THE TARGET-RADAR ALLOCATION PROBLEM

Magnus Hindsberger

LYNGBY 1998
EKSAMENSPROJEKT
NR. 09/98

IMM

Trykt af IMM, DTU

Preface

This report is a result of my Master Thesis carried out from September 1997 to February 1998 at the Institute of Mathematical Modeling, IMM, the Technical University of Denmark. When beginning to plan this work my interest was in metaheuristics. I made contact to the Danish Defense Research Establishment, DDRE, as I knew they were working on a heuristic algorithm for allocation of HAWK air-defense missiles. Being a combinatorial optimization problem, metaheuristics could also be used for solving it. So I asked if they were interested in letting me try solve the same problem, with a few simplifications, but by using one or more of metaheuristic methods and evaluate the advantages and disadvantages of such an approach compared to the existing procedure. Since they had not yet researched the usefulness of these methods, but was planning to do so, it fit well into their plans and thus the answer was positive.

The problem itself was complex, making it an interesting challenge, also because it should be solved within a more strict time limit, than metaheuristics normally are used within.

Without the help of many people, this work could not have been done in its current form. I would especially like to thank my two mentors Victor Vidal and Pedro Borges as well as all the others at IMM, staff and fellow students, who have helped me. Also a great thank to the people at DDRE, especially the HAWK research team, Jens Meng Hansen, Karl Johan Simonsen and Torben Christensen. At last I would like to thank the crew of the HAWK battery, ESK 543, for showing me the actual system.

Lyngby, February 1998

Abstract

This report covers metaheuristic algorithms: Local Search, Simulated Annealing, and Tabu Search used for allocation targeting radars to enemy targets for the HAWK air-defense system. If implemented as a final part of the system such an allocation plan should be found within 3 seconds, which is a less timespan than metaheuristics usually are used within. The methods were compared to each other as well as an existing problem specific heuristic.

On this problem the implemented Tabu Search algorithm performed best and was with respect to the maximization of the objective function value superior to the existing algorithm. But the existing algorithm generally engaged the targets at longer ranges, a wanted property that was not modelled well enough by the objective function. Improvements on this point are possible in several ways and will be discussed.

KEYWORDS: Combinatorial optimization, metaheuristics, Local Search, Simulated Annealing, Tabu Search.

Contents

Chapter 1

Introduction

The most important element of the Danish land-based air defense since the mid-sixties has been the HAWK air defense system (HAWK, Homing All the Way Killer). While the system has been updated during the period, it is no longer up-to-date because it still uses a lot of analog electronics and data communication and lacks a modern decision support system, to help the operators engage the right targets. Therefore it has been decided to upgrade the system with a new search radar and battery control central, all digital. In addition the updated system should be able to coordinate the fire of several Stinger-missile teams. The upgraded system is called DEHAWK for Danish Enhanced HAWK.

If Denmark comes under attack the HAWK system should engage enemy fighters, helicopters etc. with the available missiles so that the most threatening and valuable enemy targets are engaged first. Since the air picture is changing constantly, the allocation of the missiles to the wanted targets must be continuously updated. Unfortunately the updating is non-trivial because the number of possible allocations is increasing fast with the number of enemy targets and because of the number of restrictions to the DEHAWK system described in the next chapter.

1.1 Why this problem

As described above the number of ways, even for relative small problems, the missiles, the controlling radars, and the targets can be combined during a period is large. In addition the targets must be allocated during specific time windows and the optimal solution does not necessarily have to include all the targets.

This complex nature of this allocation problem and the fact, that a solution must be found within a strict time span in order to keep the allocation up-to-date with the air picture, makes it a very interesting combinatorial optimization problem. Also this is after all a real problem, which within the military is of great importance and is researched throughout the world.

Most combinatorial optimization problems are hard to solve exactly, i.e. finding the global optimal solution. This is not because no methods to do it are known, but because they belong to the NP class of problems where the calculation time is growing exponentially with the size (e.g. the number of enemy targets) of the problem. Thus for larger problems non-exact or *heuristic* algorithms are used. Here and in the rest of this report, the word algorithm means a step-by-step way of doing some calculation analog to a computer procedure.

A heuristic (short of heuristic method or algorithm) can either be general applicable or problem specific. The latter can only be used for solving a single type of problem like the quadratic assignment problem, QAP, while the general applicable heuristics, also know as the metaheuristics, can – with only minor modifications – be applied to a wide selection of problem types. The metaheuristics are normally used for that range of problems, which should be solved faster than an exact algorithm, but has more time available than (near) real-time decision support systems that should be solved within seconds.

When the decision to upgrade the system was made, the Danish Airforce asked the Danish Defense Research Establishment (DDRE) to assist finding an effective algorithm for the allocation problem. An effective algorithm is in this case an algorithm, which within the required time span returns a satisfying allocation. The designer of the algorithm – in this case the Danish Airforce – then has to decide when a solution is satisfying. In some cases only optimal solutions may be accepted but in normal situations a solution can be accepted, when it is within a few percent of the optimal.

DDRE designed a problem specific heuristic algorithm, which after testing has been approved for use by the Danish Airforce. This report treats the same allocation problem and how this can be solved using metaheuristics. Three different metaheuristics: Local Search, Simulated Annealing, and Tabu Search will be implemented and compared with each other as well as with the DDRE algorithm in order to investigate, whether a metaheuristic solution is useful for solving the problem.

1.2 Contents

In this report the words in the *italic* typestyle represent the introduction of a new notion to be explained. [1] is a reference to the first entry in the Bibliography and (2.1) refer to equation 1 in chapter 2.

In chapter 2 the actual HAWK system, which the problem is based upon, will be described as well as the exact allocation problem to be solved. In addition a mathematical model describing the system will be formulated.

Chapter 3 is an introduction to the metaheuristic, which each will be used in an algorithm to solve the described allocation problem.

Chapter 4 will deal with the design and implementation of the metaheuristics algorithms.

In chapter 5 the test results are shown and the performance of the implemented algorithms will be compared.

Chapter 6 describes the implementation of possible improvements and the results of those applied to the algorithms.

In chapter 7 the results of the best of the algorithms are compared to those of the DDRE heuristic algorithm. The advantages and disadventages of the different approaches will be discussed.

Chapter 8 sums up the conclusions done and points out some areas of interest, which could be investigated in the future.

Chapter 2

The allocation problem

This chapter tries to give the reader a clear comprehension of the problem. At first the actual system is described. After that follows the specific formulation of the Target - Radar allocation problem. In the last section a mathematical model of the problem is formulated.

2.1 The DEHAWK system

In generally air defense units are made up of *firing units*, which will be denoted FU's. A FU is the smallest unit, which independently can engage a target. Several FU's are organized in a squadron (sometimes named battery), which also includes different radars, a command and control element, and support facilities. Several squadrons can form an air-defense battalion, where another level of command and control elements is added.

The DEHAWK allocation algorithm described later in this report can be used both at squadron and battalion level. Knowledge to the organization and equipment of those units is necessary in order to understand the problem. An organization chart of a typical DEHAWK battery is shown at the next page, while a battalion usually is formed of 4 batteries.

Of the units shown on the chart the Stinger teams are of less interest. Although part of the DEHAWK system, they will for simplicity reasons not be implemented in the allocation algorithm. The other units interact in the

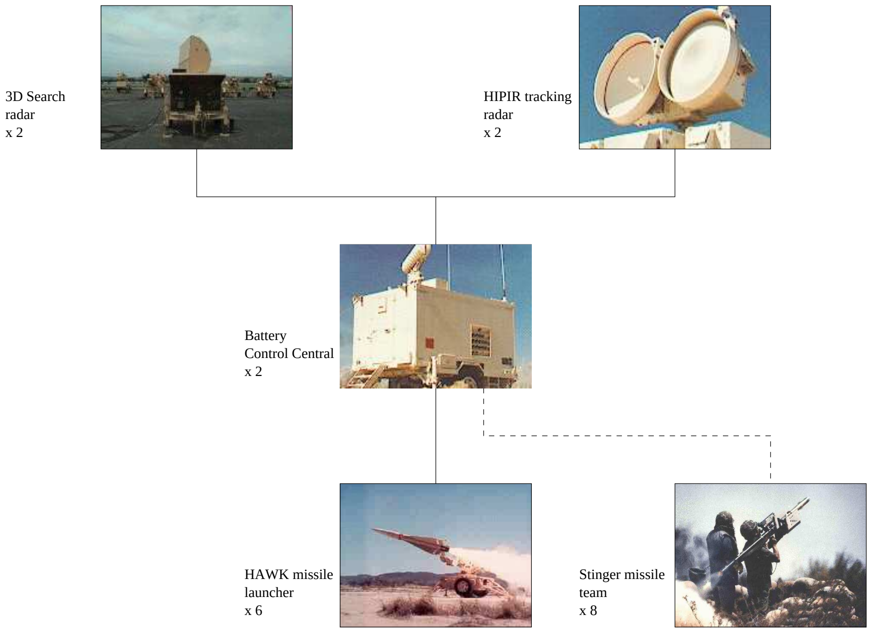


Figure 2.1: A typical organization of a DEHAWK battery

following way: The search radar will continuously search the airspace for any objects. In the battery control central the objects are sorted in friendly and enemy targets and the type of the enemy targets is determined. Now it has to be decided which – if any - of the enemy targets to engage, when it is going to happen and which of the available missiles to use. Note that the names HAWK and DEHAWK are normally used interchangeable and will from now both refer to the updated system.

2.1.1 The missiles

As mentioned two types of missiles are controlled by the system. The HAWK missiles have a maximum range of nearly 40 km. and can effectively engage target in up to 18 km. in altitude. But if a target is very close (i.e. within a few kilometers) the missiles cannot be fired, since the time of flight is to short to guide the missiles towards the target. Therefore for close range engagements the lightweight Stinger missiles are used, but the maximum effective range of those are limited to about 3 km.

If a target is to be engaged by a HAWK missile, the target has to be illuminated by one of the HIPIR tracking radars (HIPIR, HIgh-Power Illumination Radar). This means that the HIPIR starts tracking the target by continuously keeping it the target of the radaremission. Some of the emission is reflected from the target and is picked up by both the HAWK missile, which uses it to keep the guidance against the target, as well as the HIPIR to keep its track on the target. The process is sketched in figure 2.2.

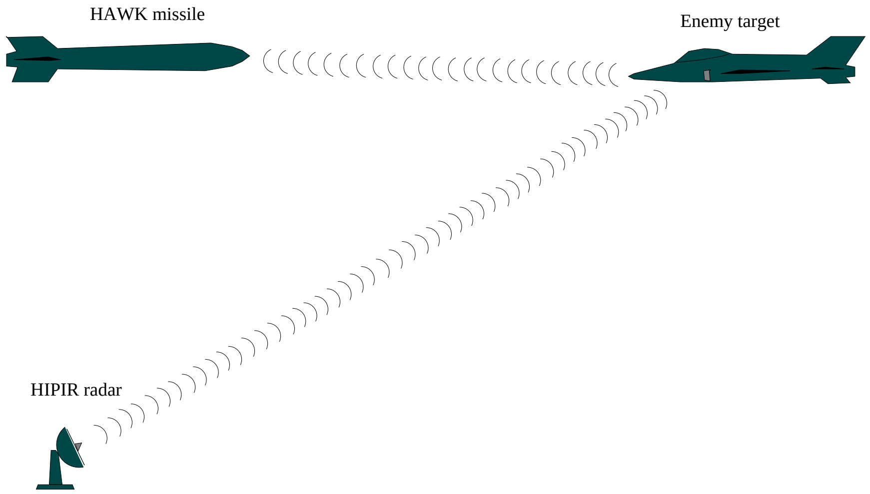


Figure 2.2: The HAWK guidance sytem

Often the targets are actually hit by the missile but else the missile warhead should ensure a kill against targets within a certain range of detonation. The guidance mentioned limit a HAWK squadron to only have two HAWK engagements running at the same time (i.e. one for each HIPIR). So the HIPIR's can be considered as the FU's.

Though excluded from the algorithm a short entry about the Stinger missile is appropriate. The Stinger is unlike HAWK a fire-and-forget missile, which uses the thermal radiation of a target to guide the missile towards it, thus eliminating the need for a tracking radar.

2.1.2 Notation

In the report some special notions will be used like target type, objects, and "kill-probability". They will be defined in the following of this chapter.

The targets the HAWK systems are to engage can be divided into *target types* e.g. airplanes, helicopters, and missiles. Similar these systems are defending some *objects* against the threat. Those objects can be the HAWK systems themselves, headquaters, airfields, etc. Actual two *missile types* exists as explained – the HAWK and the Stinger, but only the HAWK will be implemented as part of this report.

An *engagement* will denote the time sequence of allocating a single target, firing at it, until it has been determined whether or not the target was hit.

2.1.3 Intercept area

The area where the missiles can engage the targets effectively does not only depend on the physical range of the missiles. For instance will the HIPIR radar track of targets if they are not closing with a certain speed. This is because the HIPIR radar is a Doppler radar, which means it uses the change of frequency of the reflected radar signals which come from closing or diverging targets to calculate their positions. In addition the system should only engage closing targets, so the intercept area is only defined for those. Therefore if the speed vector of a target is perpendicular to the HIPIR (or pointing away), it can not or should not be engaged. It all sums up to an area for each FU where the target can and should be engaged. This will be designated the *intercept area*. Corresponding the time, where a target is inside the intercept area is designated the *intercept period*.

2.1.4 Kill probability

The probability for shooting down a target is traditionally denoted *Pkill*. This value depends mainly on the missile type (HAWK or Stinger), the target type, and the target altitude, downrange, and offrange (see figure 2.3 for definitions). The last three factors should be seen in proportion to the FU, which in this case is a HIPIR radar, since it is assumed that the missile launchers are located so close, that actual range of missiles from different launchers is insignificant.

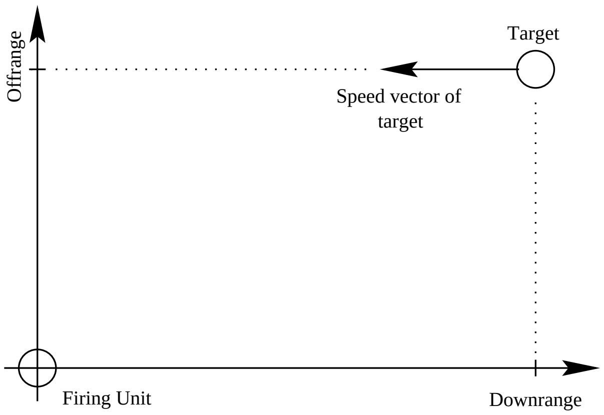


Figure 2.3: Definitions of Downrange and Offrange

2.2 Problem description

The allocation of targets is only a small – but still important – part of the total HAWK system. The problem can, when the Stinger missiles are omitted, be defined as: In a specific time period, called the *planning period*, to decide which HAWK missile launchers to engage which targets, at which time, and which HIPIR to use controlling the missiles flight. The planning period can vary, but is usually between 2 and 5 minutes. The allocation has to be updated continuously so changes in the target course and speed is taken into account. Therefore the algorithm solving the described problem must calculate a new allocation plan every time the search radar has finished a rotation. During a single rotation the information about all targets in the airspace is updated. Since a rotation only takes about 3 seconds, the time span, within which the allocation plan should be calculated, is very strict.

2.2.1 Wanted Properties

In order to evaluate whether an algorithm can be accepted as good enough, it must be decided, which criteria that it should fulfil. Some of the criteria are *strong* meaning that they should be complied with in any case.

For this problem the strong criteria that have been defined are:

- Speed: The algorithm must return a solution within the 3-second time limit
- Maximizing: The mean expected value of the objective function implemented in the algorithm should be within 5 % of the global optimum

A set of weaker criteria can be defined too, as some criteria, which is wanted but not demanded of the algorithm. The following weak criteria have been defined for this problem:

- Range: The number of targets which crosses the critical line before being engaged, should be minimized
- Robustness: The algorithm should, when run real-time, not change the recommended solution for every new 3-second run unless targets are maneuvering or new targets appear

The first of the weak criteria can possible be chained to the strong criterion of maximization of the objective value by a suitable definition of the objective function. The robustness criterion is included because the users of the system would loose confidence in a system that every 3-second changes its recommendation of the allocation plan.

2.2.2 Assumptions

When making an algorithm to solve the Target-Radar allocation problem, some assumptions have to be made to ensure that the algorithm returns a satisfying allocation plan. The assumptions made are:

- All targets within the intercept-areas are known
- The type of every target is correct determined
- Every target keeps the same speed and course within the planning period
- It can be decided within a certain time if an engagements fails

The first two points describe that full information is needed about every target in order to find a good allocation plan – an assumption, which in practice rarely would be truth. Often low-flying targets would avoid detection until well within the intercept areas. Because of this an algorithm will only optimize for the targets known to the system even though some targets might exist – unknown to the algorithm – which could increase the expected benefit if engaged. So practically only the last two points apply. The latter tells that if a missile misses the target, the operator will find out – normally within seconds. In this case the target will reappear on the threat list and will be engaged again, if it is included in the next allocation plan.

2.2.3 Algorithm interfacing

The input to an algorithm solving the named problem has to describe the system status and the threat. The system status consists of the numbers of HIPIR's, launchers, and objects as well as the positions of them all. The threat can be described as a list of the targets including the relevant data e.g. speed vector and position relative to each FU. In addition the algorithm will need several constants like Pkill of the missiles at different ranges, length of planning time, etc., which all should be input to the algorithm too.

In return the algorithm, with the properties mentioned above, should return a recommended allocation plan.

2.2.4 Military value

The objective function of the model can for instance describe the expected value of the targets shot down, summing for every engagement the product of the Pkill of the engagement and the "benefit value" that would be gained from the kill. A possible way to express this "benefit value" is for every target type to define a *basic military value* against each of the defended object types. This value expresses how desirable it is to shoot down the target, and depends upon how threatening it is to the specific object and the value of this object.

The threat is assumed highest at the *critical line*, which is the circle around the object, where the target is expected to fire at the object. Therefore when a target is at the critical line around an object the *specific military* value equals the basic military value of that target-object combination. If the range to the target is less or greater than the critical line, the military value is reduced by multiplying a range-dependent factor between 0 and 1. Figure 2.4 should illustrate this.

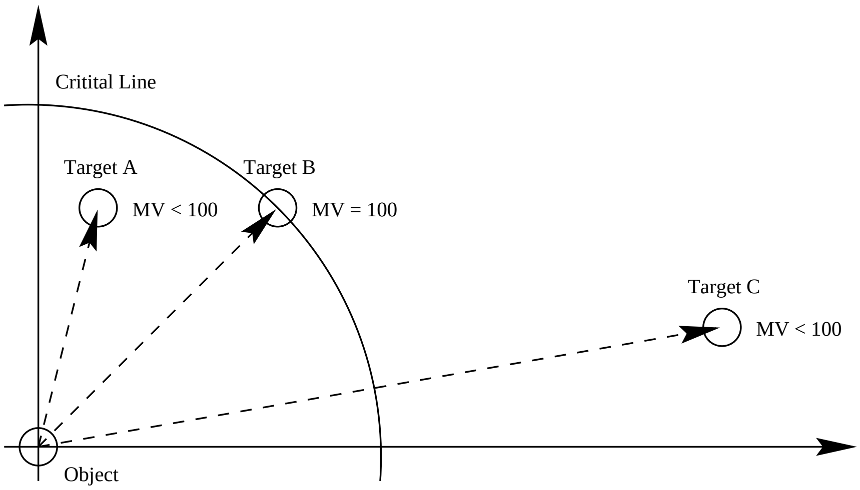


Figure 2.4: The specific military value (MV) of three targets at different ranges, all with a basic MV of 100

The specific military values of a single target against all the defined objects should be combined into a single *actual military value*. The weight which each specific military value accounts for, can for instance depend upon the size of the threats the specific values express as shown in the example below.

Example 2.1 (Calculation of the actual military value) *This example show how the actual military value of a target is calculated from the specific military values of three existing objects.*

Target specific military value against:

$$
\begin{array}{lcr}
\textit{object 1, } mv_1 & = & \textit{90.0} \\
\textit{object 2, } mv_2 & = & \textit{85.0} \\
\textit{object 3, } mv_3 & = & \textit{1.5} \\
\hline
\textit{total} & = & \textit{176.5}
\end{array}
$$

$$
\begin{array}{lcl}
\textit{Actual military value} & = & \sum_i (mv_i \cdot \frac{mv_i}{total}) \quad ; \; i = 1,..,3 \\
 & = & \textit{86.7}
\end{array}
$$

□

2.2.5 Reductions and simplifications

Since the total problem is very complex and some of the data is restricted, the problem has been simplified. Generally all data about Pkill, military values etc. are not the real ones but have been generated in a way, so that the results still will appear realistic.

First of all – as mentioned – the Stinger missiles are omitted. Test runs done by the DDRE showed that they were rarely used. Because of the limited range the targets were generally engaged at longer range by the HAWK missiles. In addition the Danish Airforce shows less interest in this part of the problem for the time being.

Another fact is that the algorithm is not to control the engagements automatically. Rather it is an expertsystem advising the users to take the right decisions. When the algorithm has recommended that a target should be engaged in a certain way, the controlling officer still has to approve the allocation and - if accepted - turn the target over to the firing officer for the recommended FU. He should direct the controlling HIPIR to lock on the target and after that has happened, he should press the firebutton at the appropriate time. In total several reaction times have to be taken into account, but in this report it has been simplified. For instance, the reaction times in the example above have been summarized to a single delay-before-pressing-button time.

The missile speed will be considered to be constant, mainly because the actual course of speed is quite complex. At last it is not taken into account that two HIPIR should not illuminate targets within a few degrees of each other, since the returns of the radaremission will confuse both.

2.2.6 Related work

Very little literature exists about this and similar problems, the most being in classified military technical reports. In addition the problem does not fit into any of the existing models of wellknown optimization problem, like the Traveling Salesman Problem, the Quadratic Assignment Problem, etc. Some similarities to other optimization problems with time windows can be seen though, and the model formulated in section 3 is thus inspired by [7] and [11].

The objective function used is the same as the one used by the DDRE research team, in order to make comparison of the easier.

2.3 A mathematical model

A mathematical model of the allocation problem will now be formulated and will – hopefully – help to understand this.

In the model the following indices are used:

$i = 1, 2, ..., n$ denotes a target from the list of targets.
$j = 1, 2, ..., m$ denotes the HIPIR used to illuminate the target.
$k = 1, 2, ..., p$ denotes the launcher used for the engagement.

The decision variables to be determined are:

$t_{ijk} \in \mathbb{R}$ denotes the time where the firebutton is pressed for a missile from launcher k to engage target i controlled by HIPIR j.
$x_{ijk} \in \{0, 1\}$ denotes if 1 that the engagement including target i, HIPIR j and launcher k is included in the allocation plan for the planned period.

The following functions are used in the model:

$mv(t_{ijk})$ calculates the military value of target i if engaged at t_{ijk}.
$Pk(t_{ijk})$ calculates the Pkill of target i if engaged at t_{ijk}.
$f(t_{ijk})$ returns the time of the intercept of target i using a missile from k launched at t_{ijk} and controlled by HIPIR j.

In addition the constants below will be used:

eng_start_{ij} is the first time where the firebutton can be pressed to engage target i – i.e. i enters the intercept area of HIPIR j.
eng_end_{ij} denotes the time where target i will exit the intercept area of HIPIR j.
r_hipir_j is the time where HIPIR j are ready for the first engagement.
r_lchr_k is the time where launcher k are ready for the first launch.
msl_no_k denotes the number of missiles on launcher k at the start time.
t_lk is the reaction time from the decision to shoot is taken and until the firebutton can be pressed due to the radar lock-on time
t_ev is the time from a target is hit and until it has been evaluated whether it was shot down. If not it probably should be reengaged.

t_lchr denotes the minimum time which must be between two launches from the same launcher.

Now the model can be formulated. For every target Pkill is the probability of shooting it down at a certain time and the military value is the benefit of the kill if made. The total expected benefit of an allocation plan can be calculated using those two functions, giving as the result the total expected military value to be shot down. Using the variable x_{ijk} that denotes the included engagements, the model can be formulated as:

$$maximize \qquad \sum_{ijk} (Pk(t_{ijk}) \cdot mv(t_{ijk}))\, x_{ijk} \tag{2.1}$$

subject to

$$\forall ij: \quad \sum_{k} (t_{ijk} \cdot x_{ijk}) \geq eng\_start_{ij} \tag{2.2}$$

$$\forall ij: \quad \sum_{k} (t_{ijk} \cdot x_{ijk}) \leq eng\_end_{ij} \tag{2.3}$$

$$\forall j: \quad \sum_{ik} (t_{ijk} \cdot x_{ijk}) \geq r\_hipir_j \tag{2.4}$$

$$\begin{aligned} \forall j: \quad & \sum_{k} (f(t_{i_1jk}) + t\_ev) x_{i_1jk} \leq \sum_{k} (t_{i_2jk} - t\_lk) x_{i_2jk} \quad i_1 \neq i_2 \quad \text{or} \\ & \sum_{k} (f(t_{i_2jk}) + t\_ev) x_{i_2jk} \leq \sum_{k} (t_{i_1jk} - t\_lk) x_{i_1jk} \quad i_1 \neq i_2 \end{aligned} \tag{2.5}$$

$$\forall i: \quad \sum_{jk} x_{ijk} \leq 1 \tag{2.6}$$

$$\forall k: \quad \sum_{ij} x_{ijk} \leq msl\_no_k \tag{2.7}$$

$$\forall k: \quad \sum_{ij} (t_{ijk} \cdot x_{ijk}) \geq r\_lchr_k \tag{2.8}$$

$$\begin{aligned} \forall k: \quad & \sum_{j} (t_{i_1jk} \cdot x_{i_1jk} + t\_lchr) \leq \sum_{k} (t_{i_2jk} \cdot x_{i_2jk}) \quad i_1 \neq i_2 \quad \text{or} \\ & \sum_{j} (t_{i_2jk} \cdot x_{i_2jk} + t\_lchr) \leq \sum_{k} (t_{i_1jk} \cdot x_{i_1jk}) \quad i_1 \neq i_2 \end{aligned} \tag{2.9}$$

Restriction (2.2) and (2.3) ensure that the targets are engaged within valid intercept areas and that the reaction time not will let a target get out of range before the actual launch. The restrictions (2.4) and (2.5) ensure that one HIPIR are not used in more that one simultaneous engagements. If the radar used for one engagement when the algorithm starts (2.4) blocks it until finished, while (2.5) is the general equation saying that the time periods of the engagement of two different missiles using the same hipir j must be disjoint.

Restriction (2.6) ensures that one target is only allocated once in the allocation plan and (2.7) that no more than the number of missiles available on a given launcher are used.

Restriction (2.8) and (2.9) are much alike (2.2) and (2.3) describing the initial and the general case respectively, and ensure that the same launcher will not launch twice within the restricted time.

Chapter 3

Metaheuristics

As mentioned in the introduction most combinatorial optimization problems in practice cannot be solved exact. This can for instance be if the size of the problem (i.e. the number of combinatorial possibilities) is so huge, that the optimal solution cannot be found within the available time. Instead you have to try solving such problems with heuristic methods. A heuristic has by Nicholson [14] been described as a procedure " ... for solving problems by an intuitive approach in which the structure of the problem can be interpreted and exploited intelligently to obtain a reasonable solution". In other words a heuristic is a method where common sense and thumb-rules are used to construct and improve possible solutions. The idea is to within reasonable time to obtain an acceptable solution.

Heuristics are often made especially for the specific problem, which should be solved, but a class of heuristics exist, which can be used in general with very little fitting needed for the different problems. These general heuristic methods can be combined to make them more efficient. The term *metaheuristic method* denotes the use and eventually combination of the general heuristic methods and the understanding of this is the purpose of this chapter and on of the main elements of this report.

This chapter will start by introducing the basics about heuristics – e.g. where they are used and which properties they should have. Next come section 2, 3, and 4, where some of the most known general heuristic methods: Local Search, Simulated Annealing, and Tabu Search are covered. The last section will sum up the important things to note, when implementing a metaheuristic method.

3.1 Introduction to heuristics

As mentioned does the name heuristic denote a non-exact method for solving a problem. For larger problems they are generally faster than the exact methods and returns often, but not necessary every time, an acceptable but seldom the optimal solution.

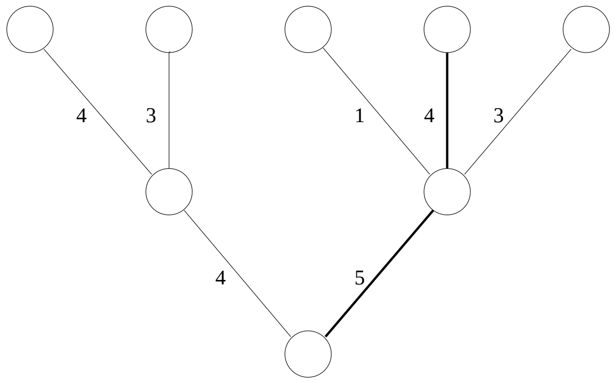


Figure 3.1: Example of tree structure

As an example the tree structure shown in figure 3.1 can be viewed. It consists of 5 leafs (the 5 circles in the upper row) and 3 nodes, where the one in the bottom is denoted the root. The elements are connected by branches as shown, each with a weight c_i. It is wanted to find the route from the root to a leaf along the branches, where the total weight (i.e. the sum of the used c_i's) is highest.

One way of doing this exact could be to write an algorithm, which would calculate the weight of all the possible routes and return the one with the highest value.

But if the tree is very big or the time available to do the calculations is very limited, an exact algorithm might not be useful. In such cases a heuristic solution to the problem is the only way. An intuitive possibility is at each note to pick the branch reaching up with the highest weight. This particular search method, which would normally be referred to as a *greedy algorithm*, is very similar to one of the Local Search strategies discussed in the next section. Using this algorithm on the example, the optimal value 9 is found by adding the weights of the marked branches.

If looking at the problem of minimizing the weight, the branch with the smallest weight reaching up at the visited nodes should be chosen instead. But in this example, the algorithm returns the value 7, which is greater than the optimal value of 6. In other examples even the worst possible solution might be returned. In generally, though, this algorithm will return a good (i.e. satisfying), but not necessary the optimal, solution to a problem, which is one of the properties heuristics are wanted to process as discussed later.

Another heuristic algorithm that could be used on the example too, is to pick randomly three routes, calculate the total weight of each and return the one with the highest value (or lowest if minimizing). This algorithm would also generally return a good solution.

3.1.1 Why use heuristics?

Some reasons to use a heuristic method have already been mentioned, but not all. To sum up, the most important reasons to use heuristics are:

- the problems, where exact algorithms to solve them still are unknown. In those cases heuristic methods of solution is the only way.
- those problems, where the computation time of the algorithm takes long. Many combinatorial optimization problems belongs to the class NPC, where the computation time will grow exponential with the size of the problem. The size could for instance be the number of nodes in a tree, number of towns in a "Traveling Salesman" problem, etc. For large NPC problems exact algorithms do not exist in practice.
- the exact algorithms that initially starts with a possible solution and subsequently tries to improve the one. These algorithms will generally terminate faster, if a good initial solution is given. An example is the "Northwest Corner principle", which is used for finding a good initial solution to transportation problem solving algorithms.

Though heuristics can be useful in many cases it has to be remembered – heuristic methods of solution are only a supplement to the exact algorithms.

Many practical problems require exact solutions and not just solutions close to the global optimum. And a heuristic algorithm is not necessary easy to develop as many have experienced.

3.1.2 Properties of heuristics

When developing a heuristic algorithm, the following should be kept in mind. The quality of all heuristics is not the same. In general the quality depends upon the following properties, which a heuristic should process:

- return good solutions with a high probability.
- similar returning poor (i.e. not satisfying) solutions with a small probability.
- fast termination, at least faster than any known exact algorithm used on the same problem.
- the heuristic should be easy understand. Many people would prefer an algorithm, "only" giving a satisfying solution, but where they understand the principles behind instead of a complex algorithm that magically returns the exact solution.

A heuristic is often designed with solving a specific problem in mind, and can for that reason seldom be used on other problems. A class of generally applicable heuristic methods exists though, which include the previously named Local Search methods. As mentioned the general heuristic methods can be combined in order to obtain better solutions or performance of the algorithm, giving us the metaheuristics. But the basic is a few general heuristic algorithms.

These are not ready-to-use algorithms, but require some thoughts about how the implementation of the particular problem should be done. The algorithms, how they work, and how they should be implemented will be described in the following sections.

Since the general applicable heuristics cannot take advantage of the structure of the problem the same way as a problem specific heuristic, you can expect the computation time or the quality of the solution of a general applicable heuristic to be inferior to that of a problem specific heuristic. So why should anyone want to use a general applicable heuristic, if he can make a better problem specific heuristic? The main reason is that new algorithms do not have to be written from the scratch every time. Once one algorithm has been implemented, the next ones will come faster than if a new method was to be designed every time. This is why much research is done in metaheuristics and why they are used more and more common.

3.2 Local Search

The *Local Search* (LS) strategies, by some called *Descent* strategies, are the most simple, but still powerful, general applicable algorithms. Local Search is an iterative method, where new solutions are sought in the *neighbourhood* of the current solution. This section should give a clear picture of how the strategies work.

Look at a combinatorial optimization problem with a well-defined solution space S. Each of the possible solutions $x \in S, x = (x_1, x_2, ..., x_n)$ can be evaluated by an objective function (or cost function) denoted $C(x)$. In addition each solution x also has a defined neighbourhood denoted $N(x)$, which could be a result of adjusting each x_i one by one. In the following it is assumed, unless otherwise stated, that the objective function value should be minimized.

Graphically, such a problem can be visualized as trying to find the lowest point in a complex and often highly contoured landscape. An optimization algorithm can thus be viewed as an explorer wandering through valleys and across hilltops searching for the topological lowest point. The grid location where the explorer is, can be likened to the current solution x, and the topological height of this location to the result of the evaluation function, $C(x)$. The neighbourhood $N(x)$ can be defined as the grid locations found in a one meter distance of x in each of the four corners of earth respectively.

Staying in the explorer analogy, two LS strategies will be introduced below:

1. the explorer walks to one of the four neighbourhood locations randomly selected, evaluates the height, and if the height is lower than the one in the location he came from he continues from the new location choosing another of the new neighbours randomly. Otherwise he returns to the old location and continues from there. If he has no changed location for a number of iterations, he is possible trapped in one of the local extremes (which might be the global one), and the algorithm is stopped. This strategy is called *Random Search*.

2. the explorer walks to each of the four neighbourhood locations and evaluates the height of each of them. He continues from the best (i.e. lowest) of them, as long as it is lower than the location he came from. Otherwise he is in one of the local extremes, and the algorithm is stopped. This strategy, which was sketched in the previous section is called *Steepest Descent* (or in case of maximizing, *Steepest Ascent*).

As it can be noted, the explorer in a LS algorithm only accepts downhill moves. This severe decreases the probability of finding the global optimal solution as seen in figure 3.2. The explorer is standing at point A, with in this example only two neighbours, B and C. He will either pick one of them randomly and evaluate the height of this or he will evaluate both B and C. But he will never choose B as the next current solution, since it is higher located than A. Instead he will walk downhill toward D and eventually reach this local minimum point and thus never come to the global minimum in E. But by rerunning the algorithm several times with different start locations, the chance of finding the optimal, or at least a better, solution is improved considerable. Using LS in this way is often very powerful.

The LS algorithms are useful, since they are simple and general applicable as long a neighbourhood and an evaluation function can be defined for the problem. A pseudo-algorithm of the Random Search strategy is showed in figure 3.3.

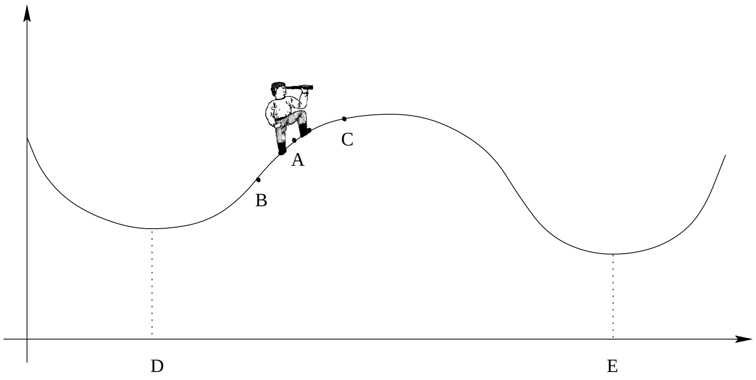


Figure 3.2: The "landscape" of a unknown function

```
procedure Random Search
begin
  choose initial solution, x ∈ S
  repeat
    choose neighbour, y ∈ N(x)
    ΔC = C(y) − C(x)
    if ΔC ≤ 0 then
      x := y
      C* := C(y)
    endif
  until stop
end
```

Figure 3.3: Pseudo-code example of the Random Search strategy

3.2.1 Implementation

Three things are of importance when implementing LS. The definition of the neighbourhood, $N(x)$, the objective function, $C(x)$, to evaluate the neighbours, and the stopping criterion. How they can be implemented is shown in Example 3.1 below.

Example 3.1 (The Traveling Salesman Problem) *The famous* Traveling Salesman Problem, *TSP, involves the design of a minimum cost path for a salesman, which has to visit a number of cities. Each city must be visited exactly once and the path has to end in the same city as it started.*

In this case the problem has 6 cities numbered 0 through 5, where the salesman has to start from (and thus end in) city no. 0.

A solution can be expressed as:

City no.	*0*	*1*	*2*	*3*	*4*	*5*
Succession no.	*0*	*1*	*3**	*2*	*4**	*5*

where the Succession number *denotes the number in the succession where the city with* City number *is visited. E.g. in the solution above, city 2 is visited as number 3, when the salesman departs from city 0 as illustrated in solution A in figure 3.4.*

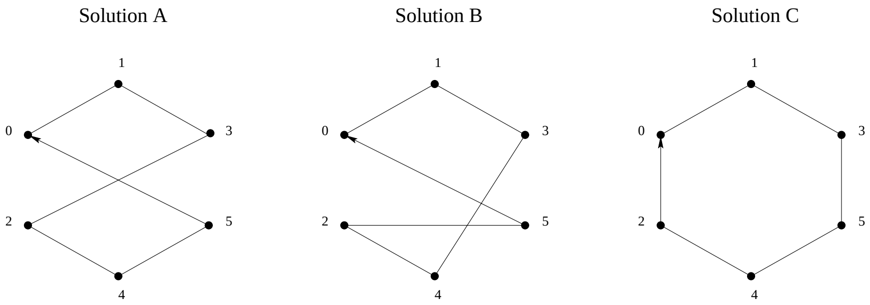


Figure 3.4: Possible solutions to the Traveling Salesman Problem

The neighbourhood: *A neighbourhood of the TSP can be defined in several ways:*

- *as moving one city in the sequence to another place in the sequence, e.g city 1 is visited last instead of first and the other cities are thus visited one "step" earlier.*
- *as an exchange of two cities in the succession they are visited, for instance the two marked with a *, resulting in the B solution of figure 3.4, where city 2 now is visited 4th and city 4 is visited 3rd. This neighbourhood is traditionally denoted* 2-OPT *in TSP litterateur, in the case it is restricted to not exchanging two cities next to each other in the succession.*

In the following of the example the neighbourhood is the latter of the two mentioned, but without the 2-OPT *restriction.*

Note that in this example the column for city 0 must be unchanged, since the city to start from is fixed. The solution space S is then given by the number of permutations of city 1 through 5, resulting in $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5! = 120$ *possible solutions.*

The number of neighbours to each solution is $4 \cdot 3 \cdot 2 \cdot 1 = 4! = 24$*. For a larger number of cities, n, the number of neighbours to each solution, corresponding to* $(n-1)!$*, would be astronomic, and the Steepest descent strategy unusable.*

Objective cost: *If the cost of traveling between two cities is assumed to be proportional to the distance, the cost of each solution can be calculated from the distance chart in figure 3.5.*

	1	2	3	4	5
0	2.0	2.0	3.5	3.5	4.0
1		3.5	2.0	4.0	3.5
2			4.0	2.0	3.5
3				3.5	2.0
4					2.0

Figure 3.5: The distance between the cities in the TSP

Assuming the current solution is A, the objective cost, $C(A)$, of this solution is found as:

$$C(A) = d(0,1) + d(1,3) + d(3,2) + d(2,4) + d(4,5) + d(5,1) = 16$$

where d(0,1) denotes the distance between city 0 and city 1.

Now a random solution, B, of the defined neighbourhood is picked. $C(B)$ of this is calculated as:

$$C(B) = d(0,1) + d(1,3) + d(3,4) + d(4,2) + d(2,5) + d(5,1) = 17$$

Since $C(B) > C(A)$ the move to B is rejected and the search from A is continued. Another random neighbour is picked from the neighbourhood, this time C from figure 3.4 by exchanging the number in the succession city 2 and 5 are visited. Evaluating the objective function as before, $C(C)$ is found as:

$$C(C) = d(0,1) + d(1,3) + d(3,5) + d(5,4) + d(4,2) + d(2,1) = 12$$

Now a better solution is found and C can be assigned as the current solution. So in the next iteration the neighbourhood of this is searched.

Stopping criterion: *It is chosen to continue as above until 20 new neighbours in a row have been rejected, since it is then assumed that a local minimum point has been reached. Other stopping criteria will be discussed in the next section.*

$\square$

Whether Random Search or Steepest Descent should be used depends upon the problem. Random Search requires only C(x) to be calculated once in each iteration, but has in general more iterations than Steepest Descent, since the latter will go more straight for the "closest" minimum point. If the size of the defined neighbourhood is big compared to the computation time of the evaluation function, Random Search would normally be most efficient. On the other hand will Steepest Decent generally do best with small neighbourhoods. The two strategies can be combined, which in some cases will be more efficient.

For both version the computation time of the procedures finding neighbours and calculating C(x) should be optimized, since these procedures will be called many times during a single run.

In the following sections the Simulated Annealing and Tabu Search method will be described. Both can basically be viewed as advanced LS strategies.

3.3 Simulated Annealing

This method is based on the annealing process of solids and is thus known as Simulated Annealing (SA). Annealing is a process where a material is heated to a very high temperature where all particles is moving randomly, without interaction between each other. Now the material is slowly cooled in order to obtain a highly ordered structure of the material with an almost minimum energy state when it freezes. SA can be introduced by showing the analogy between the annealing process and finding the minimum cost in a combinatorial optimization problem, but the easiest way of introducing SA is to show, that it is an extension to LS as mentioned in the previous section.

The LS methods are easy to understand and, depending upon the problem, are normally quite simple to implement, but have their weaknesses in being unable to overcome local minimum points. Simulated Annealing (SA) can be viewed as an advanced version of Random Search, which with the introduction of a stochastic element, in some degrees allows uphill moves. It now becomes possible to escape from local minimum points, improving the chance of finding a better – including the optimal – solution.

Like in Random Search, downhill moves are always accepted. The new part – the probability of moving uphill to a neighbour – is controlled by the *acceptance function*, formulated as $exp(-\Delta C/T_k)$, which denotes the probability of accepting the neighbour resulting in a ΔC change of C(x). The probability depends as it can be seen upon the control parameter T_k, called the *temperature*. In the beginning the temperature is so high, that almost every uphill move is allowed, but as time goes the temperature is lowered and the probability of moving uphill is decreased until practically being zero. Because of the dependence upon ΔC a neighbour solution being only slightly worse than the existing would be allowed with a greater probability than a neighbour resulting in a major increase of C(x).

Example 3.2 (The Traveling Salesman Problem - continued) *This is best illustrated by resuming the TSP example from section 2. Assuming the temperature $T_k = 100$ the probability of accepting solution B, when the current solution is A, can be found as:*

$$e^{\frac{-\Delta C}{T_k}} = e^{\frac{-1}{100}} = 0.99$$

So a random number between 0 and 1 is generated, and if it is below 0.99, solution B is assigned as the "current" solution. If the temperature instead is $T_k = 1$, the probability of accept will instead be as low as about 37 %.

□

Depending upon how often the temperature is lowered you are talking about inhomogeneous respectively homogeneous SA. A pseudoalgorithm of the latter is showed in figure 3.6.

The inhomogeneous version is special case of the homogeneous, where the temperature is lowered each iteration corresponding to $L_k = 1, \forall k$.

The mentioned stochastic element comes from the function $random[0, 1]$, which produces uniformly distributed values between 0 and 1. Because of this, the algorithm, like the Random Search strategy, is nondeterministic and will often return different solutions to the same problem for each run.

When evaluating SA with respect to the four properties that heuristics should process, it will be noted that good and poor solutions *can* be obtained with respectively high and low probabilities, but that depends upon the problem.

Returning to the explorer analogy, the explorer now has a limited amount of energy, which allow him to move uphill, but as time goes he will become

```
procedure Homogeneous Simulated Annealing
begin
  choose initial solution, x ∈ S
  initialize T_0, L_k
  k := 0
  while not stop do
    counter := 0
    repeat
      choose neighbour, y ∈ N(x)
      ΔC = C(y) − C(x)
      if ΔC ≤ 0 then
        x := y
      elseif exp(−ΔC/T_k) > random[0, 1] then
        x := y
      endif
      counter := counter + 1
      k := k + 1
    until counter ≥ L_k
    update T_k, L_k
  end
end
```

Figure 3.6: Pseudo-code example of homogeneous Simulated Annealing

tired and be less and less eager to move uphill. As his energy-resources reaches zero, he will only walk downhill. If the landscape he is exploring is highly contoured with a lot of valleys of different depth, the chance that he should end in one of the deepest is not that high, since he easily could be trapped in a "non-optimal" valley without the energy to climb the hills around him. It nevertheless generally returns much better solutions than a single LS run, and though it takes considerable more time, it turns out that SA is well suited to solve many problems.

And the method is easy both to understand and implement, since it is only a quite simple extension of Random Search.

3.3.1 Implementation

When implementing SA in order to solve a problem, some problem-specific and some generic decisions has to be made.

The problem-specific decisions are:

- Neighbourhood, how should the neighbourhood be defined in the solution space and how should it be searched.
- Objective function, how can a solution be evaluated.
- Initial Solution, should it be picked randomly or systematic.

They are all similar to the ones described in the TSP example in the previous section, and will therefore not be discussed here.

The generic decisions to make are all independent of the actual problem, and concern, except for the stopping criteria, all the *cooling strategy*. A cooling strategy consists of:

- The initial temperature, T_0.
- A cooling function to reduce the temperature.
- For the homogeneous version only – the number of iterations between each reduction in temperature, L_k.

The initial temperature T_0 should be chosen so that almost every generated neighbour is accepted initially, which can be formulated as:

$$exp(-\Delta C/T_0) \simeq 1 \tag{3.1}$$

where the left hand side as mentioned denotes the probability of accepting the neighbour, which results in a ΔC change in the objective function.

Several cooling strategies exist. The most common is the so-called *geometric cooling scheme*, which involves a geometric temperature function formulated as:

$$T_{k+1} = \alpha T_k \qquad ; \quad 0 < \alpha < 1 \tag{3.2}$$

Practice has shown that the value of α should be high, e.g. in the interval $0.8 \leq \alpha \leq 0.99$, in order to make SA run efficiently. Higher values of α gives SA a better chance to overcome local minimums but will on the other hand increase the computation time, since the cooling now takes longer. The similar situation applies to the choice of L_k. High values usually improves the quality of the results, but it takes longer to compute.

There are no rules for selecting α and L_k, so the best values must be found experimentally. Experience shows that for a fixed time several shorter runs return a better result than a single long run.

Another cooling strategy is the logarithmic cooling scheme formulated as:

$$T_k = \frac{\Gamma}{1 + log(k)} \quad ; \quad k = 1, 2, ... \tag{3.3}$$

where k is increased after the usual L_k steps and Γ is a sufficiently large constant.

Additional cooling strategies are described in [4] and [16], but generally the performance depends more upon the choice of parameters like α and L_k than the chosen strategy, and that is why no other strategies are covered in this report.

As stopping criteria there are three obvious possibilities. You can choose to stop when:

- the value of C(x) has not been improved more than ϵ_1 the last K_1 steps.
- the percentage of accepted functions is below ϵ_2 for the last K_2 steps.
- the total number of iterations has reached K_3.

The best values of ϵ (usually between 0 and 5 %), and K should be found experimentally. Alternatively the above criteria can be combined. The first two stopping rules both correspond to the "frozen" situation where a local minimum has been reached and the temperature is so low, that it cannot be escaped. The last one is the easiest to implement, but you have to be sure that the temperature has been lowered so much that the "frozen" state has occurred. Another stopping rule much equal to the latter is to stop the computation after a fixed time, which could be the time available to do the computation.

3.3.2 Proof of convergence

Some interesting theoretical results about SA has been achieved. The sequence of accepted solutions can be viewed as an inhomogeneous Markov chain, since the probability of moving to another solution, state y, only depends upon the current solution, state x, and the transition probability matrix P_T depends upon the current temperature.

As shown in [12] and [13] Hajek has used the Markov theory to prove that SA will with a probability of 1 end in the global minimum of S if the following conditions are met:

- The neighbourhood is symmetric, e.g. $y \in N(x) \iff x \in N(y)$.
- The solution space, S, is coherent, so every solution can be reached within a finite number of iterations.
- The neighbourhood is weakly reversible. That is for all x, y, T it is possible to reach y from x at the temperature, T, if and only if x can be reached from y at the same temperature.
- The temperature is cooled using the logarithmic cooling scheme (3.3).

But one thing is theory – another thing is reality. In order to ensure the convergence an infinite number of iterations has to be made, since the temperature never actual will reach zero. So the theory is of less interest when looking at practical problems.

3.3.3 Variations and improvements

The SA algorithm can be improved by other means than changing parameters. Another possibility is changing the neighbourhood generating function, so another neighbourhood are defined. An unfavorable designed neighbourhood can severe degrade the performance of the algorithm.

Some computation tricks can be used to improve performance too. Random numbers can be calculated from beforehand, being stored in a vector containing for instance 10.000 numbers. The access of a number in a array is generally much faster than calculating a new random number based upon the current.

On computers the *exp()* function calls are usually slow. Replacing this with an approximation can save much computation time. According to [13] about $\frac{1}{3}$ of the time can be saved when solving the Graph Partitioning problem if using a precomputed table of exponentials. Another possibility, instead of using the *exp()* function in the acceptance function, is to use the first terms of the Taylor expansion of the exponential function around zero, which also on most systems will take less computation time and ensure faster convergence at lower temperatures. When $P(x)$ denotes the probability of accepting solution x and ΔC is the change in the objective function value, which comes from accepting x, the standard acceptance function can be written as (3.4) and the Taylor acceptance function as (3.5).

$$P(x) = \begin{cases} 1 & \text{, for } \Delta C < 0 \\ exp(\frac{-\Delta C}{T_k}) & \text{, for } \Delta C \geq 0 \end{cases} \tag{3.4}$$

$$P(x) = \begin{cases} 1 & \text{, for } \Delta C < 0 \\ max(0, 1 - \frac{-\Delta C}{T_k}) & \text{, for } \Delta C \geq 0 \end{cases} \tag{3.5}$$

In [3] the use of the Taylor acceptance function proved to be superior to the implementation of the standard acceptance function when assigning channels in cellular mobile telecommunication systems.

Several variations of SA also exists. Some of them, which might be superior to SA in some cases, are:

- *Locally optimized SA*. In this the best solution of a randomly selected subneighbourhood is picked instead of just one random neighbour of the total neighbourhood. This can improve both the computation time and the quality of the solutions as shown on page 25 in [16].
- The *Threshold Accepting* method. Here a neighbour x is accepted if $C(x)$ is lower than the *threshold parameter*. The threshold is decreased with time making more and more worse solutions prohibited. It can be studied in [6].

3.4 Tabu Search

Tabu Search (TS) is another of the best known general heuristic methods. Compared to the SA method it is quite young and thus less studies in TS have been made, though it is by many people considered superior to SA.

TS can like SA be seen as a improved LS algorithm, but unlike SA it is build upon the "Steepest Decent" strategy. TS utilize flexible memory to remember the previous steps taken (they will be referred to as *tabu*) and will choose other steps in order to exploit new parts of the solution space by taking advantage of history. It is important to note that the same step does not have to be between the same two solutions. If solution A from the TSP example back in figure 3.4 is a result of exchanging the number in the succession city 1 and 2 are visited and the next step is to solution C, where city 2 and 5 are exchanged in the sequence, then the step where city 1 and 2 are exchange will not bring the situation back to the solution prior to A. But by remembering some information about the previous steps, TS can make sure that the algorithm does not switch between the same two solutions, and if implemented properly assure that no cycle between the same solutions occur.

In figure 3.7 is a pseudocode of TS shown. The notation is unchanged from before, except for the neighbourhood, which now is denoted $N(x,k)$, since it now also describes, which solutions at time k which is tabu.

```
procedure Tabu Search
begin
  choose initial solution, x ∈ S
  x* := x
  C* := C(x)
  k := 0
  choose V ⊆ N(x,k)
  y* := min{C(y) | y ∈ V}
  while not stop do
    x := y*
    if C(x) ≤ C* then
      x* := x
      C* := C(x*)
    endif
    k := k + 1
    choose V ⊆ N(x,k)
    y* := min{C(y) | y ∈ V}
  end
end
```

Figure 3.7: Pseudo-code of a Tabu Search procedure

A TS algorithm is not always stochastic as SA. If all the neighbourhood of each solution is searched, corresponding the situation $V = N(x,k)$ as in the Steepest Descent strategy, where the best possible solution is chosen, the algorithm is deterministic. But as mentioned in the TSP example, the number of neighbours can be so huge, that calculating the cost of all of them will be unpractical. Instead a smaller part of the entire neighbourhood should be searched as in the Locally optimized SA version of SA. Which part of the neighbourhood to be searched can either be chosen using some specific rules or random selection – making the algorithm deterministic respective stochastic.

3.4.1 Implementation

Like SA, it is quite easy to implement TS, if you already have implemented a LS algorithm. The procedure for calculating C(x) is unchanged and the one for finding $N(x)$ is easily altered to include the *tabu*-classified elements.

The new, and the hardest part, is the implementation of memory. At least short term memory of the steps taken most recent, called the *recency memory*, should be implemented, since this is very good for helping overcome local minimum points. In addition a more longer term memory can be implemented, remembering the number of times each step has been taken. This is usually denoted the *frequency memory* and can be used to prevent the same steps from being chosen over and over again, as shown in the example below.

Example 3.3 (The Traveling Salesman Problem - continued) *How TS and especially the memory can be implemented is best done by returning to the TSP example from section 2. The solution space was the number of permutations of 5 numbers, and a neighbour to a solution was the exchange of two cities in the succession the cities were visited in. A common implementation of memory when using a neighbourhood definition as the one from Example 3.1, is using a* $n \times n$ *matrix. In this example* $n = 5$ *for reasons that should become clear later on.*

In figure 3.8 the values assigned to some of the data structures of the TS algorithm during two iterations are shown. The values will be explained below.

The first part is the memory matrix. The upper triangular matrix of this is the recency memory. For iteration 10 it can be seen that the steps, which are tabu during this iteration, are: Exchanging city 1 and 2 is tabu the next 3 iterations, exchanging 2 and 5 is tabu the next 2 iterations, etc. The

Iteration 10

Memory:

Recency

	1	2	3	4	5
1		3			
2	1				2
3				1	
4	2	1	3		
5		2			

Frequency

Current solution:

City	1	2	3	4	5
Seq.	1	3	2	4	5

Objective value: 16

Best Objective value: 15

Neighbourhood:

Swap	Obj. Value	
2 - 5	12	T
2 - 4	17	
3 - 5	18	

Iteration 11

Memory:

Recency

	1	2	3	4	5
1		2			
2	1				3
3					
4	2	1	3		
5		3			

Frequency

Current solution:

City	1	2	3	4	5
Seq.	1	5	2	4	3

Objective value: 12

Best Objective value: 12

Neighbourhood:

Swap	Obj. Value
1 - 3	15
2 - 4	15
3 - 4	16

Figure 3.8: The matrix-implementation of Tabu Search Memory

frequency memory is stored in the lower triangular matrix. For iteration 10 it can be seen that city 1 and 2 has been exchanged once, city 1 and 4 twice, etc. for a total of 9 exchanges (since the current iteration are number 10).

The current solution, corresponding to solution A of figure 3.4, is stored as described in example 3.1, with city 0 left out, since it cannot be exchanged with the others. The objective function value of this solution is 16, but as it can be noted, a better solution, with a value of 15, has been found earlier (this solution is stored too, but is not pictured here).

A neighbourhood of three neighbours has been generated. Neighbour 1 is exchanging city 2 and 5 resulting in an objective function value of 12. The T *denotes, that this step is Tabu, as it can be seen in the recency memory. The other two neighbours is swapping city 2 with 4 and city 3 with 5 resulting in an objective value of 17 respective 18.*

Which of the neighbours should be chosen? If a step is tabu, it cannot be chosen unless a special condition, called an aspiration criterion, *occurs. Some common criteria will be discussed below. The number of iterations a step is tabu, which will be denoted the* tabulength, *can be constant like above, where a step is tabu for three iterations. Another option is to let the time vary between two bounds, either randomly or systematically. The latter dynamic form will normally prevent the algorithm from cycling between the same solutions.*

The most common aspiration criterion is to accept a tabu step if it results in a better solution than the best found until then. Another case where a tabu step has to be chosen, is when all the steps in the neighbourhood are tabu. Then the step with the oldest tabu-restriction should be chosen. Other useful criteria can be found in [8].

Because of the aspiration criterion mentioned it is chosen to swap city 2 and 5 resulting in solution C from figure 3.4. In iteration 11 the data structures have been updated as shown. Note than only two steps are tabu now, since a tabu step was chosen last. But since none of the solutions in the neighbourhood are tabu this time, there will be three tabu steps again from iteration 12.

Frequency memory is usually not as strict as the recency memory. If implemented at all it is normally only used when no solutions are found with a better objective function value than the current best. To the objective function values of the neigbours a penalty, depending upon the frequency of the steps until now, are added. E.g the penalty in this example could correspond to the frequency of the steps. The adjusted objective values for the three neighbours are then 15, 16, and 19, since swapping 3 and 4 has been done 3 times until now. The neighbour with the smallest adjusted value should be chosen, since it most common will take the algorithm to the least explored part of the solution space.

As stopping criterion you can continue for a fixed number of iterations or until no change of x^* or c^* has occurred for a number of iterations.

□

Because of the memory TS will often result in very good solutions for most problems. It may be a bit more complicated to implement than SA, but is still quite easy to implement, if procedures for neighbourhood generation and calculation of objective cost either exist or easily can be written.

3.5 Summary

After this brief introduction to some of the metaheuristic methods of solution, this section will sum up the most important general details.

First of all, when you are to decide a method to use solving a specific problem, you should think about whether a neighbourhood might be easily defined or not. An unclear neighbourhood structure will often lead to a time-consuming generating function, which can severe degrade the general performance. Therefore problems with no clear neighbourhood structure might better be solved using Branch and Bound or another non-metaheuristic method.

If deciding to use a metaheuristic, the following remarks should be reminded:

When defining the neighbourhood it should be noted that a neighbour should generally represent a small change compared to the current solution. In addition every part of the solution space must be reachable within a finite number of iterations from the current solution. In some cases a part of the solution space can be omitted, since it does not contain the optimal solution. But for the part included, the rules above should hold.

Concerning the implementation, the computation speed of the evaluation function and neighbourhood generation function is very important, since they are called many times during a single run.

The stopping criteria and other parameters like the tabulength, frequency penalty, T_0, α, etc. should be found experimentally, since no universal rules of choosing those values exists.

With these things in mind, the chance of getting good results from using metaheuristics should be fine.

Chapter 4

Design and implementation

It was decided to implement both LS, SA, and TS algorithms solving the allocation problem in order to be able to compare the methods. It was chosen to implement all the algorithms in the Simula programming language. This decision was taken for three reasons. First of all, it is easy to learn, especially for people knowing Pascal. Second, since the DDRE research team had chosen to implement their algorithm in Simula, it was possible to use their test scenarios and graphical display program if using similar input/output file procedures. The company, Terma, which is undertaking the missile system upgrade, will later reprogram the DDRE algorithm in C++ for greater performance. At last by implementing the program in Simula, it should be possible in some degree to compare the computation time with the one used by the DDRE algorithm, since the same CPU-server and the same Simula compiler was to be used.

With that decision taken, the overall design of the algorithm was to be made. It will be described in section 1. Due to the complexity of the problem, it was chosen to split the optimization in two parts. In one part the allocation of targets to the HIPIR's is made. I.e. it is decided which target, which should be illuminated by which HIPIR, and the succession each HIPIR should illuminate the targets. The second part optimizes the start time of the engagements assigned to a HIPIR, making the engagements start at the most optimal time. Throughout this report the engagement start time denotes the time, where the firing buttom is pushed in order to launch a missile for an engagement.

It turned out that with the simplifications made, the allocation of a missile launcher to each engagement, could be done after the (near)-optimal solution had been found, so long the number of launchers with missiles was greater or equal to the number of HIPIR's, and the number of missiles available at least was equal to the number of targets. The implementation would be rather easy, and was omitted in order to have more time to improve the optimization parts.

The design choices of the two optimization parts are described in the following sections and will include how the implementation was done in order to fit the overall design. The heuristic implemented first was LS, since both SA and TS is build upon this. So the following sections will describe the implementation of LS, while the changes which was made in order to use the other two heuristics, will be described in section 4.4 and 4.5 respectively.

4.1 Overall design

In order to design the algorithms a top-down approach was used. First a pseudo-code shown in figure 4.1 of the mainprogram was made, defining some of the larger procedures needed, like *find_best* and *time_opt*. The process was repeated for the procedures, so at last only a small main program and a few large procedures each utilizing several small procedures remained. The main program will be explained below.

In the *initialization* procedure all constants, variables, and data structures are declared and assigned initial values. During this part the algorithm reads the input-files concerning the system status and the threat.

To store this information a class of data is defined for the targets, the HIPIR's, the objects, and the missile launchers, containing all the needed information about them. But the most important data structure is for storing the allocations of both the current solution and all the solutions of the generated neighbours. The basic is the *allocation*-class, which contains all the information about a single allocation, identified with the ID-number of the allocated target. An array of allocations is defined for each HIPIR-radar, containing the allocations of that particular HIPIR. An additional

```
begin
  initialization
  curr_x := make_init_solution
  unchanged := 0
  while unchanged < max do
    Y := find_neighbourhood(curr_x)
    (best_y,best_change) := find_best(Time_opt(Y))
    if best_change > 0 then
      curr_x := best_y
      curr_obj := curr_obj + best_change
      unchanged := 0
    else
      unchanged := unchanged + 1
    endif
  end
end
```

Figure 4.1: Pseudo-code of the Local Search main program

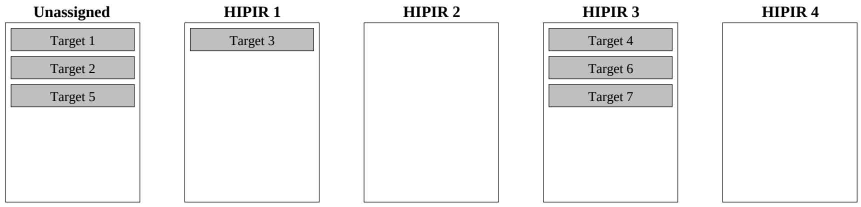


Figure 4.2: The data-structures where a solution is stored

array is defined containing all the non-allocated targets. A sample allocation plan is shown in figure 4.2. The system shown consist of four HIPIR's and seven targets. Three of the targets are unassigned; target 1, 2, and 5, while target 3 has been assigned to HIPIR 1 and HIPIR 3 should engage target 4, 6 and 7 in this succession.

At last in the initialization part the random number generator is initialized. For random number generation it was chosen to use the generator that came with Simula.

The *make_init_solution* procedure makes an initial assignment, which has been chosen always to be the zero-assignment, i.e. the assignment where all target are unassigned, and thus stored in the unassigned array. The stochastic nature of which neighbour to generate, as shown later, will make sure every possible allocation plan can be reached from this assignment. So there was no reason to start with some or all of the targets randomly assigned to the HIPIR's.

Next the variable *unchanged* is set to zero. This variable is used for the stopping criterion of the LS algorithm, as it can be seen in the while-statement. This loop is repeated until no improvement has been made for the last *max* iterations, where *max* is given as input to the program. The rest of the loop will be explained in the next section.

When running LS multiple times within the available time span another loop is added outside the loop described above, running this continuously until the time limit has been reached. The best solution obtained from the inner loop is stored in memory, again using the allocation-class data structure.

Both an implementation of LS running a single time as in 4.1 and of LS running multiple times were made. The single time version was used as basis for the SA and TS implementation.

4.2 Searching for a better allocation

The search for a new allocation is done within the while-loop of figure 4.1. At first a part of the neighbours to the current solution is found. In the following the neighbourhood definition as well as the reason for using a part of the neighbours and not one or all of them is given.

4.2.1 Neighbourhood generation

A solution consists of a set of unassigned targets that may be the zero-set, $\emptyset$, as well as a sequence of targets for each HIPIR. The sequence denotes the succession, which the targets assigned to that HIPIR must be engaged in.

The neighbourhood of the current solution is defined as one of the following changes done to the existing assignment:

- An unassigned target can be put in any position in the sequence of a HIPIR (case 1 of figure 4.3).
- An assigned target of a HIPIR can be put in any position in the sequence of any other HIPIR (case 2).
- An assigned target can be put in any other position in the sequence of targets assigned to the same HIPIR (case 3).
- One of the targets assigned to a single HIPIR can be made unassigned (case 4).

One of the described cases is randomly chosen, and all possible neighbours of this are generated and evaluated in an iteration. The number of neighbours can with this definition vary from one to the total number of targets. It depends upon the number of targets assigned to the receiving HIPIR as it can be seen from the definition. For every neighbour a temporary array is used to store the assignment.

Many neighbours exists even for a small system like 4.2. So if all the neighbours of a solution were to be tested every iteration, this part of the algorithm would take too long, since for every neighbour tested, a time optimization run, as described in the next section, has to be done. Instead it was chosen to test a smaller part, which should be an improvement over just evaluation of one neighbour, corresponding to Locally Optimized SA, one of the versions of SA mentioned in the previous chapter.

With this neighbourhood definition a neighbour is a small change to the existing solution and it assures that all possible solutions can be reached from any other solution, which were the basic properties of a neighbourhood.

4.2.2 The objective function

The objective function value, C(x), was in chapter 2 defined as the sum of the actual military value of a target times the probability of shooting that specific target down. For the evaluation of the neighbours ΔC of each is calculated.

Since both the actual military value and the probability of shooting a target down depends upon the time the missile hits (or misses) the target, you have to pick the most *optimal time* of each engagement to start. This is done by the *time_opt* procedure described in the next section, where the *optimal time* also will be defined.

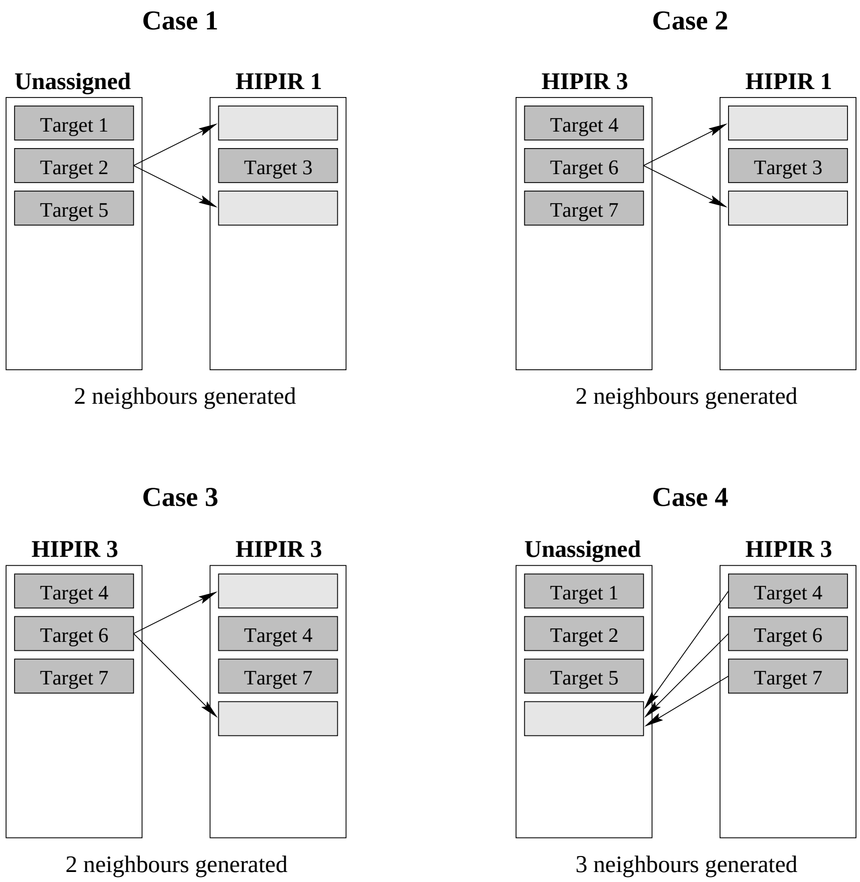


Figure 4.3: Four sample neighbour generations from the sample system shown in figure 4.2. The number of neighbours generated by each case is shown below it

Another procedure, *find_best*, calculates the change in C(x) that each neighbour results in, and returns the allocation plan and the ΔC for the one with the highest value. If this ΔC is positive the neighbour is accepted as the new current solution, since this is a maximization problem.

4.3 Time optimization

In order to find the optimal start time of the engagements of a HIPIR, this procedure is used. Using a nontrivial neighbourhood, it features the Steepest Ascent method, since all possible neighbours are checked each iteration. A pseudo-code of the procedure can be seen in figure 4.4 .

```
procedure Time optimization
check_if_possible
if no_of_targets = 1 then
  update(assign)
else
  begin
    initialization
    change := true
    while change do
      change := false
      for i := 1 to no_of_targets do
        imp1 := move_forward_check(i)
        imp2 := move_backward_check(i)
        improvement := max(imp1,imp2)
        if improvement > 0 then
          update(assign)
          change := true
        endif
      end
    end
  end
```

Figure 4.4: Pseudo-code of the time optimization procedure

When the procedure is called, it has to make sure it is possible within the allowed planning time to engage all the assigned targets. The *check_if_possible* procedure will do this as well as put the assignments in a tight succession, which is the succession, where each engagement starts as soon as the previous engagement is finished. This solution is the initial solution of the algorithm, but since it is not necessary the optimal time allocation, it has to be tested whether some targets could be engaged at more optimal times improving C(x) this way.

4.3.1 Optimal time - critical line

The *optimal time* for an engagement refers normally the time where an engagement should be started, in order to have the highest probability of shooting down the target. An exception is, if a target has crossed the critical line before it is hit by a missile fired at the optimal time using the definition above. In this case the optimal time instead denotes the engagement start time to be used, so the target is hit as it reaches the critical line. This is done in order make the algorithm engage the targets before they cross the critical line if possible, which was one of the properties the algorithm should fulfil.

4.3.2 Neighbourhood generation

When only one target is allocated to a HIPIR, it is easy. A precalculated table contains the optimal start time of an engagement for each target-radar combination. So this start time is used unless it is later than the planning time allows. In this case the target is engaged at the latest time possible instead.

With more than one target assigned it becomes more complicated, since the engagements can block for each other, so one or more of them cannot be engaged at the optimal time. Three points of time are defined for each target. The earliest possible, EST, the optimal, OST, and the latest possible engagement start time, LST. The first and the last of those can change as the engagement start time of either the previous or the next engagement is changed, but they will never cross the boundaries of the possible intercept period. It should be noted that the optimal engagement start time does not necessary lie between the other two, as seen for the third engagement of figure 4.5. In this figure engagement 2 and 3 block for each other.

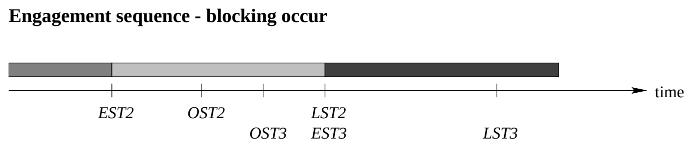


Figure 4.5: A sequence of engagements where a blocking occurs

When such a blocking occurs between two engagements, a boolean variable will be set for use in the moves explained below.

The main part of the *time_opt* procedure is the *for*-loop, which for every assigned target, will calculate the change in C(x) for each of the possible, following moves:

Forward move: The engagement is delayed, if the engagement is not blocked by the next and either the engagement start time is less than the optimal of that target-radar combination or it is blocking for the previous engagement to move forward.

Backward move: The engagement is started earlier, if the engagement is not blocked by the previous and either the current start time is greater than the optimal of that target-radar combination or it is blocking for the next engagement to move backward.

The most promising "move" can now be found, as the one with the greatest positive change to C(x). Since all possible neighbours are tested, this is a Steepest Ascent algorithm. So until no improvement can be found for any of the targets, the loop above is repeated.

This neighbourhood definition should both make a neighbour a small change compared to the existing solution as well as ensuring all "optimal" solutions can be reached within a finite number of iteration from any other solution. The notation "optimal" is used, since not all possible time displacements are defined in this implementation. But the ones defined should, with the assumption described below, include the optimal solution.

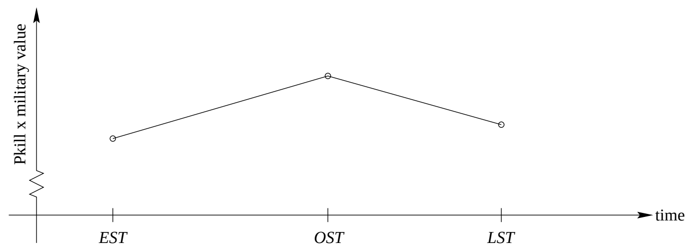


Figure 4.6: For the implementation it has been assumed, that the increase or decrease rates between the points are constant as shown

The assumption that has been made is, that the increase or decrease rate of the benefit value of each target (i.e. the Pkill times the military value) is constant between EST, OST, and LST as shown in figure 4.6.

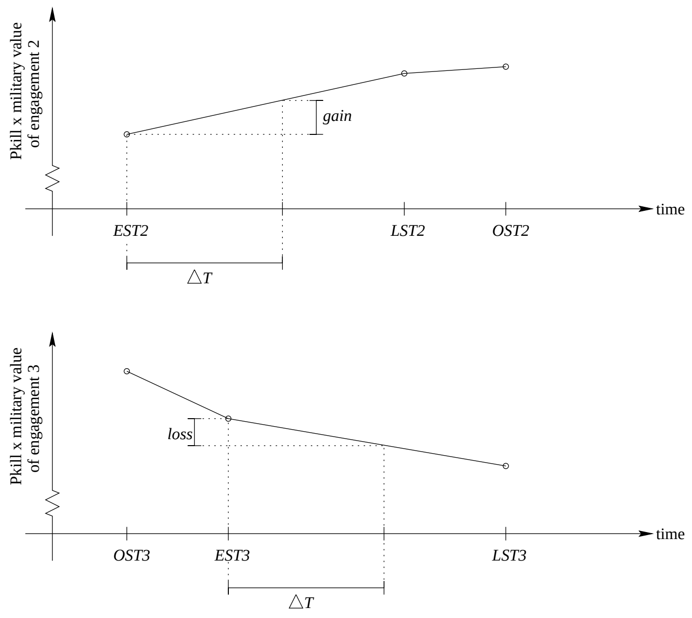


Figure 4.7: The upper plot shows the benefit of engagement 2 of figure 4.5 while the lower is the benefit of engagement 3

With the mentioned assumption the optimal solution is still contained in the solution space. This can be seen this way. If no blockings occur, the optimal solution is included, since all OST points are included. If a blocking occurs as the one in figure 4.5, both of the start times are tried to be moved ΔT forward as shown in figure 4.7. If the gain in benefit of engagement 2 is greater than the loss of benefit of engagement 3, ΔT should be as great as possible, which is the greater of the intervals $[EST2, LST2]$ and $[EST3, LST3]$. If the gain is less than the loss, ΔT should be zero, and if equal ΔT could be any of the two values. Since all the EST and LST points are included in the solution space, the optimal solution will be contained in this, because of the assumption of constant increase/decrease rates.

During the testing this procedure turned out to be very good in finding the global optimal solution since only in rare occasions an assignment blocked for another, which could have improved the solution as in figure 4.8.

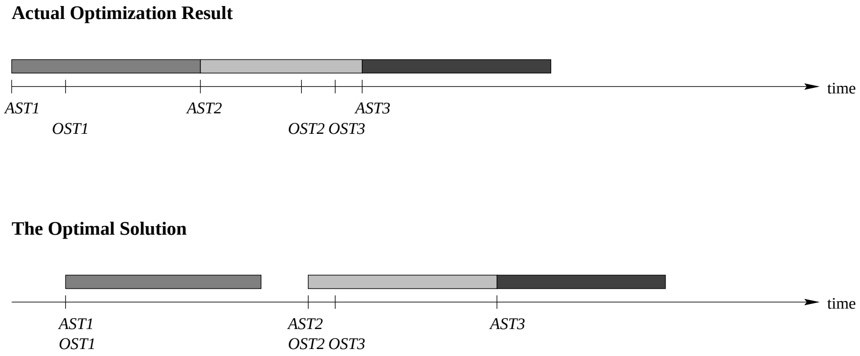


Figure 4.8: The latest of three assignments blocks for a better solution. AST and OST denotes respectively the actual start time and the optimal start time of an engagement

After the time optimization the partial objective function value of the used HIPIR is stored in memory and used by the *find_best* procedure to find the best of the generated neighbours.

4.4 Implementing Simulated Annealing

Simulated Annealing was implemented with only minor changes done to the code from the LS implementation. It was chosen only to apply SA to the allocation search procedure and not the time optimization procedure because the latter as described in the last section, returned acceptable solutions just using LS. So since the *time_opt* procedure is called many times during a single run, the total run time would be severe degraded is using SA, with only a very limited gain in the expected value of C(x). How the procedure can be improved in a SA-like style, will be discussed in chapter 6.

A pseudo-code of the Simulated Annealing main program can be seen in figure 4.9.

As it can be noted the few things needed, was the insertion of some new variables and constants taking care of the cooling strategy. Then the actual cooling strategy and the acceptance criterion was to be implemented. In addition the stopping criterion was expanded as covered later.

```
begin
  initialization
  (T_0, α, L_k) := read_cooling_strategy
  curr_x := make_init_solution
  unchanged := 0
  k := 0
  T_k := T_0
  while (unchanged < max_unchanged and k < max_total) do
    counter := 0
    repeat
      Y := find_neighbourhood(curr_x)
      (best_y,best_change) := find_best(Time_opt(Y))
      if best_change > 0 then
        curr_x := best_y
        curr_obj := curr_obj + best_change
        unchanged := 0
      elseif (exp(best_change/T_k) > random[0, 1] and T_k > 1 × 10^-10) then
        curr_x := best_y
        curr_obj := curr_obj + best_change
        unchanged := 0
      else
        unchanged := unchanged + 1
      endif
      counter := counter + 1
      k := k + 1
    until counter ≥ L_k
    if T_k ≥ 1 × 10^-10 then
      T_k := α × T_k
    endif
  end
end
```

Figure 4.9: Pseudo-code of the Simulated Annealing main program

4.4.1 Design choices

When implementing SA some choices concerning the design has to be made as noted in the SA section of chapter 3. The choices made for this implementation are described below:

Cooling strategy Since the probability of getting better results with more advanced cooling strategies than the traditional geometric cooling scheme was very little, no other strategies than this were implemented. But if additional time had been available, it could have been interesting to try nevertheless.

Homogeneous vs. Inhomogeneous It was chosen to implement the homogeneous version of SA, since this also could be used to test the inhomogeneous version by setting $L_k = 1$.

Acceptance criterion The implementation of this functionality includes that the acceptance probability is set to zero when the temperature has been lowered to a point close to zero. It was not implemented this way from the beginning, but it happened during the testing, that the algorithm began to cycle between a few solutions, where a inferior solution repeatedly were accepted, until the temperature fell below 1×10^{-32}. This resulted in a 100 times increase in the runtime compared to the normal of that particular scenario. The implementation shown will prevent such behavior and in addition make the algorithm converge faster in the end, since all inferior moves then are prohibited.

Stopping criterion The stopping criterion was expanded to include a maximum number of iterations allowed in addition to the number of iterations made without changes. It was mainly used for debugging. When the dependence of the cooling parameters had been assessed, a time limit was implemented as in the multiple run version of LS.

4.5 Implementing Tabu Search

When TS was to be implemented, it was discovered, that the neighbourhood definition used in the LS and SA implementation was unsuitable for use in TS. The function generating the neighbours creates between one and the number of targets neighbours, as it can be seen from the description. It depends upon the current solution and the random numbers drawn. If

```
begin
  initialization
  curr_x := make_init_solution
  best_x := curr_x
  best_obj := Time_opt(best_x)
  unchanged := 0
  k := 0
  while (unchanged < max_unchanged and k < max_total) do
    Y := find_neighbourhood(curr_x)
    (curr_x,curr_obj,tabu) := find_best(Time_opt(Y), tabu)
    if best_obj < curr_obj then
      best_x := curr_x
      best_obj := curr_obj
      unchanged := 0
    else
      unchanged := unchanged + 1
    endif
    k := k + 1
  end
end
```

Figure 4.10: Pseudo-code of the Tabu Search main program

only one neighbour is created and this one is tabu, this one will then be selected no matter how long it has been tabu and how much C(x) will be changed.

Since this makes the use of the memory almost useless, a need for a new neighbourhood definition and a function implementing this was recognized. Therefor more changes to the LS code was to be done than usually needed. A pseudo-code of the TS program with the new neighbourhood implementation is shown in figure 4.10.

The function, which evaluates the generated neighbours, now takes the tabu memory as parameter, returning the best of the neighbours (and the objective function value of this) depending on the current recency and frequency memory. Then the current solution are set to this and the tabu memory is updated according to this choice. The best value obtained at this point is updated if the value of C(x) of the new solution is an improvement over this.

4.5.1 The new neighbourhood

For the new implementation it was decided to create a function, which generates a fixed number of neighbours each iteration. The number to be generated can be user-specified, but has an upper limit depending upon the number of HIPIR's and targets.

Once more a neighbour is defined as the move of one target and not as an exchange of targets. Now if n neighbours are to be generated, the first one will be randomly chosen, while the rest generated will be the $n - 1$ next neighbours in a defined succession, which will be described later.

When picking the first neighbour a random HIPIR (with at least one target assigned to it) to move from is selected. From the array of this a random target to move is drawn. Note that all HIPIR's has an ID number, where 0 denotes the array containing the unassigned targets. If moving from HIPIR 0 (the unassigned) the HIPIR to move to is randomly selected between those with an ID higher than 0. Else the algorithm just pick a random one of all the arrays to decide where to move. If the HIPIR moving from is the same as the one moving to, the neighbours generated are just changes in the succession the HIPIR engages the targets. When the two HIPIR's ID numbers are different, the neighbours generated in this case, are the possible placements of the moving target in the succession of targets assigned to the receiving HIPIR.

The following steps define the succession of the neighbours, once the first one has been selected. The steps are taken until the wanted number of neighbours has been generated. Because of the number of special cases, it might look complicated, but the example below should help to clear out any confusion.

- The selected target is moved to the HIPIR with the next higher ID number, starting over from 0, when the highest number has been reached. If the target is moved from HIPIR 0 (unassigned), the unassigned number is excluded.
- If it has been tried to move the target to all possible HIPIR's, the procedure switches to the next target (in the succession) of the HIPIR moving from. If the last target of that HIPIR has been reached, the HIPIR to move from is switched to the next higher number, again starting over from 0 if the highest ID number has been reached.

Note that each time the HIPIR's or target ID is changed, the number of new neighbours, which can be created, will vary depending upon the current solution.

Example 4.1 (The Tabu Search neighbourhood) *Consider the situation where the solution pictured in figure 4.2 are the current solution and 9 neighbours are to be generated. The first neighbour is chosen the following way:*

- *One of the nonempty HIPIR arrays is chosen to move from – in this case HIPIR 3.*
- *Of the three targets assigned to HIPIR 3, the second target is randomly selected, i.e. the target with ID 6 is to be moved.*
- *Now a HIPIR to move to is chosen. In this case any would do, and HIPIR 2 is randomly picked.*

This gives neighbour 1 pictured in figure 4.11. If the steps described above are used to pick the next n-1 *neighbours, the neighbours numbered from 2 to 9 in the figure are generated. In neighbour 1 target 6 is moved to HIPIR 2. Since the array of HIPIR 2 is empty, this can only be done in one way. So when generating neighbour 2, the target now has to be moved to the next HIPIR array, which in this case is HIPIR 3 – the one moving from. Since there are more targets assigned to HIPIR 3 than target 6, it can be put in other places in the succession of engagements resulting in neighbour 2 and 3. When coming to the array of unassigned targets, there is no defined succession of those targets. So target 6 is just added to the array making neighbour 5.*

Before all 9 neighbours have been generated, target 6 has been moved to all possible positions. So the algorithm switches to the next target in the succession, which is the next target – if any – assigned to the same HIPIR. In this example it is target 7. So in neighbour 8 and 9 it is target 7 to be moved.

If even more neighbours were generated, target 7 would at a point have been moved to all possible positions too. So the next target in the succession should be used for generating the next neighbours. In this example it is target 1 of the unassigned array, since there is no more targets assigned to HIPIR 3 and no one is assigned to HIPIR 4.

□

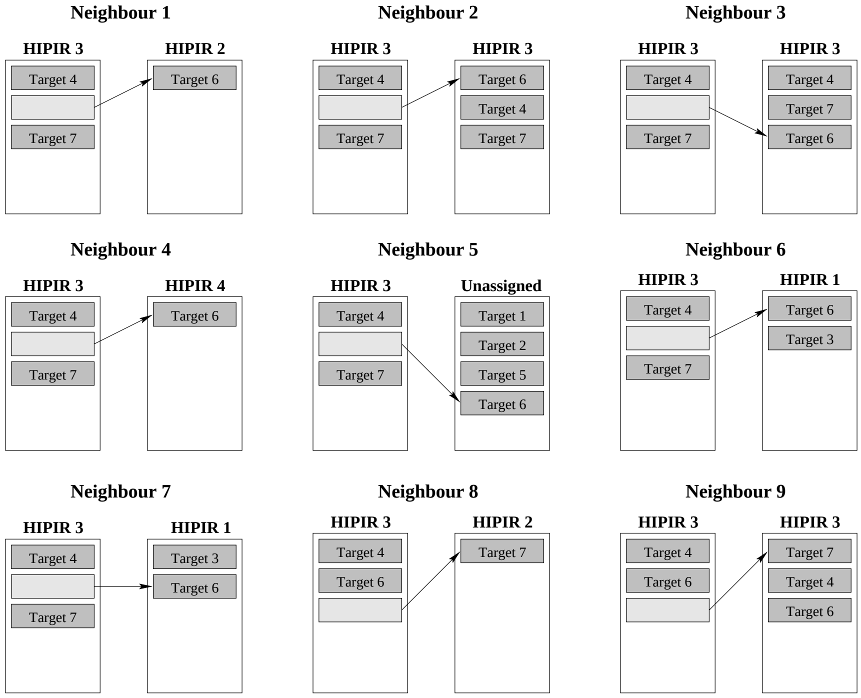


Figure 4.11: Neighbours generated in Tabu Search

4.5.2 Implementing memory

As the tabu element either just the moving target or a combination of the receiving HIPIR and the moving target can be used. E.g. if target 6 just has been moved from HIPIR 3 to HIPIR 2, either the moving of target 6 (no matter where-to) or the moving of target 6 (no matter from which HIPIR) to HIPIR 2 is declared as tabu for the next $tabulength$ iterations .

Both of the above mentioned tabu structures have been implemented in order to see any differences in performance. The frequency memory is implemented too and has to fit the tabu structure used. So in the first case the penalty added to C(x) is a certain percent of the times the target has been moved, while in the other case some percent of the times the target has been moved to that specific HIPIR. The percent is user-specified in both implementations.

Example 4.2 (The Tabu Search frequency memory) *The frequency memory, which role is to diverge the search into new areas, is implemented as a penalty to the objective function. The penalty added to the objective function is an userspecified percentage of the number of times the move to the investigated neighbour already has been taken. If 10 % has been specified and the number of times target 6 has been moved is 10, $C(x)$ of the neighbour describing this move, will be lowered by 1. If target 6 had been moved 100 times the penalty would be 10 instead, making the less taken moves more interesting.*

□

In addition to the frequency penalty both the tabu length, the number of neighbours generated, and the parameters of the stopping criterion (which was implemented as in LS and SA) are user-specified, making it possible and easy to test the dependence of those parameters. In this implementation the tabu length was fixed for the duration of a computation run.

Chapter 5

Results

During the implementation the algorithms were tested to ensure they worked properly. It was mainly done by running each procedure several times with different input and then see if they returned the expected output. The build-in debugger in the Simula compiler turned out to be very useful during this process. At last the whole program was tested with generated input to see if it behaved as it was expected to.

When the implementations of the algorithms were trusted to give the right results, a test program was written to assess the influence of different parameter choices. This test program will be described in section 1. In section 2 the scenarios representing different system data and threats, which were used for the tests, are outlined. The results from the LS, SA, and TS implementations will be treated in section 3, 4, and 5, respectively, followed by a comparison of the methods in section 6. The algorithms were tested with the normal stop criteria initially, in order to easier see the performance of the algorithms compared to the time used. This information was used to find the optimal parameter choices to use, when the algorithms were to run within the 3 seconds time limit, which was used when comparing the algorithms.

5.1 The test program

The stochastic nature of the algorithms make them in general return different solutions to the same problem. In order to find the expected value of C(x) of the algorithms for a given scenario, an automated program was written. This ran the desired algorithm with the same parameters, except for the random number seed, 100 times calculating the sample mean value and variance of C(x) and the computation time used. The results pictured in this chapter are unless otherwise stated a sample mean value of such a 100-runs batch.

To get an idea of the computation time the algorithms return both the CPU-time used, the number of times *time_opt* is called, and the total number of iterations. The CPU-time turned out to vary up to 5% for the same task, but since the test program calculates the mean value of 100 single runs, most of this variation should be smoothed out. So the mean CPU-time will be used to measure the computation time used. All the computations was done on a Digital™ VAXstation 4000-60 workstation.

The DDRE team is of the opinion, that a 100 times improvement in the computation time can be gained, when the algorithms are reprogrammed i C++ and run on a modern workstation. This will allow an algorithm to run for 300 seconds in the tests, and still be within the time limit of 3 seconds.

5.2 Test scenarios

The algorithms were tested on various scenarios, which had a different number of targets and HIPIR's. A scenario with two HIPIR's and seven targets will in this report be denoted as a 2-7 scenario, while a scenario with 8 HIPIR's and 13 targets thus is denoted an 8-13 scenario. The density of a scenario describes the number of targets compared to the number of HIPIR's. A 2-13 scenario is considered dense while an 8-7 scenario is sparse. The size of a scenario depends both on the number of HIPIR's and the number of targets. A scenario will in most cases be considered small, if it has less than ten targets or four HIPIR's.

Unless otherwise specified the scenarios used throughout this report are based upon a single 8-13 scenario. In this the HIPIR's are positioned two and two in four different locations with overlapping intercept areas, protecting a main asset (e.g. an airbase) in the middle. 13 targets are incoming, all being jet fighters. The time to plan ahead was for all scenarios set to 200 seconds.

The scenarios with 2 or 4 HIPIR's instead are the same with 3 respectively 2 pairs of the HIPIR's deleted. In the scenarios with only 7 targets, the last 6 of the original 13 targets are removed.

For none of the scenarios the global optimal value of C(x) is known for sure. So the mean values obtained will be compared with the best value of C(x) found for that particular scenario, which should be very close to, if not the global optimum. For the 2-13 and the 8-13 scenario, which will be used most widely, the best values found were 247.73 and 482.67 respectively and the "optimal" value will in this chapter, when mentioned, refer to one of these values.

5.3 Local Search results

The main question in case of LS was to find the best stopping criterion. For this, the LS version running a single search run, denoted LS-single, was used. As implemented this algorithm will stop, when it has not found a better solution in the last uc iterations. The test program was run with different values of uc and returned for the 8-13 scenario the values pictured in figure 5.1. It can be seen that the increase of the objective function value is insignificant for uc values greater than 160.

Comparing the increase in the objective function value to the computation time used results in a similar picture, as seen in figure 5.2. No significant gain is made after some 18 seconds, which also was the mean CPU-time used when demanding 160 unchanged iterations as stopping criterion.

Depending upon the scenario used (six different sized scenarios were tested), the best value of uc turned out to be between 80 and 160 with the smaller values for the smaller scenarios, since their neighbourhood can be explored in less iterations. No clear dependence upon the scenario density was shown though. The results are shown in figure 5.3.

As described in chapter 4 two versions of LS was implemented. In the LS-single, which was used above, the algorithm, hoping to find an acceptable

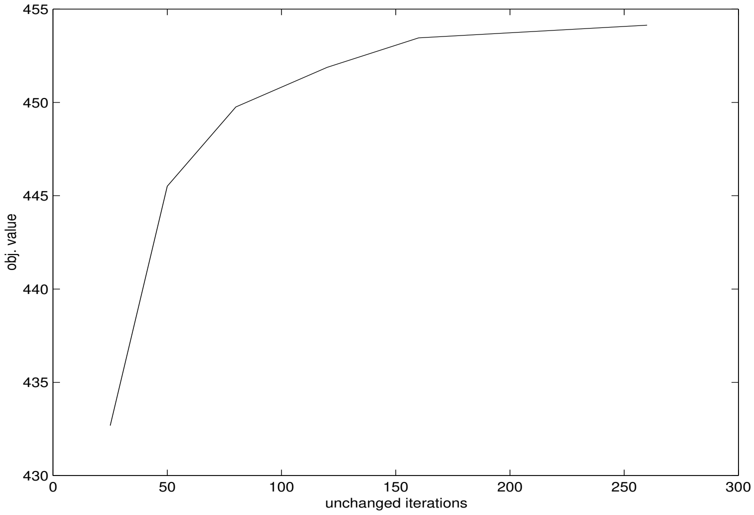


Figure 5.1: The objective function value returned compared to the number of unchanged iterations before stopping

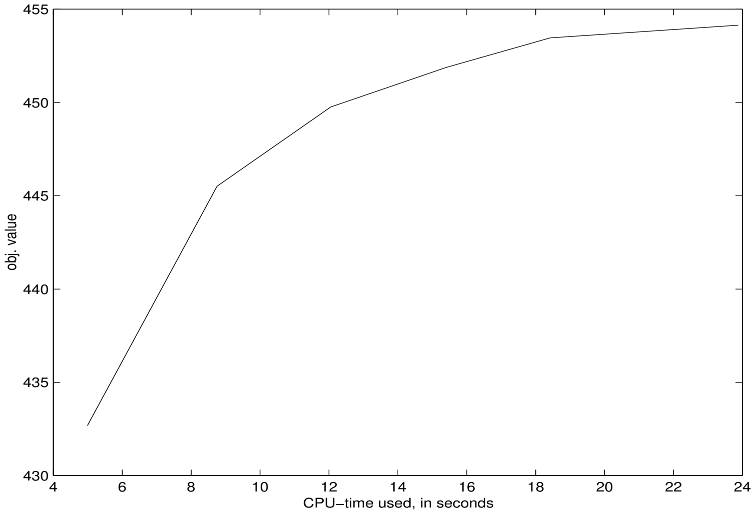


Figure 5.2: The objective function value returned compared to the computation time

solution, performs one individual LS run. The other version, denoted LS-multi in this report, runs LS repeatedly within the available time span, returning the best solution found in the end.

If the result was to be found using LS-single forgetting about any time limit, a *uc* value of 160 should ensure that (almost) no better solutions exists in the neighbourhood.

When adding the time limit, a better solution would be to let LS run for all 300 seconds. This might not give a better result, but at least you have not missed a better solution you might have found by continuing. So in fact the 300 seconds time limit works both ways. You may not exceed it and it is stupid not to use all the time available.

The optimal value of *uc* to use for the individual LS runs when running LS-multi for the 300 seconds, will be investigated later.

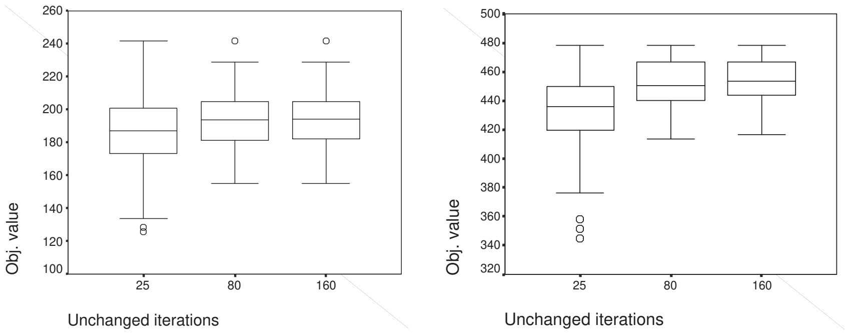
	Unchanged iterations				
Scenario	25	50	80	120	160
2-7	176.81 1.59	178.61 2.42	178.64 3.34	178.74 4.52	178.74 5.74
4-7	222.46 2.30	223.92 3.67	224.37 5.01	224.79 6.76	224.84 8.15
8-7	244.97 2.26	248.00 3.43	248.98 4.60	249.45 5.96	249.94 7.27
2-13	186.88 2.84	192.14 4.19	192.76 5.47	192.76 7.05	193.10 8.98
4-13	413.06 5.47	431.20 9.40	435.42 13.78	437.34 17.29	438.46 21.67
8-13	432.68 4.99	445.51 8.75	449.76 12.05	451.88 15.41	453.46 18.42

Figure 5.3: Table of the mean objective function value (upper value) and the mean CPU-time used (lower value) achieved for different scenario and *uc* combinations

From the table in figure 5.3 showing the results of the LS-single runs, it can be observed that with more HIPIR's the mean value of C(x) normally will increase, since the engagements can be spread out and thus be engaged more optimally. The CPU-time will increase until a certain number of HIPIR's has been added, because of the increasing number of combinations to test. But at a point it will become easier to find one of the local minima and the CPU-time will begin to drop slowly again, as more HIPIR's are added.

If more targets are added the CPU-time used will increase. As will the value of C(x) until a certain point where there is no more time for new engagements within the planning time. So C(x) will normally improve most in the sparse scenarios, since in those cases, new targets are easier added at good engagement times.

The boxplots in figure 5.4 picture the distribution of C(x) for some of the runs with some statistical data added in the table in figure 5.5. A boxplot gives a good picture of the distribution of several samples. The maximum and minimum values achieved by each are marked as the ends of the vertical lines, while the small circles drawn by the plotting program shows the outliers – values which statistically are to far from the others. The upper and lower edge of a box shows the 75 % respectively 25 % percentiles while the line through it is showing the median.



(a) 2-13 Scenario

(b) 8-13 Scenario

Figure 5.4: Boxplots of some LS runs

From the boxplots and the statistical table it can be seen that $uc = 25$ is far from enough for getting quality results, while the increase in quality from $uc = 80$ to $uc = 160$ is very limited, but anyway non-negligible in case of the larger scenarios. In general the variance of C(x) decreases for higher uc values, as you could expect.

Scenario	uc	mean	s. dev.	max	min	% best
2-13	25	186.88	21.42	241.63	125.67	24.56
	80	192.76	17.56	241.63	155.02	22.19
	160	193.10	17.46	241.63	155.02	22.05
8-13	25	432.68	24.36	478.52	344.56	10.36
	80	449.76	16.16	478.52	413.72	6.82
	160	453.46	15.08	478.52	416.47	6.05

Figure 5.5: Descriptive statistic values of the boxplot runs. The %-column shows how many percent the mean value is from the "optimal" value

When compared to the properties described in chapter 3 that is wanted for heuristics to fulfil, it can be seen that the mean values of C(x), especially for the dense 2-13 scenario, are far from the "optimal" values. So the property of returning acceptable solutions with a high probability will not hold – at least for this problem.

When running LS-multi for 300 seconds instead, the results are considerable better as shown in the table in figure 5.6. It can be seen, that the best results come from setting $uc = 50$. Each single LS-search takes less time than for $uc = 80$, so more searches can be made within the time limit, with almost as good solutions as for the higher uc-value. When using $uc = 25$ even more single LS-search are performed, but the lower quality of the results make this choice inferior nevertheless.

For $uc = 50$ the sample mean and the minimum value of C(x) solving the 8-13 scenario are within respectively 1.90 % and 3.82 % of the "optimal" value. For the 2-13 scenario the similar values are 7.00 % and 11.65 %. So compared to LS-single, LS-multi does much better fulfil the wanted properties of a heuristic.

Scenario	uc	mean	s. dev.	max	min	% best
2-13	25	229.76	6.35	243.02	220.35	7.25
	50	230.40	6.45	245.44	218.87	7.00
	80	228.93	7.08	246.30	219.30	7.59
8-13	25	471.36	4.53	481.43	461.87	2.34
	50	473.51	3.89	482.62	464.21	1.90
	80	473.50	3.29	478.52	416.47	1.90

Figure 5.6: Descriptive statistics of the LS-multi results

5.4 Simulated Annealing results

When using SA the same stopping criterion, as found to work best while running LS-single without time limit, i.e. $uc = 160$, could be used. This value should ensure, that the algorithm most likely has reached a frozen state. So unless otherwise noted $uc = 160$ will be used below.

The optimal cooling parameters were then to be found. Since the used Pkill values at optimal ranges approximately are 50% and the military value would be 100 at most, the greatest change of the objective value would be $100 \times 0.50 = 50$. If at least an initially 95 % probability of accept is wanted, T_0 can be found as in (3.1):

$$e^{-\frac{100\times 0.50}{T_0}} \simeq 0.95 \iff T_0 \simeq 975$$

T_0	α	L_k	obj. value	CPU-time
600	0.80	1	188.84	8.62
600	0.80	25	214.93	35.75
600	0.90	1	194.18	9.51
600	0.90	25	217.44	66.39
600	0.99	1	214.02	34.39
600	0.99	25	228.01	616.06
1000	0.80	1	190.16	8.45
1000	0.80	25	217.24	37.53
1000	0.90	1	195.04	9.54
1000	0.90	25	215.69	69.32
1000	0.99	1	215.72	35.09
1000	0.99	25	228.66	673.24
1400	0.80	1	189.66	8.62
1400	0.80	25	216.21	39.14
1400	0.90	1	196.23	9.95
1400	0.90	25	218.19	77.03
1400	0.99	1	217.14	36.83
1400	0.99	25	227.23	715.70

Figure 5.7: Table showing the mean value of both C(x) and the used CPU-time with different cooling parameters used on the 2-13 scenario

T_0	α	L_k	obj. value	CPU-time
600	0.80	1	451.81	17.20
600	0.80	25	461.43	75.73
600	0.90	1	450.14	19.12
600	0.90	25	466.68	123.83
600	0.99	1	462.74	78.19
600	0.99	25	472.62	804.27
1000	0.80	1	452.15	17.87
1000	0.80	25	461.57	76.06
1000	0.90	1	451.47	19.67
1000	0.90	25	465.74	124.74
1000	0.99	1	460.96	69.51
1000	0.99	25	473.64	837.83
1400	0.80	1	451.16	16.62
1400	0.80	25	462.29	76.43
1400	0.90	1	450.52	18.32
1400	0.90	25	466.36	130.51
1400	0.99	1	460.26	71.36
1400	0.99	25	473.44	879.11

Figure 5.8: Table showing the mean value of both C(x) and the used CPU-time with different cooling parameters used on the 8-13 scenario

Three values of T_0 around this value were tested; $T_0 = 600$, $T_0 = 1000$, and $T_0 = 1400$ to see, which of them that would be the best. In addition α-values of 0.80, 0.90, and 0.99 and L_k-values of 1 and 25 were tested, in order to see the performance at various cooling speeds.

Since the expected result of SA, even with the same cooling strategy, might vary depending on the scenario density, the cooling strategies were tested on both the 2-13 and the 8-13 scenario. The test results are shown in figure 5.7 and 5.8.

As expected it can be observed that larger scenarios take longer time. Which cooling strategy to prefer will be discussed below.

A boxplot (figure 5.9) has been made to picture the distribution of C(x) values obtained when using different cooling strategies on the same scenario. The descriptive statistical values of the runs are shown in figure 5.10.

For the 8-13 scenario the cooling strategy $[T_0 = 1400, \alpha = 90, L_k = 25]$

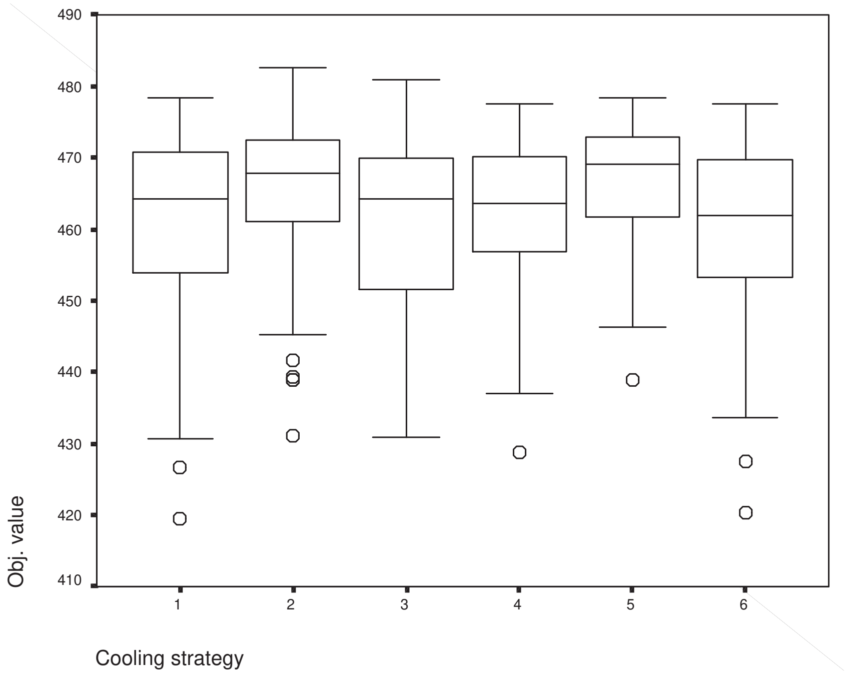


Figure 5.9: Boxplot of some runs using the 8-13 scenario and $uc = 160$. The ID number of each sample corresponds to the cooling strategy shown in the descriptive statistics table in figure 5.10

ID	T_0	α	L_k	mean	s. dev.	max	min	%
1	1000	80	25	461.57	11.4611	478.33	419.52	4.37
2	1000	90	25	465.74	9.8548	482.62	431.14	3.51
3	1000	99	1	460.96	12.0197	481.01	430.93	4.50
4	1400	80	25	462.29	10.3115	477.45	428.75	4.22
5	1400	90	25	466.36	8.4778	478.48	438.87	3.38
6	1400	99	1	460.26	11.9259	477.65	420.44	4.64

Figure 5.10: Descriptive statistics table of the different cooling strategies used on the 8-13 scenario. The %-column shows how many percent the mean value was from the "optimal" found for this scenario

looks most promising with the best mean value and least deviation, though both $[T_0 = 1000, \alpha = 90, L_k = 25]$ and $[T_0 = 600, \alpha = 90, L_k = 25]$ are very close in results. In the denser 2-13 scenario the $[T_0 = 1400, \alpha = 90, L_k = 25]$ strategy once again looks like the superior and might therefore be the best strategy in general for this problem. Due to time limitations, there was not time to test for more superior parameter values. For instance $\alpha = 95$ and $L_k = 10$ might be even better that using $\alpha = 90$ and $L_k = 25$. For larger scenarios than the 8-13 the L_k should be lowered to ensure the solution has frozen when time runs out.

For the 8-13 scenario the worst value of C(x) found in the 100 runs using the $[T_0 = 1400, \alpha = 90, L_k = 25]$ scheme is about 10 % from the "optimal" value and 75 % of the values are within 5 % of this probable optimum. So for sparse scenarios SA does fulfil the wanted property of returning good solutions with a high probability and bad solutions with a little probability.

But when used on dense scenarios like the 2-13 scenario, the mean value in percent is more than 10 % from the "optimal", as it will be shown in section 6, making the general use of SA in this form, though much better than LS-single, unacceptable for this problem.

SA might be improved in several ways though. Plots of the "current solution" at each iteration has been made and as it can be seen in figure 5.11 and 5.12, SA in both cases finds a better solution than the one it ends with in the frozen state. In case of the 2-13 scenario the frozen solution is more than 10 % inferior to the best found solution. Implementing a memory of the best solution found like in TS would therefore be natural and surely an improvement in some degree. This is covered in the next chapter.

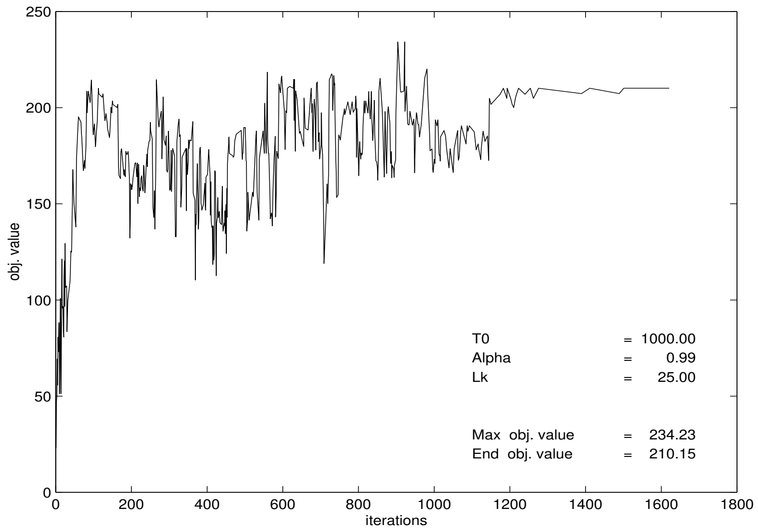


Figure 5.11: The current solution of SA each iteration solving a 2-13 scenario

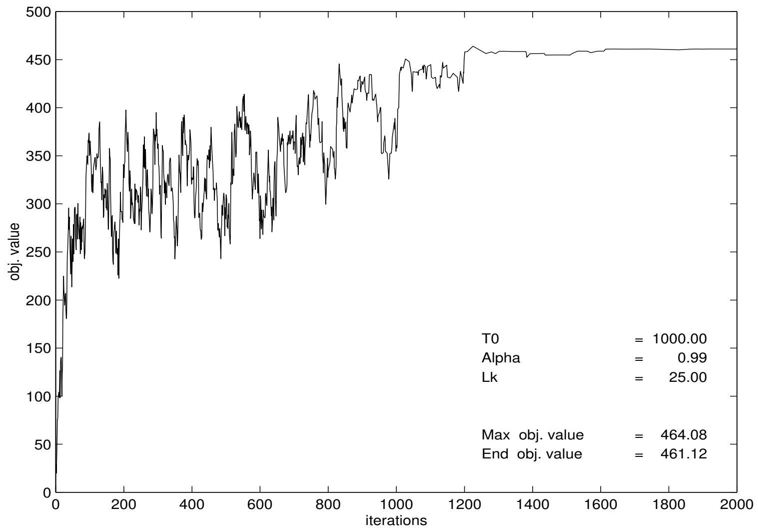


Figure 5.12: The current solution of SA each iteration solving a 8-13 scenario

5.5 Tabu Search results

In TS the new neighbour generating function was implemented making it possible to user-specify the number of neighbours to be generated. The influence from the number of neighbours and the length of the tabu list (i.e. the tabulength) were assessed first, keeping the frequency penalty and the *uc* constant. In most cases with the new neighbourhood more neighbours were generated, so *uc* was lowered to 80, since the total neighbourhood now should be explored faster.

As described in the last chapter, two versions of TS was implemented with the tabu definition being the difference. The one denoted TS-simple uses the "simple" tabu definition with only having the moving target as the tabu element, while the other, TS-advanced, uses the more advanced definition

Length of tabu list	Number of neighbours			
	4	10	16	22
4	209.93	226.24	223.50	219.49
	18.10	62.97	47.49	115.22
8	204.38	221.27	227.14	226.64
	16.08	47.49	86.35	128.83
12	199.95	214.65	218.51	218.59
	15.37	37.97	63.39	85.10

Figure 5.13: Table of the mean objective function value (upper value) and the mean CPU-time used (lower value) for TS-simple using the 2-13 scenario

Length of tabu list	Number of neighbours			
	4	10	16	22
4	211.54	226.62	226.69	222.82
	18.87	67.69	110.15	144.53
8	207.96	226.12	228.51	226.33
	17.66	59.06	101.37	139.07
12	205.43	223.86	229.01	228.28
	17.31	54.26	88.70	139.60

Figure 5.14: Table of the mean objective function value (upper value) and the mean CPU-time used (lower value) for TS-advanced using the 2-13 scenario

Length of tabulist	Number of neighbours					
	2	10	18	26	34	42
4	377.02 8.76	447.29 56.30	463.48 112.60	470.21 194.84	471.72 265.53	473.07 321.45
8	371.41 8.18	433.68 48.82	451.32 89.26	461.38 136.83	465.88 185.42	469.88 245.17
12	369.48 8.12	420.63 40.47	438.74 72.81	446.43 95.89	452.39 125.48	455.48 154.46

Figure 5.15: Table of the mean objective function value (upper value) and the mean CPU-time used (lower value) for TS-simple using the 8-13 scenario

Length of tabu list	Number of neighbours					
	2	10	18	26	34	42
4	384.81 9.31	458.49 66.53	469.98 140.02	471.44 213.62	470.91 272.07	469.66 296.88
8	385.65 8.89	453.34 57.21	468.97 121.11	470.21 190.46	471.27 261.66	472.10 339.72
12	380.83 8.75	453.04 59.37	465.99 119.80	468.52 192.61	472.77 265.47	472.80 322.59
16	382.40 9.48	450.72 60.61	463.96 80.38	466.34 162.22	471.00 237.19	470.03 268.15

Figure 5.16: Table of the mean objective function value (upper value) and the mean CPU-time used (lower value) for TS-advanced using the 8-13 scenario

having the tabu element as a combination of the moving target and the receiving HIPIR.

In tables in figure 5.13 to 5.16 shows the results of the two different versions of TS used on the 2-13 and the 8-13 scenario.

When using the simple tabu definition a 10 % frequency penalty was used, since only 13 tabu element exist, being the 13 targets. This value was a guess but from the earlier results achieved by the other methods, neighbours often differed less than 0.5 in their values of C(x), so the 10 % penalty should be enough to diverge the search to other areas, while still being small enough to not disrupt the basic search with "noise".

When using the advanced definition the number of tabu elements was 8×13, so the penalty was for this definition increased to 100 % for similar reasons as above. It will later be tested, if this values actual are the optimal.

As it can be seen for both scenarios, having the tabu list longer, or as long as the number of neighbours generated each iteration, is ineffective. The probability of all the neighbours being tabu the same time is high, making the search for the global optimum inefficient. In addition the $uc = 80$ used probably was to small for the cases with only 2 or 4 neighbours generated, though increasing uc should not make any big differences in the overall picture.

In general, when increasing the number of neighbours to be generated, better results are achieved, though the time limit might be exceeded. For both scenarios the advanced definition performed best, and this with a longer tabu list than the one used by TS-single for getting the best results.

For the 8-13 scenario a number of 34 neighbours generated looks as the optimal for both definitions, if the mean computation time are to be kept below 300 seconds. The best result with 34 neighbours is, when using the simple definition, while having a tabu list lenght of 4, while the advanced does best having a tabu list length of 12.

If using the 2-13 scenario, the optimal number of neighbours to be generated once again is the same for the two definitions – this time 16. TS-simple does best while having a tabu list length of 8, while TS-advanced still does best with a tabu list length of 12.

With this values fixed, the frequency penalty is now varied, to see if the values used until now actual are the optimal. The results can be read from the graphs in figure 5.17 and 5.18.

While TS-simple turns out to do better using a penalty of 15 % rather than the previous used of 10 %, the 100 % penalty used by TS-advanced looks optimal. The CPU-time used by the algorithms was about 250 CPU-seconds for for all the different penalty values tried.

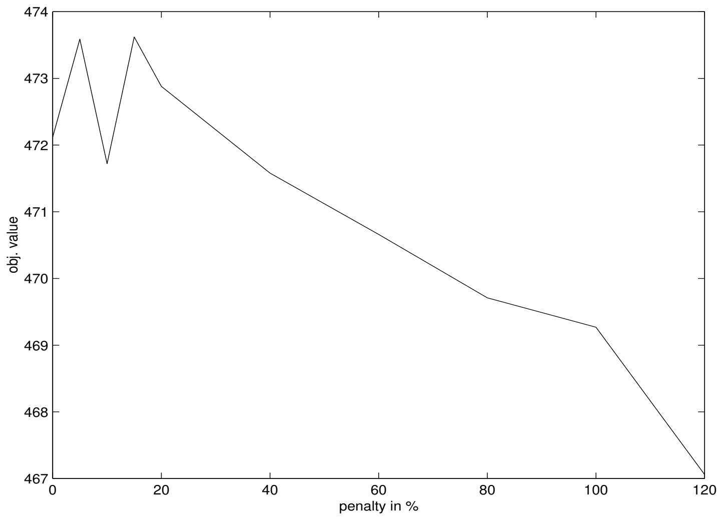


Figure 5.17: The dependence of the penalty-% for TS-simple using the 8-13 scenario

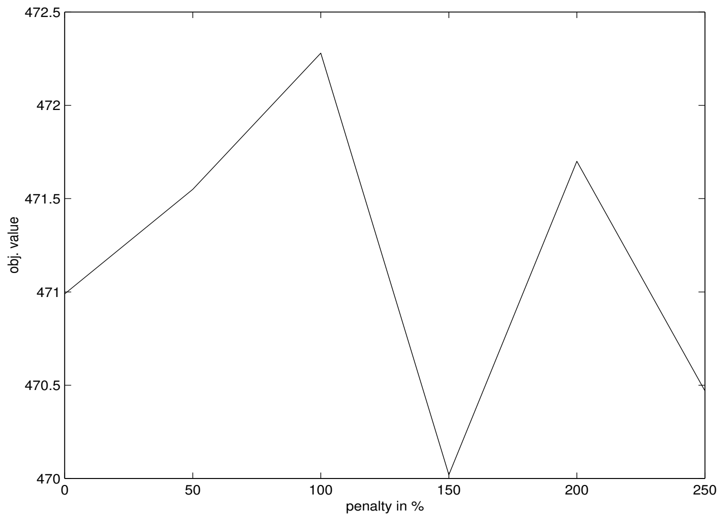


Figure 5.18: The dependence of the penalty-% for TS-advanced using the 8-13 scenario

Scenario	ID	mean	s. dev.	max	min	% best
2-13	1	227.14	8.78	247.56	205.42	8.31
	2	229.01	9.04	247.73	208.83	7.56
8-13	3	473.62	6.86	482.67	449.89	1.87
	4	472.77	6.91	482.03	438.58	2.05

Figure 5.19: Statistics on of some of the best TS runs. The ID numbers refer to the parameter values shown in figure 5.20 below

ID	method	NB	$\mid T \mid$	% penalty
1	TS-simple	16	8	15
2	TS-advanced	16	12	100
3	TS-simple	34	4	15
4	TS-advanced	34	12	100

Figure 5.20: Table showing the parameters used in figure 5.19. NB denotes the number of neighbours generated, $\mid T \mid$ denotes the tabu list length and the % penalty, the penalty to the frequency memory

Looking at the tables in figure 5.19 and 5.20 it can be seen, that the actual performance of TS-simple is equal to that of TS-advanced, unlike what you could have expected. The optimal numbers of neighbours to be generated might look high, but as shown – with a lower number of neighbours where more iterations can be performed instead, the algorithms come out with inferior solutions.

Both TS implementations look promising with respect to the properties wanted for heuristics, the quality being like the one for LS-multi and with a faster computation time.

5.6 Comparison

Until now 5 algorithms have been covered in this chapter: LS-single, LS-multi, SA, TS-simple, and TS-advanced. For a fair comparison of the algorithms, they were run for a fixed 300 CPU-seconds period. The algorithms were tried using both the 2-13 and the 8-13 scenario as earlier, in order to see any dependence of the scenario density. Note that the results in this section are based on 50 runs of each and not the 100 as used by the test program earlier in this chapter.

The parameter choices used are the same as the ones found to work best below 300 seconds in average computation time. They are shown in table 5.21 and 5.22. When now running for a fixed period, the individual runs, which took longer without the time limit should return inferior solutions, while the ones finished before the 300 seconds might obtain better results. As it can be seen later in figure 5.23 and 5.24, the mean value of C(x) is actually almost unchanged.

LS-single	-
LS-multi	$uc = 50$ for each single search
SA	$T_0 = 1400, \alpha = 90, L_k = 25$
TS-simple	$NB = 16$, $\mid T \mid = 8$, %-penalty = 15
TS-advanced	$NB = 16$, $\mid T \mid = 12$, %-penalty = 100

Figure 5.21: The parameter choices used when comparing the methods while running for a fixed period using the 2-13 scenario

LS-single	-
LS-multi	$uc = 50$ for each single search
SA	$T_0 = 1400, \alpha = 90, L_k = 25$
TS-simple	$NB = 34$, $\mid T \mid = 4$, %-penalty = 15
TS-advanced	$NB = 34$, $\mid T \mid = 12$, %-penalty = 100

Figure 5.22: The parameter choices used when comparing the methods while running for a fixed period using the 8-13 scenario

Tables of the descriptive statistics of C(x) obtained using the different methods are shown in figure 5.23 and 5.24, while boxplots showing the distributions for comparison can be found in figure 5.25 and 5.26. LSS and LSM denote the LS-single and LS-multi methods, while TSS and TSA denote the TS-simple respectively TS-advanced methods.

Method	mean	s. dev.	max	min	% best
LS-single	195.46	20.12	241.63	155.02	21.10
LS-multi	230.40	6.45	245.44	218.87	7.00
SA	218.18	13.45	246.30	185.93	11.93
TS-simple	233.31	6.93	247.56	221.97	5.82
TS-advanced	235.08	6.53	247.73	222.14	5.11

Figure 5.23: Descriptive statistics of the different algorithms when used on the 2-13 scenario

Method	mean	s. dev.	max	min	% best
LS-single	453.84	16.09	478.52	419.00	5.97
LS-multi	473.51	3.89	482.62	464.21	1.90
SA	465.92	8.72	480.00	440.10	3.47
TS-simple	475.21	4.56	482.67	459.66	1.55
TS-advanced	472.41	5.27	480.81	457.36	2.13

Figure 5.24: Descriptive statistics of the different algorithms when used on the 8-13 scenario

From the figures it is concluded that TS-advanced is the best suited algorithm overall, since it is the best used with dense scenarios and comparable with every other in the more sparse scenarios. Both TS-simple and LS-multi are very close in the results though. SA as implemented is somewhat inferior, especially for the dense scenarios. It looks like the dense scenarios have a solutions space more ill-suited for the metaheuristics in general and SA in particular. LS-single was as expected the worst and is far from the others in all aspects.

As the results show, non of the methods fulfil the property set up in chapter 2 of having the mean value of C(x) to be within 5 % of the optimal value for all scenarios. TS-advanced almost makes it though. In the next chapter various improvements to help the algorithms to fulfil the mentioned property will be discussed.

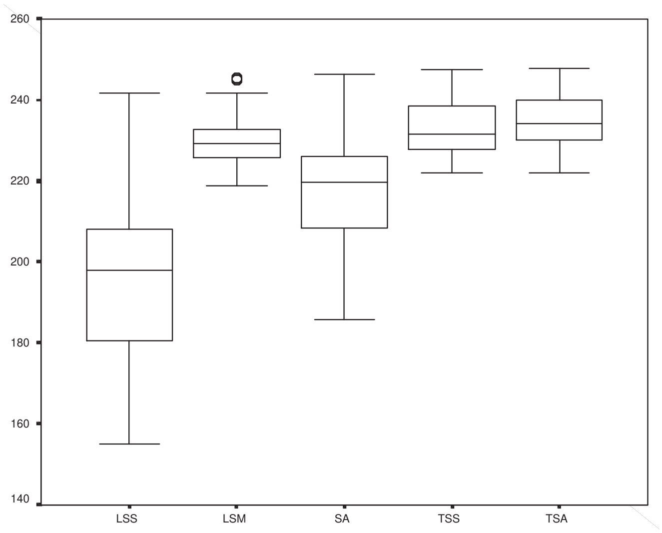


Figure 5.25: Boxplot showing the different methods used on the 2-13 scenario and with parameter choices as described in 5.21

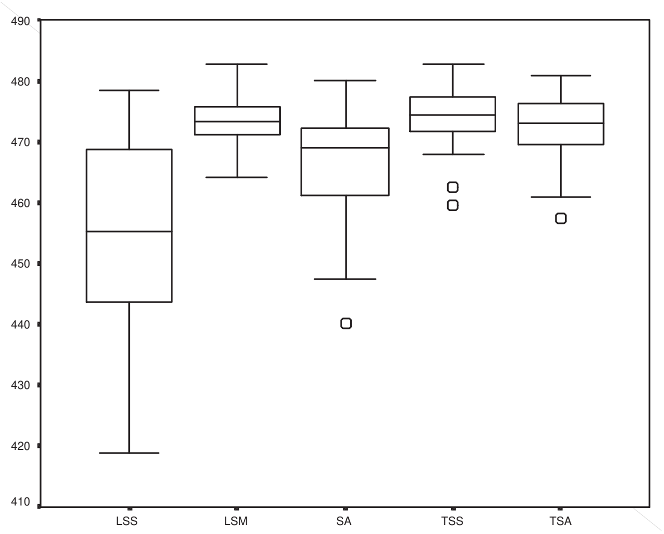


Figure 5.26: Boxplot showing the different methods used on the 8-13 scenario and with parameter choices as described in 5.22

Chapter 6

Improving the algorithms

The algorithms described in chapter 5 can be sought improved in different ways. The possible improvements mentioned in section 1 have all been tried, with the results shown in section 2. In section 3 it will be discussed whether any of the improvements should be implemented in a final algorithm for the Target - Radar allocation problem.

6.1 Possible improvements

During the implementation and the test runs some ideas of improvements to the algorithms came up. While time was insufficient to implement all of them, a number of the improvements were actually tried. They will be described below.

6.1.1 Changing the *time_opt* procedure

As mentioned it looked as the *time_opt* procedure using the Steepest Ascent strategy returned the optimal value almost every time. But the expected value might be improved if the cases, as the one shown in figure 4.4 , were corrected to the optimal solutions.

A way to try this was to accept inferior moves a small (i.e. ≤ 3) number of times, which would cause the blocking engagement to be moved. If the improvement, which could be made of the previous-blocked engagement, then was higher than moving the previous-blocking engagement back, a better solution is found. This was rather easy to implement, and can be viewed as a special case of SA, where all moves are accepted with the probability 1 (very high temperature) until L_k inferior moves have been made. Then the temperature drops instantly to (almost) zero, allowing no more inferior solutions to be accepted. This method, called *quenching* in the literature, could help getting better solutions with only a little increase in the computation time.

6.1.2 New neighbourhood definition in LS and SA

With the new neighbourhood for the TS algorithm implemented it would be natural to try LS and SA with this neighbourhood too, in order to see, if this would improve or degrade the performance, depending on the number of neighbours generated each iteration.

6.1.3 SA with memory

As it was shown in the previous chapter, SA might converge toward a value inferior to an earlier found solution, it cannot regain because of the frozen state. Therefore SA could be implemented with a memory as in TS, storing the best found solution and returning this in the end.

6.1.4 Other acceptance function in SA

Instead of using the *exp()* function, which usually takes long to compute, the first terms of its Taylor expansion around zero could be used as seen in (3.5). This would hopefully both improve the computation speed and result in earlier convergence, since no tail exists.

6.1.5 No logfile generation

During each computation ran the algorithms wrote a logfile containing information about every move and decision made. While exceptional for finding errors, it was not necessary when free of errors, and might slow the algorithm. So a version without the logging was done in order to see the effect on the computation time.

6.1.6 Tabu list clearance

As implemented in the TS algorithms described in chapter 5, the tabu list is never cleared. When a new best solution is found, a tabu element might therefore block for an effective local search performed from this solution and thus the possibility of finding an even better solution the next iteration can be missed. Clearing the tabu list when a new best solution is found will solve this. Now a local search can be performed without any restrictions from the new solution, and this will hopefully improve the results of the TS algorithm

6.2 Results

6.2.1 Changing the *time_opt* procedure

The described "improvement" was tested at LS-single only, since it turned out to be ineffective. The results are shown in figure 6.1.

For the 2-7 scenario, which is another than the previous used, we have 3.5 targets for each HIPIR. From the results we can see, that this is to much for any improvements. The engagements lie to close to be moved. In fact the results are inferior to the non-improved results in column 1.

In the 8-13 scenario we have less than two targets for each HIPIR, so the chance of having occurrences like the one shown in figure 4.4 is higher. A small improvement can be seen though, but it is only about 0.25 %, with the computation time more than doubled.

As conclusion this "improvement" is useless in the implemented form, except maybe for very sparse scenarios. If implemented to be used only in cases with few targets assigned to a HIPIR compared to the planning time, with the normal *time_opt* used in the other cases, the result might turn out better, but the work has been concentrated in other areas.

Scenario	*uc*	No. of inferior solutions accepted 0	1	2	3
2-7	25	210.23 1.23	208.39 1.77	208.65 2.25	208.65 2.90
2-7	80	215.11 2.69	213.44 3.89	213.69 4.95	213.69 6.38
2-7	160	215.38 4.59	213.44 6.54	213.69 8.57	213.69 10.63
8-13	25	432.68 2.84	434.87 6.56	434.90 8.12	434.87 9.78
8-13	80	449.76 5.47	451.56 14.94	451.59 18.55	451.56 22.35
8-13	160	453.46 8.98	454.48 23.74	454.61 29.52	454.58 35.04

Figure 6.1: Table showing the sample mean of objective function value (upper value) and the CPU-time used (lower value) for different combinations of scenarios, *uc*'s, and the number of accepted inferior moves

6.2.2 New neighbourhood definition in LS and SA

The neighbourhood definition from the TS implementation was implemented for use in the LS-single algorithm. Used on the 8-13 scenario, a number between 2 and 42 neighbours were generated each iteration. A value of $uc = 80$ was used for all the test runs, since more neighbour than with the old definition were generated in most cases. The results are shown below:

NB	2	10	18	26	34	42
C(x)	419.81	454.65	452.98	453.60	454.21	454.25
CPU-time	8.63	45.93	71.54	97.30	122.89	146.11

Figure 6.2: The sample mean of C(x) and the CPU-time used when generating *NB* neighbours each iteration, having $uc = 80$, and using the 8-13 scenario

With the old neighbourhood definition a sample mean of $C(x) = 453.46$ was achieved with a mean CPU-time usage of 18.42 seconds. So while a small improvement can be gained compared to LS-single, the longer computation time will make LS-multi inferior in comparison with the use of the old definition.

So no further tests were tried in either LS or SA, in order to use the time on more promising experiments.

6.2.3 Changes to SA

The two possible improvements to SA mentioned have been tested both individually and in combination. Figure 6.3 and 6.4 contain boxplots showing the distribution of C(x) for the 2-13 respective 8-13 scenario.

Each plot shows the results from 5 samples. The first is the unmodified SA from chapter 5, denoted SA. The next two show the Taylor acceptance function implementation, SAT, and the memory implementation, SAM. Number four is the combination of the previous two and is denoted SATM, while the last is the TS-advanced from chapter 5, which has been included for comparison reasons.

From sample 2, showing the implementation of the Taylor acceptance rule, it can be seen that the faster computation hoped for using this rule was not achieved. Only an increase of 1.18 % in the mean number of iterations performed in a single run was gained for the 2-13 scenario, while the use of the Taylor rule solving the 8-13 scenario in general resulted in fewer iterations each run. So the time used for the *exp()* function calls were insignificant compared to the time used on generation neighbours, in the *time_opt* procedure, etc.

In both cases though, the mean value of C(x) has been improved a little, probably because of the faster convergence that should happen at low temperatures.

The memory implementation turns out be a good investment. Especially when used on the 2-13 scenario, where the mean value of C(x) was improved with more than 5 % compared to the unmodified SA. The gain when used on the more sparse 8-13 scenario is insignificant though, so the two plots in figure 5.11 and 5.12 actually represents the general case quite well.

When the two improvements are combined, the achievement is a little less than the sum of the improvements, looking at them single. But as it can be seen, this implementation of SA is better than any of the other implementations of SA. The statistical values of the results are shown in figure 6.5 and 6.6.

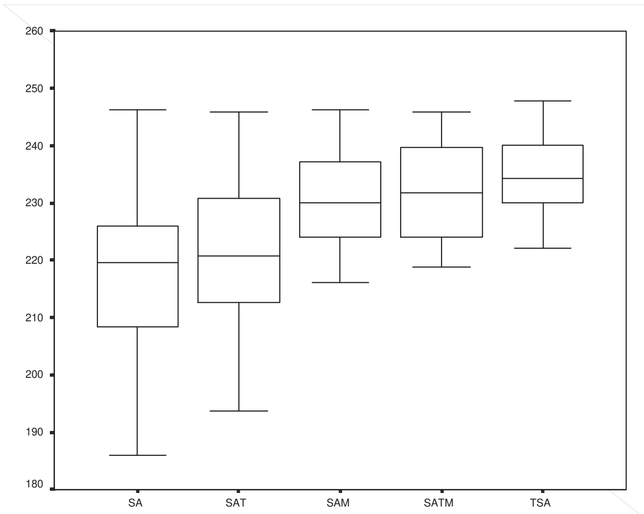


Figure 6.3: Boxplot showing the results using various SA improvements on the 2-13 scenario

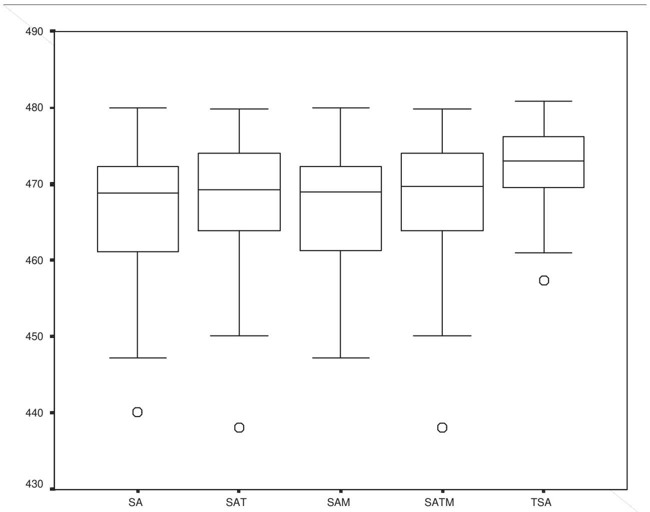


Figure 6.4: Boxplot showing the results using various SA improvements on the 8-13 scenario

Method	mean	s. dev.	max	min	% best
SA normal	218.18	13.45	246.30	185.93	11.93
SA Taylor	222.24	13.10	245.82	193.72	10.29
SA memory	230.76	8.20	246.30	216.12	6.85
SA both	231.62	8.50	245.82	218.92	6.50
TSA normal	235.08	6.53	247.73	222.14	5.11

Figure 6.5: Descriptive statistics of C(x) using different algorithms on the 2-13 scenario

Method	mean	s. dev.	max	min	% best
SA normal	465.92	8.72	480.00	440.10	3.47
SA Taylor	467.96	8.32	479.81	438.11	3.05
SA memory	466.73	7.85	480.00	447.31	3.30
SA both	467.99	8.32	479.81	438.11	3.04
TSA normal	472.41	5.27	480.81	457.36	2.13

Figure 6.6: Descriptive statistics of C(x) using different algorithms on the 8-13 scenario

6.2.4 No logfile generation

The TS-advanced version was stripped for every output command, except for the showing of the final allocation plan. When solving the 2-13 scenario a 23.5 % improvement of the mean number of iterations performed was achieved, increasing the sample mean of C(x) with about 0.6 % as seen in the table 6.7. Used on the 8-13 scenario, the number of iterations performed within the time limit once again was improved by 23.5 %, while this meant less to the achieved mean value of C(x), which only raised by 0.2 %.

Scenario	method	C(x)	iterations	*time_opt*'s
2-13	TSA normal	235.08	497.20	8669.40
	TSA -out	236.56	614.06	10725.10
8-13	TSA normal	472.41	210.16	11805.64
	TSA -out	473.31	267.86	15077.18

Figure 6.7: The gain when writing no logfile. The columns show the mean values of the results

6.2.5 Tabu list clearance

The TS-advanced implementation was changed so the tabu list was cleared every time a better solution was found. This turned out to make the search for optimums more efficient. As it can be seen in the table below, C(x) was improved with 2.19 % and 0.44 % compared to the non-changed algorithm, when used on the 2-13 respectively the 8-13 scenario.

If the output logging were removed too, as earlier described, the results were even better though not that much.

Scenario	method	mean	s. dev.	max	min	% best
2-13	TSA	235.08	5.27	247.73	222.14	5.11
	TSA cl	240.24	5.34	247.73	224.20	3.02
	TSA cl -out	240.29	5.19	247.73	227.38	3.00
8-13	TSA	472.41	5.27	480.81	457.36	2.13
	TSA cl	474.47	5.73	482.62	461.33	1.70
	TSA cl -out	475.89	3.73	479.81	438.11	1.40

Figure 6.8: Table showing the statistics of C(x) using tabu list clearance

For both the scenarios this means that the wanted quality property defined in chapter 2, which the algorithms should fulfil, now are complied with since the mean value of C(x) are below 5 % from the "optimal" value.

Time was insufficient to test the effect on TS-simple. But this algorithm should be improved too by a similar change, though it is unknown whether it is enough to make TS-simple fulfil the mentioned property.

6.3 Conclusion

From the results shown, both the change of the *time_opt* procedure and the new neighbourhood definition for LS and SA gave no improvement at all. Since only few tests have been made, the neighbourhood definition change might turn out to be an improvement for a *NB* value between 2 and 10. So this should be studied in the future.

SA was improved considerably with the two proposed improvements. It nevertheless still lacks behind in results compared to those of the TS-algorithms and in some cases the LS-multi algorithm too. Both of the mentioned improvements should be implemented if a SA algorithm was to be used, but this is unlikely since better methods exist.

TS-advanced was improved even more, making it the superior without discussion. The algorithm now fulfils the property of the quality of C(x) set up in chapter 2 as the only one of the methods. The improvement should work on TS-simple too, and might even be better, but this was as mentioned not tested.

Removing the logging file of the algorithm improved the mean value of C(x) and especially the CPU-time usage more than expected. Of course a logging of the choices and calculations made should not exist in a final running algorithm, so this improvement is natural.

Chapter 7

Comparison with the DDRE algorithm

In order to try getting an idea of the performance of the metaheuristic algorithms the most promising of the algorithms was compared to the existing DDRE algorithm on some points.

First in this chapter the DDRE algorithm will be introduced in order to make the reader see, what the metaheuristic was up against. Then the points of comparison that were used to try find the superior of them will be described. In section 3 the actual comparison will take place and the circumstances that might hinder an orderly comparison are covered. This is followed by a discussion of whether a metaheuristic solution would be preferable, based upon the results.

7.1 Outline of the DDRE algorithm

The DDRE has also chosen to split the problem in two parts – the allocation of targets to the HIPIR's and the time optimization of a given HIPIR allocation. But both of the parts are solved differently than in the metaheuristic algorithms covered elsewhere in this report, as it can be seen on the next page.

HIPIR allocation: In the HIPIR allocation part a searchtree is used. The search starts from the root, where no allocations have been made for any of the HIPIR's. A loop finds the ΔC value of every possible allocation of the currently unassigned targets to every possible HIPIR. Being a greedy algorithm, the move returning the highest ΔC value, is carried out. This procedure is repeated until all targets have been assigned to a HIPIR or no more targets can be engaged within the planning time, which is sketched in Allocation A of figure 7.1. When coming to such a "dead end" the algorithm goes back and continue with the second best "move" of the previous node until the next "dead end" is reached (Allocation B) continuing this way.
Each branch in the tree denotes a single HIPIR - target assignment and the closer the branch is to the root, the earlier the described engagement is carried out.
The procedure stops when the n_l best moves have been tried from each node (Allocation C). The value of n_l may differ depending on the level, l, of the tree (i.e. the depth of the tree – corresponding to the number of assigned targets). Any branches, which cannot improve above the best result found, are not researched, as in a branch and bound procedure.

Time optimization: A problem specific heuristic is used for the time optimization. The targets allocated to a specific HIPIR are first put in a tight sequence, starting each engagement either at the optimal time to engage it by that HIPIR or if impossible because of the previous engagement, as soon as this has finished.

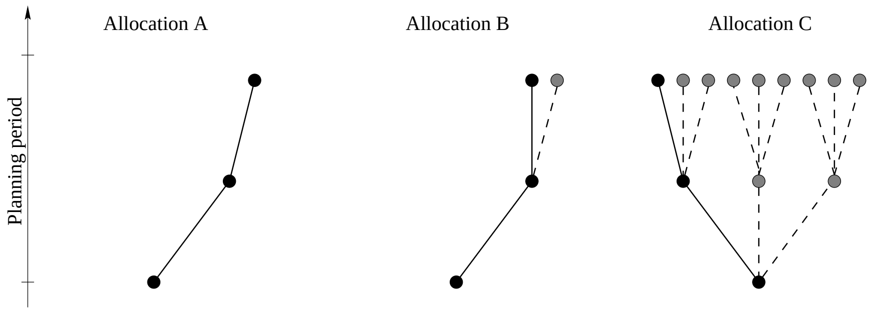


Figure 7.1: The building of a search three - in this case $n = 3$ for all levels

> This may put the last target inside the critical line before it is engaged, if it can be engaged at all. If this happens the start times of the earlier engagements are advanced, so all targets are engaged at the critical line or earlier. This again might push some engagements out of the intercept period, and the necessary engagements are delayed in order to correct this.

Note that since this algorithm contains no stochastic element, it is deterministic unlike the metaheuristic algorithms covered in this report.

The DDRE research team had a slightly enhanced definition of the military value. In the algorithm, as described in chapter 2, you have to assume, that the targets are keeping the same course and speed for the duration of the planning period. For the targets that are engaged early in the planning period, the assumption is not that strong. But for a target planned to be engaged 3 minutes later this assumption result in a degree of uncertainess.

A way to take this uncertainess into account is to reduce the actual military value of a target depending on the time it is engaged. Thus a target that can be engaged immediately, is valued with its full actual military value, while a target, which *maybe* enters the intercept area several minutes later will have its military value reduced by multiplying it with a timedependent factor, because of the possibilities of maneuvers.

This corrected military value, CMV, to be used in the algorithm can be found from the actual military value, AMV, using this expression:

$$CMV = AMV \times e^{-\lambda \cdot t}$$

where t denotes the time where the target is intercepted and λ is a sufficiently large positive constant.

This enhanced definition was not implemented in the previous tested algorithms, since it made it difficult to see, if the algorithms behaved as they should.

7.2 Points of comparison

The most promising the metaheuristic algorithms was the TS-advanced, with no output and tabu list clearance, as described in chapter 6. The implementation was changed to include the calculation of the corrected military values, as described above. The parameters used for the comparison was the same as in 5.21 and 5.22 while the Pkill values and the basic military values now were the "real ones". So the values of C(x) obtained by the TSA algorithm during the comparison tests cannot be compared to those earlier achieved.

The two algorithms, denoted modified TSA and DDRE, were as usually tested on the two standard scenarios – the 2-13 scenario and the 8-13 scenario – in order to see if the algorithms depended differently upon the target-density of the scenarios.

Being deterministic, the DDRE algorithm was only run once for each scenario, while the modified TSA algorithm was run 50 times to get the estimates of the sample mean value and variance of C(x).

The comparison was mainly done on two points: the obtained C(x) and the ranges the targets were intercepted at, though the number of targets engaged in total by each and the speed of the algorithms will be discussed shortly.

7.3 Results

Looking at the results both algorithms engaged 10 targets running the 2-13 scenario and 13 targets when running the 8-13 scenario. So at this point they performed equally.

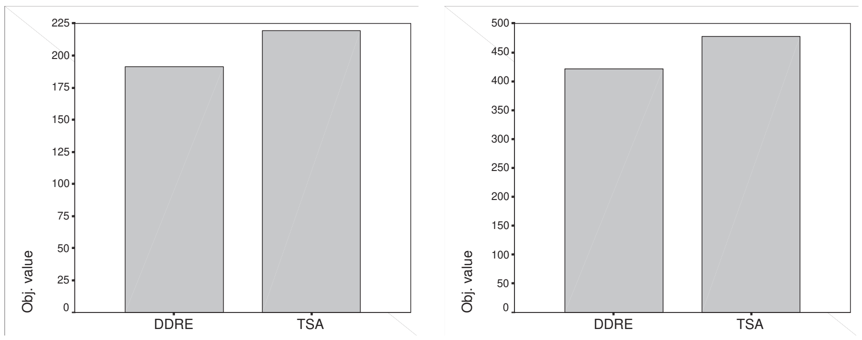
The C(x) results for the TSA algorithm are shown in the table below:

Scenario	mean	s. dev.	max	min	% best
2-13	218.64	1.03	220.24	216.89	0.73
8-13	478.72	1.96	483.02	473.95	0.89

Figure 7.2: The results of the TSA algorithm modified for comparison as noted in section 2

The value in the % best column is the sample mean value compared to the maximum value achieved in the 50 runs, i.e. 220.24 and 473.95 respectively. Note that the variances are now considerably smaller than those of the unmodified TSA algorithm. This is probably because the latest engagements using the corrected military values now contribute less to the results of C(x), and with more small contributes these results should variate less.

Comparing it with the DDRE algorithm, the modified TSA algorithm proved to be far superior looking at the objective function maximization alone as seen in figure 7.3. The modified TSA algorithm did 12.71 % better for the 2-13 scenario and 11.91 % better for the 8-13 scenario. This turned out to be because the DDRE had valued the property of engaging the targets before they reached the critical line higher than it was defined to in chapter 2 of this report. It can be seen in figure 7.4, where the sum of the ranges from the firing HIPIR to the point where the engaged targets were intercepted are pictured. These values can also be found in the table in figure 7.5.



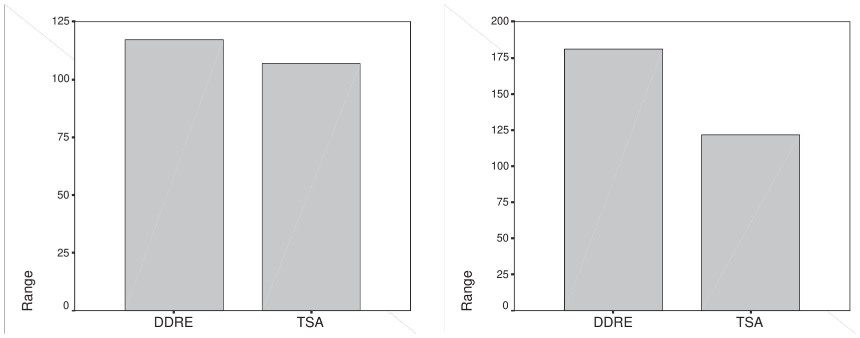
(a) 2-13 Scenario

(b) 8-13 Scenario

Figure 7.3: The C(x) obtained by the algorithms. The value for modified TSA is the sample mean

The DDRE algorithm engaged the targets at 9.66 % greater range as a mean for the 2-13 scenario, while the value for the 8-13 scenario was more than 48 %.

In addition for the latter scenario the modified TSA algorithm let two more targets cross the critical line before being intercepted than the DDRE algorithm. This results in a higher probability of taking damage from the enemy.



(a) 2-13 Scenario (b) 8-13 Scenario

Figure 7.4: The sum of the ranges where the targets were intercepted for the two algorithms

Scenario	Algorithm	Obj. fct. value	$\sum$ ranges	No. within CL
2-13	TSA	219.50	106.87	7
	DDRE	191.60	117.19	7
8-13	TSA	477.91	121.66	9
	DDRE	420.99	181.00	7

Figure 7.5: Different ways of comparing the two methods

With respect to the time used, a maximum time limit could not be set in the DDRE algorithm. The results shown though, were all obtained within 300 seconds of CPU-time.

But an actual comparison of the speed makes no sense. Both because the DDRE algorithm was much less optimized for speed (this will come when it is reprogrammed in C++), but also because it had some additional procedures implemented, like the one performing the missile launcher allocation.

7.4 Conclusion

The modified TSA algorithm proved better than the DDRE algorithm with respect to maximizing the defined objective function. But the military will nevertheless prefer their algorithm, because of the greater engagement ranges, and thus the lesser probability of taking damage from enemy attacks. The objective function as defined and used in both the modified TSA and the DDRE algorithm does not, with the current way of calculating the military value, maximize this engagement ranges. This is instead, at least in the DDRE algorithm, partly done in the time optimization part.

Clearly the modified TSA algorithm needs to improve with respect to the range property. Some ways of doing this will be discussed below.

First of all the definition of the military value could be changed to better influent the objective function to optimize for greater engagement ranges. This could be done by letting the military value drop even faster within the critical line or by only letting the range to the closest target contribute to the calculation.

Another way to make the algorithm engage the targets at longer ranges is to change the time optimization part. Utilizing the one used in the DDRE algorithm would probably help a lot, and could be very interesting to try in the future.

A last way of correcting this would be to see the problem as a multiple criteria problem. If $x = (e_1, e_2, ..., e_3)$ is a solution where e_i denotes an engagement, the definition of the objective function could then be changed to:

$$
\text{"}maximize\text{"} \quad C(x) = \begin{cases} C_1(x) = \sum_{\forall e_i} Pkill(e_i) \times BMV(e_i) \\ C_2(x) = -\sum_{\forall e_i} Penalty(e_i) \\ C_3(x) = -\sum_{\forall e_i} Cost(e_i) \end{cases}
$$

where $Pkill(e_i)$ denotes the Pkill of the engagement e_i and $BMV(e_i)$ the basic military value of the target in the same engagement. $Penalty(e_i)$ is a penalty function depending upon the range to the closest object from the point where e_i is intercepted and the type of the target engaged. At last $Cost(e_i)$ denotes the cost of the missile used in engagement e_i. The restrictions to the system should express the same as the restictions (2.2) until (2.9) did in the model formulated in chapter 2.

In order to use this in decision making a single objective function has to be made. The weighted sum program as described in [2] can be used for this:

$$maximize \quad C(x) = \sum_{i=1}^{3} \lambda_i \cdot C_i(x) \qquad \text{where} \sum_{i=1}^{3} \lambda_i = 1$$

The λ_3 should in this case be so low that the third objective function, $C_3(x)$, only makes a difference to the decision making, if one solution uses much fewer (or in case of several missiles types exist - maybe cheaper) missiles than another where the $C(x)$ values otherwise would be the same.

Solving the problem as a multiobjective combinatorial optimization problem as shown above might turn out to be more efficient than using the current model of the probem.

The modified TSA algorithm looked much more time optimized than the DDRE algorithm. So while the 100 times improvement in the computation time may come easy for the DDRE, it might not be possible to improve the computation time for the TSA algorithm that much.

No comparison of the performance within the same computation time has been made for that reason been made, since it would show more about the implementation rather than the actual performance of the algorithms.

In general the metaheuristic algorithm performed well and can probably be improved as described to take more care about the engagement ranges. It might end up still inferior to the DDRE algorithm in some respects, but it might as well be able to outperform it. Only further trials will show.

Chapter 8

Summary

Because of the complexity of the Target - Radar allocation problem, the neighbourhood generation for both the implemented optimization parts was complicated and thus time consuming, due to the many special cases. In addition the time to solve the problem was very limited. So this was not the most usual problem to use a metaheuristic approach on, though it for the same reasons, made it interesting to see the performance of the metaheuristic methods in this uncommon environment.

The results of the metaheuristic methods covered in this report: Local Search, Simulated Annealing, and Tabu Search were all compared to each other. The best of them was as well compared to the algorithm designed by the DDRE. The conclusions done from these comparisons will be covered in the next section while in section 2 some of the areas, that might be interesting to research further, will be discussed.

8.1 Conclusion

Form the results it is concluded that TS-advanced is the best of the metaheuristics implemented with TS-simple and LS-multi close behind. For SA too little time was available to work properly, since cooling should happen faster than it was effective for the solution space of the problem, especially with that of high-density scenarios.

The modified TS-advanced algorithm did well when comparing it with the DDRE algorithm. But the differences, in which properties to value highest, made an actual comparison impossible. The problem was that the defined objective function, which TS-advanced was best to maximize, did not ensured that the targets - if possible - were intercepted at the critical line at latest. This specific property is valued very high by the Danish Airforce, so they prefer the DDRE algorithm, which was much better on this point. Since longer engagement ranges result in a lesser value of the objective function, this explains why the values of the objective function obtained by the DDRE algorithm were so much inferior to the ones of the modified TS-advanced algorithm.

Exchanging the time optimization part, though returning good objective function values, with the one used by the DDRE should make the algorithms comparable, since it is in this procedure, the DDRE algorithm differ from the implemented metaheuristics in how the properties are prioritized.

Looking at the wanted properties of an algorithm defined in section 2.1 of chapter 2, the following conclusions can be made:

- Speed: If the assumed 100 times decrease in the computation time can be achieved, this property is complied with, but whether it is possible cannot be said. How much the results depend upon the time available was not assessed and this should be done in the future.
- Maximizing: It was demanded that the expected value of the objective function was within 5 % of the global optimum. TS-advanced complied with this property and had, when modified for using the corrected military value instead of the actual military value, a large margin to the 5 % demanded for all tested scenarios.
- Range: Whether the critical line was crossed by any targets before being intercepted, was not researched that much. This was because it initially was expected that a maximization of the defined objective function would take care of this too. As noted, it was not the case, and solutions to this problem were discussed in chapter 7.
- Robustness: This property is not relevant when performing a single run, since this is a static situation. None of the metaheuristics implemented were ready to run in a dynamic system updating the allocation plan for every 3-second period. Keeping the solution stable in such a system can be achieved by using the previous solution as the initial for the next run. This is described in more details in the next section.

In general the results were promising and better than expected. So at least on this problem - though the problem was not that well suited - a metaheuristic solution could be used with acceptable results. It might therefore also be true for other problems to with similar characteristics.

8.2 Further research

Much work has been done and many things tested. A lot of further research exist though, which both would be natural and interesting to do.

Most important possible ways to improve the engagement ranges should be researched. Several ways of doing this has been mentioned in chapter 7. Maybe a change of the *time_opt* procedure, for instance to the one used by the DDRE, would be enough. This might also make the overall algorithm faster - only tests will show.

The dependence of the time available should be assessed too. If the assumption of a 100 times improvement of the computation time does not hold, it would be nice to know, which quality of the results to expect when using the actual time available.

In general more tests should be done to better assess the optimal parameter values. Some possible improvements to the different metaheuristic methods could be tested too. These will be outlined below:

Local Search: For the LS-multi algorithm each new run was completely random. Jumpsearch or a similar non-random multistart technique for LS, as described in [1] and [15], could be very interesting to try instead. In those the initial solution of the next run is chosen as one in the area "close" to a good solution.
In addition more tests using the neighbourhood definition from TS could be made.

Simulated Annealing: Other cooling strategies could be tested, as could the neighbourhood definition used in TS.

Tabu Search: Implementation of a varying tabulength within a single run, instead of the fixed sized could be tested, as could the use of tabu list clearance in the TS-simple algorithm.

At last, if the results from the static runs stand comparison with the ones from the DDRE algorithm, the modified TS-advanced algorithm should be enhanced for use in a dynamic system.

The main part would be to make a procedure for testing whether the solution from the last 3-second run still is possible, since changes in speed and course of some of the targets might move their engagement start times outside the possible intercept period. Such targets should be deleted from the allocation plan and instead, together with any new targets discovered, be added to the unassigned array. This allocation plan should then be used as the initial solution in the modified TS-advanced algorithm.

In addition the algorithm should be able to choose the optimal number of neighbours to be generated, tabulength, etc., depending upon the current system status and treat.

Bibliography

[1] Boese, K. D., Kahng, A. B., and Muddu, S.
A new Adaptive Multi-start Technique for Combinatorial Global Optimization
Operations Research Letters, 16, 1994.

[2] Borges, P. C. and Hansen, M. P.
A basis for future Successes in Multiobjective Combinatorial Optimization
To be published as Technical Report, IMM, DTU.

[3] Borges, P. C. and Vidal, R. V. V.
Fixed Channel Assignment in Cellular Mobile Telecommunication Systems
Technical Report 1996-25, IMM, DTU, 1996.

[4] Collin, N. E. et al.
Simulated Annealing – an annotated bibliography
American Journal of Matematical and Management Science, 8, 1988.

[5] Cullen, T., and Foss, C. F. (Editors)
Jane's Land-based Air Defence 1997-1998
Janes Information Group Ltd., 1997.

[6] Dueck, D., and Scheuer, T.
Threshold Accepting: A General Purpose Optimization Algorithm Superior to Simulated Annealing
Journal of Computational Physics, 90, 1990.

[7] Dumas, Y., Desrosiers, J., and Gelinas, E.
An Optimal Algorithm for the Traveling Salesman Problem with Time Windows
Operations Research, 43, No. 2, 1995.

[8] Glover, F., and Laguna, M.
Tabu Search
from Reeves, C. R. (Editor)
Modern Heuristic Techniques for Combinatorial Problems
Wiley & Sons Inc., 1993.

[9] Glover, F., Taillard, E., and de Werra, D.
A User's Guide to Tabu Search
Annals of Operations Research, 41, 1993.

[10] Hansen, J. M.
Kunsten at klatre i træer
FOFT Nyt no. 2, Forsvarets Forskningstjeneste, 1996.

[11] Kolen, A., Rinnooy Kan, A. and Trienekens, H.
Vehicle Routing with Time Windows
Operations Research, 35, 1987.

[12] Laarhoven van, P. J. M., and Aarts, E. H. L.
Simulated Annealing: Theory and Applications
Reidel Publishing Company, 1987.

[13] Pirlot, M.
General Local Search Heuristics in Combinatorial Optimization
Belgian Journal of Operations Research, Statistics, and Computer Science, Vol. 32 (1,2), 1992.

[14] Silver, E. A., Vidal, R. V. V., and de Werra, D.
A Tutorial on Heuristic Methods
European Journal of Operational Research, 5, 1980.

[15] Tsubakitani, S. and Evans, J. R.
An Empirical Study of a new Metaheuristic for the Traveling Salesman Problem
European Journal of Operational Research, 104, 1998.

[16] Vidal, R. V. V. (Editor)
Simulated Annealing Applied to Combinatorial Optimization
Control and Cybernetics, 25, 1996.

[17] Vidal, R. V. V. (Editor)
Applied Simulated Annealing
Lecture Notes in Economics and Mathmatical Systems, 396, 1993.