basically 3 pure strategies can be identified. In *Depth first search* a subtree is fully explored before backtracking, while in *Best first search* we always explore the most promising node first. Finally in *Breath first search* all nodes in the same level of the tree are explored before continuing to the next level. The advantage of Depth first search is the limited memory consumption, since the number of active nodes will be proportional to the maximum level of the tree. It is as well focused on finding a feasible solution as fast as possible in order to prune the tree. Breadth first search on the other hand will have as many active nodes as the level before the leaves. Best first search can in worst case be as bad as Breadth first, but generally it is much better. In fact after finding the optimal solution, no unnecessary nodes will be considered, but the growth in memory consumption is still exponential.

In CLP, depth first search is used. When mentioning Branch & Bound and tree search, we implicitly think of *complete search* where all solutions are enumerated and optimum is guaranteed to be found. In some cases a "good" solution is sufficient and perhaps it is not possible to search the entire solution space within reasonable time. Then *incomplete search* can be useful. Here, a number of different strategies are available to heuristically reduce the size of the tree. *Bounded Backtrack Search* simply stops the search after a certain number of backtracks has been exceeded. Other examples are *Credit Search*, *Limited Discrepancy Search* and others described in [1, 3], which more cleverly tries to explore promising regions.

$$\min \sum_{j \in J} c_j y_j + \sum_{i \in I} \sum_{j \in J} d_{ij} x_{ij} \tag{5}$$

$$\sum_{j \in J} x_{ij} = 1, \qquad i \in I, \tag{6}$$

$$x_{ij} \leq y_j, \quad i \in I,\ j \in J, \tag{7}$$

$$\sum_{i \in I} x_{ij} \leq cap_j, \qquad j \in J, \tag{8}$$

$$y_j \in \{0,1\} \qquad j \in J, \tag{9}$$

$$x_{ij} \in \{0,1\} \quad i \in I,\ j \in J \tag{10}$$

Figure 1: Warehouse Location Model.

Heuristics: To be able to prune the tree as soon as possible, one needs a reasonably good feasible solution or upper bound before the Branch and Bound is started. Here, a fast construction heuristic can make the difference between solving the problem within seconds and not solving it at all. After finding an initial solution it can be improved by local search heuristics. Local search is well supported by ECLiPSe through the repair library, which allows variables to have tentative values [1, 3]. We will not discuss heuristics further in this paper.

3 Modelling and Solving in ECLiPSe

ECLiPSe can use external LP and MIP solvers as CPLEX and XPRESS-MP and can hence be used purely as a mathematical modeling tool similar to GAMS. We will illustrate this by solving the following *Warehouse Location problem*: Given is a number of customers and a number of warehouse locations. Each open warehouse can only deliver to a limited number of customers and each customer only gets delivery from one warehouse. We must now decide which warehouses to open in order to minimize costs for opening the warehouses and delivering the demanded goods to the customers. The MIP model can be formulated as shown in figure 1 where c_j is the cost of opening warehouse $j \in J$ and d_{ij} is the cost of delivering goods to customer $i \in I$ from warehouse $j \in J$. The binary variable y_j indicates the opening of warehouse j while the binary variable x_{ij} indicates delivery to customer i from warehouse

j. Constraint (6) guarantees delivery from exactly one warehouse to each customer. (6) forces delivery from only open warehouses. Finally, (6) is the limit of customers serviced by each warehouse.

3.1 Introduction to ECLiPSe

We will for readers not familiar with Prolog or ECLiPSE give a short introduction to constructs used in the following.

- In Prolog the first letter in variables must be a capital letter. If a variable is assigned a value it is said to be *instantiated*. The only way a variable can be assigned another value is by backtracking past the place it was assigned the value and hereby the variable becomes *uninstantiated*.
- A list is indicated by brackets: For `A = [X,Y,Z]`, `A` is instantiated to the list `[X,Y,Z]`.
- `dim(X,[N,M])` can be used for both constructing a matrix `X` or retrieving the dimensions of the matrix. If `X` is a given 2 dimensional matrix, the size is returned in the variables `N` and `M`. On the other hand if `N` and `M` are instantiated and `X` is not, then `X` will be initialized to a matrix with size `N` times `M` consisting of uninstantiated variables.
- `length(X,L)` can be used for both constructing and retrieving the length, `L`, of a list `X` similar to `dim` for matrices.
- `R is X[I,1..M]` sets `R` to the `I`'th row of the matrix `X`. Similarly `C is X[1..N,J` sets `C` to the `J`'th column of the matrix `X`. `R` and `C` will be lists.
- `Y = 1, X = Y+1` sets the variable `X` equal to the expression `Y+1`.
- `Y = 1, X is Y+1` sets the variable `X` equal to 2. Contrary to "=", the expression on the right side evaluated is when using `is`.

The last construct is the loop. We will give several examples to illustrate the different possibilities:

```
(for(I,1,10) do writeln(I))
```

Here the numbers from 1 to 10 are printed by the `writeln` statement.

```
(foreach(X,[1,2,3]) do writeln(X))
```

Prints the numbers 1, 2 and 3 by iterating over the list. `foreach` can also be used for constructing lists:

```
(for(I,1,3), foreach(J,L) do J is I+1), writeln(L)
```

Note that it is not a nested loop, but simply two iterators. Here the list, `L` equals `[2,3,4]` constructed and afterwards `L` is printed outside the loop. Loops can also be used for constructing expressions:

```
(for(I,1,3), foreach(J,L) do J = I+1), writeln(L)
```

Here the list `[1+1,2+1,3+1]` is printed. `I` and `J` are local variables in the loop while `L` is only accessible outside the loop. To make variables accessible within the loop `param` is used:

```
X is 3, (for(I,1,3), foreach(J,L), param(X) do J is I+X), writeln(L
```

Here `X` instantiated to 3 is available inside the loop and the list `[4,5,6]` is printed outside the loop.

3.2 MIP model in ECLiPSe

The ECLiPSe code of the Warehouse Location model is shown in figures 2 to 5. Calling the function `whouse(Cap,D,C,X,Y,Cost)` will return the optimum value in the variable `Cost`. `Cap` is the vector of capacities, cap_j for warehouses $j \in J$ from the model in figure 1, `D` is the matrix, d_{ij}, with distances from customer i to warehouse j and `C` is the vector of warehouse setup costs, c_j. Line 2 returns the dimensions of the matrix `D`, such that `NCust`$=|I|$ and `NWh`$=|J|$. `NWh` is used for initializing the vector `Y` of y_j's in line 3, and `NCust` and `NWh` are used for initializing the matrix `X` of x_{ij}'s in line 4. In line 5 `cross_prod(1..NCust,1..NWh,Pairs)` returns the list `Pairs` of pairs `[I,J]` for all values of `I` and `J` in the intervals $1, ..., NCust$ and $1, ..., NWh$ to be used later.

```
 1: whouse(Cap,D,C,X,Y,Cost) :-
 2:        dim(D, [NCust, NWh]),
 3:        dim(Y,[NWh]),
 4:        dim(X,[NCust,NWh]),
 5:        cross_prod(1..NCust,1..NWh,Pairs),
```

Figure 2: Warehouse Location MIP model in ECLiPSe – Data.

In figure 3, the variable domains are defined. Line 6 creates a list `YList` of y_j variables, which are defined to be binary variables in line 7. When solving with an external solver we have indicate that integer variables should be used with the call, `eplex: (integers(YList))`. Otherwise the LP-relaxation is solved instead, which can be useful in other circumstances. In line 9 a list `XList` of x_{ij} variables is extracted from the matrix `X`. In line 10 the variables are again defined to be binary. Here we do not need to to use integer variables, since the constraint matrix is unimodular.

```
 6:        YList is Y[1..NWh],
 7:        YList::0..1,
 8:        eplex: integers(YList),
 9:        matrix_to_list(X,XList),
10:        XList::0..1,
```

Figure 3: Warehouse Location MIP model in ECLiPSe – Variables.

In figure 4 we set up the constraints corresponding to (6), (7) and (8). First we loop over the customers where the variable `I` takes values from 1 to `NCust`. Line 14 constructs a list of x_{ij} variables for row i or here `I`. Line 15 sets up a constraint by summing the variables in the list and requiring this sum to be 1. Lines 16 to 18 iterates over all customer-warehouse pairs setting up constraint (7). Finally for each warehouse j, here `J`, row j is extracted from the matrix in line 21 and constraint (8) is set up in line 22.

In lines 23 to 26 of figure 5 the customer part of the cost function is collected in the list `CCost` by iterating over all customer-warehouse pairs. Similarly, the warehouse setup costs are collected in lines 27 to 29 and stored in the list `WCosts`. In line 30 the objective function is finally set up and in line 31 solved by calling the external solver with `optimize`.

It is clear that the above program is not as compact as the mathematical formulation or a formulation in another modelling language as for instance GAMS, but it is still relatively short considering it being written in a programming language.

3.3 CLP model in ECLiPSe

Now we set up the same problem as a CLP model shown in figures 6 to 9. Instead of using binary variables, y_j's, we can think of choosing a *set* of warehouses to open. In line 3 we initialize a variable, `Y`, with `intset` to be a *set* with numbers between 1 and `NWh`.

```
11:        (for(I,1,NCust),
12:         param(NWh,X)
13:        do
14:         XI is X[I,1..NWh],
15:         eplex: (sum(XI) =:= 1)
           ),

16:        ( foreach([I,J],Pairs),
17:          param(X,Y)
           do
18:          eplex: (X[I,J] =< Y[J])
        ),

19:        ( for(J,1,NWh),
20:          param(X,NCust,Cap)
        do
21:          XJ is X[1..NCust, J],
22:          eplex: (sum(XJ) =< Cap[J])
        ),
```

Figure 4: Warehouse Location MIP model in ECLiPSe – Constraints.

Instead of the binary variables, x_{ij}, specifying if customer i is serviced by warehouse j, we introduce variables for each customer taking values corresponding to open warehouses. The variables are initialized in the list `XList` in line 4 and in lines 5-7 they are constrained to take values `in` the set `Y` of open warehouses. We emphasize that `in` is a constraint, which could be mathematically stated like $x_i \in Y$, where x_i is the index of the chosen warehouse of customer i in the set-variable `Y` consisting of indices corresponding to open warehouses.

Note that instead of supplying the warehouse capacities in a vector, we now supply them in a list named `CapList`.

In lines 8-13 of figure 7 the capacity constraints are set up. We introduce the auxiliary variable `CapVar` in line 12 for each constraint. In line 13 we use the occurrences constraint, which constrains `CapVar` occurrences of warehouse `J` in the list `XList`. In line 14 we add the redundant constraint making the sum of `CapVar`'s equal to the total number of customers. As we will see later redundant constraints can have a significant impact on CPU times.

To get the cost of supplying the customer from the chosen warehouse, we use the constraint `element` in line 20. `CostList` is the list of costs for the

```
23:        ( foreach([I,J],Pairs),
24:          foreach(CCost,CCosts),
25:          param(D,X)
           do
26:          CCost = X[I,J]*D[I,J]
           ),

27:        ( for(J,1,NWh),
28:          foreach(WCost,WCosts),
             param(C,Y)
           do
29:          WCost = Y[J]*C[J]
           ),

30:        Objective = sum(WCosts)+sum(CCosts),
31:        eplex: optimize(min(Objective),Cost).
```

Figure 5: Warehouse Location MIP model in ECLiPSe – Objective.

given customer and `element(X, CostList, CCost)` constrains `CCost` to take the `X`'th element of the list `CostList`. The costs are collected in the list `CCosts` as specified in line 17.

The costs for opening the warehouses, `WCosts`, defined by the set `Y`, is calculated with the `weight` constraint in line 21, where `C` is the vector of setup costs. Basically the weights corresponding to the set `Y` are summed in `WCosts`. For instance if the set is $\{1, 3, 7\}$, then the weights with index 1, 3 and 7 are added together.

Remaining is now to specify the search procedure in figure 9. In line 24 `minimize` is called. It is a Branch & Bound procedure available in ECLiPSe. The procedure does not use a lower bound, which means that the performance of the search can be very much dependent on, in which order we consider variables and values. Therefore, in line 23 we call a procedure, which for each customer sort the warehouses in increasing distance from the customer to warehouses. In line 26 we specify how to search the set of warehouses to open. `increasing` indicates that the cardinality of the sets we try are increasing, i.e. first we try sets with one element, then two, etc. In line 26 we call the search procedure shown in lines 28 to 32. Here the sorted list of warehouse id's are used to select values of `X`. `member(X, OrderedWarehouseIds)` basically iterates through the different values `X` can take.

```
 1: whouse(CapList,D,C,XList,Y,Cost) :-
 2:       dim(D, [NCust, NWh]),
 3:       intset(Y, 1, NWh),
 4:       length(XList,NCust),

 5:       ( foreach(X, XList),
 6:         param(Y)
          do
 7:         X in Y
          ),
```

Figure 6: Warehouse Location CLP model in ECLiPSe – Variables.

```
 8:        ( for(J, 1, NWh),
 9:          foreach(CapVar,CapVars),
10:          foreach(Cap,CapList),
11:          param(XList)
           do
12:          fd: (CapVar :: 0..Cap),
13:          occurrences(J,XList,CapVar)
           ),
14:        sum(CapVars) #= NCust,
```

Figure 7: Warehouse Location CLP model in ECLiPSe – Constraints.

3.4 Comparison of CLP and MIP

People from the CLP community would now argue that the CLP formulation is "closer" to how you would think of the real-life problem. OR people would argue instead that MIP models are just as easy to construct with a bit of modelling practice.

Another issue comparing CLP and MIP is performance. We have tested the two models on a small example with 19 warehouses and 20 customers for different warehouse capacities. We have also tested different formulations where extra redundant constraints are added in order to improve performance. The following different models are tested:

MIP-S: Formulation in section 3.2.

MIP-LB: The following trivial lower bound on the number of open ware-

```
15:         ( for(I, 1, NCust),
16:           foreach(X, XList),
17:           foreach(CCost, CCosts),
18:           param(Y,D,NWh)
          do
19:           CostList is D[I, 1..NWh],
20:           element(X, CostList, CCost)
          ),
21:         weight(Y,C,WCosts),
22:         Cost #= WCosts + sum(CCosts),
```

Figure 8: Warehouse Location CLP model in ECLiPSe – Objective.

```
23:         order_warehouses(D, CustOrderedWarehouseIds),
24:         minimize((
25:             insetdomain(Y, increasing, _, _),
26:             labeling(XList, CustOrderedWarehouseIds)
27:         ), Cost).

28: labeling(XList, CustOrderedWarehouseIds) :-
29:         ( foreach(X, XList),
30:           foreach(OrderedWarehouseIds, CustOrderedWarehouseIds)
31:         do
32:           member(X, OrderedWarehouseIds)
          ).
```

Figure 9: Warehouse Location CLP model in ECLiPSe – Search.

houses is added:

$$\sum_{j \in J} y_j \geq \left\lceil \frac{|I|}{\max_{j \in J} cap_j} \right\rceil \tag{11}$$

MIP-W: It is well known that

$$\sum_{i \in I} x_{ij} \leq cap_j y_j, \quad j \in J \tag{12}$$

is a weaker formulation than (7), but fewer constraints are necessary.

CLP-R: Formulation in section 3.3.

CLP-RLB: CLP-R with addition of same lower bound as for MIP-LB.

CLP-LB: As CLP-RLB without the redundant constraint in figure 7:

```
sum(CapVars) #= NCustomers.
```

CLP: As CLP-R without the redundant constraint

The tests were done with ECLiPSe 5.5 and XPRESS-MP 13.26 and the results are shown in table 1. All warehouses have the same setup cost and each column reports the results for a given capacity – the same for all warehouses. The necessary number of warehouses are then 2, 3, 4, 5, 7 and 10 for the given capacities. The cost matrix can be supplied by contacting the author.

The results of the aggregated and the stronger version of the MIP formulation are quite surprising. The weaker formulation is much faster on the difficult instances with many warehouses. The explanation is most likely that that the y's are less fractional with many warehouses and the LP problem easier to solve with much less constraints compared to the strong version. Adding the redundant lower bound on the number of warehouses improves the performance significantly.

This is also the case for the CLP formulation. Here the lower bound is the difference making the problem solvable at all. Redundant constraints are often not necessary in MIP models, but for CLP they are. Experiments are as always needed to conclude if the constraints have any effect or if they are useless and even lead to increased runtimes. The other redundant constraint is not as powerful as the lower bound and the combination of the two is worse than the lower bound alone.

The reason the lower bound constraint is so effective, is the way the different sets of open warehouses are tried; first all sets with one warehouse open, then all sets with two open warehouses, etc. For each of the tried sets with too few open warehouses, customers are anyway allocated to warehouses, but without success.

We have earlier mentioned the importance of trying values in a specific order. In line 23 of figure 9 the warehouses are sorted in order of cost for each customer, which means that when trying to assign warehouses to customers it is done in increasing order of the cost. Without the ordering we get the runtimes shown in table 2, which are significantly higher than CLP-LB.

Earlier, we mentioned that branching should be done on important variables first. In this case the most important is, what warehouses to open. To illustrate the impact of making the right choice, we have tried the opposite order with the results shown in table 3.

Capacity	10	7	5	4	3	2
MIP-S	0.2	0.3	0.3	0.6	7.7	42.2
MIP-LB	0.2	0.2	0.1	0.1	0.1	0.4
MIP-W	0.2	0.1	0.1	0.1	0.2	0.1
CLP	4.5	318.2	-	-	-	-
CLP-R	4.7	8.8	19.3	50.0	251.3	1081.3
CLP-LB	4.5	7.9	14.8	34.0	126.8	566.8
CLP-RLB	4.7	7.9	15.2	35.0	132.9	585.8

Table 1: Comparison of CLP and MIP for the Warehouse Location Problem.

Capacity	10	7	5	4	3	2
CLP-LB	4.8	8.4	25.14	43.0	158.8	839.3

Table 2: CLP-LB with no ordering

The basis of the comparison is quite limited – only one quite small and simple prototype example. The intention of the comparison was however not to make the definite choice between MIP and CLP, but to merely illustrate some points.

On this instance of the problem MIP is superior to CLP, but it is not fair to generalize the result to other problems. Problems for which there is a large gap between the upper and lower bounds are not well-suited for MIP. This for instance is the case when disjunctive constraints are necessary in scheduling and packing problems. Here CLP looks particularly promising.

4 Conclusion

We have in this article described CLP from the view point of MIP. The two paradigms have many similarities, but also the potential to complement each other and create even more powerful *hybrid solvers*.

We gave an example of modelling and solving the Warehouse Location Problem in the ECLiPSe environment with both MIP and CLP. We argued that MIP is superior when linear models can be formulated where the gap

Capacity	10	7	5	4	3	2
CLP-LB	130.5	456.7	-	-	-	-

Table 3: CLP-LB with branching on customer variables before warehouses.

between upper and lower bounds is relatively small. CLP lacks the ability to use bounding to prune the tree, but can instead use domain reduction to reduce the size of the search tree.

Currently much interesting research is focused on using methods originating from the OR area to do domain reduction in CLP and as well using domain reduction in Branch & Bound. A definite conclusion is not yet near regarding, which components should enter into the future solver of NP-hard problems.

References

[1] A. M. Cheadle, W. Harvey, A. Sadler, J. Schimpf, K. Shen, and M. G. Wallace, *ECLiPSe – An Introduction*, IC-Parc, Imperial College, London, August 2002.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer, 4. ed., 1994.

[3] IC-Parc, *ECLiPSe Constraint Library Manual*, Imperial College, London, 2002.

[4] K. Marriott and P. J. Stuckey, *Programming with Constraints – An Introduction*, MIT Press, 2. ed., 1998.

Paper VII

CIAMM: A Knowledge Driven Methodology for Development of Combinatorial Optimization based Information Systems

CIAMM: A Knowledge Driven Methodology for Development of Combinatorial Optimization based Information Systems

Andrea Carugati, IPL, DTU
Jesper Hansen, IMM, DTU
June 2003

Abstract

This paper presents a methodology for information systems development (ISD) specifically designed for the development of combinatorial optimization (CO) based systems.

The usual approach in ISD is to invest significant resources in the beginning of the project in order to reduce the risk of solving the wrong problem. However this approach fails to consider that, in developing CO based IS, it is difficult for the users to specify what they request from the IS until they actually see it in action. Therefore for CO based IS the initial system analysis and design will always be inaccurate and incomplete.

Experiences in developing CO software systems point to two important factors differentiating projects with CO from other ISD projects:

1. Knowledge exchange between developers and users is particularly difficult due to the complexity of the CO technique.
2. Software based on CO is difficult to divide into smaller development tasks to be used as prototypes.

We therefore propose a methodology based on an agile approach focused on the facilitation of knowledge exchange between developers and users. This is achieved through frequent encounters focused on the discussion of a model, a prototype, or a release that act as boundary objects during the discussion. Frequent encounters require short development cycles and therefore create the need to subdivide the CO development task is subdivided into smaller tasks.

Developing CO based systems normally requires long periods of non-interaction due to the complexity of developing the CO core. We propose a way to subdivide the development of CO systems into smaller tasks that present increasing level of complexity. These smaller tasks are manageable in relatively short periods of time hence making possible the use of the ideas behind agile methodologies.

High frequency poses a limit to the increments in software complexity therefore allowing the users to comment at full capacity on the work done by the developers. The methodology focuses on making the users able to give well grounded reasons when they ask the developers to change the software or extend it.

This openness to change in goals and objectives is reflected in the methodology in the fact that the idea of a well-defined initial definition of the project goal is not pursued, but rather it is recommended to allow for an emergent process of goal determination within a larger interest area. The specific path followed to reach the final goal emerges during the execution of the project according to the decisions taken collaboratively by the groups of developers and users and is hence impossible to define in advance.

The users-developers interaction is carried out at fixed rhythm to make sure that the project is keeping on going in the right direction and to keep the users motivated in giving their feedback to the developers.

Finally, the process, focusing on short concentrated meetings, is designed not to interfere with the users' daily work, hence making the methodology feasible in most users' environments.

Index

Abstract ii
1 Introduction 1
1.1 Assumptions and Target Group 2
1.2 Structure of the paper 3
2 Combinatorial Optimization Problems 4
2.1 Applications of CO 5
2.2 Embedding CO in Software 5
3 The Concept of Models 6
3.1 Models in Software Development 7
4 Methodologies and Goals 8
4.1 Basic Ideas 10
4.2 The Evolving Nature of the Project Goals 11
4.3 Knowledge Exchange 13
5 CIAMM Methodology 15
5.1 The Business Problem 18
5.2 Conceptual Object Model and Usage Model 18
5.2.1 The Usage Model 18
5.2.2 The Conceptual Object Model 19
5.2.3 Supplementing the Models for CO Development 20
5.2.4 Summary and Final Remarks 21
5.3 Iteration and Release Planning 21
5.4 Modeling, Design and Implementation 23
5.4.5 Implementation Issues 25
5.4.6 Improving optimization methods 25
5.4.7 Test and Validation 26
5.5 Evaluation 26
5.6 Integration 26
6 Project Managers: Users and Developers 27
6.1 Users' Manager 27
6.2 Developers' Manager 28
7 Summary and Conclusions 29
Appendix 1: Experiences From the Inventory Case 31
References 33

1 Introduction

A high failure percentage in information system development (ISD) projects is a well-known problem. A large number of different processes or *methodologies* have been proposed to reduce the risk of ending up with a failed project. A methodology can be described as a suggested way of doing things – a description of a process, which in this case is the process of developing software [Pries-Heje 1996]. The most widely used software development methodologies are not designed specifically for particular software techniques but are designed for general applicability. In this paper we consider a special type of system development projects where there is a potential gain by including combinatorial optimization (CO) techniques. We discuss the issues differentiating these projects from usual ISD projects and propose a methodology that specifically supports the development of CO based systems.

Throughout the paper we refer to experiences gained from participation in a research project involving the development of an information system with a substantial CO component. The research project was called CIAMM (Center for Industrialized Application of Mathematical Models), and was initiated with the purpose of developing theory, methods and tools to contribute to the diffusion of information systems with CO in the manufacturing industry. One of the cases in the project was carried out at a shipyard needing software for the scheduling and control of their steel plate inventory. The project involved team members from several research institutions and a consultancy company [Carugati 2002, Hansen and Kristensen 2003a-c]. Since the project acronym is CIAMM, the methodology will be called the CIAMM Methodology. The content of the methodology is based on empirical research carried out in the period 2000-2003 on the basis of the inventory case [Carugati 2002 (a, b, c), Hansen and Kristensen 2003(a-c)].

In the paper the expression *information system* is used in a broader sense than just a synonymous of software. Information system is intended as the system comprising of people and technologies, hardware and software, constructed with the purpose of improving some sets of organizational activities.

The CIAMM development case highlighted that CO based IS present additional challenges with respect to traditional software development processes. Research in techniques of CO has primarily been developed in theoretical settings where the use of simplified models of complex phenomena has been sufficient to study the characteristics of the techniques. This simplification does not work in industrial settings where complex factors are an essential part of the problem [Brooks 1987]. Second, in industrial settings it is essential to use IS prototypes to elicit requirements from the users [Brooks 1987], but CO is not a technique that easily lends itself to breakdown into independent development efforts and therefore it is not suitable for rapid prototyping. Third, to be able to use the techniques it is necessary that users and developers reach a high reciprocal understanding of each others' problems and possibilities. CO methods are very complex for the expert to explain and for the users to understand. At the same time the developers need a very detailed knowledge of the users' system to exploit the techniques, but this is also difficult to obtain because of the high degree of tacit knowledge usually present in the users' environment. While this last issue is true for any ISD case, it is particularly damaging for systems based on CO because these systems function only if the model of the users' environment is well described in the code.

To support the development of CO based IS this paper presents a methodology that is partly based on established methodologies and terminology, but which also tackles the new problems specific of CO. Firstly, the methodology focuses on creating a viable process that facilitates knowledge exchange among projects participants. Secondly, different methods will be proposed to divide the development task in subtasks that can take form of either prototypes or finished products, hence making it possible to have frequent demonstrations and releases supporting the dialog between

developers and users. In order to make clear when the methodology can be profitably applied, the next paragraph makes explicit the context of application of the methodology.

1.1 Assumptions and Target Group

One of the characteristics of commonly used ISD methodologies is that they are proposed for general validity and the basic assumptions about organizations and technologies on which they rest have become black-boxed and the application of one or another methodology in a particular development group becomes almost axiomatic.

However the discussion of assumptions, meanings and expectations present in the contexts in which methodologies are produced and applied is very important to understand whether a specific methodology is adequate for a specific development case. The experience from the inventory case has shown that the axiomatic and unreflective use of methodologies can hide unexpected traps.

This paragraph presents the assumptions at the base of the methodology presented in this paper. The methodology resulted from the experiences of the inventory case and therefore it follows that the assumptions for its use are also to be found in the development context of the case.

The development of the information system was carried out in an organization where the members were working from different locations. The locations were several kilometers apart but there was not time-zone difference between team members.

The team members were formally employed in different organizations and had responsibilities both for the inventory case but also for their organizations. At times the requirements of the organizations and of the development project would interfere causing the team members to make choice on priorities.

Each member of the development team did mainly tasks pertinent to his/her specific interest area and the interest area of the employing organization.

The span of control of the project manager of the development team was relatively limited because the team members were working for different companies. Control was achieved through negotiation.

The developers had a deep theoretical knowledge of the CO technique. Knowledge of programming was necessary but subordinate to knowledge of CO. There was no initial knowledge of the customer's site or of the customer's problem.

The customer was totally new to the CO technique. The customer knew that the situation was not optimal and that CO could help in changing the situation but they were not aware of the total possibility of the technique. The customer was very knowledgeable of his environment and of its constraints and possibilities.

The organizational task to be supported by the IS was complex because neither the customers nor the developers had an exact idea of what the processes would have looked like when the IS would have been finished.

To summarize, the development context for this methodology involved a development organization, which was distributed, loosely coupled and where the span of control of the project manager was limited. Furthermore the developers knew about the technology but not about the customer's system and the customers knew their problem and believed that CO could help to solve it, but had no idea of the real possibility of the technique. Consequently the definition of the problem was unclear and open to discussion. All these conditions where in the minds of the authors during the writing of this paper and are addressed explicitly or implicitly in the methodology.

Results from the inventory case presented in another case [Carugati 2002b] show that this methodology is necessary when the development organization is distributed and loosely coupled. However it can be contended that the methodology would work well also in case of co-located and hierarchical organizations or other combinations.

The methodology is recommended for development projects where the technique employed, CO in our case, is new for the customer organization. In these cases a high degree of reciprocal learning is required: learning about the problem domain for the developers and learning about the technique for the customers. The methodology is, on the other hand, not recommended if both developers and customers are knowledgeable about each other's areas. In these cases more structured approaches, e.g. the waterfall model, are recommended.

It is hoped that these explanations can help a project manager to answer the three questions:

Is this methodology suitable for our development organization?

Is this methodology suitable for our customer organization?

Is this methodology suitable for the technology on which the system under development is based?

These are the three basic questions that should be addressed by any group at the beginning of a ISD project and that far too often are instead neglected because of traditions, habits, or rush to begin.

1.2 Structure of the paper

The paper is structured in the following way:

Chapter 2 – describes CO problems and introduces different types of optimization methods to solve these types of problems. Further, problems solvable with use of optimization techniques are introduced, hereby making it possible for non-experts to identify potential business cases.

Chapter 3 – presents the concepts of models and the models used in ISD.

Chapter 4 – gives an overview of some of the existing methodologies and presents the basic ideas on which the methodology is based. In particular the definition of the business problem and the issues of knowledge exchange including the difference in using knowledge exchange vs. system success as the goal for ISD. These ideas emerged during the empirical investigation of the inventory case.

Chapter 5 – presents the methodology and elaborates on the basic ideas introduced in chapter 4. The framework of the methodology is an iterative process that forces frequent interaction between developers and users. Going through the process many times and each time discussing prototypes of a small part of the IS facilitates a debate, which is focused and at the same time keeps every stakeholder informed of the actual state of the technology and the users' system. The chapter focuses on the issues specific for developing CO based IS.

Chapter 6 – considers the main managerial challenges for using the methodology. The focus is essentially on the use of the methodology in organizational settings accustomed to more structured approaches.

Chapter 7 – gives a summary and the conclusions.

Appendix 1 – lists many of the experiences of the development case at the shipyard, their consequences and the way in which the methodology is going to affect them.

2 Combinatorial Optimization Problems

In this section we introduce Combinatorial Optimization (CO) problems and issues to consider when dealing with Information System Development (ISD) projects with a key content of CO. There is no single definition of CO, but in the following we describe the main elements. One or several objectives are given, which we want to minimize or maximize. The set of solutions is discrete and countable. The problem sounds like a simple one to solve: Try all combinations and pick the best of the feasible solutions. Assume that our problem is to find the best sequence of only 10 tasks, for instance at a production line. This results in evaluating 10! = 3.628.800 potential combinations referred to as the solution space. Assume that we, by use of a computer, can evaluate as many as 1.000.000 orderings per second. Evaluating all solutions would then take only 3.6 seconds. Now consider the case where we double the number of tasks to 20, resulting in 20! sequences. For this problem it would take more than 77000 years to find the best sequence! This phenomenon is called the *combinatorial explosion*.

In most cases we are satisfied with "good" solutions and not necessarily the proven optimum. Fortunately, search methods exist, which investigate only promising regions of the solution space, resulting in good solutions within reasonable computation time. These methods are called *heuristics* and do not guarantee that an optimal solution is found. *Exact methods,* on the other hand, guarantee to find an optimal solution. These methods search the entire solution space, but can often deduce that the optimum is not in certain regions and hence only consider these implicitly. The problem with exact methods is that they require long computation time but these methods can also be used as heuristics to search for good solutions only doing a partial search of the solution space in a limited time span. The search will hence not necessarily reach an optimal solution and the possible optimality of the solution cannot be proved. In the following we discuss the components of a CO problem.

The *output* of a CO problem is a set of decisions. For instance in "Project Selection", we must decide, which of say 20 projects to initiate. This is basically 20 yes-no decisions leading to 2^{20} =1.048.576 different solutions, of which of course many most likely are infeasible or very poor solutions. We typically model yes-no decisions with use of *binary variables*, where 1 means yes and 0 means no.

The problem owner should decide which of the potential solutions are preferred according to his preferences or *objectives*. These objectives must then be translated into a mathematical function, which can be maximized or minimized according to his preferences. Assume that each initiated project in the Project Selection problem has a value to the problem owner, which is independent of which other projects are initiated. The objective function is then the sum,

$$\max \sum v_i x_i$$

Here v_i is the value of project i and x_i is a binary variable, which is 1 if project i is initiated and 0 otherwise. In most cases some of the solutions in the solution space defined by the decision variables are not feasible. The values of the variables are *constrained.* Perhaps a limited budget is available for initiating the projects resulting in the following constraint:

$$\sum c_i x_i \leq b$$

Here b is the budget and c_i is the cost of project i. Further, assume that project 1 and 2 cannot both be initiated. The constraint $x_1 + x_2 \leq 1$, will remove that solution from the feasible set of solutions, since only one of the variables can be 1. This is a so-called "hard" constraint, which **must** be fulfilled. The number of variables and constraints represent the *problem size*.

"Soft" constraints are constraints that are **preferred** to have fulfilled, but where for instance few or small violations are accepted. Soft constraints are added to the objective function as a penalty:

$$\max \sum v_i x_i - p(\max\{0, x_1 + x_2 - 1\})$$

If both projects 1 and 2 are initiated a penalty factor *p* is subtracted from the original objective function.

To summarize, we focus on CO problems where the goal is to optimize one or several objectives. Typical problems are combinations of yes-no decisions, which are mutually dependent or constrained. Determining a sequence of tasks is also a typical example.

2.1 Applications of CO

A large number of problems originating in real-life can be solved with CO. Below we show some examples of the most common problem types occurring in industry:

1. **Cutting Stock** and **Nesting Problems:** A set of items is to be cut out of larger objects in 1 or 2 dimensions, minimizing costs on raw materials.
2. **Bin Packing Problems:** A set of 3 dimensional items is to be packed in a minimal number of larger bins. This is a typical problem in container and pallet loading.
3. **Production Mix:** Given products to produce, availability constraints and costs on raw materials and machine resources, determine the production plan minimizing costs.
4. **Project Planning:** Given a set of projects with expected profit and cost, maximize expected profits.
5. **Production Sequencing:** Given a set of products to be produced at an assembly line. Determine the optimal sequence with different constraints on the sequence.
6. **Job-shop Scheduling:** Determine the optimal sequence of operations or tasks on a given set of machines in a job-shop.
7. **Staff and Shift Scheduling:** The demand for staff with certain qualifications over a period of time, possibly divided into shifts is given. The problem is to create an optimal working schedule for the staff covering the demand for the period.
8. **Crane/Robot and Vehicle Routing/Scheduling**: Determine the optimal route/schedule performing all tasks or visiting all customers.
9. **Facility Location**: Given a set of potential facility locations and customers with demands, decide in which location to open a facility in order to minimize the cost of opening facilities and delivering goods to customers.

Problems in real-life often have some characteristics in common with the above problem types. Perhaps the problems are a combination of the above types or have additional constraints. The purpose here is only to give a flavor of the type of problems we are considering.

With limited knowledge of CO it can however be a difficult task to identify a problem as belonging to the class of CO problems. In general, the possibilities of using CO should be investigated for problems where a solution is to be chosen among a large set of possible solutions - remember the combinatorial explosion of solutions.

2.2 Embedding CO in Software

In most cases the optimization task is only a subpart of a larger IS. For example, in the inventory case, the CO algorithm was used to generate a sequence of operations to be used as support for decision making for the operators in the inventory. The problem solved with CO is only a subset of the business problem, which is managing all activities in the inventory. CO software requires input data about parts, resources, costs, gains etc. and delivers an output list where a sequence of

operations is specified. Interfaces with other systems and users are required to make the optimization possible and useful for the user. Figure 2.1 shows the situation when considering CO for optimization of production scheduling.

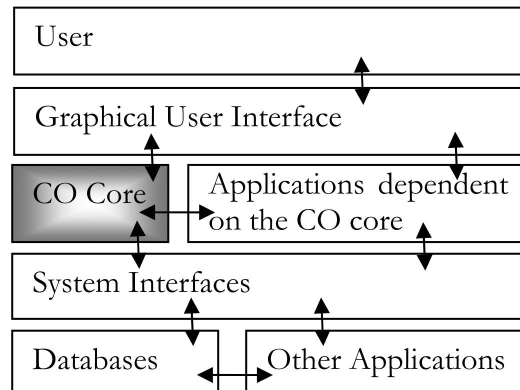

Figure 2.1. Combinatorial Optimization and other system components

The CO algorithm belongs to the core of the system. It takes inputs from the company databases and the production plans through system interfaces. It also takes input from the users through a graphical user interface. Possibly there could be other applications that interact with the CO algorithm. These applications could for example manage the request of an extra order or track orders on the production floor. Then the CO algorithm returns output data to the databases, applications and users. Since the CO algorithm is embedded in the overall system, we use the term *optimization core* to define the specific part of the system that implements the optimization methods for solving the problem.

Thus the *optimization core* must have access to relevant data and after finding a satisfactory solution, it returns this to the system for further processing. The main technical objectives in projects including CO are the following:

1. Modeling a given optimization problem. Designing and implementing an optimization method to solve the model.

2. Specifying and implementing the interfaces between the CO core and the rest of the IS.

Achieving the technical objectives is not a trivial task and it often takes a significant amount of time, but even then, the main problem that has been evidenced through the inventory case is to define in detail the business problem and exchange the relevant knowledge between developers and users [Carugati 2002].

3 The Concept of Models

In this section we introduce the concept of a model, since models are a significant part of ISD. A model is an abstract representation of reality in any form (including mathematical, physical, symbolic, graphical, or descriptive form) to present a certain aspect of that reality for answering the questions studied [def. ISO 15704, GERAM]. Within a system development methodology there are multiple purposes for building models: Models help people to exchange knowledge [Carlile 2002] and develop shared understanding of the problem setting, to coordinate their action and to facilitate communication [Wand et al. 1995]. Models can also be used for documenting collected knowledge or as guidance in project planning.

As a corollary to the intended purposes of models, it is useful to state some of the possible problems in using models. Models are only a representation of reality, but they are not reality. Therefore a model of an IS only represents some aspects of the IS and should not be treated as

the final product. Models are not fixed in time, since they have to be changed as interests, knowledge, priorities, purposes, and assumptions change. The use of outdated models diminishes the ability of the model to facilitate knowledge exchange [Carlile 2002]. Finally, models are built for specific reasons and by specific people to serve, willingly or not, specific interests. Through a representation of specific views of reality, models enable and constrain different ways of acting.

In section 3.1 we discuss models, which can be used to facilitate the process of ISD, minding the pitfalls explained above.

3.1 Models in Software Development

According to Brooks [1987], one of the essential characteristics of software is that it is "invisible and unvisualizable" (page 41) and this makes it impossible to create single and easily understandable models of software (as opposed to physical artifacts that is represented with geometrical models). In order to capitalize on the many advantages of models explained above, software has to be represented via several models superimposed upon each other in order to provide the richness of details that enable shared understanding, coordination of actions and communication between the developers and the users of the IS[1].

The software of the inventory case was developed using the object oriented (OO) paradigm, which is widely used and has proven to be very effective for the development of CO based systems. We will therefore only consider the OO paradigm in the following.

In software development the goal is to develop software solving some problem arising in the real world system. For the software to be useful for the users there must be a relation between the software and the problem it tries to solve. To create this relation it is necessary to achieve an understanding of how the stakeholders of the system perceive the real world. This understanding becomes the basis of the *conceptual object model.*

A conceptual object model includes the *objects* in the real world system, their *relations*, *activities* they perform, *attributes* they posses and possible *states* they can be in. *Processes* are created by sequences of *events* and *activities*. Objects are instances of *classes*. A pen for example is a class, while the red and blue pens lying on my desk are objects. A class is in other words a template or pattern of an object describing classes of objects, which are in some way related. As conceptual object models are focused on modeling objects, they are often referred to as *object-oriented models*.

The *usage model* [Mylopoulos 1992] is a model of how the software system to be developed is going to be used. This includes all interactions with the system by users and other systems through system interfaces.

A *CO model* is basically a logical or mathematical formulation of the CO problem at hand. It is based on the conceptual object model, but it additionally specifies constraints between objects, decisions to be taken and system objectives. A CO model of a CO problem can then be seen as an extension of the conceptual object model. For some of the objects, decisions have to be taken in order to minimize or maximize a mathematical function. The decisions are modeled with variables, which can take values within specified sets of possible values. For instance returning to the problem of selecting projects, each project is an object instance of the class "project". For each object we can decide to select it or not, associating to it a variable, which can take values in

[1] To be true, physical artifacts cannot be fully represented by geometrical models either; for example material properties and surface treatment are not part of geometrical models. However with geometrical models the function of an artifact can be understood intuitively, which is not true when many different models have to be used to explain the function of software.

the set {0,1}[2]. The class has some attributes as well, i.e. the cost of the project and the profit. Note that the CO model is specific for ISD projects with CO.

With the conceptual object model, CO model and usage model, the developers can built a model of the software system to be developed. This is usually referred to as the design phase resulting in a *design model*.

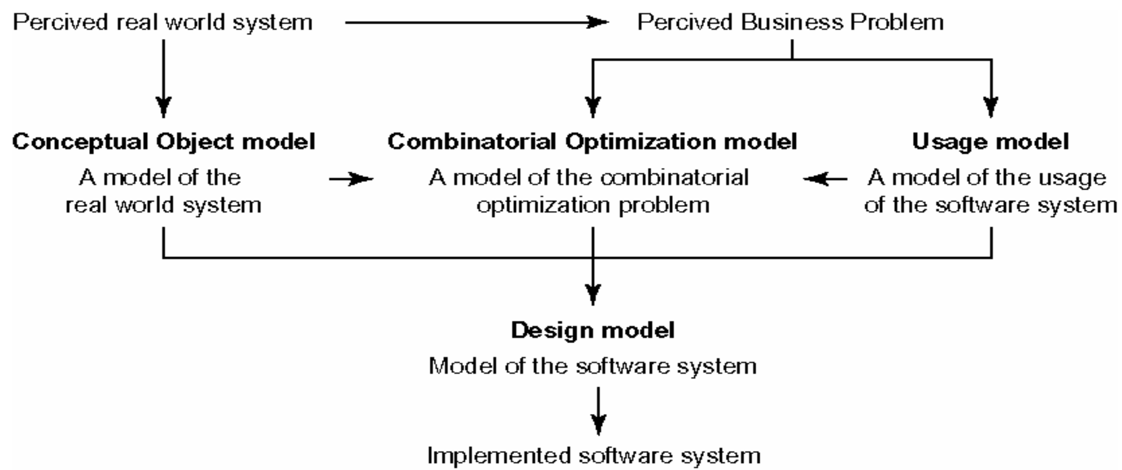

Figure 3.1 Models used for developing CO based IS.

The way the customer perceives the real world is the basis for the definition of the conceptual object model and the business problem. In practice the information collected from the customer will not be focused on one model at a time. The developers will have the task of separating the information into the different models.

Building models requires a modeling language that complements the use of natural language. The standard modeling language within object-oriented modeling is the Unified Modeling Language (UML) [Fowler and Scott 1999], which can be used for building models in software development. The language mainly consists of different diagram types for instance Use Cases for building usage models and the Class, State and Activity diagrams for building conceptual and design models. The Object Constraint Language (OCL) of UML has means to describe constraints on objects and among objects. OCL is however quite laborious to use, which suggests supplementing the constructs of UML with use of natural language and mathematical formulations. Modeling is further discussed in section 5.2.

4 Methodologies and Goals

In this section we describe some of the most commonly used ISD methodologies. These descriptions will not be comprehensive, but the ideas underpinning each approach will be used as source of inspiration for the methodology proposed in the next chapter.

Most methodologies for ISD are conceptually based on the idea that the development process can be organized in stages. These stages describe the evolution of the software over time from being an idea, through design, development and operation. The most basic life cycle stages are known as System Development Life Cycle (SDLC) and comprise: system planning and selection, system analysis, system design, system implementation and operation in Valacich et al. [2001]. This paper follows the ISO 15704 guideline that separates the development stages from the activities to be perform during system development. In ISO 15704 the identified activities are:

[2] Another possibility is to have a separate object representing the set of selected "project" objects.

Identification, Concept, Requirements, Preliminary Design, Detailed Design, Implementation, Operation and Decommission. It is important to discern between stages and activities because knowing which activities are performed in a particular stage helps formalizing the content and extent of the stage itself.

The most commonly used methodology is a linear process in which the SDLC phases are accomplished sequentially with relatively little iteration; this methodology was one of the first to be used and it is called the Waterfall Model [Bohem 1988, Valacich et al. 2001].

Another approach to development is the Spiral Model [Bohem 1988] where the different phases, with focus on requirements definition, are repeated a series of times and includes risk assessment and customers evaluation as check points for taking decision about the continuation of the project. The major focus of the spiral model is on institutionalizing the evaluation of project continuation.

Rapid Prototyping (RP) [Pries-Heje 1996] is another approach that focuses on re-iterations. Mock-ups and simplified versions of the software are prepared to elicit requirements from the users. RP is mostly focused on requirement elicitation. Lacking focus on implementation, RP is mostly used during the requirement definition phase of the SDLC.

Rapid Application Development (RAD) [Valacich et al. 2001] is an approach that focuses on quick re-iteration of the phases of system design and development. RAD emerged from the need for speed requested by the continuously changing environment of modern businesses. RAD is based on high customer participation in the development.

A number of methodologies have emerged from RAD and RP e.g. Dynamic Systems Development Method (DSDM) [DSDM Consortium 2003] and lately, Extreme Programming (XP) [Beck 1999]. DSDM and XP are based on developments of small subsets of functionality that can be integrated in the customer system every three to four weeks. This is made possible by having the customer defining *user stories* or problems that can be tackled separately within short periods of time. The major focus of XP is to constantly provide value through frequent integration at the customer site. The principles of DSDM and XP are very similar. XP is however more focused on programming practices while DSDM is more focused on the customer's role in the development process.

The first two methodologies and especially the waterfall model are also referred to as *heavyweight* methodologies for their relative lack of flexibility in revising the completed phases once a new phase is started. This is because a heavy use of resources is committed to performing every phase in the best possible way and therefore there is lack of motivation to revise the work done. As we move towards less rigid processes we move towards what are referred to as *lightweight methodologies* [Fowler 2000] or more recently *agile methodologies* because of their focus on continually revising the understanding of the problem domain. In this second type of methodology much less weight is given to the planning and analysis phases. Lightweight methodologies are based on the hypothesis, that the environment is changing faster than the system is developed and therefore there is potentially no gain in using months, if not years, on the development of a system, which when completed is based on obsolete analysis and requirements.

In the rest of the paper we focus on a methodology specifically designed for the characteristics of software based on CO. Obviously the development of CO based software is subject to all the problems that plague the development of any other system. As already mentioned, there are issues related to the novelty of the technology that are specific to projects including CO. The Chic-2 project [Chic-2 1999] had a similar focus when developing the Chic-2 Methodology for combinatorial applications. This methodology is modeled after the waterfall model however the proponents of Chic-2 conclude that the CO technique requires an iterative approach to the design, development and validation of a solution. Therefore they propose an evolutionary

approach within the specific stages of the lifecycle. As in the spiral model, the evolutionary process within each waterfall stage allows risks to be taken into account in the management of each stage. Our methodology includes many of the same issues, but for reasons that will become apparent below we choose a more agile approach. We have taken inspiration from the cyclical nature of the spiral model and we capitalize on the use of prototypes and on the ideas of XP, but we propose an alternative way to gain insights from the users without removing them from their daily activities.

4.1 Basic Ideas

Our methodology builds on multiple ideas. These ideas have emerged from the experiences of the studied inventory case. The majority of these ideas are confirmed in results of previous studies while some ideas are original. The full list of problem items of the inventory case can be found in appendix 1. In the following part the problem items have been clustered and converted into constructive ideas:

1. It is a complex activity to **decide the detailed objectives for the IS**. Developers and users have different perceptions of the problem and of the possibilities of the technology. The methodology must account for these perceptions as well as being open to changes in the definition of goals. This problem is, among others, reported in [Checkland and Scholes 1999].
2. **Priority to the solution of the business problem.** This has the priority for the customer. It has to have the same priority for the developers to keep the customer motivated to participate constructively in the project. The hypothesis is that the customer will ask for more releases depending on how satisfactory the previous releases are and how far the customer believes the latest release is from the finished system.
3. **Knowledge exchange** is fundamental in the creation of a satisfactory system. Since the customer's knowledge of CO is usually low and the developers' knowledge of the business problem is limited we have to create a means of knowledge exchange. Knowledge exchange goes beyond information exchange: it entails a reciprocal (customer and developer) understanding of one another's worldview [Churchman 1968, Checkland and Scholes 1999].
4. **Invested knowledge** is a barrier to changing and revising practices for the customer and to rebuilding the software for the developers. It is a known phenomenon that we tend to continue doing our work in ways that we know are not perfect, because we have invested a significant amount of energy in learning how to do it and we do not want to unlearn. In a similar way software developers do not want to discard months' worth of coding because new conditions emerge. The methodology has to ensure that the knowledge invested by both parties in the software is susceptible to change with little inertia [Carlile 2002].
5. **An iterative process based on short iterations** allows for keeping both customer and developers motivated and focused on the solution of the business problem. The hypothesis is that long time between meetings diminishes the users' participation and therefore their involvement. This leads to greater risk of dissatisfaction with the system [Hartwick and Barki, 1998]. Furthermore, the inventory case shows that with low presentation frequency, the developers have too much space for interpretation. The effect is that the software is difficult to understand for the users during the rare presentations.
6. Short time between releases encourages the developers to produce **simple releases:** simpler for the developers to code and for the customer to evaluate. With simple releases, the level of developers' interpretations (e.g. simplifications) is kept low and the customer can more easily comment on them [Carlile 2002]. The challenge in software development with CO is to break the CO part into smaller tasks that the customer can evaluate, while the usual approach is to develop the CO core in one long iteration.

7. Short time between releases allows **regular monitoring on the business validity of the work done and planned**. The customer's environment tends to change over time because of internal and external pressures. Frequent releases put customers and developers close to each other and therefore diminish the risk of working on outdated requirements [Valacich et al. 2001].

8. The customer perceives value only during presentations or after a release. Many activities of data collection have little value for the customer but consume a lot of time. Developers have to focus on activities that **create value for the customer** in order to engage them in continuous and useful debate [Beck, 2001].

9. The CO core is a **black box** for the user. It takes data and produces an output in a way hidden from the user. The user has to trust the system to be satisfied with its use … or to use it at all. Frequent iterations with presentation of simple parts of software make the users active participants in the creation process and they are therefore more willing to trust the output of the system as a product of their own work [Hartwick and Barki, 1998]. This again calls for opening the black box, splitting the development task into smaller pieces that can be visualized and explained for the customer.

In sections 4.2 and 4.3 we discuss the *evolving nature of the project goals* and *knowledge exchange* since we find these to be the fundamental elements in shaping developers' and customers' worldviews. The issue of decomposing the CO core into smaller components suitable for development in short and simple iterations is discussed in sections 5.3 and 5.4.

4.2 The Evolving Nature of the Project Goals

The ultimate goal of any software project is to bring value to the customer. Value for instance through increased productivity for the users of the software and the entire organization, new possibilities for improving processes in the organization, better planning of activities, etc. Customer value will be central in every aspect of the proposed methodology.

The main requisite for delivering value is that the IS works as required by the customer. The problem is that very often it is difficult for the customer to define in detail the requirements of an IS at the beginning of a project. The reason for this is that when dealing with a new technology, like CO, the possibilities granted by the technology are not clear to the customer until they can see the system in operation.

At the same time developers often do not have sufficient knowledge of the application domain and the knowledge must be gained from the users and domain experts within the customer organization in order to clearly understand goals and objectives for the IS. A dialog between the developers and the customer is required for the customer to externalize their goals and objectives and for the developers to successfully capture the requirements and model the customer's problem. This process is shown in figure 4.1, where goals and objectives of both sides change as a constructive dialog between developers and customers is established.

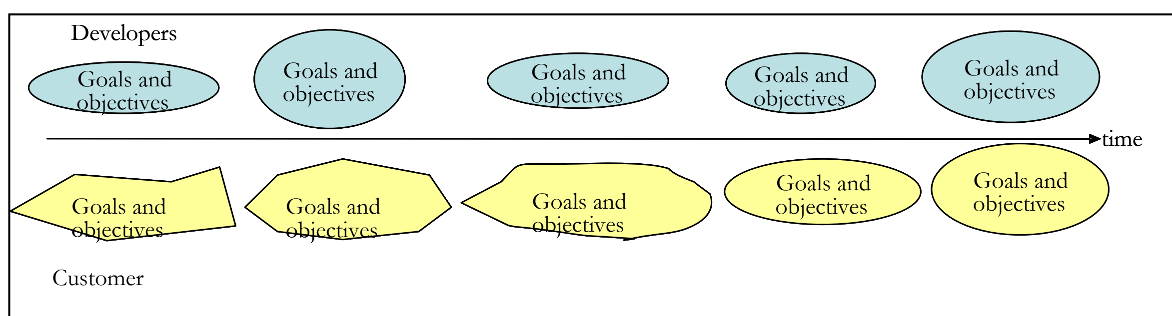

Figure 4.1. Evolution of goals and objectives for customer and developers

Despite the difficulties with the changing nature of goals and objectives, at the beginning of an ISD case, the customer is usually able to describe the business problem, which is more generic than the specific IS functionality. With reference to figure 4.2, the general business problem is called the *outcome space* [Remenyi et al. 1997] of the IS. Throughout a development project, goals for the IS are refined and objectives become more detailed as explained above. This idea is showed graphically in figure 5, where the arrows 1, 2, 3, 4, in the cone represents systems development efforts concluded with a prototype demonstration. The numbers 1, 2, 3 and 4 in the outcome space correspond to the specific goal for the IS during the respective efforts if the development had continued without changing course.

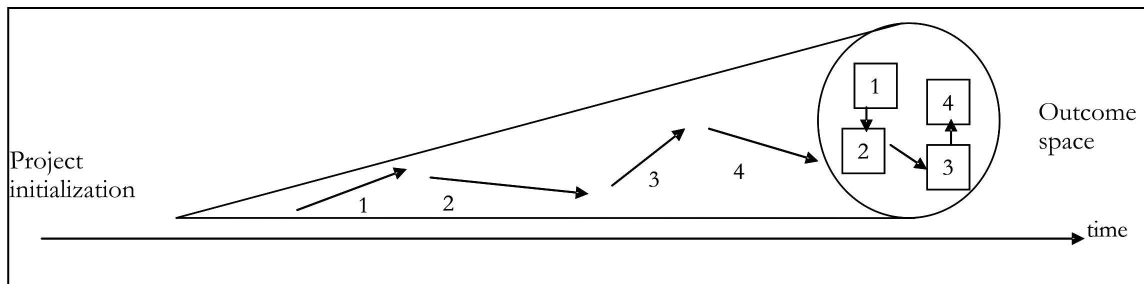

Figure 4.2. Outcome Space

The conical form of the development project is not indicating that more freedom is obtained the further we move in the development case, but rather that there is a boundary within which the project continues and outside where the project might be stopped. The open shape of the cone also indicates that the more the stakeholders know about the possibilities of the technology, the more precisely they can shape it to fit their needs.

A typical problem in the development of large IS emerges when the requirements, albeit agreed on by both parties, are treated as static and used throughout the development project. In this case the system provider develops the system with little contact with the customer and in many cases the result is not what the customer expected in the first place. This might result in a dispute between the parties on whether the system reflects the initial requirements or not.

1. From the Diary of the Inventory Case

The developers and the shipyard stakeholders used the first five months of the project to discuss the detailed requirements of the IS, but the programming activities were not yet started. After one more meeting the following lines were written by the project manager in an email to the development team:

"I think that we have now "milked" the shipyard people for information and thoughts about the inventory to such a degree that nothing more comes out! Furthermore I think that the information that we have now are not much more valuable than what we had [3 months ago]. The next step is therefore that we use our skills and the information in our possession to show how the problem can be solved.

It is my hope that this new suggestion to organize our work will succeed because I seriously think that this is our last chance to maintain the respect of the shipyard and get some solutions delivered to them! Otherwise there is a risk that they will totally lose faith in our ability to ever finish the job and as they said: "It might be that we can solve the problem better on our own!"

In the inventory case, after six months of data collection and many meetings to determine the system requirements there was still debate on what the system should do. The shipyard considered this situation very negatively and consequently the project manager of the developers mandated the developers not to investigate any further the details of the business problem. After this command

(see previous "diary" window) the developers proceeded on their own to develop the IS attempting to interpret the needs of the users, making simplification, and second-guessing some of the physical relationships of the inventory. The prototypes were shown to the users in very advanced stages when the developers envisaged that the software could provide real insights and value to the customer. The first prototype was shown to the users during the 16th month of the project and it was so complete - or complex - that the users were not able to give compelling comments either of the appropriateness of the simplifications or of the software rendering of the physical relationships of the inventory.

With respect to the considerations presented above, the diary from the inventory case shows that discussing the software without a concrete example on which to focus the discussion, allows very little progress towards detailed requirements. It also shows that the decision of having the developers working alone impeded the evolution of common goals and objectives (figure 4.1) and also the path-changing behavior within the outcome space (figure 4.2). The developers' interpretation and simplification were actions that, outside the control of the users, prioritized the developers' goals to test and try the CO techniques to the cost of the users' goals.

Summarizing, often the customer does not initially have a clear view of the requirements for the IS, it takes a long time to decide the detailed system requirements, and the requirements agreed upon can change during the project when the customer collects more knowledge about the technology used. The knowledge exchange has to be actively supported otherwise only the developers will carry out the shaping of the technology. Furthermore, organizations of today live in increasingly dynamic environments where it makes little sense to consider the requirements as fixed. Therefore, in the search for a method to develop CO based software we have to sustain a process that supports active and continuous dialog for business problem definition and redefinition.

4.3 Knowledge Exchange

The issues in the definition and redefinition of the business problem expressed in the previous paragraph highlight a different way of looking at ISD where the successful process aims at the exchange of knowledge between developers and users. Certainly, the aim is still to develop a successful IS, but the consideration shown above led to consider this aim as secondary in the sense that the chances for system success[3] are increased if knowledge is exchanged properly among the stakeholders. The main advantage in using knowledge exchange as aim in the development process is that there is always progress even when what is learned indicates problems with the IS. Using system success as driver for ISD is conducive of a behavior that focuses on satisfying the requirements rather than investigating their appropriateness. Taken to the extreme, this behavior might induce total absence of communication between developers and users in the name of "developing a successful IS that satisfies the requirements". This is very similar to the behavior illustrated in the previous diary window where, after a certain time, learning was not seen as positive, but as a waste of time. This perception ultimately resulted in 10 months of no communication between users and developers.

Knowledge exchange among different communities emerges as one of the main hurdles in development projects [Carlile 2002]. Carlile [ibid.] discovered that when dealing with projects with high levels of novelty, like CO based systems, effective knowledge exchange between

[3] Note that here is used the wording "system success" and not "information system success". System success refers to the improvement of the larger system in which the IS under development is only a part. Indeed system success can be obtained even if the IS is not developed at all, when the knowledge gained from the development process is used such that the overall system is improved. Furthermore, if the project is stopped because new conditions emerge that make it unnecessary, then the overall system will save on the development costs which otherwise would be incurred without returns.

groups, like users and developers, is particularly difficult because what one group needs to know to complete a task is not independent from the knowledge needed by the other group. Furthermore Carlile [ibid] identified defensive behaviors in groups to protect the integrity of their hard worked knowledge. He called this phenomenon *invested knowledge* to indicate that knowledge is not a commodity good but it presents inertia when trying to change it.

In this case, according to Carlile [ibid.], knowledge exchange can be facilitated by centering the dialog between developers and users on artifacts, called boundary objects, especially designed to ease the passage of knowledge. Boundary objects in ISD can be models, documents, diagrams, prototypes or software releases. During meetings these boundary objects become the focal point around which discussion revolves.

The experience from the inventory case has shown that software prototypes are the most efficient boundary objects while written documentation is very inefficient and works better as an information repository. However the inventory case has also shown that prototypes, to become boundary objects have to have specific characteristics and have to be used consciously.

2. From the Diary of the Inventory Case

During the project the developers met the users for the last time in the 5th month of the project and then showed them a quite complete prototype during the 16th month. At this last occasion the developers asked questions related to the modeling choices made for the physical layout. For the users it was very difficult to take a position on the question, since they could not foresee the consequences of their answers for either the software functionality or the operability of the software in the inventory. The issues went unresolved and were picked up repetitively at other presentations. The software prototypes became an impediment to transfer of knowledge because the users did not know enough about the prototypes to comment properly. Without compelling reasons to change the prototypes, the developers continued their work as if everything was fine.

The choice of effective boundary objects is not as straightforward as it might seem. In cases similar to the one presented in our case (Diary 2), the prototype becomes an impediment to the passage of knowledge because knowledge about the prototype is unevenly distributed. This is a known phenomenon where the inappropriate use of boundary object strengthens the power position of one group on the other and reinforces the boundary rather than bridge it [Wenger 2000].

Boundary objects must represent one groups' knowledge to another group making it explicit Nonaka and Takeuchi [1996] but they must also be targeted towards the needs of the recipient group. Boundary objects are not created perfect, estimation and trial and error approaches are necessary to refine the object's ability to bridge knowledge boundaries [Boland and Tenkasi 1995].

Wenger [2000] provides three characteristics for objects to work as boundary bridges. First, boundary objects must be something to interact about or according to Carlile [2002]: *everybody must be able to use them*. Second, they must show real differences as well as common ground. Real differences are needed to make the object interesting. Common ground is needed because otherwise the object cannot be understood completely by one of the parties. Third, they must present dependencies to translate knowledge between groups' repertoires so that experiences and competences can actually be adjusted.

Let us evidence the characteristics of a boundary object applied for example to a possible prototype created for the inventory case.

The boundary object must be visual [Carlile 2002, Brooks 1985]. Visual artifacts are easy to inspect and quick to understand. The software prototype has to replicate the environment of the users such that they can verify whether the developers understanding of it is accurate enough. Visualization responds to the need of making knowledge explicit.

The boundary object must be usable/functional [Brown and Diguid 2001]. Not all knowledge can be made explicit by visualization. Some knowledge that remains tacit can only be demonstrated through action. By working with the prototype the users enact their daily routines and can immediately identify the misunderstandings of the developers. Visual and functional boundary objects will also facilitate the establishment of a common language [Wenger 2000]

A boundary object must be up-to-date [Carlile 2002]. The prototype has to be the latest product of the developers. The main function of the prototype is that the developers can take home the comments of the users, change their understanding of the problem and create more accurate solutions. If the developers present to the users an obsolete prototype while they are already working on newer versions two problems might happen. First, the developers will be focused on newer problems and will miss the importance of the users' feedback. Second invested knowledge will create inertia to do rework.

The boundary object must work both ways [Boland and Tenkasi 1995]. Prototypes are built for the users to learn about the system's possibilities but also for the developers to collect feedback. Mechanisms must be built in the prototypes to facilitate the collection of the feedback. For example the prototype can be built in with a "recording" mechanism so that everything done with the prototype and everything said can be recorded and replayed at will. This feature will give the developers a chance for retrospective sensemaking and facilitate the improvement of the software

Once the importance of software prototypes as boundary objects is clear, the next step is to determine the right pace at which the prototypes are used. The rule of thumb is to show prototypes when their level of complexity is not too high, such that the users can relate to the changes and give constructive comments, but not so often that the variation becomes trivial. Proponents of agile methodologies suggest to present prototypes every three to four weeks [Beck 2000]. Short cycles of development followed by software presentation and discussion should support the knowledge exchange that allows the common evolution of goals and objectives (figure 4.1), and the change of direction within the outcome space (figure 4.2).

5 CIAMM Methodology

In this section we present in detail the methodology for the development of CO based IS. The ideas presented in section 4 provide a rationale for the methodology. The difficulty in determining the business problem leads to the design of an iterative process that allows for frequent revision. The need for knowledge exchange requires the creation of boundary objects as focus points in discussions. To reduce the negative effects of invested knowledge short iterations and simple prototypes are required. These points call for an iterative process where development cycles are short. This in turn reflects on the need for splitting the software into smaller development tasks that can be tackled within the length of one iteration. The subdivision point is of technical nature and will be dealt with in sections 5.3 and 5.4.

There are six main stages in the methodology:

1) Business Problem Definition

2) Conceptual object model and Usage Model (creation of)

3) Iteration and Release Planning

4) Design and Implementation

5) Evaluation

6) Integration

The methodology focuses on a process that facilitates frequent and constructive encounters between developers and users while at the same time trying to minimize the amount of time used for discussion without the support of the proper boundary object. The process involves repeating the phases of the methodology many times in an iterative fashion in order for the knowledge exchange to take place. The idea is to use a relatively low amount of resources in the initial definition of the business problem and instead to put effort in the activities of evaluation and validation, redefinition of use cases, and iteration and release planning that are more meaningful when based on a software prototype.

Figure 5.1 shows an example of how the methodology phases might unfold over a series of three iterations. For reasons of space the stages of the methodology are indicated by numbers from 1 to 6 in figure 5.1. In the figure the methodology stages are mapped against the system life cycle activities proposed by the ISO 15704 to show which activities are involved in each stage. The figure shows that in the first iteration few resources are used to define the business problem and then in the second iteration the business problem is revised due to the learning of the first cycle. In stage four the developers design, implement and test the system. In stage 5 the users evaluate and test the system. In accordance to the characteristics of boundary objects presented above, the prototype should be functional and therefore evaluation is a stage that involves operation of the system. The barred box in stage 6 represents the integrated system only for evaluation and testing purposes by the users. In the third iteration, stage 6 represents integration of the system in production. Integrating software whenever possible is not only source of immediate value for the users but software releases represents the best boundary objects since the users can provide feedback to the developers from actual usage.

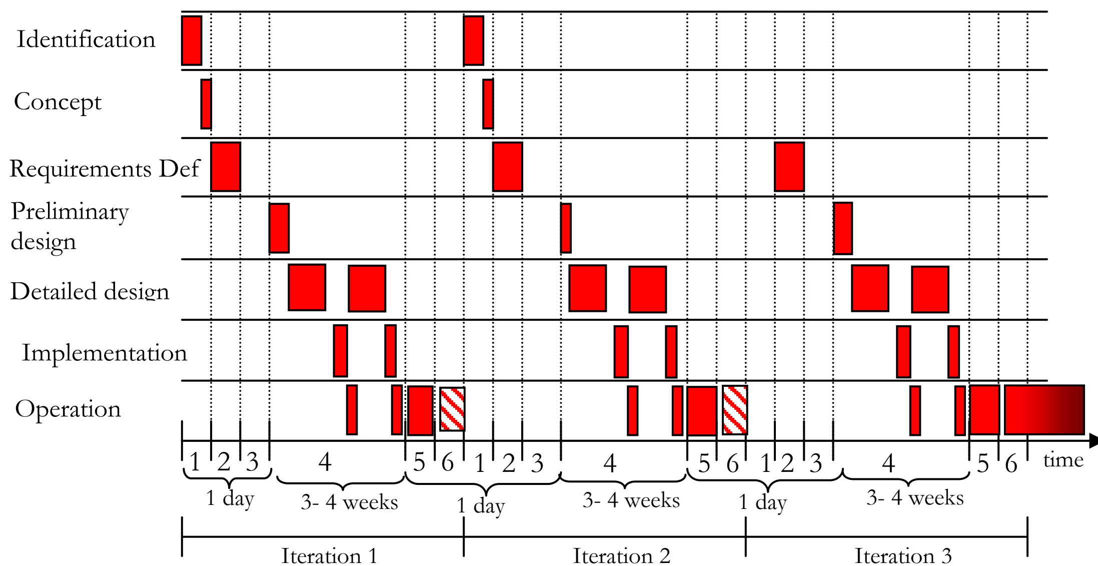

Figure 5.1. An example of iterations over time

The situation is further explained in figure 5.2. The figure refers to the evolving nature of goals and objectives (paragraph 4.2) and shows the representation of goals and objectives of the developers as shown in figure 5.2. At the beginning of the first iteration the developers have an initial understanding of goals and objectives and they proceed to develop the first prototype (the squares in the figure) as a boundary object. When they show the prototype to the users, they (together) achieve a new understanding of the goals. They find out that a part of the first prototype was out of scope because goals and objectives have changed or were not clear and other parts of the prototype were developed with mistakes, maybe because of misunderstandings among the groups. In the second iteration, the developers proceed to develop the second prototype and

correct the first, represented by the area of overlap of the squares marked with 1 and 2. In the following presentation the situation is repeated. Parts of the second prototype are out of scope and parts have to be corrected. The process continues in this fashion until the work done during the entire development effort covers all the needs for the final version of the goals and objectives. Obviously the shape of goals and objectives does not only change because the software prototypes help the users to understand what they want from the IS, but they can also change because something in the environment has changed.

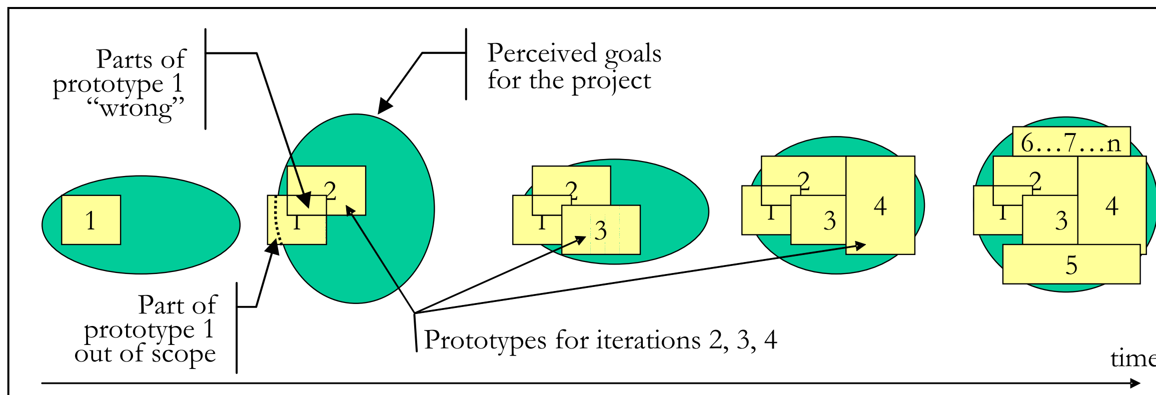

Figure 5.2. Evolving goals and objectives and development efforts

Figure 5.3 depicts the methodology for developing CO based IS. The figure shows an example of 6 iterations. At the beginning, a first version of the requirements for the IS are formulated based on the initial definition of the business goals. In the first iteration the focus is on creating a basic system that can become a boundary object to focus the discussion between developers and users. In the first iteration a graphical user interface (GUI) and a database connection (DB) are therefore developed. This is to prepare at the very beginning visual and useable prototype. In the following iterations, more elements of the CO core are added in common agreement between users and developers. The details of the division of the CO problem in sub-problems are explained below in paragraph 5.4.5.

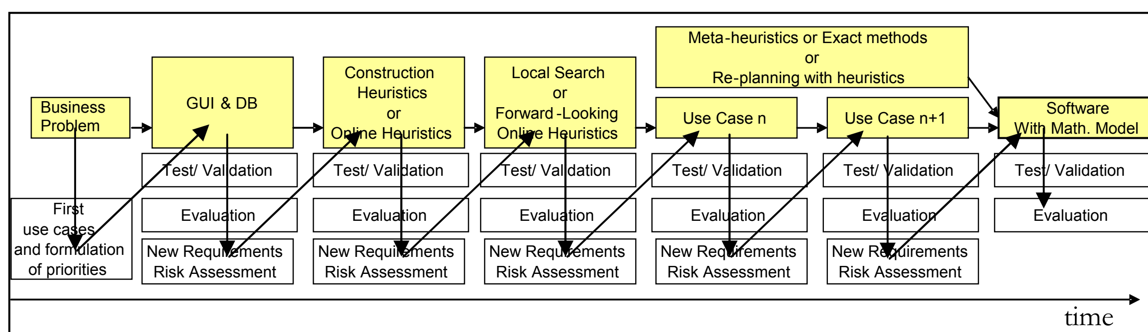

Figure 5.3. CIAMM methodology for information systems with CO

In the next sections the phases and practices of the methodology will be explained in detail, but before moving on we will point out the major difference between this methodology and other agile ones like XP. XP is based on the assumption that rework costs do not increase over time [Beck 2000, p. 23], but rather is constant if the code is kept simple and continuously tested. The result is that programming decisions can be postponed as late as possible in the development process without incurring additional problems. Though we do subscribe to this last point we do it for a different reason. To develop a CO based IS we need mutual understanding of goals and objectives among users and developers. This understanding is the result of a debate around boundary objects that cannot be done prior to the development of parts of the system.

Knowledge exchange and a process that supports it, is therefore necessary in developing CO based IS independent of other cost considerations for ISD projects. Programming decisions must hence be postponed until the necessary knowledge is available.

5.1 The Business Problem

In the Business Problem phase the customer identifies the opportunities for improving current practices with use of CO. The major players in this phase are the different stakeholders in the customer organization possibly supported by the developers facilitating the process of defining the business problem. In this phase, as explained above, it will be sufficient to define rough goals for the system – the outcome space. The process can be supported via the use of workshops as the ones proposed in the *Soft Systems Methodology* [Checkland and Scholes 1999]. The output of this phase is a short document containing an initial definition of the business problem including the overall objectives. Interrelations to other parts of the organization and other business problems should be described. The customer is responsible for writing the document. This phase should not take more than a few days and it is important that both users and developers acknowledge the results of this activity as temporary and therefore subject to change.

5.2 Conceptual Object Model and Usage Model

In this phase the definition of the business problem is detailed resulting in a *conceptual object model* of the system domain and a *usage model*. These models can function as boundary objects facilitating knowledge exchange at an early stage enhancing the developers' understanding of the domain. The models are the basis for design and are hence not directly related to how the software system is built. This is the purpose of the design phase. The usage model and the conceptual object model can be built either in sequence or in parallel. From our experience the sessions should be relatively small to be effective: A few developers, a domain expert and few end-users (see also [Beck 1999]). This phase of the life cycle should be finished within a few weeks. This process is often referred to as the requirements capture and analysis [Bennett et. al. 1999] resulting in a *requirement model*.

5.2.1 The Usage Model

The usage model consists of *use cases*, which are descriptions of functionality. The set of functionality included in each use case solves a specific task necessary for a user or some other interfacing system. Basically, a use case includes the role of the user, the functional requirements and the business reason for the use case. The business reason might seem insignificant, but the purpose is to validate the need of the requirement. The stories are identified at a meeting where both developers and users participate. The use cases can either be written on note cards, or a CASE tool can be used. The user will later detail each use case when it is going to be implemented. The collection of use cases gives a rough description of the future usage of the system. The usage model is obviously revised after every iteration.

In Table 5.1 is shown an example of a possible use case from the inventory case. The crane operator is notified if the plate requested is not on top of the stack, the crane is moving to. The developers can add notes on the card including risk assessments and time estimates for implementing the story.

<table>
<tr><td>Story name: Manual registration of misplaced plate</td><td>Date:</td></tr>
<tr><td>Risks: Low</td><td>Time Estimate: 1 day</td></tr>
<tr><td colspan="2">User roles: Crane operator</td></tr>
<tr><td colspan="2">Story: If the top plate in a stack is not the plate registered in the system, then the expected top plate is added to a "warning" list. Else if the operator finds a plate that should be in another stack he/she can manually register the correct placement of the top plate.</td></tr>
</table>

Business reason: When registration errors occur these must be corrected to restore the consistency of the database.
Notes:

Table 5.1 Example of a use case.

Another use case might be to get the output from the CO method for the defined CO problem. For example in the inventory case, this would be to request sequences of movements for the two cranes.

5.2.2 The Conceptual Object Model

The conceptual object model is as mentioned earlier a simplified representation of the system of interest. The use cases constitute a foundation for identifying the objects and classes in the system: All nouns are potential candidates.

For each class we are interested in the attributes determining the characteristics of the class, the relations or associations to other classes and the responsibilities of the class. The different possible conditions of the object are described by different states and state changes are triggered by events. It is often easier to grasp a visual representation of a class than a written. Here, different types of UML diagrams are useful. Class diagrams can be used for describing the different objects and classes of objects in the system as well as their relations.

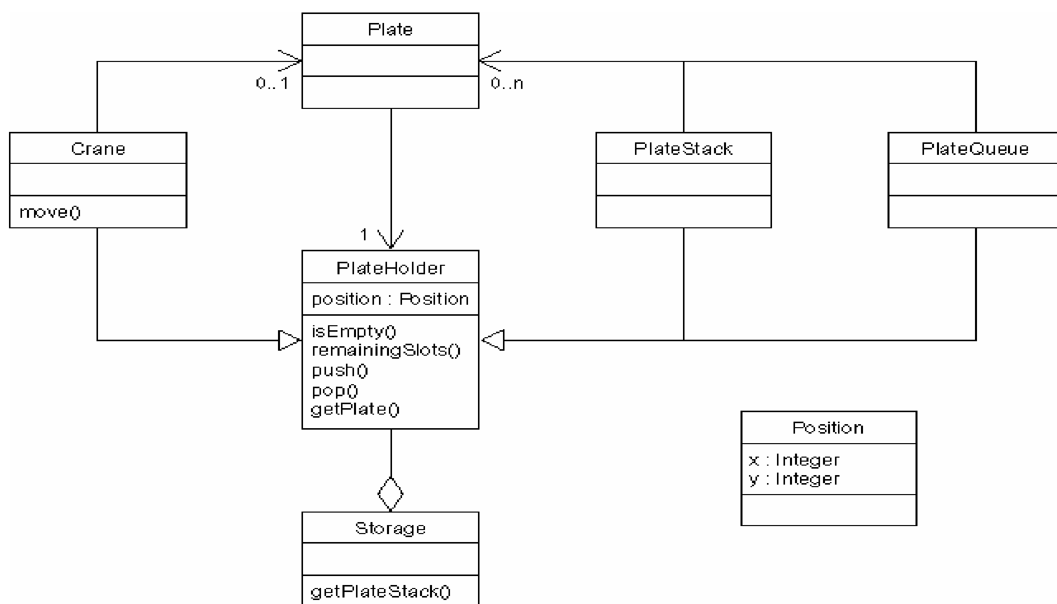

Figure 5.4. An example class diagram for the inventory case.

Figure 5.4 provides a simplified example from the inventory case. "Crane", "PlateStack" and "PlateQueue" are all subtypes of the general "PlateHolder" class. "PlateHolder" is not related to a physical object, but is an abstraction of an object that can "hold" a plate. For all plate holder classes a plate can be "pushed" into or on to it, e.g. a crane can lift a plate or a plate can be dropped on a stack. Similarly a plate can be "popped" or removed from a crane of stack. Further a crane can move. The arrow to the plate class indicates an association, which refers to the fact that a crane can hold zero or one plate. Similarly 0 to *n* plates can be placed on the plate stack and queue. Characterizing all subclasses of "PlateHolder" is that they have a position in the inventory, where the position is an x-y coordinate. The operation isEmpty returns true, if the "PlateHolder" is not holding a plate. The association from "Plate" to "PlateHolder" indicates that given a plate, we can find out where on the storage it is positioned.

Another useful diagram is the state diagram. Here, the different states and activities of an object are illustrated as well as the events triggering state changes. Figure 5.5 shows an example of the crane class from the inventory case. The two key states are the crane waiting for further instructions with or without holding a plate. An arrow to another state indicates a state change and the legend describes the event triggering the change. The brackets are so-called *guards* or conditions for triggering the event. For instance when the crane is waiting, it can only start moving, if no collision with another crane would occur. Similarly a plate can only be lifted from a stack, if the stack actually holds a plate.

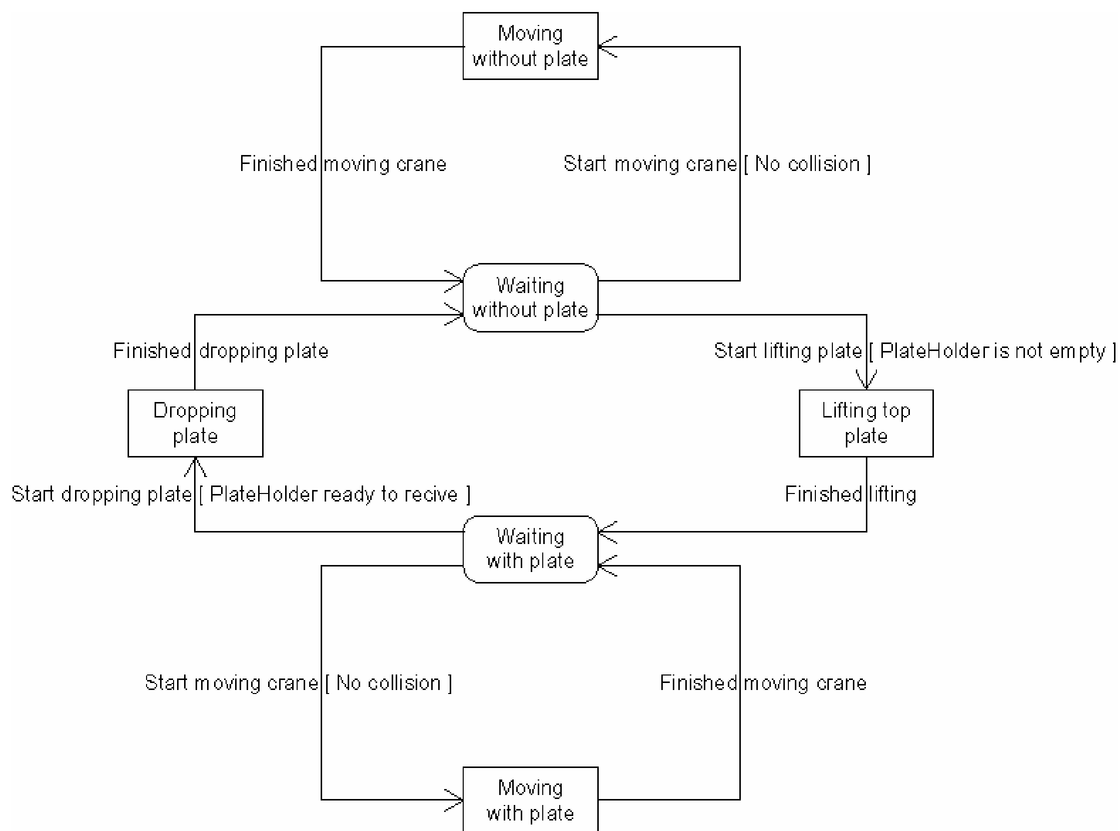

Figure 5.5. An example state diagram for the crane class in the inventory case.

5.2.3 Supplementing the Models for CO Development

A number of optimization specific issues are usually not captured in the conceptual object model and the usage model. We propose to extend the models to include inputs, outputs, constraints and objectives. Inputs to the CO problem are most likely already captured in the object model, but more detail is needed. The degree of uncertainty on input data must be evaluated: Is the problem deterministic or stochastic? Is the problem static, or dynamic – i.e. are data available a priori or will they become available during execution? What types of disruptions or events should the system be able to handle during execution of the achieved solution? Should the user be alerted in case of certain events occurring or should it be handled automatically by the system? If data are not available, initiatives must be taken to collect these. Some of these issues relate to the usage model and others to the conceptual object model, which might need to be updated when these issues are resolved. Additional information on inputs is noted on the class diagram typically as attributes and relations.

Objectives identified in the phase of determining the business problem is validated at this stage. In case of multiple objectives, it has to be decided how they are ranked or how the objectives are aggregated to form a single objective. Further, should the user be given a single solution or choose from multiple solutions presented? Together with the developers the customers

determine the requirements on solution quality and available computation time. These issues are all related to the usage model.

In Table 5.2 it is shown an example of a checklist, which can be used when collecting information on objects or classes, which are part of the problem definition. Again we use the inventory case, considering the crane class. The focus is on analyzing what types of input, output, constraints and objectives are needed to model the crane and its relations to other objects.

Class/Object name: Crane	**Date**:
Deterministic/static input: Start position, costs of lifting/dropping plates and moving crane.	
Deterministic/dynamic input: Request for movement jobs and changes in movement jobs.	
Stochastic/static input:	
Stochastic/dynamic input: Distributions over movement speed in X and Y direction and lift/drop times.	
Disruptions: Breakdown/halt of crane.	
Constraints: Avoid collision with other cranes. Can only hold one plate at a time.	
Output: Sequence of movements performed by the crane.	
Objectives: Min. no. of lift/drops, moved distance and duration of movement sequence.	

Table 5.2 CO checklist for the crane class.

5.2.4 Summary and Final Remarks

At the end of this phase all the models are ready for further use in the design and implementation phase. The models have maximal importance in the first iterations in the methodology since they have the function of boundary objects. In later stages, when software prototypes are ready, then the models' informative content becomes less important. During the multiple iterations the goals and objectives are going to change making the initial models outdated. Decision must be taken on whether to maintain the models up to date during the project or not. Models must be updated under any circumstance at the end of the development project to create the necessary documentation for software maintenance and upgrade. Creating the documentation at the end of the development will save lots of resources in maintaining documentation, which is destined to become obsolete anyway. Comparing the first models with the last provides an occasion for reflection on the process and concrete evidence of the variability of goals and objectives.

5.3 Iteration and Release Planning

The iterations and releases are planned on the basis of the initial collection of use cases and the conceptual object model. One iteration in the project life cycle must be limited to weeks not months according to our basic ideas described on page 10. This will increase the amount of feedback from the customer thereby increasing knowledge exchange.

3. From the Diary of the Inventory Case
This project was very long … it is no secret – I even heard the developers' manager saying it – that we could have done ourselves a big favor by having held more frequent technical and comprehensive meetings to keep track of what [the developers] were doing. It would have put us and our production department closer to the development; it would perhaps also have assured that the project hadn't been running on such a wide spectrum or even in totally another direction than planned. So more milestones over this VERY long period … We felt that some

people withdrew to their ivory tower, and cultivated their own interests there, where it perhaps could have been more advantageous for them and us if we had more dialogue in that period.

Project Manager - Customer side.

Given the short duration of the iteration and the developers' estimates on how long it will take to develop each use case (including possible dependencies between use cases), the customer decides what use cases the developers shall develop in the current iteration. In this phase it should be possible to make a preliminary plan with the dates when the use cases will be integrated in the customer's existing systems. The plan is revised regularly during the development when more knowledge is gained.

In projects including CO, the design and implementation of the use cases in which the users request a solution from the CO core will usually take a long time to be completed. CO software is quite complex and the development of such a use case might take months. In order to exploit the methodology for the CO use case, we need ways to split the development task into smaller components that fit into the short iterations. The developers drive the process of splitting the complex use cases into smaller components. Possible ways of splitting the development can be done for instance by considering:

- Smaller sub-problems.
- Different optimization methods.
- A subset of the constraints or other simplifications.

When requirements are neglected, plans should be made for their reintroduction in the system.

In the following is given an example of the headlines of the planned development in the first 4 iterations of a development project including CO:

1. A database and simple GUI to do manual planning and visual simulation.
2. Constraint checks and objective calculation. The user uses the software manually but is alerted if constraints are violated.
3. Simple construction heuristic that can automate the work of the user. The user can compete against the heuristic in an optimization game. Here visualization is essential to create a realistic test environment for the user.
4. A local search heuristic that improves the manual work of the user or the construction heuristic and a GUI to adjust optimization parameters.

After the completion of the first iteration, the software prototype will support discussions between the customer and the developers by visualizing what the users do when planning manually and by simulating the planned solutions. In the following iteration the implementation includes constraints that are checked for violation during manual planning and simulation. The constraints are implemented based on the developed CO models. The objective value is calculated in order to compare solutions. In the third iteration a computer game is created where the manual plan is compared to the plan achieved with a simple heuristic. We refer to section 5.4 for details on heuristics and other optimization methods. Now the user has the opportunity to validate that the notion of a feasible solution in the system is correct. Again, visualization of the solutions is the key to focus the discussion. It should be noted that the development until iteration 4 is relatively low risk, while developing more complex optimization methods than construction heuristics are medium to high risk. It is not trivial to split an optimization development task between smaller iterations and still keep the added functionality interesting for the users to test and discuss. The idea is to identify smaller subsets of the optimization methods that can be graphically illustrated for the user. In this way the length and risk of the iteration is

reduced and visualizing the underlying components of the CO methods increases the users' knowledge of optimization methods.

5.4 Modeling, Design and Implementation

The main activity in every iteration is design and implementation of the chosen use cases. We will specifically focus on development related to the CO core. Given the conceptual object and usage models, the development team initiates the building of the mathematical or logical model of the problem. First the problem is analyzed for specific characteristics hereby identifying suitable model types and optimization methods, e.g. a scheduling problem or a resource allocation problem. The model includes variables, constraints, parameters and the objective function. Secondly, the developers determine the problem size and size of the solution space to give an indication of the expected computation time. Perhaps the problem can be decomposed into smaller sub-problems, which separately can be solved more efficiently. The above tasks require experience from similar projects and a good overview of CO methods and applications. Otherwise there is a risk that the same and possibly wrong tool is applied to all problem types even if other more suitable tools exist: "Once you know how to use a hammer everything in the world looks like a nail". [Williams 1999] is an indispensable source of inspiration when building Mathematical Programming (MP) models. MP models are in our setting models where the objective is a mathematical function and the constraints are inequalities and equalities consisting of mathematical functions. Often the functions are linear functions. MP models constitute a subset of Constraint Logic Programming (CLP) models. CLP models are more general since any logical relationship between variables can be expressed, but the optimization methods are mathematically less sophisticated [Marriott and Stuckey 1998]. In cases where the models become too complicated to build or solve, the alternative is a simulation model [Pidd 1998] or models suitable for local search heuristics [Pirlot 1992].

We propose that the development team initially develops the simplest form of optimization method for the given CO model and afterwards gradually improve it or complement it with more complex methods. Unless the model is straightforward to solve with standard optimization software, the simplest thing to do is usually to *construct* a solution by repeatedly taking greedy decisions without regret – a so-called construction heuristic. The advantage of this approach is that the heuristic is fast, easy to develop and the developers will quickly get feedback from the customer. Changes to the CO model will be necessary and this is cheaper to do with a simple approach than a more complex one. Table 5.3 shows the appropriate methods based on the requirements on solution quality and available computation time.

	Low Solution Quality	**Medium Solution Quality**	**High Solution Quality**
	Short Computation Time	**Medium Computation Time**	**Long Computation Time**
Static Data	Construction heuristics	Local Search Heuristics	Meta-heuristics or Exact methods
Dynamic Data	On-line heuristics	Forward-looking On-line heuristics	Re-planning with heuristics

Table 5.3: Appropriate methods based on problem and solution characteristics

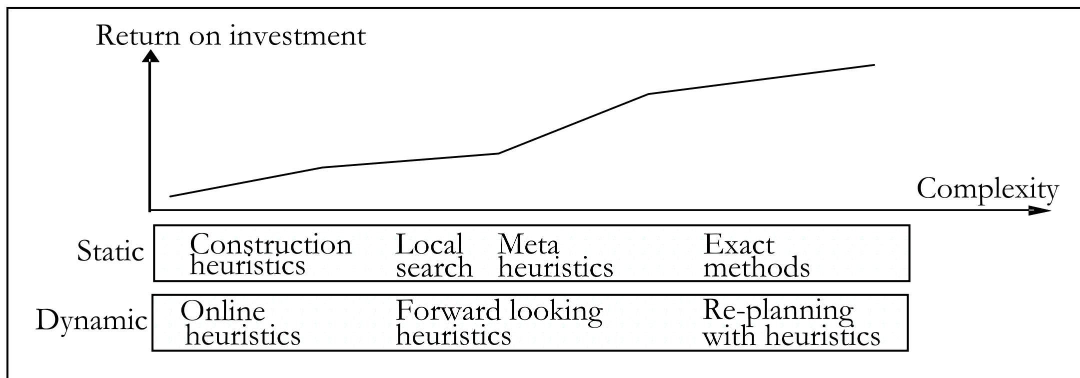

Figure 5.6. Return on Investments versus the CO methods' complexity

Figure 5.6 shows the different methods in a complexity vs. return on investment diagram. The plot illustrates common beliefs, but the shape of the curve and the scales are of course depending on the skills of the development team, the characteristics of the problem, and the effectiveness of the development process. The developers should inform the customer of the costs, risks and benefits of choosing different methods for a given problem. If data are static or given before optimizing, the first row is appropriate otherwise the methods in the second row are proposed.

It is always a good step to first develop a construction or on-line heuristic, because this is the simplest thing to do and a prerequisite for developing more complex methods. Local search heuristics for instance require a constructed solution to improve. An on-line heuristic is basically the same as a construction heuristic. The only difference is that the heuristic decisions are made on-line. The forward-looking on-line heuristic as the name indicates looks a few steps into the future in order to avoid making very shortsighted decisions, similar to a chess-player trying to foresee the opponent's move in order to make his best move.

Local search heuristics try to improve a solution by iteratively making small or local changes to it. A local search is composed of the following components:

1. A *representation of a solution* to the problem.
2. Operators to make local changes to a solution – also called the *neighborhood structure*. Solutions reachable by applying an operator once on a solution in all possible ways are called the *neighborhood solutions*.
3. To make an efficient local search it must be fast to determine the objective value and feasibility of the neighbor solutions – i.e. *evaluating a neighbor*.
4. S*top criteria*: A class of local search heuristics called Descent algorithms stop when no improving neighbor solution exists. The solution is then a *local optimum*, but not necessarily a global optimum of the entire solution space. Other criteria are computation time, number of iterations, etc.

There are other methods than the Descent algorithms that allow the escape local optima still without guarantee to reach a global optimum. The most widely used local search heuristics are Simulated Annealing and Tabu Search [Pirlot 1992]. The set of local search heuristics is a subset of the class of heuristics called Meta-heuristics which also includes Genetic Algorithms. Exact methods and heuristics based on exact methods are generally more complicated to implement than local search heuristics, but in cases where the model of the problem or sub-problem fulfill certain structural characteristics, very efficient methods exists [Nemhauser and Wolsey 1988]. Often, however, the textbook models need to be extended to take into account special constraints occurring in practice, which might destroy the special structure that made the problem relatively easy to solve. In our methodology we focus on flexibility and simplicity, which are the strengths of heuristics. A mathematical model with linear constraints and objectives is not required for building heuristics. Any model consisting of any type of constraints and objectives can be solved with heuristics.

The components of a local search heuristic might suggest a way to split the development into smaller tasks as suggested in section 5.3. First of all, the solution should in some way be visualized. It is more difficult to illustrate the neighborhood structure: Given a solution the user suggests neighbor moves predefined by the neighborhood structure that might improve the solution. Alternatively the user can iterate through the possible neighbor moves, which are evaluated by the system. These features can also be combined with the optimization game to aid the user. After this iteration one or several different Meta-heuristics can be developed in separate iterations and different neighborhood structures can be tried, etc.

5.4.5 Implementation Issues

Given the CO model and the chosen solution method, implementation can start. Remember that design is not followed by implementation. The developers change between designing and implementing when necessary in order to revise the design to cope with additional functionality. When designing complex optimization software small sketches and/or UML diagrams can be of help, but completely specifying the system before starting implementation is not recommendable [Beck 1999]. The design should include interfaces to other external systems. This is often straightforward with static data, but if the CO core is part of a system used in a dynamic environment, integration can be complex. With static data, these are normally loaded into an appropriate object model and the solution is returned after optimization. In a more dynamic or on-line environment ensuring consistency of data and concurrency are issues to be handled. It is however outside the scope of this paper.

The choice of development platform should not be limiting the set of possible methods to apply. The best choices for heuristics are languages such as Java and C++, since they are object-oriented, fast and widely used. Use of available software libraries and frameworks should be used in order to decrease development time. Examples include libraries for solving LP, IP and CLP models, Meta-heuristic frameworks and Branch & Bound/Price/Cut frameworks. Most of them have interfaces to or are built in Java and/or C++. An alternative is the use of modeling languages and other high-level languages especially suitable for CO. We suggest choosing languages, which have interfaces to Java or C++.

5.4.6 Improving optimization methods

Performance should be considered when the developers and the users are confident that they have reached a common understanding of the business problem and that the CO problem is correctly modeled and implemented: Is the solution quality sufficiently good? How long is the computation time of the method compared to the requirements? Should time be invested in improving the quality and/or the computation time? Often the first straightforward shot at a working method is not satisfactory. The developers identify potential improvements and estimate the expected gain, risks and working time. Here a profiler is an indispensable tool for finding computational bottlenecks. Again we suggest trying the simple improvements before the more complex and time consuming. Generally a significant gain must be expected to pursue more complex improvements. Implement only one improvement at a time in order to precisely measure and track the improvements. Analyzing and keeping track of code-changes, improvements and experimental results can be a tedious task. ExpLab [Hert et. al.] attempts to simplify this task by providing a set of tools for running experiments, documenting the environment of the experiment and afterwards analyzing the results. Developers can be too focused on reaching the best possible performance, but often it is not a big issue for the users. Clear performance goals should be set up together with the customer before spending time on tuning the performance. The customer should prioritize the improvement tasks together with the other development activities.

5.4.7 Test and Validation

Testing is the last subject to be discussed within design and implementation. We distinguish between two forms of tests: *Developer tests* and *customer tests* [Beck 1999].

Developer tests are as the name indicates performed by the developers. Often it is by accident that that a bug is discovered in the CO core. In some cases long computation time of CO methods makes it difficult to find the reason for the bug and a significant amount of time might be necessary to find the source of the error and afterwards correcting it. When trying to locate bugs, debuggers or print statements are often used. A lot of time can be saved if invariants as well as pre- and post-conditions are used. In CO tests checking consistency of the object model, feasibility of neighbor moves and change in objective value is a must. When doing time-consuming computations, it is a good idea that the program at certain checkpoints saves the state of the system. If an error appears, debugging can begin from the last saved state instead of starting the run from scratch. This will significantly reduce time spent on debugging and frustration in the developer team. A more radical possibility is automatic unit testing. One writes code, which automatically checks that the program is working properly [Jeffries 1999]. The test code must be separate from the actual program.

User tests are so-called functionality tests based on the use cases. For each use case the user writes a set of tests including input data and required output data. The tests are afterwards coded by the developers and included in the test suite. In CO software, a functionality test that must be implemented is a check that solutions fulfill all the constraints and has the right objective value. Note again that the test code obviously is separate from the code doing the actual optimization. The tests should be implemented as soon as possible in order to create confidence that the achieved solutions are valid. A supplementary test is to generate instances, where the optimum solution is known.

5.5 Evaluation

In the evaluation phase the iteration and progress of the entire project is evaluated. This is done at a meeting where the developers present the results of the last iteration. The results are discussed and the prototype is demonstrated to the users and evaluated. Since the intention is to use the software prototype as a boundary object to facilitate knowledge transfer, it is important that time is scheduled either at the meeting or before the meeting for the users to use and discuss the software with the developers. In this phase the users get acquainted with the technical possibilities of CO. The use of the software will be the foundation for generating new knowledge and hence new requirements for the IS as well as to comment on the correctness of the representation of the problem in the software.

At the same meeting it is discussed the next iteration and the revision of the plan. Basically iteration/release planning is done at the same meeting immediately after evaluation, hence initiating a new iteration.

5.6 Integration

Releases are integrated into the user environment and used in production by the users as soon as possible. This is the best way of getting feedback. The users should choose the smallest useful subset of use cases for the first release. New versions of the software should be released as often as possible to the users and hence give value to the customer as early as possible. Potential obstacles for integrating the software must be identified as soon as possible in the project. The plan for their removal must also be part of the project plan even though not necessarily done by the developer team.

The costs of preparing integration might involve large initial investments, which will only be justified, if the software system introduces a significant value to the organization. The risk is hence that the customer will only accept to integrate the system when it is nearly or completely finished. If the software cannot be implemented stepwise, initiatives must be taken to create a test environment simulating the integrated software to avoid problems during the final integration.

6 Project Managers: Users and Developers

The methodology described above has its fundament in the customer's business problem and its knowledge aspect, however the explanation of the phases in section 5 focuses almost solely on software creation activities. This section is focused on specific activities that the project managers of users and developers should focus on in their daily managerial activities.

6.1 Users' Manager

Business Problem: The main activity for the users' manager in the definition of the business problem is to understand the nature of the problem that he has to solve and treat it accordingly. Since the problem will only be partially clear in the beginning of the development case, the advice is to exploit the CIAMM methodology to incrementally provide a solution to the evolving understanding of the business problem. It is very important for the success of the project that the project manager is a "champion" for the project [Ryan 2002]. He must argue and speak in support for the project and the methodology internally in the organization to ensure the support from management and participation from the users.

Problem identification and specification of the use cases: According to the CIAMM methodology the customer and developers will go through the phase of problem definition and use case specification many times, once for each iteration. The main concern is to create acceptance for the methodology for those involved in the project and those outside. Customers of IS are used to receive the final requirements document and sign it off before the developers begin coding. Signing off is a traditional act as well as a legal requirement. There is a high probability that the iterative process proposed in the CIAMM methodology could be considered too informal and the project mistaken for being "out of control".

The task of the project manager for this phase is mainly educational. He has to educate the other project managers and high level managers about the idea, that the direction of the project is emerging and the final goals are detailed during execution. He also has to educate the users to work, and be satisfied with, semi-finite requirements at each stage. It has to be clear that the high frequency of the presentations sets the rhythm of the development and that therefore the users have to prioritize the use cases in cooperation with the developers in order to focus the development in the available time.

Iteration and releases planning: In this phase it is important to keep in mind the basic ideas of the methodology and in particular the focus on knowledge exchange. In planning the iterations, it is very important that a boundary object in form of a graphical prototype is developed as soon as the very first iteration. This is because graphical representations are the only concrete way [Brooks 1987] in which the users can evaluate the developers' understanding of the problem. At the end of the first iteration the users should receive a graphical representation of their physical system and they will be asked to perform the same activities as they would do in regular operation. Through their comments on the representation errors, on the limitations of the system etc., the developers can begin to understand the users' worldview and see how much of it has been captured in the first usage and conceptual object model. At the same time the users can understand the state of the software and therefore they can decide their priorities for the following iterations.

Customer's Reflection: While the developers proceed to develop the code, the project manager should invite other potential stakeholders in the company for a "moment of reflection" about the work accomplished and present newly emerged ideas. The goal with these reviews is to re-examine the software delivered in the previous iteration in order to investigate what types of opportunities and threats it can bring to the overall system. These meetings will increase the pay-off of the software system as well as avoid potential disruptions of the overall company's activities. A secondary result of this activity is to demonstrate, to external stakeholders, that the project is proceeding in the right direction. The results of the meetings will be discussed with the development team in the following presentation.

Evaluation and Validation: Using a process that takes emergent issues into account brings the development project into a status of continuous revision and change. Consequently the delivered software will also be subject to revisions. It is important that this idea is well understood by all stakeholders including especially the users. Misunderstandings must be tolerated as part of the process that brings the two teams towards a shared understanding of the goals and objectives for the project. The goal is knowledge exchange. The users have to be able to deal with evolving software such that the developers can get concrete feedback at each iteration. It must be stressed here that the discovery of misunderstandings has to be received as a positive sign of good cooperation and not as a sign of failure. Misunderstandings will always emerge because of the knowledge boundaries. If they are not surfaced e.g. because of fear or bad practice, they will evidently be incorporated in the system generating consequences that are not easily remedied later on. The consequences might be a final system with little resemblance to what the users expect and of little value to the customer.

6.2 Developers' Manager

When using the methodology, the project manager on the developers' side will most likely encounter problems related to the traditional ways in which system developers attack the development process. That is, the worldview common in software development that software has to be delivered completely finished and functional and that the goal is "IS success". A large amount of energy has to be used to change this worldview because it can easily result in late deliveries that will hinder the knowledge exchange process. Therefore, also for the developers, the main objective must be knowledge gained from users' feedback and not delivering an error-free system at each iteration. A typical example can be found in the diary 5 of the inventory case that tells about the decision on when to present software to the users.

5. From the Diary of the Inventory Case

During the 15th month of the project the developers felt ready to present the software to the customer and proposed some dates for the presentations. These dates were communicated by mail to the customer's project manager. After reading the mail he responded:

"We are surprised about these late dates. In our last meeting in [13th month] you proposed a date at the beginning of [17th month] which is already much later than the planned date in [14th month]. These late dates make it difficult for me to keep the users interested. … We would therefore prefer to have the first two presentations before the summer holidays"

The developers' manager answered: "What you propose is probably possible but there is a higher risk that unexpected problems will incur during development, which will result in us not being able to deliver the promised results. This risk was the original reason for adding extra time to the original plan. … Now that you are aware of this risk you can decide accordingly when to schedule the presentation."

The customer decided to accept the risk and scheduled the presentations before the summer.

In this case the central concern for the developers is to present something functional … "the promised results". In order to do this they request more time delaying the presentation. Notwithstanding the importance of the customer's point in keeping his users motivated. The note from the diary shows a typical risk and conflict avoidance behavior that could prevent the methodology from being effective. Only through an appreciation of the positive effect of feedback it will be possible for the developers to avoid this behavior and capitalize on the users' frequent feedback. Also in this case the metric of performance should be linked to parameters pointing at increased knowledge exchange. Quantifiable parameters could be hours spent with the customer, frequency of meetings and plan revisions, quantity of feedback received (vocal or mails) and so on. Rewards should not be given for performances that encourage "ivory tower" retreats like the some times used "lines of code produced".

For the developers the focus on knowledge exchange is more problematic than for the users because showing prototypes that are, to some degree, "wrong" puts them in a vulnerable position and exposes them to criticism. For this reason it is very important that there is full agreement and understanding between the users' and developers' teams on the reasons behind the use of the methodology and the advantages that are sought with it.

7 Summary and Conclusions

We have, as outlined in the introduction, proposed a methodology for developing CO based software. We have identified the basic ideas for the methodology based on experiences gained from a practical case. The basics of the most commonly used software methodologies have been presented as source of inspiration and we have adopted some of the ideas from agile methodologies like XP.

The usual approach in ISD is to invest significant resources in the beginning of the project in order to reduce the risk of solving the wrong problem. However this approach fails to consider that, in developing CO based IS, it is difficult for the users to specify what they request from the IS until they actually see it in action. Therefore, for CO based IS the initial system analysis and design will always be incomplete.

The experiences from the inventory case points to two important factors differentiating projects with CO from other ISD projects:

1. Knowledge exchange between developers and users.
2. The subdivision of the CO core into smaller development tasks visualizable for the users.

In CO projects knowledge exchange between the stakeholders is particularly difficult because of the complex nature of the solution provided. Our choice of using an agile approach emerged from this recognition and is focused on sustaining a process that facilitates knowledge exchange between developers and users. This is achieved through frequent encounters focused on the discussion of a model, a prototype, or a release that act as boundary objects during the discussion.

Developing CO based systems normally requires long periods of non-interaction due to the complexity of developing the CO core. We have therefore proposed a way to subdivide the development of CO systems into smaller tasks. These smaller tasks are manageable in relatively short periods of time hence making possible the use of short iterations.

High frequency poses a limit to the increments in software complexity therefore allowing the users to comment at full capacity on the work done by the developers. The methodology focuses on making the users able to give well grounded reasons when they ask the developers to change the software or extend it.

This openness to change in goals and objectives is reflected in the methodology in the fact that we do not look for a well-defined initial definition of the project goal, but we allow for an emergent process of goal determination within the outcome space. The specific path followed to reach the final goal will emerge during the project according to the decisions taken collaboratively by the two groups.

Continuing the users-developers interaction at fixed rhythm is helpful to make sure that the project is kept on going in the right direction and, as underlined in the experience from the inventory case, to keep the users motivated in giving their feedback to the developers.

Finally these activities together should respond to the ideas and issues presented in paragraph 4.1:

1. Decide the detailed objectives for the IS.
2. Priority to the solution of the business problem.
3. Knowledge exchange to reach a common view on requirements.
4. Avoidance of problems in relation to invested knowledge.
5. An iterative process based on short iterations.
6. Simple releases for the customer to understand resulting in constructive feedback.
7. Regular monitoring on the business validity of the work done and planned.
8. Create value for the customer through frequent prototype demonstrations and releases.
9. Open the CO black-box for gaining the customer's trust in the system.

The focus is all the time on the business problem and its evolution. The frequent interaction between the groups puts the business problem on the agenda every time [ideas 1, 2 and 7].

Knowledge exchange and problems in connection with invested knowledge are taken care of by the frequent iterations and the simple releases focused on boundary objects [ideas 3, 4 and 6].

Short iterations with frequent presentations increase customer participation, leading to involvement and consequently satisfaction with the system [idea 5]

Value is given to the customer as often as releases or demonstrations are planned. Little time used on requirement elicitation and more time used on using the system maximizes the value that the customer can expect during development [idea 8].

The customer participates in all phases of the development and is therefore well informed on how the CO core provides a solution. Resistance to the use of a black-boxed system is minimized [idea 9].

To conclude, the methodology provides a way of doing software development, which responds to issues, which are common in many software development efforts, but which are accentuated by the complex nature of CO. The complexity of CO makes it difficult for the customer to see the possibilities granted by the optimization methods. Showing the possibilities of CO to the customers will change their perception of the problem and therefore reshape their objectives for the IS. However, in order to make the process most effective the users and the developers need to go through frequent discussions based on boundary objects. The goal with these boundary objects is primarily to facilitate knowledge exchange and secondarily to fulfill the requirements for the IS. In this view it is sensible to make the development efforts as simple as possible only solving the problem specified in each iteration. Short iterations result in less resistance to rework because the level of invested knowledge remains low. The proposed way of subdividing the development of the CO core is necessary for the developers to code the core in short iterations. Further it gives an opportunity to transfer the necessary knowledge of CO to the customer for them to realize the possibilities of CO.

Appendix 1: Experiences from the Inventory Case

Risk Scenarios	Consequences	Countermeasures
The customer organization has not allocated sufficient time for the employees in the project. Employees are not directly rewarded for project participation.	Lack of commitment and participation in the project.	Quick integration of the application in the organization creates instant value, which justifies time used.
End-users not committed in project.	Software will never be used.	Frequent tests by users of new application releases.
Culture of fire fighting focused on the solution of current problems instead of development projects.	Impatience and decreasing commitment.	Produce rapid and frequent application releases with clear user benefits.
Too much time spent on requirements capture, delaying the first delivery of a prototype.	Customer organization looses faith in the project.	Quickly produce a first release of the application based on coarse require-ments to engage further debate.
Software so complex that not even programmers can understand it fully	Difficult to detect errors and extend the system to take further require-ments into account.	Frequent releases do not guarantee simple software. Always keep the software design as simple as possible. Use re-factoring.
The customer does not understand the concept and possibilities of combinatorial optimization.	The customer looses interest in the project.	Create an optimization game where the users compete against the compu-ter or an interactive demo.
The developers do not understand the business domain.	The developed software does not solve the problem of the customer.	Ensure frequent communication between the domain expert and the development team based on applica-tions.
Difficult for the customer to determine the relevant requirements.	The application solves the wrong problem.	Frequent releases and feedback based on applications reduce time wasted on rework.
Communication gap between customers and developers.	Even using the same language the problems are understood in very different way. The wrong solution will be implemented.	Frequent iterations based on prototype discussions.
Developers not sufficiently skilled in the deployed technology.	Risk avoidance and long time used on polishing.	Simple and short releases will help to create the experience base to tackle problems of rising difficulty.
Lack of agreement on the deployed methodology.	The process cannot be monitored and controlled	Agree on methodology and go through the necessary education phase.
Difficult for the developers to estimate time to deliver the requested requirements.	Project gets delayed if optimistic. Use rule: "estimate and double".	Be open in the discussion of risk and complexity. Follow up on estimates and learn. Always deliver something on time.
Everyone is afraid to expose ones ignorance of the business domain or	Silence … problems not discussed.	Discuss the prototypes … discussion will emerge naturally.

optimization methods.		
The user wants to automate the "pain" of performing manual tasks.	The software does not respond to a real business problem	Look for possibilities before the development is started. Redesigning processes before is better than after.
The customer stakeholders have different views on requirements.	The developers get confused and no clear consensus is reached on the features of the system.	Use releases as a focal point for discussion. Be ready to waste a few designs in favor of reaching agreement.
Difficult to show the real value of the system	System is not appreciated. Difficult to justify expenses.	Frequent releases provide added value at low cost.
Difficult to show progress of the system	During long "quite" phases the customer cannot see progress. It gives the impression of "nothing is happening"	Force frequent presentations, despite fear of looking bad.
Commitments with one's own institution/company creates tradeoffs with project activities	Less effort is given to the project than necessary. Work quality is low.	Make the project very visible in the organization through frequent demos.
Difficult to coordinate programming activities for common parts	Bugs and time lost on integration and rework.	Always use simple design. Modular design helps integration
Graphical representations enhance mutual understanding	Software is not physical. Only a prototype can provide a good image of the product.	Always start with GUI and then refine it.
Plans are useful if updated often	Old plans are not used and do not provide control on project	Revise plans at least at every start of a new iteration.
Time consuming (costly) to agree on what to do	Discussion is stopped without proper reason	Use iterations as means to achieve agreement.
The customer's project manager is passive and reluctant to take any decisions regarding the project.	Lack of guidance and clear direction for the project from customer side. Business decisions left to developers and the system may never be used.	This might result because of lack of knowledge about CO or lack of trust in the technology. The use of prototypes should provide the validation necessary to gain commitment.
Project manager has no commitment in the project.	No effort to involve domain experts and end-users in the project. Business decisions left to developers and the system may never be used.	Showing gradual improvements in the software can act as stimulus for the project manager to be more involved in the project.

References

1. C.M. Beath, Orlikowski W.J., *The Contradictory Structure of Systems Development Methodologies: Deconstructing the IS-User Relationship in Information Engineering*, Information Systems Research, Vol. 5, No. 4, 1994 .
2. K. Beck: *Extreme Programming Explained: Embrace change*, 1.ed., Addison Wesley, 1999.
3. K. Beck and W. Cunningham: *A Laboratory For Teaching Object-Oriented Thinking*, Proceedings of OOPSLA 89, 1989.
4. S. Bennett, S. McRobb and R. Farmer: *Object-Oriented Systems Analysis and Design using UML*, 1. ed., McGraw-Hill, 1999.
5. B. Bhoem, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, 1988
6. R.J. Boland, Tenkasi R.V., *Perspective Making and Perspective Taking in Communities of Knowing*, Organization Science, Vol. 7, No. 4, 1995
7. J.S. Brown, Diguid P., 2001, *Knowledge and Organization: A Social-Practice Perspective*, Organization Science, Vol. 12, No.
8. P.R. Carlile, *A Pragmatic View of Knowledge and Boundaries: Boundary Objects in New Product Development*, Organization Science, July/August 2002
9. A. Carugati, *New Challenges for the Management of the Development of Information Systems Based on Advanced Mathematical Models*, proceedings of the Global Information Technology Management (GITM 2002) Conference, New York, USA, 2002a.
10. A. Carugati, *Information System Development: Can Traditional Project Management Practices be Successful in Distributed Organizations?* Proceedings of the Seminar in Operation Management and Innovation, Fredericia, Denmark, 2002b.
11. A. Carugati, *Perspectives of IT Artifacts: Information Systems Based on Complex Mathematical Models*, proceedings of the Business Information Technology Management (BIT 2002) Conference, Manchester, United Kingdom, 2002c.
12. P. Checkland, J. Scholes, *Soft Systems Methodology in Action,* Wiley, 1999
13. Chic-2: *Engineering of Optimization Projects*, http://www-icparc.doc.ic.ac.uk/chic2/chic2_methodology/, 1999.
14. C. Collins and R. Miller: *XP Distilled*, http://www.rolemodelsoftware.com, 2001.
15. DSDM Consortium: Dynamic Systems Development Method, http://www.dsdm.org, 2003.
16. M. Fowler: *Keeping Software Simple*, Dr. Dobb's TechNetCast, 2000.
17. M Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts: *Refactoring: Improving the Design of Existing Code*, 1.ed., Addison-Wesley, 1999.
18. M. Fowler and K. Scott: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2.ed., Addison-Wesley, 1999.
19. J. Hansen and T. H. Kristensen: *Crane Scheduling for a Plate Storage in a Shipyard: Modelling the Problem*, Technical Report IMM-04-03, 2003.
20. J. Hansen and T. H. Kristensen: *Crane Scheduling for a Plate Storage in a Shipyard: Solving the Problem*, Technical Report IMM-11-03, 2003.
21. J. Hansen and T. H. Kristensen: *Crane Scheduling for a Plate Storage in a Shipyard: Experiments and Results*, Technical Report IMM-12-03, 2003.

22. S. Hert, L. Kettner, T. Polzin and G. Schäfer: ExpLab – A Tool Set for Computational Experiments, ver. 0.6, http://explab.sourceforge.net/, Max-Planck-Institut für Informatik, 2002.
23. F. S Hillier and G. J. Lieberman: *Introduction to Operations Research*, 7.ed., McGraw-Hill, 2001.
24. J. Hughes, T. Wood-Harper, *An Empirical Model of the Information Systems Development Process: a case study of an automotive manufacturer*, Blackwell Publishers Ltd. Oxford, 2000.
25. R. E. Jeffries: *eXtreme Testing – Why aggressive software development calls for radical testing efforts*, Software Testing & Quality Engineering, 1999.
26. K. Kellogg, W. Orlikowsky, J. Yates, *Enacting New Ways Of Organizing: Exploring The Activities And Consequences Of Post-Industrial Work*, Academy of Management Conference Proceedings, Denver, USA, 2002.
27. K. Marriott and P. J. Stuckey: *Programming with Constraints – An introduction*, 2. ed., MIT Press, 1998.
28. J. Mylopoulos: Conceptual Modelling and Telos, Chap. 2 in: P. Loucopoulos and R. Zicari (eds.) Conceptual Modelling, Databases and CASE, John Wiley, 1992.
29. G. L. Nemhauser and L. A. Wolsey: *Integer and Combinatorial Optimization*, John Wiley and Sons, 1988.
30. I. Nonaka, Takeuchi H.: *The knowledge Creating Company: How the Japanese Companies Create the Dynamics of Innovation*, Oxford University Press, New York, 1995
31. M. Pidd: *Computer Simulation in Management Science*, 4.ed., John Wiley & Sons, 1998.
32. M. Pirlot: *General Local Search heuristics in combinatorial optimization: a tutorial*, Belgian Journal of Operations Research, Statistics and Computer Science, 32, 1992.
33. J. Pries-Heje: *Nyere systemudviklingsmetoder: Hvad følger efter vandfaldsmodellen og strukturerede metoder,* Økonomistyring og Informatik, 11, Nr. 2, 1995/1996
34. D. Remenyi, T. White, M. Sherwood-Smith: Information Systems Management: The Need for a Post-Modern Approach, International Journal of Information Management, Vol. 17, No. 6, 1997
35. D. Ryan: Presentation at Nordic Summer Course on Applied Optimization and Modelling, Bornholm, Denmark, 2002.
36. J. S. Valacich, J. F. George and J. A. Hoffer: *Systems Analysis & Design*, Prentice Hall, Upper Saddle River, 2001.
37. K. Weick, *The Social Psychology of Organizing* (second edition), McGraw-Hill, New York, 1979
38. E. Wenger: *Communities of Practice and Social Learning Systems*, Organization, Vol. 7, No. 2, 2000.
39. H. P. Williams: *Model Building in Mathematical Programming*, 4.ed., Wiley, 1999.