Design, Implementation and Comparison of Randomized Search Heuristics for the Traveling Thief Problem

Rasmus Birkedal

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The Traveling Thief Problem (TTP) – a composition of the Traveling Salesman (TSP) and Knapsack (KP) Problems – has been recently proposed as a benchmark problem to feature a specific type of complexity – called *interdependence* of component problems – which it is claimed is typically missing from the benchmark problems used to demonstrate the performance of metaheuristics; particularly nature-inspired heuristics.

In this thesis is presented some relevant background on the TSP and KP, and a more careful literature study of the TTP. The study gives an impression of what strategies are expected to deal well with the interdependence, and which ones are not. Particularly, it is conjectured that heuristics that ignore the interdependence, and focus on solving the component problems in isolation, will fare poorly. However, this has not been demonstrated convincingly.

The well known, nature-inspired metaheuristic Ant Colony Optimization (ACO) is adapted as an algorithm for TTP. This adaption is documented and the thesis concludes with a computational study of the implemented algorithm. Said briefly, the idea is to base the algorithm around the naive approach, which solves the component problems in isolation in some nondeterministic way, yielding a solution for the TTP, and repeats this process. When this isolation is broken, the sequence of component-solutions are, conceptually, "chained" together, the construction of each being guided slightly by the previous solution to the other component problem. As the implementation is consistent with this description, the "chain" can be safely broken by reinstating the isolation, yielding a functional algorithm, and the two can be compared. This will facilitate a comparison that is well suited to concluding that isolating the component solvers sacrifices

easily realizable performance gains.

For this approach to work, it is necessary that at least one of the subproblems can be solved very efficiently. However, no method exists in the literature which deals specifically with the TSP as a component of the TTP, and those that exist for the KP are not good enough. Thus the existing algorithms *Simple Heuristic* and *Density-Based Heuristic* are taken, by subtle adjustments, to realize a much greater potential than previously demonstrated.

Because the TTP arose to answer a call for better benchmark problems, an excellent suite of such problems exists from [PBW+14]. For the nine problems studied here, the results achieved are far better than previous results, and I think that it is safe to conclude that mine is the best published algorithm. But there is yet very little competition.

Summary (Danish)

The Traveling Thief Problem (TTP) – en komposition af The Traveling Salesman Problem (TSP) og Knapsack Problem (KP) – er for nyligt foreslået som et nyt basisproblem der præsterer en bestemt type af kompleksitet – kaldet *indbyrdes afhængighed* af komponent problemer – der påstås typisk at mangle i de basisproblemer der benyttes til at demonstrere metaheuristikkers evne til at præstere. Det er især de naturinspirerede heuristikker der henvises til.

Denne afhandling præsenterer relevant baggrundsviden om TSP og KP, samt en mere omhyggelig litteraturanalyse af TTP. Analysen giver et indtryk af hvilke strategier der bedst håndterer den indbyrdes afhængighed. Mere konkret påstås det, at heuristikker der ignorerer den indbyrdes afhængighed for at fokusere på isoleret at løse komponentproblemerne vil klare sig dårligt. Dette er dog ikke demonstreret overbevisende.

Den velkendte, naturinspirerede metaheuristik, Ant Colony Optmization (ACO), vil blive tilpasset som algoritme til TTP. Denne tilpasning dokumenteres, og afhandlingen konkluderes, med en analyse af den praktiske præstation af den implementerede algoritme. Kort sagt er ideen at basere algoritmen omkring den naive tilgang hvor komponentproblemerne løses isoleret på en ikke-deterministisk måde der leverer en løsning til TTP, og derefter gentages. Når isolationen derpå brydes, vil sekvensen af løsninger til komponentproblemerne konceptuelt "kædes"sammen, således at deres konstruktionen ledes af den sidst sete løsning til det andet komponentproblem. Da implementationen stemmer overens med denne beskrivelse, kan "kæden"sikkert brydes ved at genindføre isolationen, for derved at opnå en tilsvarende algoritme således at der er to der kan sammenlignes eksperimentelt. Sådan en sammenligning egner sig til at konkludere, at den

isolerende løsning går glip af en let realiserbar præstationsforbedring.

For at denne tilgang kan virke, er det nødvendigt at mindst ét af underproblemerne kan løses meget effektivt. Der findes dog ingen metode i litteraturen der gør dette specifikt for TSP i rollen som komponent i TTP; og de der findes for KP er ikke tilstrækkeligt gode. Derfor tages de eksisterende algoritmer – *Simple Heuristic* og *Density-Based Heuristic* – til et højere præstationspotentiale, gennem subtile justeringer.

Fordi TTP opstod for at besvare en efterspørgsel på bedre basisproblemer, findes en fremragende suite af sådanne problemer fra [PBW+14]. For de ni problemer undersøgt i afhandlingen, er det opnåede resultat langt bedre end de forudgående, og jeg tror at det er sikkert at konkludere at min er den bedste publiserede algoritme. Dog er der endnu kun få konkurrenter.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with understanding and solving the newly proposted Traveling Thief Problem. The aim is to provide the first serious attempt of an algorithm that solves large instances of the TTP as a whole.

Lyngby, 26-June-2015

Rasmus Birkedal

Acknowledgements

I would like to thank my advisor, Carsten Witt, for his support throughout this masters project. Carsten has with his broad understanding of the field provided perspective that has enabled me to better found this work into the existing literature.

Due to my tendency for tunnel-vision, I have made mistakes rather sought solutions. During our conversations, Carsten has exposed these, saving me from much confusion. For the same reason, I have had trouble staying focused on the big picture. But Casten has mentioned this every so often, and I credit to this in part the wholeness of the final thesis – for this aid I am especially grateful.

I would also like to thank my family and friends for their general support – and especially my dad for giving solid and well-founded advice.

Finally I thank Carsten for introducing me to the Traveling Thief Problem in the first place. It has been fun to work with, and this project has definitely been a positive and challenging experience for me.

Contents

Summary (English) i

Summary (Danish) iii

Preface v

Acknowledgements vii

1 Introduction 1

2 Background - Component Problems 5
2.1 The KP - A Shallow Introduction 6
2.1.1 Greedy Heuristic 7
2.1.2 Random Local Search 8
2.1.3 (1+1) EA 10
2.2 The TSP - A Shallow Introduction 10
2.2.1 2-opt 11
2.2.2 The Lin-Kernighan Heuristic 12
2.2.3 Ant Colony Optimization 19

3 Background - The Traveling Thief Problem 23
3.1 An Explanatory Example 25
3.2 Calculation of the Objective Value 28
3.3 Algorithms for the TTP 29

4 Design of a Complete Solver for the TTP 35
4.1 Overview 35
4.2 An Improved Packing Heuristic for The Traveling Thief Problem 37
4.3 A Faster GDH 43

4.4 Design of Visualization software 44

5 Implementation **47**
5.1 Framework and Objective Function 47
5.2 Lin-Kernighan Algorithm for the TSP 48
5.3 ACO Algorithm for the TSP 48
5.4 GDH and HH for the KP component of the TTP 49
5.5 Visualization Software 49

6 Experiments **51**
6.1 The Experimental Setup 51
6.2 Computational Study of the Packing Plan Algorithms 52
6.3 Computational Study of ACO and LK for the TSP 56
6.4 Computational Study ACO for the TTP 57

7 Discussion **59**
7.1 IGDH and IHH 59
7.2 The ACO-based TTP Algorithm 60

8 Conclusion **63**

Bibliography **65**

Chapter 1

Introduction

This text documents a particular approach to solving the Traveling Thief Problem (TTP).

The TTP was presented for the first time in [BMB13], in 2013. It is a *composite problem* that *composes* two older and more familiar optimization problems, the Traveling Salesman Problem (TSP) and Knapsack Problem (KP). In the remainder of this text, these two will be called the *component* or *sub-problems* to emphasize their role as part of the TTP. Two models, TTP_1 and TTP_2, were proposed in [BMB13]. Only the first is considered here, following the trend of [PBW+14] and [BMPW14].

The term "Traveling Salesman Problem" is taken from nineteenth-century writings concerned with the logistic challenges of actual salesmen. These needed to tour a set of cities as efficiently as possible. As a mathematical problem, the solution is the shortest tour which visits all cities from a given set exactly once.

This general formulation of the TSP has turned out to abstract many specific problems. Since solving problems efficiently often means taking advantage of specific characteristics, less general variants of the TSP are often considered. A TSP can be *symmetric*, meaning that all distances are independent of direction. It can be *metric*, meaning that any distance between a pair of distinct cities is symmetric, greater than zero, and obeys the triangle inequality. Even more

strictly, an *Euclidean* TSP requires that the distance is the Euclidian distance[1], meaning that it is calculated by use of the Pythagorean theorem. The TSP instances considered in this text are Euclidean and therefore symmetric.

The knapsack problem provides a container – the knapsack – of limited capacity, and a set of items – each associated with a *weight* and a *profit* – and asks for a subset of the items that will fit in the knapsack, maximizing the sum of the profits of the packed items. Among variants of the KP, the *unbounded* KP permits inclusion of any number of copies of each item, as long as the knapsack capacity is not exceeded, while the *bounded* KP limits the number of copies to some finite amount (that is permitted to vary among the items). As a special case of the bounded KP variant, the *0-1 knapsack problem* sets the limit to one for all items. It is the 0-1 knapsack problem that constitutes the KP component of the TTP, and so, when nothing else is stated, the term KP is used in the following to refer to the 0-1 knapsack problem.

The TTP combines these two problems. It involves a set of items distributed among a set of cities. In this context it is a *thief*, not salesman, who must visit each city exactly once. The thief is equipped with a knapsack, which is packed *during* the tour. As the amount of free space in the knapsack decreases, so does the speed of the thief. The value which must be maximized is the total item profit minus the total travel *cost*. The cost is the time required to complete the tour multiplied by a constant called the rent rate.

Each of the two component problems are NP-hard. Thus it is believed that no polynomial-time algorithm exists for them. However, due to the ubiquitousness of the problems, they have been the subject of much study, and many polynomial-time heuristic algorithms, which do not guarantee optimal solutions, exist. Some of these are presented in Chapter 2, while Chapter 3 presents algorithms for solving the TTP itself.

A major goal of this project is to use heuristic algorithms for the TSP and KP as subroutines, and combine them into an overall, *composite* algorithm for the TTP. To help the subroutines better deal with the interdependence of the component problems, they are put into a generic algorithmic framework based on Ant Colony Optimization (ACO), which is a nature inspired metaheuristic. ACO can be applied to construction algorithms, which build solutions by iteratively and irrevocably adding components to an initially empty solution. In the case of the TSP, for example, a construction algorithm typically builds a tour by adding one city at a time. At each step, the choice of component is randomized, with extra weight given to components that are better according to

[1] Note that Euclidian TSP benchmarks are often restricted to two dimensions. As are the benchmark problems used in this thesis.

some heuristic, and to components with more *pheromone*. It is the pheromone concept which gives rise to the name of ACO. Real ants leave behind a trail of pheromones that serve to guide other ants. To model this inspiration, after a number of solutions have been constructed, some of them are chosen as being better than the rest, and the components that are a part of the chosen solutions are marked by an additional level of pheromone, while other components have their level decreased. This process of decreasing the level of pheromone is called *evaporation*. A more thorough introduction to ACO is given in Chapter 2.

Alternatives to ACO in this role will not be explored. The choice was made because of the versatility of ACO – it has been shown to perform well on a wide range of problems [DS10]. Additionally, the introductory paper, [DMC96], as well as future papers [SH00][BHS97], demonstrate the proposed ACO algorithms by application to the TSP, thus giving relevant context to the case at hand.

The design of the composite algorithm is documented in Chapter 4, while its implementation is presented in Chapter 5

As explained in [BMB13], the TTP is a benchmark problem intended to address concerns presented in [Mic12] that there is a gap between theory and practice in the field of metaheuristics for combinatorial optimization problems. In [BMB13] it is claimed that the definition of complexity is a main difference between the benchmark problems used in theoretical work, and the real-world-problems to which the results are intended to be applied in practice. For the benchmark problems, it is claimed, there is a tendency to equate complexity with size – e.g. number of cities for the TSP – while real world problems usually include additional sources of complexity, such as the interdependence of component problems, as intentionally featured by the TTP.

Consequently, a solver for the TTP which first solves the component problems to optimality in isolation, and subsequently combines the results into one for the TTP, is *not* guaranteed to yield an optimal solution [BMB13]. To see this, consider the TTP-instance where all items have small profits and large weights, so that any optimal solution involves picking none of the items. For the ordinary KP, if any item fits in the knapsack, the optimal solution contains at least one item. The isolated KP solver then produces a non-empty knapsack, the inclusion of which into the solution for the TTP-instance precipitates suboptimality. Less pathological examples are provided in [BMB13] and [PBW+14]. Further, some of the benchmark problems for the TTP that are used later, appear to be solved best by leaving plenty of extra space in the knapsack.

These considerations apply to exact solvers, but the TTP was introduced specifically to challenge heuristic solvers, and it is not as easy to reject naive application of heuristic algorithms to the component problems. In fact, such approaches

have been tried with modest success in [PBW+14] and [BMPW14], albeit with primary focus on achieving an increased understanding of the nature of the TTP, and with little focus on actually finding good solutions. The algorithm, *CoSolver*, was introduced in [BMPW14], and provides, to an extent, a contrast, by allowing "separate modules/algorithms" to "communicate with each other", as opposed to the other algorithms which do not make use of such communication.

So for the TTP there exist different kinds of heuristic solvers which feature different levels of communication, and there appears to be an expectation that such communication is necessary. As new algorithms are developed, it would be interesting to see how well this expectation is met. To facilitate this, another algorithm is introduced in Chapter 4, which takes care to feature none of this communication, while offering significantly better performance than any previous such algorithms for the TTP.

The comparison is based on a computational study that is documented in Chapter 6, and it is discussed in Chapter 7. The study makes use of nine benchmark problems, chosen from the set of 9720 presented in [PBW+14].

Finally, Chapter 8 concludes the thesis.

Chapter 2

Background - Component Problems

This chapter presents background for the two component problems – the traveling salesman problem (TSP) and the knapsack problem (KP). The composite problem, the traveling thief problem (TTP), is dealt with in the next chapter. The TTP is a "young" problem – the introducing paper, [BMB13], is from 2013 – and I have found only three papers (the two others being [PBW+14] and [BMPW14]) that deal with it specifically, so the background given in this thesis aims to cover the current state of the art for the TTP. For the two component problems, however, too much material exists for this to be possible, so I present primarily procedures and heuristics that are used in the following chapters.

All three problems are *combinatorial optimization problems*. Thus a number of *feasible* solutions generally exist, not all of which are *optimal*. For example, a solution to the TSP is any tour. As long as all cities are visited exactly once, the tour is a solution, although not necessarily optimal. These solutions can then be compared based on a *fitness* or *objective value*, which defines the optimum, but also orders feasible, suboptimal solutions. Algorithms presented in this chapter are optimization algorithms; they repeatedly construct new solutions based on some currently best solution, which is replaced whenever an improvement is found.

The decision variants of the considered problems are NP-complete[1], so determining whether the achieved result is optimal is, in general, a super-polynomial-time operation (unless P = NP). Thus, faster algorithms must contend with being unable to know whether an achieved solution is optimal or not[2].

Conceptually, some of the algorithms are search procedures that search the space of solutions for an optimum. In this case the search may be unable to advance from a suboptimal solution, because any reachable solution is worse than the current one. However, since all reachable solutions are worse, this is in practice often taken as an indication that the reached solution is *good*. In any case, if the search cannot advance, the solution is called a *local* optimum, with the actual optimum being called the *global* optimum. Note that the global optimum is always a local optimum.

2.1 The KP - A Shallow Introduction

An instance of the KP comprises the following.

a) A knapsack of capacity W.

b) A set, M, of m items.

c) To each item, $I_i \in M$, is associated a pair, (w_i, p_i), being the weight and profit of the item.

A solution, P, is a subset of M subject to the constraint that the sum of weights of items in P is at most W, such that the sum of the profits of those items is maximized. In other words, P is a solution iff $c(P) =$ true as given by equation 2.1 and it's total profit, $c(P)$, given by equation 2.2, is at least as great as the total profit, $p(Q)$, of any $Q \subseteq M$ such that $c(Q) =$ true.

$$c(P) = \sum_{I_i \in P} w_i \leq W \tag{2.1}$$

[1] In all three cases, the decision variant asks something like "does a solution exist which is at least *so* good?".

[2] There are optimization algorithms which guarantee an optimal solution. As an example, branch and bound, an algorithm design paradigm, can be used to solve TSPs and KPs exactly. It does this in super-polynomial time however.

$$p(P) = \sum_{I_i \in P} p_i \tag{2.2}$$

The KP is unusual among NP-hard problems, because there exists a pseudo-polynomial algorithm for it – the well known dynamic programming algorithm. The benchmark problems that are solved in later chapters lower bound W by $\frac{m}{11}$[3] – i.e. W is linear in m – so, assuming that the running time of the dynamic programming algorithm is $O(mW)$, the running time in the present context is at least a quadratic function of m.

However, recall from the discussion in chapter 1 that guaranteed optimality of component problem solutions does not transfer into the composite problem. For this reason, the dynamic programming algorithm is discarded in favor of heuristic approaches, which do not guarantee optimality of solutions, but are faster. Three such heuristics are presented in the following three sections.

2.1.1 Greedy Heuristic

There is a simple yet effective greedy approximation scheme for the KP. Each item, I_i, is assigned a score, $s_i = \frac{p_i}{w_i}$, and the knapsack is filled repeatedly with the highest scoring, available item, until no item fits. Algorithm 1 below gives an approach where the items are sorted prior to beginning the process of adding them to the knapsack. This makes the for-loop in line 5 a simple, linear traversal of the m items, and the worst-case, asymptotic running time is due to the sorting: $O(m \log m)$.

While this pseudocode assumes the ordinary 0-1 knapsack problem, a similar formulation can be made for the bounded and unbounded KP variants. For the unbounded KP, this greedy approach is a $\frac{1}{2}$-approximation – it guarantees a total profit that is at least half as good as the total profit of an optimal solution. But for the bounded KP, there is no such guarantee. To see this, consider the following 0-1 knapsack problem instance.

- I_1: $w_1 = 1$, $p_1 = 2$
- I_2: $w_2 = 100$, $p_2 = 100$
- $W = 100$

[3] The knapsack capacity, W, of benchmark TTP-instances from [PBW+14] is $\frac{mC}{11}, C \in \{1, 2, \ldots, 10\}$ if all item weights are equal to 1. Otherwise it is greater.

Algorithm 1 Greedy KP Heuristic

Input: A KP-instance **Output:** A solution, P

```
1: for all I_i ∈ M do
2:     s_i = p_i / w_i
3: I[] := array of all I_i ∈ M, sorted in descending order of s_i
4: P := ∅
5: W_P := 0
6: for all I_i ∈ I[] do
7:     if W_P + w_i ≤ W then
8:         Add I_i to P
9:         W_P := W_P + w_i
10: return P
```

The first item, I_1, has the best score, so the greedy algorithm will pack it first, leaving room for no other items. Thus, the algorithm produces $P = \{I_1\}$, with total profit 2. But the best solution is $P = \{I_2\}$, with total profit 100, which is better by a factor of $\frac{p_2}{p_1}$. As $W = w_2 = p_2$ increases to infinity, so does the factor by which this example is solved worse by the greedy heuristic than any optimal solver. It follows that no non-trivial guarantee can be given for application of the greedy heuristic to the bounded KP.

This worst-case behavior is rare, and the greedy heuristic sees use even in application to bounded KPs. Particularly, it provided foundation for the algorithms SH and DH, which will be introduced later.

2.1.2 Random Local Search

Random Local Search (RLS) is an algorithm that belongs to the class of *genetic algorithms* – a sub-class of the nature inspired metaheuristics. Thus RLS can be applied in many contexts with varied success. In fact, for solving the KP, RLS is likely a poor choice[4], while it is much better at solving the subsuming KP-sub-problem of the TTP, as we shall see later. Since it is useful in this latter case, it is convenient to introduce RLS as an algorithm for the ordinary 0-1 KP.

For genetic algorithms in general, solutions are often represented as a string of bits. This can be easily done for the KP. The most straight forward way is to have the i'th bit represent inclusion of the i'th item in the solution, with

[4] A more general version of RLS – which uses a population of size greater than one, and applies crossover – I do not claim to be a poor choice.

1 meaning that it is included and 0 meaning that it is not. For example, for $m = 5$, the string $[1, 0, 1, 0, 0]$ represents the solution $P = \{I_1, I_3\}$.

RLS repeatedly *mutates* some initial solution by flipping one, uniformly at random chosen bit. After each flip, the solution is evaluated by use of a *fitness function*, and the flip is undone, unless the evaluation indicates improvement. Equation 2.3 gives the fitness function for the KP, and Algorithm 2 gives a detailed description of application of RLS to the KP.

$$f(P) = \begin{cases} p(P) & \text{if } c(P) \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

Algorithm 2 Random Local Search (RLS)

Input: A KP-instance with m items and knapsack capacity W **Output:** A string of m bits that represents the solution

1: P is initialized to a bit-string of length m that contains only zeros
2: **while** Terminal condition is not met **do**
3: $P^* := P$
4: r is chosen uniformly at random from $\{1, 2, \ldots, m\}$
5: the r'th bit of P^* is flipped
6: **if** $f(P^*) \geq f(P)$ **then**
7: $P := P^*$
8: **return** P

There is no general way of knowing how many iterations of RLS should be used – hence the vague condition of the while-loop in line 2. In practice, the rate of improvement per iteration declines the longer the algorithm works on a particular solution; so the algorithm can simply be stopped after a given number of consecutive iterations without improvement.

Regarding application to the KP, algorithm 2 adds items to the solution at random, with no preference for *good* items, and never removes any item, since that always results in a decrease to the fitness of the solution (assuming P does not exceed the knapsack capacity, which is guaranteed if $W > 0$). Eventually, enough items will have been added that there is room for no more in the knapsack, and no future iteration will add an item to the solution. Since the algorithm is incapable of removing items, it is stuck, and all future iterations are no-ops. In this case we say that a *local optimum* has been reached, whether or not an actual optimal solution to the KP-instance was found. If this was the case however, we further say the the local optimum is a *global optimum*.

2.1.3 (1+1) EA

The algorithm (1+1) EA is different from RLS only in the way the solution is mutated. A single mutation iterates over every bit in the solution, flipping each independently with probability $\frac{1}{m}$.

Algorithm 3 (1+1) EA

Input: A KP-instance with m items and knapsack capacity W **Output:** A string of m bits that represents the solution

1: P is initialized to a bit-string of length m that contains only zeros
2: **while** Stopping criteria is not met **do**
3: $P^* := P$
4: **for all** $i \in \{1, 2, \ldots, m\}$ **do**
5: the i'th bit of P^* is flipped with probability $\frac{1}{m}$
6: **if** $f(P^*) \geq f(P)$ **then**
7: $P := P^*$
8: **return** P

Since every bit is potentially flipped in a single mutation, a single mutation can yield any solution, regardless of what P is mutated. So for (1+1) EA there are no local optima that are not global. Despite the superiority to RLS in this regard, (1+1) EA suffers the drawback that a mutation involves producing m random numbers, while RLS only needs one such. A compromise that I have found to be effective in practice (in application to the TTP), is to use RLS to quickly find a local optimum, and then alternate between RLS and (1+1) EA afterward.

2.2 The TSP - A Shallow Introduction

This section introduces the *Euclidean* traveling salesman problem as well as a selection of optimization algorithms for it.

A problem instance is traditionally given as a set, N, of n cities along with *coordinates* – a pair of integers – for each. The distance from city i to city j is then calculated using the Pythagorean theorem – this makes it possible to avoid storing a distance for every pair of cities; e.g. in an n by n matrix. Since a distance calculated in this way may be a real number – e.g. if the catheti are of length 1, the hypotenuse is of length $\sqrt{2}$ – some finite level of precision must be agreed upon, so that researchers can accurately compare results. For this reason benchmark problems often define the distance to be rounded to an

integer. For example, the distance is rounded down in "symmetric traveling salesman problem" instances from the widely used set of benchmark problems TSPLIB [Rei95]. Confusingly, the benchmark problem instances of the TTP presented in [PBW+14] round the distance up, despite basing the instances on ones from TSPLIB.

The solution to a TSP-instance is a *tour* of *all* n cities for which the total distance is minimal, i.e. such that no tour has shorter length. A tour is an undirected Hamiltonian cycle, i.e. a cyclic path of n edges which visits each city exactly once. A path is a *sequence* of edges. But a *set* of edges known to constitute a path that does not visit any city twice can represent only that path and that of opposite direction. Since a TSP-tour has no direction, it can be represented unambiguously by a set of edges. Note that this does not apply to a TTP-tour, for which a reversal of direction may significantly impact the tour quality and overall solution fitness.

Alternatively, a tour can be represented as a permutation of the numbers from 1 to n, implying an edge between cities that are consecutive in the permutation, and an edge from the first to the last city of the permutation. Thus, a tour of an eight-city instance that traverses the cities in order of index can be represented as | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |.

2.2.1 2-opt

2-opts [Cro58], 3-opts [Lin65], and 4-opts [LK73] are used as subroutines in algorithms that solve the TSP. Generally, a k-opt is a *move* that replaces k edges of a tour, i.e. k edges are deleted, disconnecting the tour, and a disjoint set of k edges are added such that the result is a valid tour. Of course, there is only an improvement if the sum of the distances of the deleted edges is greater than that of the added edges.

The 2-opt is often presented as a method for resolving the problem of two crossing edges. Figure 2.1 gives an example application of a 2-opt that does this.

When using the array representation, a 2-opt is applied by reversing the order of the cities between the two edges. For example, the 2-opt applied in figure 2.1 changes this tour: | 1 | 2 | 6 | 5 | 4 | 3 | 7 | 8 | to this tour:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |. Note that for any cycle there are two choices for "the cities between two edges". This other choice results, for the same 2-opt, in the tour | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |. Although it does not apply in

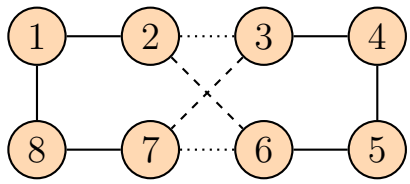


Figure 2.1: The circles are cities of a TSP-instance. A tour is given by the solid edges, and the two crossing, dashed edges, $\{2,6\}$ and $\{3,7\}$. A 2-opt that deletes the dashed edges, *must* introduce the dotted edges, $\{2,3\}$ and $\{6,7\}$. If, for example, the edges $\{2,7\}$ and $\{3,6\}$ were introduced instead, the result would not be a tour, as there would be two disconnected components.

this specific case, one of the alternatives typically involves reversing the order of fewer cities than the other, and an implementation of the 2-opt should take advantage of this to save computation time, as explained in [ABCC99].

Regarding the choice of k, it must be in the interval $\{2,3,\ldots,n\}$, as deleting only one edge leaves only one feasible choice for the edge to be added – the same edge as the one that was deleted – and, because there are n edges in the tour, it does not make sense to delete more than n edges. However, in practice, k is usually chosen to be small, as computational complexity rises sharply with k [LK73].

Algorithm 4 A structure for TSP optimization algorithms

```
1: Generate initial tour T as a random permutation of the numbers [1,...,n]
2: while Stopping criteria is not met do
3:     Apply k-opt to T, yielding T'
4:     if T' is shorter than T then T := T'
5: return T
```

2.2.2 The Lin-Kernighan Heuristic

The Lin-Kernighan Heuristic (LK) [LK73] for the TSP is a local-search procedure which uses the structure of algorithm 4. It remained the best choice for producing approximate solutions to the TSP in the following two decades [ACR03].

Algorithm 5, which gives pseudocode for a simple implementation of LK, delegates the bulk of the work to procedure 7, which searches for a k-opt, and, if

successful, applies that k-opt.

Algorithm 5 Lin-Kernighan

1: Generate initial tour T as a random permutation of the numbers $[1, \ldots, n]$
2: $C := N$ ▷ C is the set of cities that have yet to be processed
3: **while** $C \neq \emptyset$ **do**
4: $t_1 \in C$
5: $C := C/\{t_1\}$
6: call procedure 7 with input: T, t_1
7: **if** the procedure was successful **then** $C := N$
8: **return** T

The cornerstone of LK is a heuristic procedure which searches for a k-opt as a *sequential exchange*, i.e. a sequence of k *exchanges*. A single exchange is illustrated in figure 2.2, and a version of it is detailed by procedure 6. Because it is edges that are exchanged, tours are represented by sets of edges in the following.

Procedure 6 Exchange

Input: A tour, T, and distinct cities t_1, t_2, and t_3, such that t_1 and t_2 are connected by an edge in T

1: Of the two cities connected to t_3 by an edge in T, t_4 is chosen as the city that will be reached first when traversing the tour in the direction $[t_1, t_2, ..]$ (see figure 2.2).
2: The edge from t_1 to t_2 is called x_1.
3: The edge from t_2 to t_3 is called y_1.
4: The edge from t_3 to t_4 is called x_2.
5: The edge from t_4 to t_1 is called y_2.
6: x_1 is removed from T
7: y_1 is added to T.

Note that once T, t_1, t_2, and t_3 – the input to procedure 6 – have been chosen, there is only one choice for each of t_4, x_1, x_2, y_1, and y_2. This is used in line 6 of procedure 7.

After an exchange is performed, the tour is invalid, so some other operation must follow. There are two options.

1. The tour is *closed*. To do this, add y_2 to the tour, and remove x_2 from the tour. After this, the tour has effectively been subject to a k-opt if procedure 6 was applied $k-1$ times before being closed with this option.

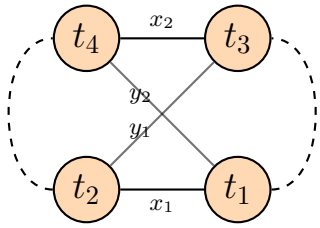


Figure 2.2: The circles are cities and the dashed lines represent the part of the tour which is not shown. The edges x_1 and x_2 are part of the tour. A single exchange is of x_1 with y_1, i.e., x_1 is removed from T and y_1 is added.

2. Choose t_3', set $t_2' = t_4$ and $t_1' = t_1$, and apply another exchange with input T, t_1', t_2', t_3'.

After each exchange, option two is always chosen, but only after evaluating how good the k-opt resulting from choosing option one would be. The best observed k is saved, and when the search eventually ends, the algorithm applies the corresponding k-opt. The criteria for ending the search involves the so called *gain*, g_i. Specifically, to the i'th exchange is associated $g_i = |x_1^i| - |y_1^i|$ – where x_1^i is the edge removed and y_1^i is the edge added – which is a value that quantifies the improvement achieved by applying that exchange. The continuance of the search is contingent on the *gain criterion*, which fails immediately before the i'th exchange iff the total gain, $G_i = \sum_{j=1}^{i} g_j$, is negative. With this G_i, the best k-opt found during the search is the one for which $G_k^* = G_{k-1} + |x_2^i| - |y_2^i|$ is maximized.

The value G^* is the improvement found, i.e. it is the amount by which the improved tour, T^*, is shorter than the original one. Thus, for any improving k-opt, G^*, which is a sum of a sequence of numbers, is positive. Lin and Kernighan derive the gain criterion from this observation. The derivation uses the following result: *" if the sum of a sequence of numbers is positive, then there exists a cyclic permutation of the sequence such that every partial sum is positive"*, which is proven in [LK73]. So if some sequence of exchanges yields a k-opt with positive G^*, there is a "rotation" of the sequence that a search will find without encountering a negative G_i for $i < k$. In practice, any such rotation is achieved by starting the search at an appropriate t_1.

This is the reason that the original Lin-Kernighan algorithm stops only after every choice of starting point fails consecutively – i.e., after n searches in a row with different t_1 fail to improve T. This is reflected in algorithm 5 by maintaining a candidate-set, C, which holds all cities that have yet to be processed as input to

Procedure 7 Sequential Exchange

Input: A tour, T, and a city, t_1 **Output:** Success iff T was improved

```
1:  for both choices of t_2 do
2:      G := 0                                              ▷ The gain
3:      G* := 0                                  ▷ The best improvement so far
4:      while true do
5:          choose a city t_3
6:          update variables t_4, x_1, x_2, y_1, y_2 to reflect t_2 and t_3
7:          G := G − |x_1| + |y_1|
8:          if G < 0 then break the while-loop
9:          perform exchange using procedure 6 with input: T, t_1, t_2, t_3
10:         G*' := G − |x_2| + |y_2|      ▷ Improvement achieved by closing T now
11:         if G*' > G* then
12:             G* := G*'
13:             T* := (T ∪ {y_2})/{x_2}
14:         t_2 := t_4
15:     if G* > 0 then
16:         T := T*
17:         return Success
18:     else
19:         Undo all changes made to T in the while loop
20: return Failure
```

procedure 7. The set holds initially all cities, and each time one is used as input to the procedure, it is removed from C. Whenever the procedure successfully improves the tour, C is set to once again contain all cities.

In [ABCC99] it is noted that this approach is too time consuming for larger instances. It is suggested that, after a successful sequential exchange, only a subset of the cities involved in the exchanges should be added to C. This trades off result quality in favor of faster algorithm running time.

After algorithm 5 stops, there is still a possibility of finding an even better tour, by restarting the algorithm with a new, random permutation for T. This concludes the description of the basics of the search procedure, but a few essential tweaks have yet to be explained.

Backtracking

The search procedure described thus far is very narrow, in the sense that few alternatives are tried before the search is abandoned altogether, and a new search is started with a different initial t_1. The original paper details a search that branches by backtracking. Some backtracking is already incorporated in the search as it is described thus far. The first line of procedure 7 essentially requires the procedure to be repeated for the other choice of t_2 (but with the same choice of t_1) if the first one does not lead to an improvement. In general, backtracking occurs when the gain criterion fails and no improvement has been found. When this happens, some t_i is chosen to be replaced with an alternative, while retaining choices for $t_j, j < i$, and the search continues from the new t_i. Generally, if there are multiple t_i's that can be replaced, the one with the highest i is chosen first. The following is a complete list of backtracking as described in [LK73].

- In the first exchange, both choices of t_2 are tried. Procedure 7 above already reflects this type of backtracking.

- In the first and second exchange, five choices per exchange are tried for t_3. The remaining exchanges do not backtrack on t_3. Thus the level of backtracking on t_3 can be given as a sequence $[5, 5, 1, 1, \dots]$, or just $(5, 5)$, indicating a level of backtracking of 5 for the first two exchanges, and a level of one for the remaining ones. In [ACR03], 8 alternatives for $(5, 5)$ are tested computationally. The results show that $(5, 5)$ has very good, overall performance, but they choose $(4, 3, 3, 2)$ for their implementation, because of slightly better performance in tests with long execution time.

- In the first exchange, both choices for t_4 are tried. Recall that procedure 6 defines t_4 unambiguously. But there is one other choice – the other neighbor of t_3 – which can be tried. However, using this alternative t_4 dramatically complicates the next two exchanges, where the choice of t_3 must be limited in a rather involved way. Furthermore, this choice precludes closing the tour with $k = 2$, and possibly with $k = 3$. The specifics are explained in [LK73] and [ABCC99]. Lin and Kernighan admit that this type of backtracking is relatively complicated to implement, but insist that the performance improvement is worth the effort. In my experience the improvement is relatively small, especially when fast execution time is important.

Non-overlap of Added and Removed Edges

Regardless of the level of backtracking, the description in [LK73] constrains the choice of t_3 by requiring every x_2 to not have been among the edges that were added in the current branch of the search, i.e., $x_2^i \notin \{y_1^1, y_1^2, \ldots, y_1^i\}$. Similarly, no y_1 that was previously deleted should be added, i.e., $y_1^i \notin \{x_1^1, x_1^2, \ldots, x_1^i\}$.

Choice of t_3

The choice of t_3 defines the edges x_2 and y_1. Even if the tour is closed after the exchange for which t_3 is chosen, x_2 will be removed from T. Thus, t_3 is chosen to maximize the difference in length of x_2 and y_1, viz. $|x_2| - |y_1|$. The algorithm described in the original paper chooses t_3 from a precomputed set of the five nearest neighbors of t_2, such that the difference is maximized. Alternative approaches exist. For example, the LKH implementation [Hel00] by Helsgaun qualifies edges not by their length, but by a measure, dubbed *α-nearness*, of proximity to a particularly calculated minimum spanning tree.

Choice of Starting Tour

In [ACR03], alternatives to using a random permutation as starting tour are studied. The tested starting tours, in order of performance, worst to best, are: *Random*, *Nearest Neighbor*, *Christofides*, *Greedy*, *Quick-Borůvka*, and *HK-Christofides*. Thus the random permutation provides the worst starting tour, and HK-Christofides, the best. I do not explain how to produce these tours, but make a few comments.

- The HK-Christofides tour is slow to calculate. Compared to the second best, Quick-Borůvka, it is shown to be slower by a factor of 800 on a 100,000 city instance.
- The results indicate that the random starting tour is the fastest to produce, while the nearest neighbor tour is the second fastest, and Quick-Borůvka just a little slower.
- The improvement of using a better starting tour is great when the algorithm is stopped quickly, but in long runs, the performance gap shrinks, and the random starting tour is not much worse than the others. I imagine that this is because random tours tend to be longer than the other ones,

which, in a sense, get a head start. But the value of the head start is inflated, as worse tours are more rapidly improved by the LK algorithm.

- If the method for producing the start tour can produce only very few unique tours, it potentially weakens repeated application of the LK algorithm, becuase it is deterministic – the same result is produced if the same start tour is used.

- Interestingly, the two best starting tours are calculated by use of minimum spanning trees. Thus, the heuristic that it is good to guide the LK toward minimum spanning trees exists in some form in both of [ACR03] and [Hel00].

Nonsequential Exchanges

While every sequential exchange is a k-opt, the other direction does not hold. For example, the so called *double bridge move* constitutes a 4-opt which is unattainable through a sequential exchange.

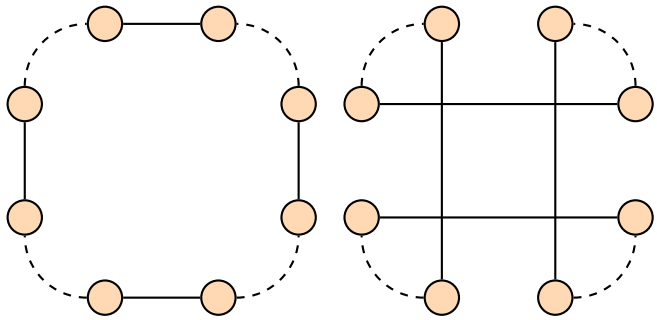


Figure 2.3: With the dashed lines indicating paths (not necessarily single edges), the 4-opt called a *double bridge move*, changes the tour on the left to the tour on the right

The original LK suggests appending to the end of the main search – algorithm 5 – another search, for which the aim is to find an improving double bridge move. Further work has generalized this, naming the double bridge one of many possible *kicks*, and it is suggested that, instead of restarting the algorithm from a new starting tour, applying a kick to a locally optimal tour, even if the result is worse, may allow LK to continue and reach a better local optimum; i.e., the kicked tour effectively provides the initial tour of the next application of LK. In [ACR03], these ideas are condensed, and the resulting heuristic is dubbed Chained Lin-Kernighan.

Crossover

A crossover-operator takes at least two distinct, parent solutions, and returns one or more child solutions by combining the parents. Genetic crossover in biological reproduction provides inspiration. If applied to tours – solutions to the TSP – a crossover-operator provides, conceptually, another way to *kick* tours. An example of successful application of crossover to the TSP is given in [WHH09] and [WHH10], which respectively introduce *Partition Crossover* and *Generalized* Partition Crossover.

The subject of crossover escapes the scope of this thesis. I mention it as a more recent development of the idea of chaining together applications of the LK algorithm. Additionally, I believe that crossover constitutes a prime candidate among procedures that *might* be successful in solving the TTP as a whole.

2.2.3 Ant Colony Optimization

Ant Colony Optimization (ACO) is a nature-inspired metaheuristic. Thus it is not designed specifically to solve the TSP, though its performance has historically been illustrated through application to the TSP [DMC96][BHS97][SH00]. The aim of this section is an introduction that establishes terminology and provides a foundation solid enough that later adjustments can be justified. To this end, the introduction here is to application of ACO to the TSP. A general and more thorough description and overview of ACO is provided in [DS10]. For the TSP, the main framework is the following.

Algorithm 8 ACO for the TSP

Input: A TSP-instance

```
1: while Stopping criteria is not met do
2:     for Every ant in the population do
3:         Construct a tour using procedure 9
4:         Optionally, apply local optimization, e.g. LK, to the constructed tour
5:     Update pheromones with procedure 10 followed by procedure 11
6: return The best seen tour
```

The tour construction is based on ordinary random walk. That is, some city is designated as the starting point, and the next city to visit is chosen from the yet unvisited ones at random, until all cities are visited. However, unlike ordinary random walk, the next city is not chosen *uniformly* at random. Rather,

a probability distribution is created at every step of the walk. The probability, p, of choosing a particular edge, e, depends on η, a heuristic value which for the TSP is usually $\eta(e) = \frac{1}{|e|}$, and τ_e, which is the amount of pheromone on e. If at a given step the set of feasible edges is E, the equation below gives the probability of choosing the edge e.

$$p(e, E) = \frac{\tau_e^\alpha \cdot \eta(e)^\beta}{\sum_{e' \in E} \tau_{e'}^\alpha \cdot \eta(e')^\beta} \tag{2.4}$$

In the above, α and β are parameters of the algorithm the ratio of which define the ratio of influence of the heuristic and pheromone information on the probability distribution. When β is high compared to α the heuristic information becomes more influential than the pheromone information, and vice versa. Furthermore, each of the two parameters have an important effect independently of the other, as a higher value helps contrast the distribution. For example, by increasing β, the degree by which short edges are deemed better than long edges is increased. At the extreme, when $\beta = 0$, only pheromone levels influence the distribution, and when $\beta \to \infty$, only edges, e, with $|e| \to 0$, are associated to nonzero probabilities. In [DMC96] it is concluded that, for the TSP, good choices are $\alpha \in \{0.5, 1\}$ and $\beta \in \{1, 2, 5\}$.

The tour construction is summarized below.

Procedure 9 Ant-Tour Construction

Input: A TSP-instance **Output:** A tour, T, as a sequence of cities

1: $C := N$ ▷ Initially, C is the set of all cities
2: $c \in C$
3: $C := C/\{c\}$
4: $T := [c]$ ▷ The tour is initially a sequence of one city
5: **while** $C \neq \emptyset$ **do**
6: $E = \{(c, c') \mid c' \in C\}$ ▷ The set of edges from c to any city in C
7: $e = (c, c')$, $e \in E$, is chosen with probability $p(e, E)$, given by equation 2.4
8: $c := c'$
9: $C := C/\{c\}$
10: $T :=$ concatenate$(T, [c])$ ▷ The new c is concatenated to the end of T
11: **return** T

The pheromone-update comprises evaporation followed by deposition of additional pheromone on edges that are part of the best tours. Evaporation involves the algorithm-parameter $\rho \in (0, 1]$, the *evaporation rate*, and is given below.

Procedure 10 Pheromone Evaporation

Input: A TSP-instance where N is the set of cities

1: **for all** $e \in N \times N$ **do**
2: $\tau_e := (1 - \rho) \cdot \tau_e$

This simulates the evaporation that would occur in realitiy, and helps avoid a stalemate, where the accumulated mass of pheromone of previous iterations is too great for further additions to be significant. As another measure to avoid stalemate situations, Max-Min Ant System [SH00], enforces upper and lower bounds, τ_{max} and τ_{min}, for the amount of pheromone on individial edges. The upper and lower bounds are not reflected by the pseudocode given here, but they are a part of the implementation presented in the next chapter.

With $|T| = \sum_{e \in T} |e|$ being the length of T, the below procedure details how pheromone is deposited on a set, S, of tours.

Procedure 11 Pheromone Deposition

Input: A set, S, of tours $T \subseteq N \times N$ of a single TSP-instance

1: **for all** $T \in S$ **do**
2: **for all** $e \in T$ **do**
3: $\tau_e := \tau_e + \frac{1}{|T|}$

While the solution proposed in the introducing paper, [DMC96], deposits pheromone on every tour, later papers, [BHS97][SH00], suggest an *elitist strategy*, in which deposits are made on only few tours, typically the iteration or global best.

Chapter 3

Background - The Traveling Thief Problem

Specifically, an instance of the TTP comprises the following.

- A set, $N = \{0, 1, .., n-1\}$, of n cities.
- For every pair of cities, i and j, a distance $d_{i,j} = d_{j,i}$ is defined.
- m items, $I_i \in M, i \in \{0, 1, .., m-1\}$.
- Every item, I_i, has a weight w_i, a profit p_i, and it is associated with exactly one city, $c_i \in \{1, .., n-1\}$, which is its location. Note that there are no items located at the first city, i.e. $c_i \neq 0$ for all i.
- The capacity, W, of the knapsack, which is the maximum weight the thief can carry.
- The renting rate, R, which is the cost for the thief of spending one unit of time.
- The maximum speed, v_{max}, and minimum speed, v_{min}, of the thief. In all cases addressed in the remainder of this paper, it is assumed that $v_{max} = 1$ and $v_{min} = 0.1$.

Some values are not explicitly a part of the TTP, but can be derived from the definition of the objective value which is given at the end of this section. One such is the speed, $v(w)$, of the thief as a function of the currently carried weight, w.

$$v(w) = v_{max} - \nu w \ , \ \nu = \frac{v_{max} - v_{min}}{W} \tag{3.1}$$

Another is the time, $t(d, w)$, it takes the thief to travel the distance d while carrying the weight, w.

$$t(d, w) = \frac{d}{v(w)} \tag{3.2}$$

A solution to an instance of the TTP comprises a tour, Π, and a packing plan, P. In the context of the TTP, it is practical to use the array representation of the tour; so it is a permutation of the cities, i.e. $\Pi = \pi_0 \pi_1 \dots \pi_{n-1}$, $\pi_i \in N$, and $\pi_i \neq \pi_j$ for $i \neq j$. Note that it is required that the tour starts at the first city, i.e. $\pi_0 = 0$. Also, notation will be abused by writing π_n for π_0.

The packing plan, P, has the same definition as the solution to the ordinary KP. It is called a packing *plan*, to emphasize the fact that, given a tour, there is an ordering imposed on the items of P – the order in which they are picked up by the thief. This is not a complete ordering, as the items of P located at city i, for some i, are picked up simultaneously. It will be useful to be able to conveniently extract the weight of these items, so the function, $w_P(i)$, is defined as the weight of the items located at city i which are in the packing plan, P.

To every solution is associated an objective value, $Z(\Pi, P)$. To define Z, we first define the so called *cumulate weight*, W_{π_i}, which is the total weight of all items in the packing plan, P, which are available at the first $i + 1$ cities of the tour, viz. $\pi_0 \pi_1 \dots \pi_i$.

$$W_{\pi_i} = \sum_{k=0}^{i} w_P(\pi_k) \tag{3.3}$$

In the expression for $Z(\Pi, P)$ below, a binary variable, $y_i \in \{0, 1\}$, is used to indicate whether item I_i is packed, viz. $y_i = 1 \Leftrightarrow I_i \in P$. Note that the

constraint from the ordinary KP, that the capacity of the knapsack must not be exceeded, equation 2.1, still applies.

$$Z(\Pi, P) = \sum_{i=0}^{m-1} y_i p_i - R \sum_{k=0}^{n-1} \frac{d_{\Pi_k, \Pi_{k+1}}}{v(W_{\pi_k})} \tag{3.4}$$

In words the first term is the sum of profits of all items in the packing plan; and the negative term is the sum over all n edges, of the cost of traveling for the duration of time it takes to cross that edge with the total weight the thief has accumulated throughout the part of the tour which comes before that edge.

3.1 An Explanatory Example

This section presents a simple, example TTP-instance, and an informal discussion of how it can be solved. It will become apparent that some of the intuition and rules of thumb that apply to the component problems, do not apply to the TTP. Furthermore, this section presents intuition that is specific to the TTP; i.e., that does not apply to the TSP or KP.

The example is given by figure 3.1, which presents the item distribution of the TTP-instance with knapsack capacity $W = 9$ and rent rate $R = 1$. By insertion of these parameters in equation 3.1, the speed of the thief is $1 - \frac{0.9 \cdot W_{\pi_k}}{9} = 1 - \frac{W_{\pi_k}}{10}$, where W_{π_k} is the current knapsack weight.

Notice that the TSP and KP components are trivially solved. There is only one feasible TSP-tour, and all items fit in the knapsack. Nevertheless, solving this TTP-instance is *not* trivial.

First, let us use a shortest tour, $\Pi = [0, 1, 2, 3]$, and an empty packing plan, $P = \{\}$, and calculate the objective value.

$$Z([0, 1, 2, 3], \{\}) = 0 - 4 \cdot \frac{2}{1 - \frac{0}{10}} = -8$$

Since no items are picked up, the objective value is the negative of the time, 8, it takes to traverse the tour, times the rent rate, $R = 1$. If the item I_1 is picked up, the objective value increases:

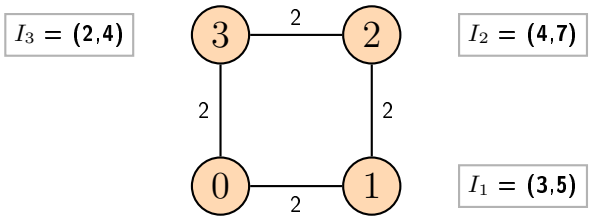


Figure 3.1: The four circles are cities with the initial city labeled by a 0. Edges are labeled by their length, and the three items appear adjacent to the city in which they are located. The weight and profit of the items are given in that order in parenthesis, i.e. $I_i = (w_i, p_i)$.

$$\begin{aligned} Z([0,1,2,3],\{I_1\}) &= 5 - \frac{2}{1-\frac{0}{10}} - 3 \cdot \frac{2}{1-\frac{3}{10}} \\ &= 5 - 2 - 3 \cdot 2.857 = -5.57 \end{aligned}$$

Since this item is picked up in the beginning of the tour, it is intuitively a good idea to try to reverse the tour, as the thief will then need to carry the weight for a shorter distance. As mentioned earlier, this reversal has no effect in the context of the ordinary TSP. For the example here however, reversal makes a difference:

$$Z([0,3,2,1],\{I_1\}) = 5 - 3 \cdot 2 - 2.857 = -3.857$$

In this case, the change of tour improves the objective value, despite the length of the tour remaining unchanged. From this point, adding I_3 further improves the objective value:

$$Z([0,3,2,1],\{I_1,I_3\}) = 5 + 4 - 2 - 2.5 - 2.5 - 4 = -2$$

The alternative, I_2, is equally good:

$$Z([0,3,2,1],\{I_1,I_2\}) = 5 + 7 - 2 - 2 - 3.33 - 6.67 = -2$$

But adding both, which uses the optimal solution for the KP component, is not a good idea:

$$Z([0,3,2,1],\{I_1,I_2,I_3\}) = 5+7+4-2-2.5-5-20 = -13.5$$

Notice that in this case it takes 20 time units to traverse the last edge, four times as many as the previous edge, despite the knapsack weight increasing only by a factor 1.5 between those two edges. In contrast, if the same amount of weight was added while the knapsack was empty, traversing the last edge would take only $\frac{2}{1-\frac{3}{10}} = 2.857$ time units to traverse. So in general, the impact to the objective value of adding I_1 depends on what other items were added. Conversely, the impact of adding the other items depends on whether or not I_1 is going to be added in a later stage. Specifically, adding to P first I_3 and then I_2 yields in both cases an improvement to the objective value:

$$Z([0,3,2,1],\{I_2\}) = 7-4-\frac{4}{1-\frac{4}{10}} = -3.67$$
$$Z([0,3,2,1],\{I_2,I_3\}) = 7+4-2-2.5-10 = -3.5$$

But then adding I_1 results in a reduction of Z. Notice that this is actually the order in which the greedy heuristic algorithm for the KP would add the items, as I_1 has the poorest score, $s_1 = \frac{5}{3}$. The reason that it is bad to pick up I_1 at this point is that, although the knapsack has room for I_1, the thief has added too much weight in the early part of the tour. Stated more constructively: when deciding to pick up items in the early parts of the tour, it should be factored in that adding much weight so early, impairs the ability of adding items in the later part of the tour. This can be done by penalizing the score of items that appear early in the tour, and that is exactly what will be done later, when the greedy heuristic algorithm for the KP is extended to the TTP.

As a final remark, despite the initial intuition that it is best to reverse the tour, so that the relatively heavy I_1 can be picked up near the end, the optimal solution is actually to omit I_1 entirely, and go back to the original $\Pi = [0,1,2,3]$. Then the objective value is:

$$Z([0,1,2,3],\{I_2,I_3\}) = 7+4-2-2-3.33-5 = -1.33$$

Even though there were only two possibilities for Π, and an informed choice was initially made, the choice turned out to be wrong. So even in this relatively simple example, the attempt to separate the components by first choosing a tour and then creating the packing plan for that tour, failed to produce the optimal solution. This goes to show the strength of the interdependency of the component problems of the TTP.

Note also that if the length of the edge $(0, 3)$ were doubled, then all of the above discussion remains valid, except that this last "turn of events" no longer applies. With this change, $Z([0, 1, 2, 3], \{I_2, I_3\}) = -6.33$ is not optimal, but $Z([0, 3, 2, 1], \{I_1, I_3\}) = Z([0, 3, 2, 1], \{I_1, I_2\}) = -4$ is.

3.2 Calculation of the Objective Value

In [BMPW14] an upper-bound of $O(nm)$ is given for the time it takes to calculate the objective value of equation 3.4. The first sum clearly requires at most $O(m)$ operations. The second sum is over n cities, but each uses a unique W_{Π_k}, which is calculated as the sum of at most m items, hence the $O(nm)$. However, with some effort, an upper-bound of only $O(n+m)$ for calculating the objective value can be achieved.

Notice that W_{π_k} has the recursion formula given below.

$$W_{\pi_k} = \begin{cases} W_{\pi_{k-1}} + w_P(\pi_k) & \text{if } k > 0 \\ 0 & \text{if } k = 0 \end{cases} \tag{3.5}$$

Then, in every term of the second sum in equation 3.4, the value W_{π_k} is calculated by adding the previous value, $W_{\pi_{k-1}}$, to the extra weight, $w_P(\pi_k)$, of the items in the current city, k. This added weight is the sum of at most m items. But because each item is located at a unique k, it can be added only once, so the *amortized* number of item-additions over the entire tour is at most $O(m)$. In other words, the second sum is over n terms, each of which require an average of $O(\frac{m}{n})$ additions to calculate. So there are the $O(m)$ item-additions *and* the n additions from adding together the terms of the sum. This yields the claimed, total upper-bound – for both sums – of $O(n + m)$. This can be simplified to $O(m)$ in the context of the benchmark problems that are introduced later, as these enforce $m \geq n - 1$.

3.3 Algorithms for the TTP

The TTP was studied in [PBW+14] and [BMPW14], where a host of algorithms were applied to proposed benchmark sets of TTP-instances.

In [PBW+14], three algorithms are presented and compared. Each takes as input a solution to the TSP-component – a tour, Π – and solves only the KP-component by producing a packing plan, P, that maximizes $Z(\Pi, P)$. The first presented algorithm is *simple heuristic* (SH), which extends the greedy heuristic for the ordinary KP given by algorithm 1. The details are given in the next section. The other algorithms are RLS and (1+1) EA, already introduced in algorithm 2 and algorithm 3 for the KP. In the present context, these latter two are applied with the only modification being to the fitness function, formerly equation 2.3. In the context of the TTP, the function $c(P)$, equation 2.1, is unchanged, and the fitness function is given below.

$$f(P) = \begin{cases} Z(\Pi, P) & \text{if } c(P) \\ -\infty & \text{otherwise} \end{cases} \tag{3.6}$$

The test-results of the paper indicate that RLS and EA perform much better than SH, but that they are much slower; so much so that SH actually outperformed both for very large instances in the time limited tests.

In the second paper, [BMPW14], the algorithms *density-based heuristic* (DH) and *CoSolver* are introduced, and they are compared using another set of TTP-instances. These instances are limited to at most 36 cities for the TSP-component, and 150 items for the KP-component; whereas the instances introduced in [PBW+14] are based on TSPLIB, and may thus contain up to 85900 cities (in the case of the pla85900 TSP-instance) and nearly ten times as many items. While DH is similar to SH, CoSolver is designed to deal with the interconnection of the component problems, and as expected it is shown to produce much better solutions than DH. However, CoSolver, as it is described, solves to optimality the sub-problems, and so it is a super-polynomial time algorithm. Thus it is likely too slow to be used feasibly on the much larger benchmark problems of [PBW+14].

Defining SH and DH

Both of SH and DH were studied only in a context where they are given the shortest possible tour, Π, as input. This Π was pre-calculated with the Chained Lin-Kernighan algorithm introduced in [ACR03]. Thus, like the greedy heuristic for the ordinary KP, SH and DH produce P, a subset of of M, such that a fitness function, $f(P)$, is maximized. However, as discussed in section 3.1, the TTP severely complicates calculating meaningful score values for the items.

So, accordingly, SH and DH differ from the greedy heuristic for the KP in how the item-scores are calculated. Both SH and DH use the same score, s_i, which is based on a crude estimate, t_i, of the time it takes to complete the tour after I_i is picked up. The value t_i can only be an estimate, because, while the length, d_i, of the path that the thief must traverse to complete the tour is known, the thief's speed depends on what other items are picked up, and this information can of course not be known by an algorithm tasked with deciding what items to pick up.

The estimate, t_i, then, is the time it takes a thief carrying *only* I_i to travel the remaining tour distance d_i, from the city c_i, which has tour index $\Pi(c_i)$.

$$d_i = \sum_{k=\Pi(c_i)}^{n-1} d_{\pi_k,\pi_{k+1}}$$

$$t_i = t(d_i, w_i) = \frac{d_i}{v_{max} - \nu w_i} \tag{3.7}$$

The score, s_i, is given below.

$$s_i = p_i - Rt_i \tag{3.8}$$

Notice the similarity to the objective value. Like it, s_i is a profit from item(s), and a travel-cost. As the time, t_i, only depends on the weight of the item in question, it may seem that the heuristic intuition from section 3.1, that the score should take other items into account, has no influence on s_i. However, s_i does penalized items that appear early in the tour, because d_i is larger for these items.

Once all items have been assigned a score, they are sorted in descending order of s_i, and processed one at a time in this order. As the items are processed, they

are added to the packing plan if they fit, and an additional requirement, r_i, is satisfied. Thus, apart from this additional requirement, the only change from the greedy heuristic is the item scores. Algorithm 12 summarizes the process, which is different for SH and DH only by virtue of the form of the requirement r_i.

Algorithm 12 SH and DH

1: $D_C := 0$ ▷ The distance to the end of the tour
2: **for** $i = n-1, n-2, \ldots, 1$ **do**
3: $\quad D_C := D_C + d_{\pi_i, \pi_{i+1}}$
4: $\quad$ **for all** $I_j \in P$ available at π_i **do**
5: $\quad\quad t_j := t(D_C, w_j)$
6: $\quad\quad s_j := p_j - Rt_j$
7: $I[] :=$ array of all $I_i \in M$, sorted in descending order of s_i
8: $P := \emptyset$
9: $W_P := 0$
10: **for** $i = 0, 1, \ldots, m$ **do**
11: $\quad I_j := I[i]$
12: $\quad$ **if** $W_P + w_j \leq W$ and r_i **then** ▷ r_i is an implementation dependent requirement
13: $\quad\quad$ add I_j to P
14: $\quad\quad W_P := W_P + w_j$
15: **return** P

For DH, r_i is true iff $Z(\Pi, P \cup I_i) > Z(\Pi, P)$, i.e., I_i is added to P only if the addition improves the objective value. This makes the for-loop in line 10 of algorithm 12 become the dominating constituent of the running time. The asymptotic running time of DH is then $O(m(n+m))$, since the objective value can be calculated in time $O(n+m)$, and it must be calculated m times in the worst case, where all items fit in the knapsack. Note that due to the tighter upper-bound on calculation of the objective value, the presented running time for DH is correspondingly tighter than the upper-bound of $O(nm^2)$ given in [BMPW14].

For SH, r_i is true iff $u_i > 0$, where u_i is the so called *fitness value*[1] given by equation 3.9.

$$u_i = p_i - R(t_i - t'_i) \tag{3.9}$$

[1] The fitness value of an item should not be confused with the fitness value of a solution. I call u_i a fitness value because it is called so in [PBW+14] which introduces SH.

$$t_i = t(d_i, w_i)$$

$$t'_i = t(d_i, 0)$$

In words the fitness value of I_i is its profit, p_i, minus the cost of spending time equal to the *difference* between the durations of traversing the tour with no items in the knapsack and, when just I_i is picked up.

The intuition for using u_i is that it expresses the gain in objective value of adding I_i to the empty packing plan. Thus, adding an item with $u_i \leq 0$ to any packing plan, can result in no increase in objective value. A proof is given in [PBW+14].

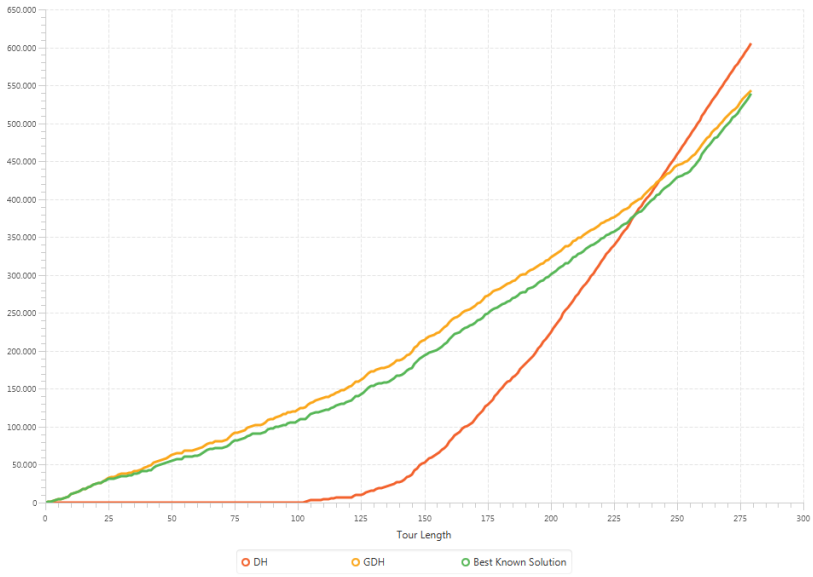
The SH described in [PBW+14] additionally checks before termination if the packing plan produced is no better than the empty packing plan, returning the empty plan in that case. This condition would be met, for example, if SH were to solve the TTP-instance given by figure 3.1, since all items would be packed, yielding an objective value less than $Z(\Pi, \{\}) = -8$. The condition is not met in any of the experiments performed later.

Once again, a tighter upper-bound than the $O(nm)$ from [BMPW14] can be given. The $O(nm)$ is from the final, single calculation of the objective value that determines if the produced packing plan is an improvement over the empty packing plan. So, like in the regular, greedy heuristic for the TSP, the for-loop in line 10 takes linear time, the most time consuming operation is the sorting, and so the running time of SH can be upper-bounded by $O(m \log m)$.

The Item Scores of SH and DH

Here an indication is given about the shortcomings of the score, s_i. Figure 3.2 shows three graphs, each of which correspond to a particular solution-pair of tour and packing plan for the TTP-instance *u280_5_5usw*. The y-axis value of the graphs indicate what the knapsack weight is at each city in the tour. For the three solutions, the same, shortest tour is used. Only the packing plan is different.

Figure 3.2: For the TTP-instance *u280_5_5usw*, the graphs are of the knapsack weight on the y-axis and the cities $\pi_0\pi_1 \dots \pi_{n-1}$ on the x-axis for three solutions; all of which use the same tour, but different packing plans. The solution of the orange graph (which, of the three, achieves the greatest, final knapsack weight) uses the packing plan produced by DH, that of the yellow graph uses the plan of GDH, and the green graph is for the solution with the best known packing plan for the tour (which achieves the least, final knapsack weight of the three).



The objective value achieved by the solution which uses the packing plan produced by DH, $Z(\Pi, P_{\mathrm{DH}}) = 43,630.42$, is significantly less than the optimal one for the same tour, $Z(\Pi, P_{\mathrm{opt}}) = 104,365.73$. The graphs in the figure shows that the packing plan produced by DH includes no items from the first 100 cities, which is in stark contrast to the best plan. It seems reasonable to conclude, that the score, s_i, does not accurately reflect the quality of items in this instance. In the next chapter, it is shown how to calculate a better score in constant time. The resulting algorithm is called *generalized* DH (GDH).

To give an indication of how accurate we can expect the scores to get, the comparison here includes GDH. The objective value achieved $Z(\Pi, P_{\mathrm{GDH}}) = 103,141.76$, is much closer to that of the optimal packing plan for this tour.

Chapter 4

Design of a Complete Solver for the TTP

This chapter proposes a solution-procedure for the TTP which is based on MMAS for the TSP. It involves two subroutines which are described individually. First, an overview is given in the next section.

4.1 Overview

The idea is to apply ACO as it would be to the TSP, but after a tour, Π, is constructed, a packing plan, P is produced by a procedure similar to SH and DH, resulting in a solution for the TTP. Since the components that are chosen by the ants are edges, pheromone is not used to guide the construction of the packing plan. Thus, with this solution, the TSP is solved in isolation, i.e., with no regard as to what tours are inclined to yield good solutions to the TSP.

This algorithm is summarized below, when the choice for procedure 11 is made in line 7. In this case, the algorithm is called ACO_{tsp}, to indicate that the pheromone values reflect the quality of the solution in a TSP context. The alternative choice – for procedure 14 in line 7 – causes pheromone values to reflect component quality in the TTP context, i.e. pheromone values are highest for

edges that tend to be a part of tours that are well suited to yielding a high TTP objective value. The corresponding algorithm is accordingly dubbed ACO_{ttp}.

Algorithm 13 ACO for the TTP

Input: A TTP-instance

1: **while** Stopping criteria is not met **do**
2: **for** Every ant in the population **do**
3: Construct a tour, Π, using procedure 9
4: Apply LK to Π using algorithm 5
5: Create a packing plan, P, for Π
6: Evaporate pheromones with procedure 10
7: Deposit pheromones as normal with procedure 11, *or* by using the TTP-specific procedure 14
8: **return** The best seen pair (Π, P), i.e., such that $Z(\Pi, P)$ is maximized

Thus, ACO_{ttp} involves pheromone deposition that is based on the objective value, $Z(\Pi, P)$, of the TTP solution, instead of the tour length. In this case, the pheromone trail should converge toward tours that tend to yield better packing plans. This constitutes communication between procedures that deal with the respective component problems.

In specifying procedure 14, which is given below, there is a difficulty in line 3. Recall that the corresponding line in procedure 11 was $\tau_e := \tau_e + \frac{1}{|T|}$, where $|T|$, the tour length, conveniently was guaranteed to be positive, nonzero, and typically small enough that pheromone can be deposited on the same edge, e, many times, without causing τ_e to increase beyond 1. This last point is important, because MMAS, [SH00], uses this 1 as the enforced upper-bound for τ_e.

Line 3 in procedure 14 reflects an attempt to meet these requirements, with the values $\text{UB}(I)$ and $\text{LB}(I)$ for the TTP-instance I given below.

- $\text{UB}(I) = [W \cdot \frac{p_i}{w_i} - n]$ such that $\frac{p_i}{w_i}$ is the greatest among the $I_i \in M$. This yields an upper-bound for Z which is very coarse.

- $\text{LB}(I)$ is equal to $\min(-Z(\Pi, P), 0)$, for the solution, (Π, P), with least $Z(\Pi, P)$ of all those observed during execution of algorithm 13. In other words, $\text{LB}(I)$ is zero, unless the worst encountered solution has negative Z, in which case $\text{LB}(I)$ takes on this value.

Procedure 14 TTP Pheromone Deposition

Input: A set, S, of pairs (Π, P) that are feasible solutions for the TTP-instance I

1: **for all** $(\Pi, P) \in S$ **do**
2: **for all** $e \in \Pi$ **do**
3: $\tau_e := \tau_e + \frac{\mathrm{LB}(I)+Z(\Pi,P)}{\mathrm{UB}(I)}$

If using the pheromones in this way is to work, clearly it is important that the procedure which produces the packing plan in line 5 of algorithm 13 produces the plans quickly and with consistent quality. It is not necessary that the produced plan, P, is optimal, or even good, as long as the objective value, $Z(\Pi, P)$ is a good indicator of how good Π is compared to other tours. Then, when the algorithm ends, the plan can optionally be improved further. Of course, producing an optimal P ensures that $Z(\Pi, P)$ is an exact indicator of the quality of Π in the TTP context, but this P takes too long to calculate.

Of the algorithms considered so far, none are good enough. RLS and EA (1+1) are too slow (except perhaps on very small instances), and SH and DH are too inconsistent. A better method is thus required, and it is introduced in the following section.

To further intensify the selection strategy, the *elitist* pheromone update scheme is adopted. After every iteration, pheromone is deposited on only two tours – the best tour of the iteration, and the best global tour. In continuation of the above, what constitutes a *best* tour is determined by its length if the algorithm is ACO_{tsp}, and by the TTP objective value if the algorithm is ACO_{ttp}.

Unfortunately, while the tours constructed by the ants in ACO_{ttp} are guided by pheromones toward good TTP tours, the subsequent local optimization is tuned only to create short tours. Thus, some of the edges which are good from a TTP point of view, may be deleted by LK because they are not consistent with short tours. Nevertheless, some of the "communication" will likely leak through.

4.2 An Improved Packing Heuristic for The Traveling Thief Problem

This section introduces a new algorithm, *generalized density-based heuristic* (GDH). Because GDH is just a generalization of DH, algorithm 12 roughly

gives the pseudocode. The differences are the condition, r_i in line 12, and the score-calculation in lines 5 and 6.

The new score-function is based on the fitness value, u_i, given by equation 3.9. Recall that u_i is the exact gain in objective value of adding I_i to the *empty* packing plan. The score derived for GDH in the following, is an *estimate* of the gain in objective value of adding I_i to an *approximation* of the *optimal* packing plan.

Recall from the discussion in Section 3.1 that it is important that the score takes into account the weight of the other items in the knapsack. To help explain the extent to which scores take account, I identify two issues such that the first is dealt with to some degree by the scores of SH, DH, and CoSolver, and the second is globally ignored. They are given below.

a) When I_i is picked up, the knapsack usually already contains some weight, which is greater the closer to the end of the tour I_i is picked up.

b) After I_i is picked up, the speed of the thief will continue to decrease as more items are added and the knapsack weight increases. So when I_i is added to the packing plan at some tour index $\pi_j = c_i$, this causes some additional amount of time, Δt_j, to be used to traverse the next edge (π_j, π_{j+1}), and, similarly, some additional amount of time, Δt_{j+1}, to traverse the edge after that (π_{j+1}, π_{j+2}). Of course Δt_j is not necessarily equal to Δt_{j+1} because the two edges are not necessarily of equal length. But even if they are, the two numbers may be different. Specifically, if the edges are of equal length, Δt_{j+1} is greater if extra weight is added at π_{j+1}, and otherwise, $\Delta t_j = \Delta t_{j+1}$. More generally, if the packing plan already contains one or more items to be picked up at π_{j+1}, then the *time per distance* increases: $\frac{\Delta t_j}{|(\pi_j, \pi_{j+1})|} < \frac{\Delta t_{j+1}}{|(\pi_{j+1}, \pi_{j+2})|}$. So, from adding I_i, the consequent amount of extra time *per distance* is a growing function of the total distance traveled by the thief. Nevertheless, the scores used by SH, DH, and CoSolver incorporate the estimate of the total extra time which assumes that there is no growth: $\frac{\Delta t_j}{|(\pi_j, \pi_{j+1})|} d_i$, where d_i is the remaining tour distance.

The algorithm CoSolver deals with a) in a way that is used in the design of GDH. CoSolver uses as scores so called "relaxed profits", $\bar{p}_i$, which is essentially a modification of u_i that takes into account a packing plan, P_{prev}, produced in a previous application of CoSolver.

$$\bar{p}_i = p_i - R(\bar{t}_i - \bar{t}_i{}') \tag{4.1}$$

where

$$\bar{t_i} = t(d_i,\ W_{\pi_{j-1}} + w_i)\ ,\ c_i = \pi_j$$
$$\bar{t_i}' = t(d_i,\ W_{\pi_{j-1}})\ ,\ c_i = \pi_j$$

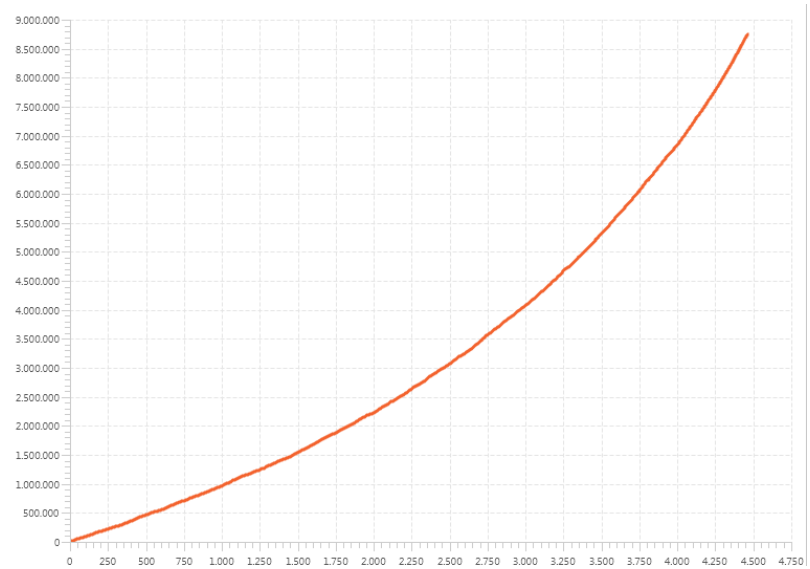
In the above $W_{\pi_{j-1}}$ is the cumulate weight of the city that, in the tour, comes before the one where I_i is located. This cumulate weight is with respect to P_{prev}. As usual, d_i is the remaining tour distance.

The meaning of $\bar{t_i}$ is the time necessary to complete the tour from the city where I_i is picked up, but without picking up any more items after I_i. The meaning of $\bar{t_i}'$ is similar, except that I_i is not picked up.

Comparing to u_i, the only difference is the addition in two places of $W_{\pi_{j-1}}$. But if $P_{prev} = \varnothing$, then $W_{\pi_{j-1}} = 0$ and $\bar{p}_i = u_i$ for all i.

So u_i is just $\bar{p}_i$ for one particular choice of P_{prev}. Note that in calculation of $\bar{p}_i$, the information needed from P_{prev} is the weight of the knapsack at each city. Figure 4.1 gives an example of this for a good packing plan.

Figure 4.1: The graph of $y = W_{\pi_x}$ for a particular tour of *fnl4461* and a good packing plan for the TTP-instance *fnl4461_5_5usw*. Thus, the knapsack weight is given on the y-axis, and the tour-indexes are given on the x-axis



Let us say that the graph presents the function $y(x)$ and that the corresponding packing plan is the P_{prev} used to calculate $\bar{p}_i$. Then $W_{\pi_{j-1}} = y(j-1)$. This illustrates how the $\bar{p}_i$ score deals with the issue a) above. Correspondingly, the issue b) has to do with the function $y(x)$ in the interval $j \leq x < n$. The value $\bar{p}_i$ assumes that $y(x) = y(j-1)$ for all $x \geq j$. However, as can be seen in the figure, this assumption far from holds. In the following is described a score which takes into account the steady increase of the knapsack weight.

By modification to $\bar{p}_i$, this can be achieved by replacing $\bar{t}_i$ and $\bar{t}_i{}'$ with $T_i(w_i)$ and $T_i(0)$ respectively, using the following definition of $T_i(w)$, which assumes as usual that $c_i = \pi_j$[1].

$$T_i(w) = \sum_{k=j}^{n-1} \frac{d_{\pi_k,\pi_{k+1}}}{v_{max} - \nu \cdot (W_{\pi_j} + w)} \tag{4.2}$$

The problem with the resulting score function,

$$p_i - R(T_i(w_i) - T_i(0)),$$

is that it requires up to $O(n)$ additions to calculate.

The next step is to approximate $T_i(w)$ in constant time. For GDH, this is done by approximating the running weight – for example, as expressed by the function $y(x)$ in figure 4.1 above – by a quadratic function in the distance traveled by the thief, i.e., $y(x) = k \cdot x^2$, for some appropriate k. So the city-weights which were previously obtained empirically from a packing plan, P_{prev}, are now calculated as a function of the tour.

Figure 4.2, which gives the equivalent of figure 4.1 for additional TTP-instances, indicates that different instances call for different such functions; i.e. different polynomials approximate the various graphs best. Despite the differences, the function $y(x) = k \cdot x^2$ appears to be a reasonable approximation if one choice must cover all instances. A discussion of what function is actually best, and whether it is possible to choose between alternatives during algorithm execution, will be left to future work.

If D is the total distance of the tour, and W_{opt} is the final knapsack weight,

[1] For $T_i(w)$, the running weight used is W_{π_j}, the one for the *current* city, while the value used to calculate $\bar{t}_i$ and $\bar{t}_i{}'$ is for the previous tour city. The change has little practical impact, but greatly simplifies further calculations.

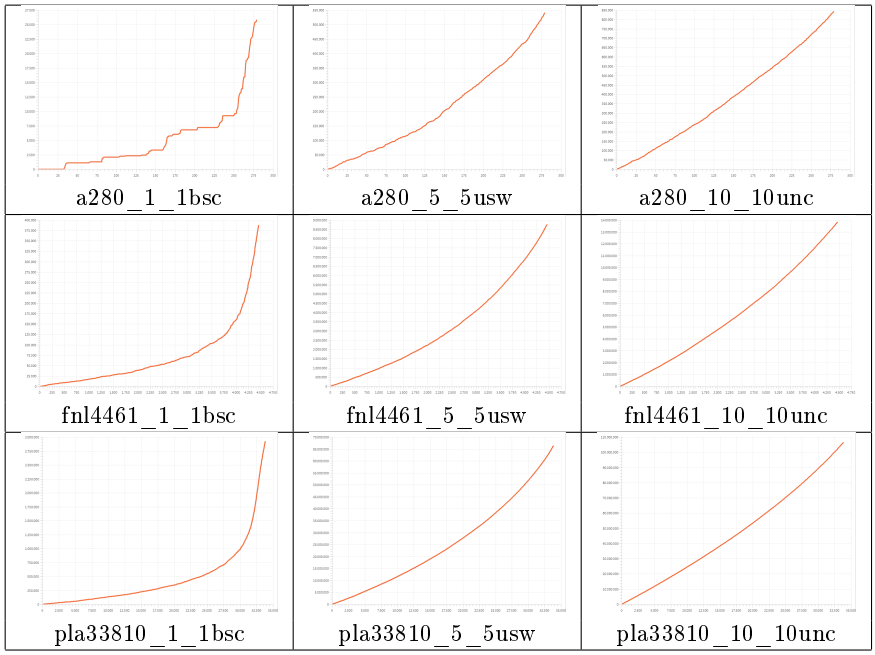


Figure 4.2: These graphs give the weight of the knapsack on the y-axis, and the tour-index on the x-axis, similarly to the graph of figure 4.1

equation 4.3 gives the approximation.

$$W_{\pi_j} \approx W_{opt} \frac{(D - d_i)^2}{D^2} \tag{4.3}$$

This approximation is more naturally expressed as a function, $W(d'_i)$, $d'_i \in \mathbb{Z}$, of the distance, $d'_i = D - d_i$, of the part of the tour leading to I_i.

$$W(x) = W_{opt} \frac{x^2}{D^2} \tag{4.4}$$

By using $W(x)$ to approximate W_{π_j}, $T_i(w)$ can be estimated as a sum over every unit of distance left of the tour after picking up I_i. This estimate, $T_i^{\mathbb{Z}}(w)$, extrapolates the sequence of edges into an integer sequence.

$$T_i(w) \approx T_i^{\mathbb{Z}}(w) = \sum_{x=d'_i}^{D} \frac{1}{v_{max} - \nu \cdot (W(x) + w)} \tag{4.5}$$

The expression for $T_i^{\mathbb{Z}}(w)$ is an estimate of $T_i(w)$, because it reflects a gradual decrease in speed during travel between each pair of cities in the tour, while the speed is actually constant in these intervals.

The last step is to estimate $T_i^{\mathbb{Z}}(x)$ by expressing it as an integral, by extrapolating from the integers to the real numbers.

$$\begin{aligned} T_i^{\mathbb{R}}(w) &= \int_{d_i'}^{D} \frac{1}{v_{max} - \nu \cdot (W_{opt} \frac{x^2}{D^2} + w)} dx \\ &= \frac{D}{\sqrt{\nu}\sqrt{W_{opt}}\sqrt{v_{max} - \nu w}} \\ &\quad \left[\operatorname{arctanh} \left(\frac{x\sqrt{\nu}\sqrt{W_{opt}}}{D\sqrt{v_{max} - \nu w}} \right) \right]_{x=d_i'}^{x=D} \end{aligned} \tag{4.6}$$

The expression has the convenient form

$$T_i^{\mathbb{R}}(w) = \frac{C_1}{C_2} \left[\operatorname{arctanh} \left(\frac{x}{C_1 C_2} \right) \right]_{x=d_i'}^{x=D} \tag{4.7}$$

where C_1 and C_2 are constant once a tour has been fixed, and a W_{opt} has been chosen.

$$\begin{aligned} C_1 &= \frac{D}{\sqrt{\nu}\sqrt{W_{opt}}} \\ C_2 &= \sqrt{v_{max} - \nu w} \end{aligned}$$

Finally, the score used in GDH is given by below.

$$s_i^{GDH} = p_i - R(T_i^{\mathbb{R}}(w_i) - T_i^{\mathbb{R}}(0)) \tag{4.8}$$

As noted earlier, apart from using this new score-function, GDH shares with DH and SH the pseudocode given by algorithm 12, but has a unique requirement in line 12. The value r_i is set to be the conjunction of the values of r_i for DH and SH, albeit with a different fitness value. The fitness value used is s_i^{GDH}

calculated with a reduced value for W_{opt}; the present implementation uses the value $0.8 \cdot W_{opt}$.

The value W_{opt} is a parameter of the algorithm, and choosing it well is difficult.

The derivation of s_i suggests that the best choice for W_{opt} is the total knapsack weight of the best known packing plan. While the two values are typically strongly correlated, they are not necessarily equal.

Nevertheless, using repeated application of GDH to produce better choices for W_{opt} is useful. In the following IGDH(x) (*Iterative*-GDH) denotes the algorithm that applies x iterations of GDH, updating W_{opt} after each iteration to take into account the packing plan created in the previous iteration.

4.3 A Faster GDH

It is possible to greatly reduce the computation time of GDH. This comes at a cost to the objective value of the resulting packing plan; but the cost is usually very low, and sometimes there is actually an improvement compared to GDH. The intuition is that, since all the items to be considered are ordered by their score, i.e. how good they are, there is no need to check for all items (in $O(n+m)$ time for each) whether it is advantageous to add them, but only to find the turning point and add all items before it.

So, ideally, all items before the i'th, for some initially unknown i, must be added, and the remaining ones must not be added.

Although there is not necessarily any such absolute turning point, the intuition holds. The fast version of GDH which uses this approach will be called *hybrid heuristic* (HH) (and IHH(x) for the iterative version), since it uses at most $O(\sqrt{m})$ objective value calculations where DH uses at most $O(m)$ and SH uses $O(1)$.

Specifically, HH adds the items from the sorted list in chunks of $\sqrt{m}$ items, checking for each chunk – rather than for each item – whether the addition results in an improved objective value. When the turning point is reached – a chunk is added, decreasing the objective value – that chunk is removed. Then all items from the removed chunk *and* the next one are processed normally; i.e. they are added individually, the objective value is calculated each time, and the item is then removed again – before moving on to the next – unless there was an improvement.

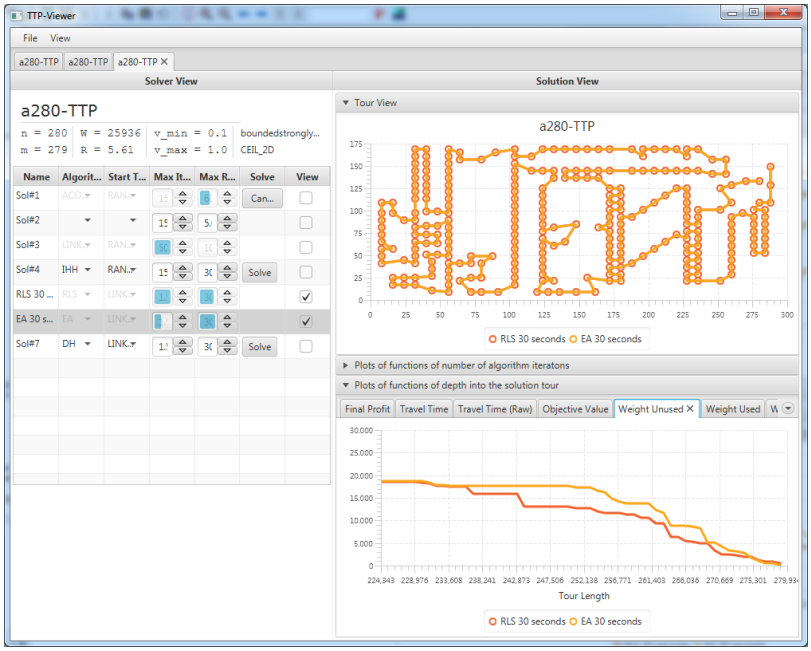
Thus there are a total of at most $3 \cdot \sqrt{m}$ objective value calculations, and the running time of HH is upper-bounded by $O(\sqrt{m}(n+m))$; an improvement over the $O(m(n+m))$ of GDH.

The obvious drawback is that the objective value takes linear time to calculate, while the G used in LK for the TSP takes only constant time. Before d issue, it

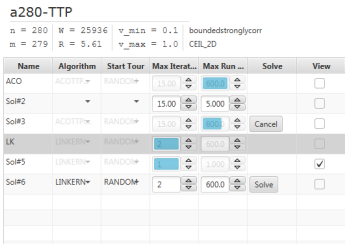
4.4 Design of Visualization software

As a part of this project, a graphical user interface (GUI) based visualization software has been produced. The purpose is to enable rapid and dynamic visualization of solutions for the TSP, KP, and TTP, to facilitate analysis and documentation in this thesis. Figure 4.3 gives a typical view of the GUI.

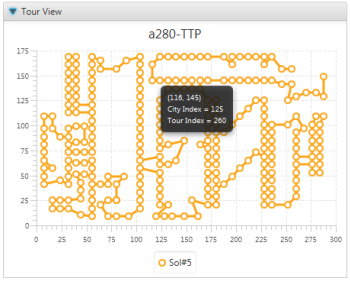
Figure 4.3: The GUI of the visualization program



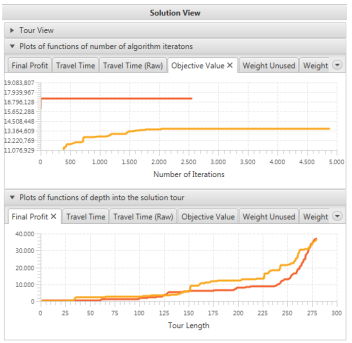
The following are features of the program.



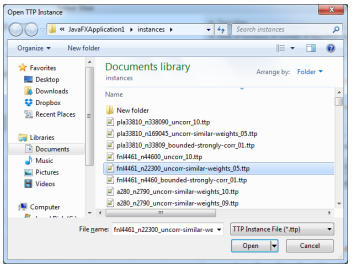
- **Solver View** Multiple algorithms (solvers) can run in parallel. In-GUI selection of parameters such as start-tour, number of iterations, and maximum run-time is possible. There are progress indicators for duration and number of iterations. Execution can be canceled. Solvers can be renamed. They can also be duplicated and deleted.



- **Tour View** Tooltips allow querying coordinates, city-index, and tour-index of cities. Multiple tours can be displayed simultaneously. The user can click and drag to pan, use the scroll wheel to zoom

- 

  Graph View Two choices for x-axis are available, and there are eight choices for y-axis values, for a total of 16 graphs. Pan and zoom is possible. Multiple solvers can be displayed in the same view. Solvers can be individually hidden and re-added indefinitely.

- 

  Open From File A TTP-instance file can be specified by its path and opened. Multiple instances can be open simultaneously in separate panes.

Chapter 5

Implementation

This chapter deals with implementing the designs of the previous chapter. Focus is on presenting challenges and issues that have come up, as there is no neat, overarching structure for the entire implementation.

5.1 Framework and Objective Function

Available from `http://cs.adelaide.edu.au/~ec/research/combinatorial.php` is a java implementation of RLS, EA and a framework for loading TTP-instance files, representing them in memory, and calculating the objective value for this representation. The designs presented in the previous chapter are implemented in java around this framework. A great advantage of using this framework is that it provides confidence that good results cannot be explained by incorrectly calculated objective value, or incorrect interpretation of benchmark problems.

The framework uses the Pythagorean theorem to calculate distances, and never saves these in memory. Particularly the square-root is expensive to calculate and the distances must be frequently fetched by algorithms such as LK. So, intuitively, some time can be saved at the cost of memory by saving these

distances in an $n \times n$ matrix. Since all instances are of symmetric TSPs, less than $\frac{n^2}{2}$ distances actually need to be stored. Nevertheless, I ran out of memory for some of the larger instances, so I do this only for sufficiently small n.

Importantly, java's virtual machine checks for out-of-bounds errors at every array-look-up. This is great for debugging, but there is a small, constant time-cost associated to every look-up. The cost is great enough that the matrix representation of distances offers no improvement to algorithm running-time. However, it is possible to disable the out-of-bounds-check by using a class called *Unsafe*. Using this class – which is rather roundabout, as it is made to be used by developers of the java language – I managed to achieve an improvement in algorithm running time by storing distances in memory.

5.2 Lin-Kernighan Algorithm for the TSP

The implementation of LK implemented here is optimized for fast, single runs, i.e. where LK is stopped after a starting tour cannot be further optimized, rather than repeating with new starting tours. The original paper [LK73] suggests (in section 2.C titled "Reduction") using statistical analysis of what edges occur frequently in several such repetitions. Accordingly, this has not been implemented as suggested. Additionally, this is essentially what ACO achieves with the pheromone updates, so implementing it would likely be redundant in this context. This bars direct implementation as suggested in section 2.D of [LK73] of a closing "Nonsequential Exchange", such as the double bridge move, because the statistical analysis must be used to simplify the search for the double bridge move, by ruling out deletion of the most frequent edges.

For the same reasons, using kicks, as suggested in [ACR03], to chain together consecutive calls to LK, is not featured by the implemented version. However, the discussion on alternative starting tours to the random permutation is relevant, as the tour provided in the ACO context is similar in nature to the Nearest Neighbor starting tour.

5.3 ACO Algorithm for the TSP

For ACO to work, it seems that pheromones must be stored in memory for every edge. As mentioned in Section 5.1, there not be enough memory available for this to be possible. However, because of the design choice to deposit pheromone only

for very few, best tours, the majority of edges will have the minimum possible amount of pheromone. Thus a hash table can be used to store the pheromone values. With care, removing entries that are evaporated to the minimum, it seems that the amount of memory required to hold the pheromones can be kept within a constant factor of n.

A related issue arises from the need to create, at each of n steps in the tour construction phase, a probability distribution over $n - i$ edges when choosing the i'th component. When i is low, most edges are associated with a very low probability. In fact, this issue is also closely related to one of the Lin-Kernighan algorithm, which at any given time limits itself to considering only the nearest five neighbors, i.e., the shortest five candidate edges. Using this inspiration, the probability distribution is created over the $\log n$ shortest, outgoing edges minus those that enter cities that are already visited. If this set is small, i.e. most of the nearest neighbors are already visited, this typically means that $n - i$ is very small, and so the probability distribution is over all feasible candidates in this case. In addition to inclining search additionally toward short tours, this adjustment significantly reduces running-time.

5.4 GDH and HH for the KP component of the TTP

Due to the similarity of SH and DH, these were implemented first. In the case of SH, this has allowed for proof of concept, as the algorithm is deterministic, and [PBW+14] provides results for public benchmark problems. Curiously, I was unable to exactly reproduce the results, despite trying every variation I could think of. The variation that is – as far as I can tell – *as defined* in [PBW+14], is the one that produces results that provide the closest match; which is close enough that I find reasonable the conclusion that the cause of the discrepancy is some minor implementation detail.

5.5 Visualization Software

In addition to facilitating incorporation of some basic GUI elements like buttons, text, and tables, the new *JavaFX* has been particularly useful for its *LineChart* – extending the *XYChart* – class, which is used to display the graphs and to visualize the tour. These take a list of two-dimensional data-points in a class called a *Series*, which is displayed by drawing a line from point to point. For

the tour view there was a complication, as the XYChart sorts data-points by x-coordinate prior to drawing the graph-line, thus breaking the order of the cities. To get around this, I had to override the *LayoutPlotChildren* class, to disable the sorting. Conveniently, this was also a natural place to add the tooltips for the cities. I used the *Tooltip* class for this.

The XYChart's of JavaFX do not offer native support for "Google-maps"-style pan and zoom. I implemented this by using mouse-input-listeners – which all JavaFX elements do support – to trigger appropriate axis-range modification.

A downside of XYChart is that it is not intended to be used with quite as much data as necessary in this application. Thus, whenever a new Series was added to a chart, the program would freeze for a few seconds (or hours for very big series with many data-points), as all JavaFX must run on the main thread. To alleviate this I did the following two things:

a) In the case of the graphs, a sequence of three or more consecutive data-points on a line can be equivalently represented by the first and last of the sequence. Identifying and removing this redundancy due to linearity made a big difference. Further, as it does not make use of JavaFX, this identification-process can be moved out to a daemon thread, allowing process-indicators and cancel buttons.

b) When the user requests a solution to be displayed, it is processed only by views that are open, i.e., not all 17 possible graphs are created, but only those corresponding to the up to three open charts.

Chapter 6

Experiments

In this chapter, the implementations are exposed to a number of tests. There is a set of nine benchmark problems used for all the tests. Details for these are given in the next section. The following three sections presents three sets of experiments that compare the three different types of algorithms presented in this thesis:

a) Algorithms that solve the KP component of the TSP

b) Algorithms that solve the TSP

c) Algorithms that solve the TTP

6.1 The Experimental Setup

The setup uses the nine TTP-instances from the benchmark-set described in [PBW+14], which have the names given below.

a280-1-1bsc	a280-5-5usw	a280-10-10unc
fnl4461-1-1bsc	fnl4461-5-5usw	fnl4461-10-10unc
pla33810-1-1bsc	pla33810-5-5usw	pla33810-10-10unc

They are based on the TSPLIB, TSP-instances *a280*, *fnl4461*, and *pla33810* which have 280, 4461, and 33810 cities respectively. The TTP-instance names, e.g. *a280-1-1bsc*, all include two numbers, the first of which is the number of items per city, and the second of which is the capacity category value, C, specifying the capacity of the knapsack as $\frac{C}{11}$ times the sum of all item weights. So *a280-1-1bsc* has one item per city and $C = 1$.

The last three letters of the instance name specify its *knapsack type*. There are three possibilities: *uncorrelated* (unc), *uncorrelated with similar weights* (usw), and *bounded strongly correlated* (bsc). Uncorrelated KP instances are usually the easiest to solve, while the strongly correlated ones tend to be very hard.

All experiments are performed on one core of an Intel i5-3570K CPU.

6.2 Computational Study of the Packing Plan Algorithms

This section presents experiments which measure the performance of DH, IGDH, and IHH.

Since these algorithms only process the KP part of the TTP, the experiments use just one tour per TSP-instance. This, too, is the approach used to test SH, RLS, and EA in [PBW+14]. In fact, the tours – which were produced using the *Chained-Lin-Kernighan* algorithm from [ACR03] – used in that paper were made public[1], and so the experiments documented here use the same tours, and the results can be compared to those from the paper (which they are in table 6.4).

Table 6.1 and 6.2 present data on the experiments. They were stopped after 20 iterations with no improvement, or ten minutes, whichever came first.

[1] The tours are included with the java implementation of RLS and EA available from `http://cs.adelaide.edu.au/~ec/research/combinatorial.php`

TTP-instance	#iters	time total	time per iter	resulting Z
a280-1-1bsc	27	93	0.21	15,773.77
a280-5-5usw	26	144	2.37	104,226.32
a280-10-10unc	26	261	7.37	411,549.42
fnl4461-1-1bsc	55	4,417	80.31	247,782.75
fnl4461-5-5usw	26	20,756	798.31	1,478,144.43
fnl4461-10-10unc	32	88,345	2,760.78	6,259,156.83
pla33810-1-1bsc	79	295,057	3,734.90	1,708,313.14
pla33810-5-5usw	8	663,665	82,958.13	1.46965×10^7
pla33810-10-10unc	2	640,351	320,175.50	5.676597×10^7

Table 6.1: Experimental results for IGDH(x). The first column gives the TTP-instance on which the experiment is performed. The second gives x, i.e. the number of calls to the algorithm. The third column gives the duration in milliseconds of the experiment. The fourth column is the average duration of one call to the algorithm. For the experiments of duration less than three seconds, this average was calculated for a separate experiment with greater x. The final column gives the resulting objective value

TTP-instance	#iters	time total	time per iter	resulting Z
a280-1-1bsc	27	80	0.15	15,773.77
a280-5-5usw	25	101	0.84	104,202.52
a280-10-10unc	33	144	2.17	411,459.29
fnl4461-1-1bsc	48	236	3.06	247,750.05
fnl4461-5-5usw	35	2,239	57.17	1,478,086.99
fnl4461-10-10unc	35	5,819	166.26	6,246,448.19
pla33810-1-1bsc	26	1,182	46.96	1708719.81
pla33810-5-5usw	29	55,599	1,917.21	1.46742×10^7
pla33810-10-10unc	42	252,832	6,019.81	5.665097×10^7

Table 6.2: Experimental results for IHH(x)

The reason that some of the experiments took longer than ten minutes is that the algorithms were not stopped mid-iteration.

To help compare the objective values obtained by the various algorithms, I adopt the scheme of [ACR03] to express the result as a percentage deficit of the best result. In that paper the results were tours for the TSP. An equally elegant calculation of the percentage deficit of a solution of a TTP-instance is probably not possible. The method used in the following requires calculating a *best* and

worst objective value for each of the nine instances. Then the experimental results can be given by the percentage deficit from this best value in the following way.

$$\frac{100 \cdot (best - result)}{best - worst}.$$

These best and worst values are presented in table 6.3.

TTP-instance	best seen Z	empty packing plan Z
a280-1-1bsc	16156.397	−14658.93
a280-5-5usw	104365.731	−189965.1
a280-10-10unc	411714.790	−544888.89
fnl4461-1-1bsc	256910.094	−259989.8
fnl4461-5-5usw	1478962.570	−3205302.82
fnl4461-10-10unc	6261433.187	−9051359.18
pla33810-1-1bsc	1727869.528	−1987561.74
pla33810-5-5usw	1.4704801579×10^7	-2.186317914×10^7
pla33810-10-10unc	5.6808627336×10^7	-6.227693452×10^7

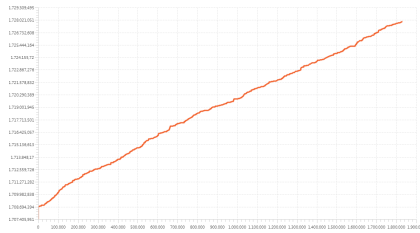
Table 6.3: *Best* and *worst* values for the nine instances

The *worst* values in the right-most column are the objective values achieved by using the empty packing plan.

The *best* values are produced by first building an initial packing plan using IHH, and subsequently further optimizing it using RLS and EA. I have aimed at producing *best* values which at least appear to be locally optimal, i.e., where relatively many applications of RLS and/or EA find no improvement.

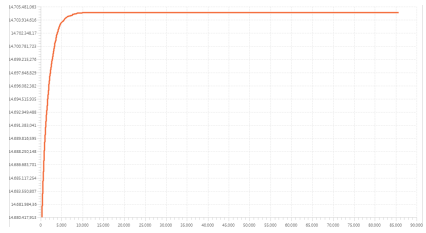
One instance – pla33810-1-1bsc – was too "hard" for this to be possible in reasonable time using the method described. The *best* value listed in table 6.3 for it is the result of an optimization process presented in figure 6.1. IHH produces the initial packing plan with $Z = 1,708,719.81$ in about one second, and after the second stage – where an iteration includes 100 RLS applications and 0-5 EA applications – the result is $1,727,869.53$. This second stage of the process took over 8 hours, yet the result is clearly not optimal, since the rate of improvement per iteration did not greatly slow down.

Figure 6.1: This graph presents the progress of an optimization process for pla33810-1-1bsc which took over 8 hours. The objective value is given on the y-axis, and the number of iterations on the x-axis



In a similar attempt at obtaining a *best* value for pla33810-5-5usw presented in figure 6.2, a local optimum, which was $1.89 \times 10^{-7}\%$ worse than the best known result, was not escaped despite nearly 60,000 applications of EA.

Figure 6.2: The progress of the same optimization process applied to pla33810-5-5usw. The process took 5 hours, and it resulted in $Z = 1.470480155 \times 10^7$.



These two examples also serve to indicate that the problem difficulty is heavily influenced by other factors than instance size. The second instance has five times as many items as the first, yet the second appears to admit a near optimal solution much more readily than the first.

So the *worst* and *best* values are not necessarily the minimal and maximal values achievable. But all the experimental results fit in the range, and the corresponding percent-deficits are given in table 6.4. With these values, 0 is best and 100 is worst.

In the table I have included the results for SH, RLS, and EA from [BMPW14]. The results for the nondeterministic RLS and EA are averages over 30 trials, each of which was stopped after ten minutes or 100,000 consecutive iterations with no improvement.

Additionally, I have included some results from my implementation of DH.

	SH	DH	RLS	EA	IGDH	IHH
a280-1-1bsc	13.0818	-	5.7631	1.5072	**1.2417**	**1.2417**
a280-5-5usw	36.3928	20.6331	**0.0006**	0.0006	0.0474	0.0555
a280-10-10unc	24.2848	-	**0.0000**	0.0004	0.0173	0.0267
fnl4461-1-1bsc	16.9287	16.6953	10.7635	4.7778	**1.6512**	1.6576
fnl4461-5-5usw	31.0149	17.0515	**0.0084**	0.0553	0.0175	0.0187
fnl4461-10-10unc	23.3733	3.4203	0.2241	8.5935	**0.0149**	0.0979
pla33810-1-1bsc	18.9917	17.4395	3.1539	7.3687	0.5264	**0.5154**
pla33810-5-5usw	41.1326	19.6068	53.2629	67.9612	**0.0047**	0.0657
pla33810-10-10unc	30.6967	3.9120	87.2424	92.6544	**0.0324**	0.1290

Table 6.4: This table gives the percent deficit of the best seen objective values; lower is better. The results for SH, RLS, and EA are taken from [PBW+14]; while the results for DH, IGDH, and IHH are the results of original experiments detailed in this section. The best results are highlighted in bold.

6.3 Computational Study of ACO and LK for the TSP

This section demonstrates with a few experiments the performance of the ACO and LK implementations for the TSP.[2].

	mean	std
LK	2613	0
ACO	2613	0

Table 6.5: Tour length after 1 second of solving the benchmark TSP *a280*. The results are averaged over 10 trials – all of which found the optimum of 2613.

[2]These tests are consistent with the choice of rounding all distances up to the nearest integer. For this reason, the optimal tour lengths differ slightly from those which apply to the TSPLIB instances of the same name

	mean	std
LK	187 200	58.076
ACO	187 038	66.572

Table 6.6: Tour length after 10 minutes of solving the benchmark TSP *fnl4461*. The results are averaged over 5 trials. The optimum is 185707.

6.4 Computational Study ACO for the TTP

This section presents results from application of the composite algorithms ACO_{tsp} and ACO_{ttp}. Due to the nondeterministism of ACO, each result is the mean of five trials, and the sample standard deviation (std) is supplied.

In the two tables below – one for each of the algorithms – a final column gives the percentage std per mean. This is useful to quickly compare the experiments by how spread out the results are. To conclude this section, the results of the two tables are condensed in table 6.9, which compares the two algorithms.

	mean	std	$\frac{\text{std}}{\text{mean}} \cdot 100\%$
a280-1-1bsc	16896.74583009378	679.05111593977	4.0188%
a280-5-5usw	107046.08308917252	2779.97768282294	2.5970%
a280-10-10unc	425523.1502678897	3571.4695704274	0.8393%
fnl4461-1-1bsc	223901.3412126741	5376.1549418388	2.4011%
fnl4461-5-5usw	1560736.5967671007	7965.9859607386	0.5104%
fnl4461-10-10unc	6355652.977072473	87412.773809482	1.3754%

Table 6.7: This table gives the results of the experiments for ACO_{tsp}. Algorithm execution was stopped after 10 minutes. The mean is the average over five separate such executions.

	mean	std	$\frac{\text{std}}{\text{mean}} \cdot 100\%$
a280-1-1bsc	17684.645617329500	186.27158482922	1.0533%
a280-5-5usw	110255.43862388225	56.07233574242	0.0509%
a280-10-10unc	429082.458074452	6.981533989	0.0016%
fnl4461-1-1bsc	232906.6316696956	689.992173397	0.2963%
fnl4461-5-5usw	1572362.4543070450	1774.5933783021	0.1129%
fnl4461-10-10unc	6400666.984278966	1959.480393837	0.0306%

Table 6.8: This table gives the results of the experiments for ACO_{ttp}. Algorithm execution was stopped after 10 minutes. The mean is the average over five separate such executions.

With these results, it is clear from comparison to table 6.3 that, with the ex-

ception of *fnl4461-1-1bsc*, these composite algorithms handily outperform the previously presented algorithms, which only had one tour to work with.

	$\left(1 - \frac{\mathrm{ACO}_{tsp}}{\mathrm{ACO}_{ttp}}\right) \cdot 100\%$	$\left(1 - \frac{\max(\mathrm{RLS,EA})}{\mathrm{ACO}_{ttp}}\right) \cdot 100\%$
a280-1-1bsc	4.4553%	11.2679%
a280-5-5usw	2.9108%	5.3433%
a280-10-10unc	0.8295%	4.0476%
fnl4461-1-1bsc	3.8665%	0.5437%
fnl4461-5-5usw	0.7394%	5.9651%
fnl4461-10-10unc	0.7032%	2.7115%

Table 6.9: The center column of this table relates the results of the two previous tables by giving the percent deficit of the ACO_{tsp} result mean from that of ACO_{ttp}. In a similar fashion, the rightmost column relates the best previously published result to ACO_{ttp}. Note that these percent deficits use the value 0 as the "worst" value, as opposed to the previous section where the worst value was the objective value for the empty knapsack.

In addition to the summarizing results presented by the tables, I add that only for the instance *fnl4461-10-10unc* did ACO_{tsp} produce any solution with objective value greater than at least one of the five produced by ACO_{ttp}. In fact, the two best solutions of the total 10 produced for this instance, were produced by ACO_{tsp}.

Chapter 7

Discussion

The results of the previous chapter are discussed here.

7.1 IGDH and IHH

The results presented in table 6.4 show that, like SH and DH, the new IGDH and IHH algorithms cannot compete with the random climbers, RLS and EA, when enough time is given relative to the instance difficulty. However, while DH outperforms RLS and EA on only the two largest instances, IGDH and IHH have the best results on six of the nine instances. Furthermore, in the three cases where IGDH or IHH do not produce the best result, the results they do produce are well within 1% of the best.

In fact, the worst result achieved by any of the two is only 1.6576% worse than the best known result, while all the other algorithms have at least one result that is at least 20% worse than the best.

These experiments indicate that the performance of IGDH and IHH is much more stable than that of the other algorithms. This makes it suitable for use in the composite algorithm discussed in the next section. The stable performance is crucial in this context, because incorrectly deeming one solution better than

another, will propagate to future solutions through the pheromone deposition, which is a self-perpetuating mechanism.

Furthermore, the improvement of algorithm running time from IGDH to IHH is an enabling factor. Consider *fnl4461-10-10unc*, where IHH is faster by 2.5 seconds per iteration. An iteration of ACO_{tsp} or ACO_{ttp} takes about 3.4 seconds on this instance, and it calls IHH(3), i.e. there are three iterations, so 7.5 seconds are saved, cutting algorithm running time by a factor $\frac{7.5}{3.4+7.5} = 0.69$.

Even when much time is available, IHH is likely the better choice, as the small performance gap quickly can be closed by local search algorithms like RLS or EA.

This was the technique used to produce the upper-bounds for the nine instances in the previous section. Producing these revealed one case – pla33810-1-1bsc – where an acceptable result was not reached even after eight hours. The results in general indicate that the instances with the suffix "_1_1bsc" – one item per city, $C = 1$, *bounded, strongly correlated* – are much harder to solve. This is as expected from the *bsc* item types, but it has not here been tried to separate the *bsc* tag from the two other parameters, so it cannot be concluded that it is the cause.

As a final remark, notice that SH provided the previously best known results for *pla33810-5-5unc* and *pla33810-10-10unc*, which are 41% and 31% worse than the best solution (for the same tour) presented in this thesis[1]. These new results thus give better context for reasoning about the relative performance of the other algorithms. For example, it is now clear that for these instances, RLS and EA (1+1) do not even surpass the half-way mark in their climb from the empty packing plan toward the optimal one; whereas it could be previously said that RLS comes within $\left(1 - \frac{53.26\%}{100\% - 41.13\%}\right) \cdot 100\% = 9.52\%$ of the best known result.

7.2 The ACO-based TTP Algorithm

Table 6.8 presents results for ACO_{ttp} that are consistently better than the results for ACO_{tsp}. Additionally, the standard deviation is much higher for the ACO_{tsp} results. This is consistent with the expectation that ACO_{tsp} is capable of happening upon a short tour that lends itself well to being part of a solution to the TTP, but is equally likely to converge toward a tour that is short, but lends itself poorly. On the other hand, ACO_{ttp} is designed specifically to favor

[1] The results for DH are better, but they have not yet been published

those edges which in previous iterations were part of tours that in turn were part of the best TTP solutions. Thus, it is not suprising that the experimental results were relatively more consistent for this algorithm.

Let us examine more closely the two instances *fnl4461-1-1bsc* and *fnl4461-5-5usw*. In both cases ACO_{ttp} was better, but in the former case the difference was 3.87%, and in the latter it was 0.74%. Apparently, the degree by which ACO_{ttp} outperforms ACO_{tsp}, varies by the instance. The experiments presented in the previous chapter do not preclude the existence of instances for which this degree of "outperformance" takes a dip into the negative.

But there is no need for such preclusion. Instead the varying performance should be taken as an indication of the strength of the interconnection of the component problems. In that case, the instance *a280-1-1bsc* should be the most strongly interconnected of the six. While the experiments have not been thorough that I dare make this claim, this seems like the most intuitive conclusion. For that instance, there are very few items, and they are all very heavy and of very similar weight. So the packing plan is nearly dictated by the tour, as there is little other choice than to pack those items near the end of the tour. For this reason, the objective value will be devastated by choosing a tour – which may be short – but which is a poor TTP-tour for placing longer edges close to the end of the tour. It is exactly this kind of situation which ACO_{ttp} is capable of actively avoiding.

At the other end of this spectrum are *fnl4461-10-10unc* and *a280-10-10unc*. These have high item density, and the item weights vary greatly. This gives the packing plan construction algorithm great freedom to adapt to a poor tour. The results in figure 6.9 seem to agree with these observations.

In addition to these points which can be made by comparing the two algorithms, they are unique in the context provided by the literature, as no other published (to my knowledge) algorithm which solves both component problems is applied to any benchmark problem of comparable size. This makes it difficult to reason about the overall performance of the algorithms, but makes them all the more useful, as future work is now provided the means to do this.

Chapter 8

Conclusion

This thesis has presented background on the Traveling Thief Problem (TTP) and its component problems, the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP). Based on this, procedures and algorithms from the literature has been selected, adapted, and combined into an overall algorithm. It solves the TTP through an iterative procedure in each step of which the component problems are solved separately, but with respect to the context in which the solution is used.

Specifically, the iterative procedure is founded around an Ant Colony Optimization framework, where the pheromone level of edges reflects the quality of the TTP solution, rather than the TSP solution. The KP component is solved by an algorithm, IHH, based on the simple greedy heuristic for the ordinary KP which assigns to each item a score equal to its value divided by its weight. The TTP-specific version uses a score which is derived in the thesis, and which outperforms scores previously suggested.

Between the overall algorithm and IHH, new best results are found for the nine benchmark problems considered, with a significant gap to the previous best.

Bibliography

[ABCC99] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. *Finding tours in the TSP*. 1999.

[ACR03] David Applegate, William Cook, and André Rohe. Chained lin-kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92, 2003.

[BHS97] Bernd Bullnheimer, Richard F Hartl, and Christine Strauss. A new rank based version of the ant system. a computational study. 1997.

[BMB13] Mohammad Reza Bonyadi, Zbigniew Michalewicz, and Luigi Barone. The travelling thief problem: the first step in the transition from theoretical problems to realistic problems. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1037–1044. IEEE, 2013.

[BMPW14] Mohammad Reza Bonyadi, Zbigniew Michalewicz, Michal Roman Przybylek, and Adam Wierzbicki. Socially inspired algorithms for the travelling thief problem. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 421–428, New York, NY, USA, 2014. ACM.

[Cro58] Georges A. Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.

[DMC96] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, 1996.

[DS10] Marco Dorigo and T. Stützle. Ant colony optimization: Overview and recent advances. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, chapter 8, pages 227–263. Springer, New York, 2010.

[Hel00] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[Lin65] Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal, The*, 44(10):2245–2269, 1965.

[LK73] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the travelling-salesman problem. *Operations Research*, 21:498–516, 1973.

[Mic12] Zbigniew Michalewicz. Quo vadis, evolutionary computation? In *Advances in Computational Intelligence*, pages 98–121. Springer, 2012.

[PBW+14] Sergey Polyakovskiy, Mohammad Reza Bonyadi, Markus Wagner, Zbigniew Michalewicz, and Frank Neumann. A comprehensive benchmark set and heuristics for the traveling thief problem. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 477–484, New York, NY, USA, 2014. ACM.

[Rei95] Gerhard Reinelt. Tsplib. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`, 1995.

[SH00] Thomas Stützle and Holger H. Hoos. Max–min ant system, 2000.

[WHH09] Darrell Whitley, Doug Hains, and Adele Howe. Tunneling between optima: Partition crossover for the traveling salesman problem. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 915–922, New York, NY, USA, 2009. ACM.

[WHH10] Darrell Whitley, Doug Hains, and Adele Howe. A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover. In *Parallel Problem Solving from Nature, PPSN XI*, pages 566–575. Springer, 2010.