THALES UNIVERSITY OF TWENTE.

Generic Scheduling in Radar Systems

Appendices not included due to confidentiality

Author
Roelof Spijker

Supervisor UT
Johann Hurink

Supervisor Thales
Ruben Marsman

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors, Ruben Marsman from Thales and Johann Hurink from the university, for their advice and proofreading my thesis *many* times. Furthermore, I would like to thank all my coworkers at Thales for helping with small and large issues I encountered during the project. Finally, I would like to thank Jan-Kees van Ommeren for being a member of my graduation committee.

Introduction

A scheduler decides how to assign resources to various tasks in order to optimize some objective. In our specific instance we want to allocate time to different tasks a radar system needs to perform. The decisions that need to be made are to select which tasks to process and to determine the order in which the selected tasks are processed. The intent is to create a generic scheduler. The scheduler should only have (or need) information about the essential aspects relevant to schedule the tasks, such as release time, processing time, priority, etc. Currently the schedulers designed at Thales are relatively specific to a certain project. There are project-specific scheduling rules in them, which makes it hard to port such a scheduler to a different system without having to redesign it. The aim of creating a generic scheduler is to be able to use it on multiple systems without the need to redesign it completely or in large parts.

This subject has already been researched during my internship[1]. The focus there was to show whether or not it is possible to use a more generic approach to solve the scheduling problem. The conclusion was that it is possible, but finding an optimal solution in the available amount of time is infeasible. Therefore the main question we try to answer in this report is whether we can find solutions of acceptable quality in a feasible amount of time.

This report is divided into multiple chapters. The following chapter covers the problem analysis. Some background is introduced and a concrete problem description is provided. The third chapter deals with the different methods we propose to use in solving the problem described in chapter two. In the fourth chapter we discuss the approach used to evaluate the methods discussed in chapter three. Chapter five covers the implementation of the prototype used to evaluate the solution methods. In the sixth chapter we present and discuss the results of the computational experiments we conducted. Finally we offer conclusions and recommendations in chapter seven.

Problem Description

In this chapter we introduce the problem. We begin with describing the problem with its background and then give a more formal mathematical description of the underlying scheduling problem.

2.1 Background

There are different kinds of radar systems with different responsibilities. In this report we consider shipborne radar systems with military applications. We can roughly divide this group into three smaller groups based on the tasks a radar system is designed to handle. First there are systems designed for surveilling a volume of space, called search radars. Second there are systems designed to accurately track one or more targets and guide projectiles, called tracking radars. Finally there are systems that combine these two tasks, called multi-function radars. All three of these types of systems use the same basic principle of reflected radio waves. However, there are different approaches to using this principle. We consider two different approaches, Frequency Modulation Continuous Wave (FMCW) and Doppler radar.

An FMCW radar continuously transmits radio waves with a frequency changing over time defined by a predetermined pattern. A reflection of these radio waves implies the presence of an object. From the properties of this reflection, the shift in frequency due to the Doppler effect and the time passed between transmission and reception of the reflection, the radar system can determine the location, direction and speed of the object. To accurately detect objects a sweep is made over a certain frequency range which is then modulated by a certain known pattern and sent out over a period of time. We call such a sweep a burst.

A Doppler radar system works by transmitting electromagnetic pulses and listening for possible reflections of these pulses. A reflection implies the presence of an object. The radar system can determine the location of the object as well as its direction and speed by analyzing these reflections. Figure 2.1 illustrates the functioning of the Doppler radar. To accurately determine the properties of the object multiple pulses have to be transmitted. We denote the transmission of a set of zero or more pulses and the corresponding listening time as a burst.

To transmit and receive the radar system uses an antenna. We distinguish between the physical antenna and the system used to control the antenna, which we denote the antenna system. The antenna is used to transmit the individual

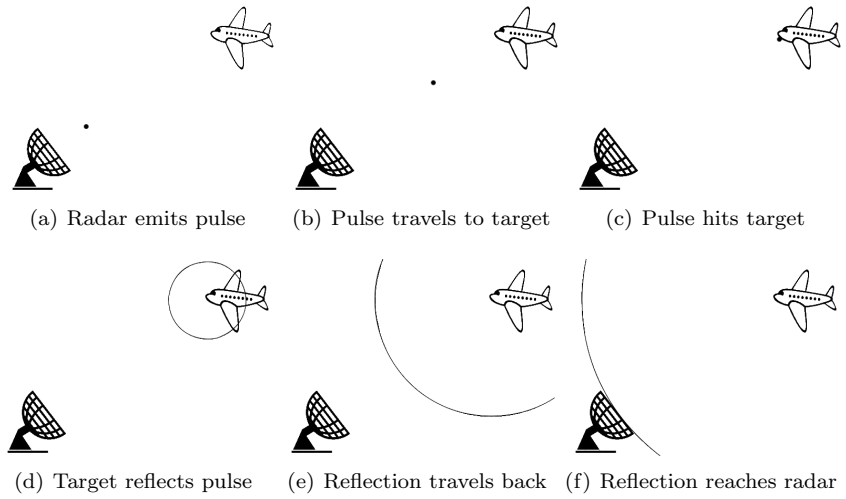

Figure 2.1: Doppler Radar Operation

pulses and listen for reflections. The antenna system processes the bursts and can only process one burst at a time. The tasks the radar system has to perform, like surveilling a volume of space, tracking a certain target or guiding a missile, all consist of one or more bursts for the antenna system to process. In order to complete the task all bursts that make up the task have to be processed. A radar system may have to perform many of these tasks in parallel. For instance we might want to search for new targets by sweeping 360 degrees around us, while keeping track of a number of targets we have found in the past and guiding a missile to a target we wish to engage. Since the antenna can only process one burst at a time we have to decide the order in which we process the bursts on the antenna and the exact moment at which we process each burst. The tasks to be performed are requested by a user, who can request more tasks than can be executed in the time available. Therefore we might also have to decide to reject certain requests.

Additionally there are some constraints to be considered. First, the requests made are time-sensitive. The user does not want to wait too long for a search he requested (search tasks are usually continuous, we always want to look around us) and if we wait too long when tracking a target it might make an unexpected maneuver and we might lose it. Therefore there is a time window associated with each burst within which it must be processed. Second, some tasks are more important than others. Missing an update on tracking a slow moving target far away is less important than missing an update when guiding a missile. The user can indicate importance by assigning priorities to tasks. Third, bursts might depend on each other. For instance, we might want to use the results of a burst measuring the activity of electronic countermeasures to determine the optimal parameters for a subsequent tracking burst. Therefore there can exist a minimum and maximum amount of time between two related bursts. In the mentioned example the minimum would serve to allow enough time to analyze the first burst and the maximum would serve to make sure that by the time we transmit the second burst the information we gained from the first is

still relevant. Finally the length of a burst often only becomes known shortly before it is transmitted. This is due to the fact that the burst length depends on parameters that have to be calculated and depend in turn on e.g. the position of the ship on the waves which is hard to predict. Since these true lengths only become known shortly before transmission they can not be taken into account when the decision on the bursts has to be made. However, we do have to deal with the consequences of the change in duration.

To some extent the above described problem is already being solved in every radar system, since these decisions need to be made in order for a radar system to operate. However, the problems that are actually being solved are special instances of the problem that are specific to the type of radar system they are solved in. This approach has as a drawback that the problem has to be solved again for each new radar system, keeping in mind its specific requirements. This increases the development time of new radar systems as well as increasing maintenance costs. Our aim is to design a generic method instead that solves the part of the problem that all types of radar systems have in common. If a specific system requires more or different functionality this part of the problem can be solved by an extra layer on top of the generic method. This limits the amount of work to be done to implementing the additional features and defining correct parameters for the generic method.

2.2 Problem Definition

The problem can be split up into two parts: the scheduling problem and the change in duration problem. The following subsections provide a short introduction to the two problems, connecting the terms used in sketching the background to more familiar scheduling terms.

2.2.1 Scheduling Problem

We have a single machine (the antenna) scheduling problem. Jobs (Tasks) arrive over time and consist of operations (bursts). The machine can only process one operation simultaneously and the processing of an operation cannot be interrupted. The operations that a job consist of have to be processed in a given order. Each operation has an associated window within which it has to start. Furthermore there can be relations between operations, indicating there has to be a minimum and/or a maximum amount of time between them. These relations can also exist between operations belonging to different jobs. Every job has an associated priority, indicating its importance. To schedule a job we have to schedule all of the operations of which the job consists. It is allowed to reject jobs. Each operation has an associated processing time, the expected time required for the machine to process the operation. The problem is then to make the following two decisions: which jobs do we reject/accept and when exactly do we start the operations belonging to the jobs that we chose to accept. Our objective is to minimize the amount of rejected jobs, respecting priorities, and minimize the makespan (maximum completion time) of the schedule of the accepted jobs.

2.2.2 Duration Change Problem

In Section 2.1 we stated that at the time of scheduling only expected processing times are known. Once the schedule is being executed the actual duration of each burst becomes known shortly before the scheduled transmission of the burst. Whether or not such a change leads to a direct problem depends on the nature of the change and the situation. Whenever the actual duration is shorter than the expected duration there is not directly a problem, the transmission is finished before the next burst is scheduled to start. If this is the case we might want to check if it is possible to start subsequent bursts earlier, since that leaves more room for possible changes in duration for these bursts. When the actual duration of a burst is longer than the expected duration and the longer duration causes the transmission of the burst to finish after the next burst is scheduled to start, we have a direct problem. The schedule has to be updated to accommodate these changes. This means subsequent bursts have to be delayed and/or rejected. When the actual duration of a burst is longer than the expected duration but the burst still finishes transmission before the next burst is scheduled to start, no action has to be taken.

We should take the subsequent nature of these problem into account when searching for a solution to the scheduling problem. It is of interest to develop methods to solve the scheduling problem which already take into account that the durations of bursts might change afterwards.

2.3 Problem Analysis

In this section we discuss the problems in more depth. We discuss the place of the scheduler in the radar system, look at the input we get from the system, what our output should look like and how we can compare different solutions. We start with some additional background information specifying the role and place of our problems in the system.

2.3.1 Place in the System

In this subsection we discuss the parts of the radar system that are relevant to the problem we are trying to solve. There is a system, we denote it *Radar Management (RM)*, that deals with the outside world. It communicates with the user of the system (e.g. an actual operator) and other systems present on the ship. It also relays tasks requested by these outside entities to the correct subsystem inside of the radar system. For instance, a radar system with multiple physical antennas also has multiple subsystems to control these. RM ensures that tasks are routed to the correct subsystem. Each of these subsystems has a component which we denote as *Waveform Calculation (WFC)*. During the design phase of this component every possible type of task that might have to be executed by this subsystem is specified as a waveform. A waveform is a collection of bursts in a specific order. When a task request comes in, WFC translates it into an ordered list of bursts, determines the specific parameters suited for this specific situation and passes these on as a job request to the scheduler. The scheduler is then responsible for creating a schedule, sending status updates to the requesting party regarding their requests, initiating parameter calculation

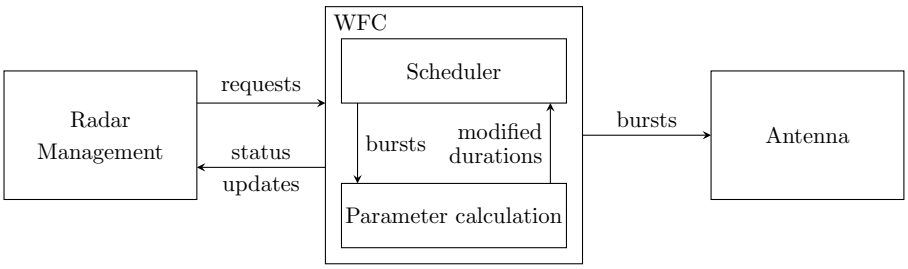

Figure 2.2: System Architecture

on time and sending the bursts to be transmitted to the antenna. Figure 2.2 depicts a schematic overview of the system.

2.3.2 Scheduling Problem

In this subsection we discuss the scheduling problem in greater depth. We consider the input to the scheduling problem, discuss what makes up a solution and how these solutions can be compared. We also discuss the complexity of the scheduling problem.

Input

As we have seen in the previous subsection, the scheduler receives requests from a requester. In this subsection we explain how these requests are constructed and what information they contain. These requests together with the contained information constitute the input to the scheduling problem as defined in Section 2.2.1.

Each request is a request for a specific job to be executed. A job j in turn consists of a set of n_j operations $O_j = \{o_{j1}, o_{j2}, \ldots, o_{jn_j}\}$, which need to be executed in this order. Furthermore, each job has a priority p_j which represents its relative importance. These priorities are chosen from a finite set of n_P priorities $P = \{\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_{n_P}\}$, such that $\bar{p}_1 < \bar{p}_2 < \ldots < \bar{p}_{n_P}$. We denote J_k as the set of jobs with $p_j = \bar{p}_k$. Each operation o has a window $[\alpha_o, \beta_o]$ within which it has to start.Furthermore, it has an expected processing time p_o and a set of $n(o)$ relations $R_o = \left\{(o_{r_1}, T^-_{o,o_{r_1}}, T^+_{o,o_{r_1}}), (o_{r_2}, T^-_{o,o_{r_2}}, T^+_{o,o_{r_2}}), \ldots, (o_{r_{n(o)}}, T^-_{o,o_{r_{n(o)}}}, T^+_{o,o_{r_{n(o)}}})\right\}$, where each relation is of the form $\left(o_r, T^-_{o,o_r}, T^+_{o,o_r}\right)$ and denotes a relation between the starting times of o_r and o respectively, stating there has to be a minimum difference of T^-_{o,o_r} and a maximum difference of T^+_{o,o_r} between the start of o_r and o. Formally this is expressed by the constraints: $s_{o_r} + T^-_{o,o_r} \leq s_o$ and $s_{o_r} + T^+_{o,o_r} \geq s_o$, where s_o denotes the starting time of operation o. Figure 2.3 illustrates this relation.

Solution

In the following we discuss the characteristics of a solution to the scheduling problem. A solution consists of a set of accepted jobs and a starting time for each of the operations that belong to an accepted job. Formally, we have for each job $j \in J$ a variable A_j, where $A_j = 1$ if j is accepted and $A_j = 0$ otherwise

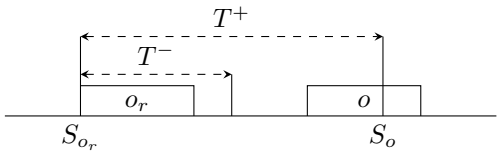

Figure 2.3: Relation between operations

and a starting time s_o for each operation o belonging to a job j with $A_j = 1$. We denote the set of jobs that are accepted by A with J^A. That is to say, $J^A = \{j \in J | A_j = 1\}$. We denote a schedule as a tuple $S = (A, s)$, where s is the set of starting times for the operations belonging to jobs in J^A. The starting times in s have to define a unique ordering of the operations of the accepted jobs. On the other hand, for any given ordering of the operations there are in general infinitely many possible starting times. However, since we are mainly interested in schedules where the operations start as early as possible, we can restrict our attention to that schedule for a given ordering, where each operation starts as early as possible. If we can construct an efficient method to find the set of earliest possible starting times given a certain ordering of operations, we can uniquely define a schedule as a tuple $S = (A, \vec{O})$, where $\vec{O}$ denotes an ordering of the operations of the jobs in J^A. By confining ourselves to orderings instead of starting times, we can drastically limit the amount of possible solutions we have to consider, since there are infinitely many different sets of starting times, but only $n!$ possible orderings of n operations.

In the following we show that finding the earliest possible starting times for a given ordered set of operations is equivalent to solving a longest path problem. Finding longest paths can be accomplished efficiently ($O(nm)$, where n is the number of vertices and m the number of edges) using the Bellman-Ford algorithm [2, 3]. Given an ordering $\vec{O} = (o_1, o_2, \ldots, o_n)$ of n operations, we consider a directed graph G on $n + 1$ vertices. We introduce one vertex s as a starting vertex and n vertices corresponding to the n operations. Next we introduce arcs such that the length of the longest path from s to a vertex o_i corresponds to the earliest possible starting time of operation o_i if all requirements are met. The earliest possible starting time of an operation o_i is influenced by a number of constraints. First, the operation cannot start before the start of its window: $s_{o_i} \geq \alpha_{o_i}$. Second, the operation cannot start before the preceding operation has finished: $s_{o_i} \geq s_{o_{i-1}} + p_{o_{i-1}}$. Third, the operation cannot start before $T^-_{o_i,o_r}$ time has expired since the start of a related operation o_r, i.e. for all $\left(o_r, T^-_{o_i,o_r}, T^+_{o_i,o_r}\right) \in R_{o_i}$ we have the constraint: $s_{o_i} \geq s_{o_r} + T^-_{o_i,o_r}$. Finally, the operation cannot start more than $T^+_{o_i,o_r}$ after a related operation o_r is set to start, i.e. for all $\left(o_r, T^-_{o_i,o_r}, T^+_{o_i,o_r}\right) \in R_{o_i}$ we have the constraint: $s_{o_i} \leq s_{o_r} + T^+_{o_i,o_r}$. The first constraint can be expressed by an arc from the start vertex s to vertex o_i with weight α_{o_i}. The second constraint can be expressed by an arc from vertex o_{i-1} to vertex o_i with weight $p_{o_{i-1}}$. The third constraint can be expressed by one arc from vertex o_r to vertex o_i with weight $T^-_{o_i,o_r}$. The last constraint can be expressed by one arc from vertex o_i to vertex o_r with weight $-T^+_{o_r,o_i}$. To clarify these relations, we consider the following example.

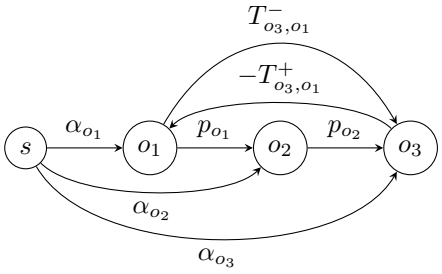

Figure 2.4: Graph representation of earliest possible starting times

Example 2.3.1 We have a single job j with 3 operations o_1, o_2 and o_3. Furthermore, there is a maximum and a minimum relation between operations o_1 and o_3. Figure 2.4 shows the graph corresponding to the example.

If we calculate now in the resulting graph the longest path l_{o_i} from s to all other vertices o_i ,$(i = 1, 2, \ldots, n)$, the schedule where operation o_i starts at time l_{o_i}, $(i = 1, 2, \ldots, n)$, has ordering $(o_1, o_2, \ldots, o_n)$, respects the lower bounds of the time windows and all relations and leads to the earliest possible starting times fulfilling all these constraints. By checking if these starting times also respect the upper bounds of the time windows, we can either conclude that we have found the earliest possible feasible schedule or that no feasible schedule exists for this ordering.

In Section 2.2.1 we introduced some objectives for the solution to the scheduling problem. Our primary aim is to schedule as many jobs as possible, respecting priorities. Given two solutions $S_1 = (A_1, s_1)$ and $S_2 = (A_2, s_2)$ we consider the number of jobs of each priority scheduled in descending order of priority. We prefer S_1 to S_2 if in the first place where these numbers differ, the number of S_1 is greater than that of S_2. Formally we denote for a schedule $S = (A, s)$ by $n_k(S)$ the number of accepted jobs of priority $\bar{p}_k$, i.e. $n_k(S) = |\{j \in J_k | A_j = 1\}|$. We say $S_1 > S_2$ if and only if there exists an $r \geq 1$ with $n_k(S_1) = n_k(S_2)$ for $k = 1, \ldots, r-1$ and $n_r(S_1) > n_r(S_2)$. $S_1 > S_2$ now signifies that we prefer S_1 above S_2. We note that S_1 and S_2 are equally good if and only if $n_k(S_1) = n_k(S_2)$ for all k. When two solutions are equally good in terms of the above defined preference relation, we prefer the solution with the smallest makespan. If there are multiple solutions which are equally good in terms of the above defined preference relation and have the same makespan, we prefer the most robust solution. Robustness is a quality we define in regards to the duration change problem as described in Section 2.2.2. There it is explained that the duration of an operation can change after the schedule is created. The impact these changes have depends on the robustness of the schedule.

We define robustness in terms of *float*. Float is a concept used in project planning, specifically critical path analysis [4, p. 434]. In project planning problems we have a set of activities that need to be completed in order to complete the project. These activities can depend on each other, imposing some constraints on the order in which they may be performed. We refer to the completion of a set of activities as an event. Two special events are the initial event where no activities have finished, the start of the project, and the final event where all activities have finished, the end of the project. Two important concepts are the

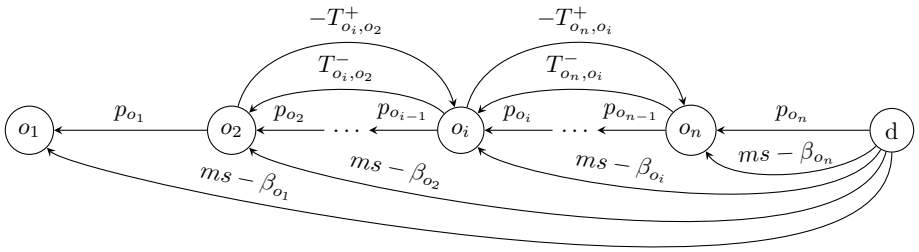

Figure 2.5: Late event time graph

early event time ($ET(o)$) and the *late event time* ($LT(o)$), denoting the earliest time at which event o can occur and the latest time at which event o can occur without delaying the completion of the overall project respectively. The early event times correspond to the earliest possible starting times we determine for operations as discussed above. From these earliest possible starting times we find a makespan ms for the schedule, the maximum completion time. The late event times then correspond to the latest possible time each operation can start without increasing the makespan of the schedule. Determining the late event times is done in a similar fashion to determining the earliest possible starting times, through solving a longest path problem.

Given an ordering $\vec{O} = (o_1, o_2, \ldots, o_n)$ of operations, we can define the late event time of an operation o_i as follows:

$$LT(o_i) = \min \begin{cases} LT(o_{i+1}) - p_{o_i} & \\ LT(o_j) - T^-_{o_i,o_j} & \forall o_j : (o_i, T^-, T^+) \in R_{o_j} \\ LT(o_h) + T^+_{o_h,o_i} & \forall o_h : (o_h, T^-, T^+) \in R_{o_i} \\ \beta_{o_i} & \end{cases} \tag{2.1}$$

for $i = 1, 2, \ldots, n-1$. Note that the last operation o_n cannot start any later than $ms - p_{o_n}$ and so we set $LT(o_n) = ms - p_{o_n}$. For all other operations o_i, we now have to find the values $\bar{l_{o_i}}$, which represent the minimum amount of time that has to pass between their start and ms. Similar to the case for the earliest starting times, we define a corresponding graph to find these minimum amounts of time. We begin by adding a dummy vertex d, corresponding to an operation that starts directly after the schedule has finished at ms. Furthermore, we add a vertex for each operation in the ordering. We add an arc between d and o_n of length p_{o_n} and an arc between o_{i+1} and o_i of length p_{o_i} for $i = 1, 2, \ldots, n-1$. For each operation o we add an arc between o and every related operation o_r of length T^-_{o,o_r} and an arc between o_r and o of length $-T^+_{o,o_r}$. Finally we add an arc between d and each operation o of length $ms - \beta_o$, to respect the latest possible starting time β_o of operation o. We note that the longest path $\bar{l_o}$ between d and vertex o in this graph, corresponds to the minimum amount of time that has to pass between the start of o and ms. We can now define $LT(o_i)$ by $LT(o_i) = ms - \bar{l_{o_i}}$. Figure 2.5 illustrates an example of such a graph.

The early and late event times allow us to define *free float* and *total float*.

Definition 2.3.1. The Free Float of an operation o_i, $FF(o_i)$, is defined as follows: $FF(o_i) = ET(o_{i+1}) - ET(o_i) - p_{o_i}$.

Definition 2.3.2. The Total Float of an operation o_i, $TF(o_i)$, is defined as follows: $TF(o_i) = LT(o_{i+1}) - ET(o_i) - p_{o_i}$.

Until now we have looked at the amount of float available when the makespan has to stay the same. However, makespan minimization is only the secondary objective. Our primary objective is to maximize the number of jobs scheduled. Therefore, if we want to examine by how much the duration of an operation may increase without leading to infeasibility, we are interested in the available float of the operations where the given sequence stays feasible, but the makespan is allowed to increase. For this purpose we introduce *latest feasible event time* ($LFT(o)$). This corresponds to the latest possible time an operation can start without making the schedule infeasible. These times can be determined by solving the same longest path problem as we did to find the late event times, only now we note that the last operation can't finish any later than $\beta_{o_n} + p_{o_n}$, instead of ms. If we replace ms by $\beta_{o_n} + p_{o_n}$ in Figure 2.5 we obtain the graph used for finding $LFT(o)$. The longest path $\tilde{l_o}$ between d and vertex o in this graph, corresponds to the minimum amount of time that has to pass between the start of o and $\beta_{o_n} + p_{o_n}$. This allows us to define $LFT(o)$ by $LFT(o) = \beta_{o_n} + p_{o_n} - \tilde{l_o}$. We can then use this quantity to define *feasible float*.

Definition 2.3.3. The Feasible Float of an operation o_i, $PF(o_i)$, is defined as follows: $PF(o_i) = LFT(o_{i+1}) - ET(o_i) - p_(o_i)$.

When the duration of an operation changes, the operations preceding it should be fixed in place, meaning their starting times, early event times, late event times and latest feasible event times are all the same and can no longer be changed. The reason is that at that point these operations are too close to their execution time to be adjusted (or have already been processed). These fixed operations may influence the time windows of operations that still have to be scheduled though. Therefore, we do have to include them when determining the earliest, latest and latest feasible event times of later operations. Now we can use these three measures to determine the effects of the change in duration.

Let us consider operation o_i whose duration changes from p_{o_i} to $p'_{o_i} = p_{o_i} + \Delta_{o_i}$. We then distinguish the following cases:

Case 1: $FF(o_i) \geq \Delta_{o_i}$
If this is the case then p_{o_i} can increase by Δ_{o_i} without affecting the rest of the schedule. By definition, $FF(o_i) = ET(o_{i+1}) - ET(o_i) - p_{o_i}$, so $ET(o_{i+1}) - ET(o_i) - p_{o_i} \geq \Delta_{o_i}$. Now since $p'_{o_i} = p_{o_i} + \Delta_{o_i}$, we have $ET(o_{i+1}) \geq ET(o_i) + p'_{o_i}$ and so the given schedule remains feasible. In this case we don't need to take any action.

Case 2: $TF(o_i) \geq \Delta_{o_i} > FF(o_i)$
If this is the case then an increase of p_{o_i} by Δ_{o_i} makes the schedule where every operation starts at its earliest possible starting time infeasible, due to $\Delta_{o_i} > FF(o_i)$ the same derivation as in case 1 would yield $ET(o_{i+1}) < ET(o_i) + p'_{o_i}$. However, since $TF(o_i) \geq \Delta_{o_i}$ and $TF(o_i) = LT(o_{i+1}) - ET(o_i) - p_{o_i}$, we find that the following operation can still start at or before its latest possible starting time $LT(o_{i+1})$. From the definition of the latest possible starting times it then follows that the schedule remains feasible and the makespan is not altered.

Case 3: $PF(o_i) \geq \Delta_{o_i} > TF(o_i)$
If this is the case then an increase of p_{o_i} by Δ_{o_i} makes any schedule with

the same ordering and makespan infeasible. Since the following operation will not be able to start at or before its latest possible starting time. However, since $PF(o_i) \geq \Delta_{o_i}$ there does exist a schedule with the same ordering, but greater makespan that is still feasible. From the definition of feasible float $PF(o_i) = LFT(o_{i+1}) - ET(o_i) - p_(o_i)$, we find that the following operation can still start at or before its latest feasible starting time, it then follows from the definition of latest feasible starting time that a feasible schedule exists on this ordering.

Case 4: $PF(o_i) < \Delta_{o_i}$
If this is the case then p_{o_i} can't increase by Δ_{o_i} without causing the schedule to become infeasible, because the following operation will not be able to start at or before its latest feasible starting time. In this case we can't simply move operations back. We have to perform rescheduling and perhaps reject this or upcoming jobs.

Based on the above discussion, we define the robustness of a schedule in terms of feasible float. We define the feasible float of a schedule $S = (A, s)$ as the smallest change in duration of any operation belonging to a job in J^A that makes the schedule infeasible.

Definition 2.3.4. The feasible float of a schedule S, $PF(S)$, is defined as follows: $PF(S) = \min\{PF(o)\}$ over all o belonging to a job in J^A. Furthermore, we call solution S_1 more robust than solution S_2 if $PF(S_1) > PF(S_2)$.

Complexity

In the following we discuss the computational complexity of the considered scheduling problem. Specifically, we proof the following theorem.

Theorem 2.3.1. *The stated scheduling problem is NP-Hard in the strong sense.*

Proof. We show its NP-Hardness by a reduction from 3-PARTITION. First we introduce the 3-PARTITION problem. Given a multiset of $3m$ integers $S = \{x_1, x_2, \ldots, x_{3m}\}$, the problem is to decide whether S can be partitioned into m disjunct subsets $S_1, S_2, \ldots, S_m$ such that the sum of the elements of each subset is equal, i.e. $\sum_{x_i \in S_k} x_i = B$ for all $k \in \{1, \ldots, m\}$ where $B = (\sum_{x_j \in S} x_j)/m$. The subsets $S_1, S_2, \ldots, S_m$ must form a partition in the sense that they are disjoint and cover S. Given an instance $S = \{x_1, x_2, \ldots, x_{3m}\}$ of 3-PARTITION, we define a corresponding instance of the scheduling problem with $4m$ jobs. The first m jobs are dummy jobs d_i each consisting of one operation o_{d_i} with the following parameters: $p_{o_{d_i}} = 0$, $\alpha_{o_{d_i}} = \beta_{o_{d_i}} = iB$, $i = 1, 2, \ldots, m$. The remaining $3m$ jobs each consist of one operation o_j with the following parameters: $p_{o_j} = x_j$, $\alpha_o = 0$ and $\beta_o = mB$, $j = 1, 2, \ldots, 3m$. We note that the dummy jobs partition the timeline in m separate pieces, as depicted in Figure 2.6. A solution to this scheduling problem in which every job is scheduled and which has a makespan of mB will have each of the m pieces of the timeline completely filled with operations, since the sum of the processing times of the operations is mB and each of the m pieces is of length exactly B. If we define S_i to be the set of operations starting between $(i-1)B$ and iB, the sets $S_1, S_2, \ldots, S_m$ are disjoint and cover the set of all operations. Furthermore $\sum_{o_k \in S_i} o_k = B$ for each $i \in \{1, \ldots, m\}$ and so we have a yes-instance of the corresponding 3-PARTITION problem. On the other hand,

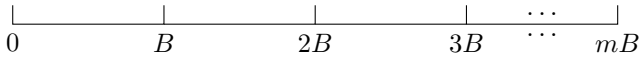

Figure 2.6: Timeline partitioned by dummy jobs

any optimal solution in which not every job is scheduled corresponds to a no-instance of the corresponding 3-PARTITION problem. Since we optimize with respect to the amount of jobs scheduled, the only reason for a job not to be scheduled is that it does not fit. This implies there is no distribution of the jobs over the pieces of the timeline such that each of them is exactly filled. Any solution with all jobs scheduled but makespan greater than mB will have an operation that starts at exactly mB, since operations cannot start after mB (due to $\beta_o = mB$) and any operation starting before mB will have to finish before mB as well (due to the dummy operation d_m starting at mB). We can then use the same argument as in the case where not all jobs are scheduled to show this also corresponds to a no-instance of the corresponding 3-PARTITION problem. Combining these three results shows that we have a yes-instance for 3-PARTITION if and only if we find a solution in which all jobs are scheduled with makespan mB to the corresponding scheduling problem. Since 3-PARTITION is shown to be NP-Hard in the strong sense [5], so is our scheduling problem. □

2.3.3 Duration Change Problem

In this subsection we discuss the duration change problem in greater depth. We consider the input to the problem and discuss what makes up a solution.

Input

In a given schedule of the scheduling problem, each operation represents a burst to be processed by the antenna. Shortly before this burst has to be processed parameters for the burst are determined. These parameters can influence the duration of the burst. When this is the case and the burst duration is affected, we have to solve the duration change problem. This may lead to an updated schedule which is being executed from that point on. Shortly before the next burst is scheduled to be transmitted according to this updated schedule, parameter calculation is again performed and, if necessary, the schedule is updated again. In this manner we iteratively change the schedule by solving a sequence of duration change problems. As a consequence, the input to the duration change problem consists of a schedule $S = (A, s)$ as defined in Section 2.3.2 and an altered processing time p'_o for one operation o in the schedule.

Solution

A solution to the duration change problem consists of a (possibly) modified schedule which accommodates the change in duration. More formally, the solution to the problem consists of a new schedule $S' = (A', s')$, where $J^{A'} \subseteq J^A$ and s' may have different starting times compared to s for operation o and operations starting after o. Since the solution to this problem is again a schedule,

the same methods as discussed in Section 2.3.2 can be used to compare the quality of two solutions.

Complexity

The complexity of the problem on its own is the same as that of the scheduling problem. Even though we are given a current schedule, there is no reason for the solution to use the same order of operations or the same set of accepted jobs. We simply have a (slightly smaller) scheduling problem to solve with some additional constraints, due to the fact that the part of the schedule before the altered operation is fixed.

Methods

In this chapter we discuss possible solution approaches to the scheduling problem as well as the duration change problem.

3.1 Scheduling Problem

In this section we discuss the possible solution approaches to the scheduling problem. We consider different kinds of approaches and draw a comparison between them.

3.1.1 On-Line

In the previous chapter we stated that requests for jobs arrive over time which makes our problem an on-line problem. Jobs arrive continuously over time, however we do require them to be requested some amount of time before they are allowed to start. Let us call this time the *minimum request delay* and denote it by m_{rd}. Since we do not want the machine to be idle and since jobs can start shortly (the minimum request delay) after they are requested, we can't postpone making scheduling decisions. If there are requested jobs available, we always want to have jobs scheduled for the immediate future. On the other hand we don't want to commit to a certain schedule too far in the future, due to the possibility of higher priority jobs being requested in the near future. This means that in the extreme case it may happen that we have to determine a new schedule every m_{rd} time units (in the case where each time a new job arrives directly after determining the schedule), to ensure the machine does not become unnecessarily idle and to ensure we make decisions about arriving jobs in time. Let us call the i^{th} time we determine a schedule the i^{th} *scheduling run*, let t_i denote the time at which the i^{th} scheduling run takes place and let c_i denote the time between the $(i-1)^{\text{th}}$ and i^{th} scheduling run: $c_i = t_i - t_{i-1}$. At time t_i we have to consider the schedule we have so far, which in general extends beyond t_i, and at the least the requests for jobs that wish to start before the next scheduling decision at t_{i+1}. Making these scheduling decisions takes a certain amount of time. The exact amount depends on the current schedule, the amount of requests, the amount of operations each requested job consists of and the level of inter-relatedness (in terms of the relations discussed in Section 2.2.1) between the operations. Let s_i denote the maximum amount of time required to find a feasible schedule in scheduling run i. During the i^{th} scheduling run we only need to consider jobs that wish to start at or after $t_i + s_i$, because for jobs

#	α_o	β_o	p_o	priority
1	0	0	$l_i/5$	1
2	0	l_i	$l_i/5$	5
3	0	l_i	$l_i/5$	5
4	0	l_i	$l_i/5$	5
5	0	l_i	$l_i/5 + 2\epsilon$	5
6	$l_i + \epsilon$	$l_i + \epsilon$	2ϵ	5

Table 3.1: Operation parameters for Example 3.1.1

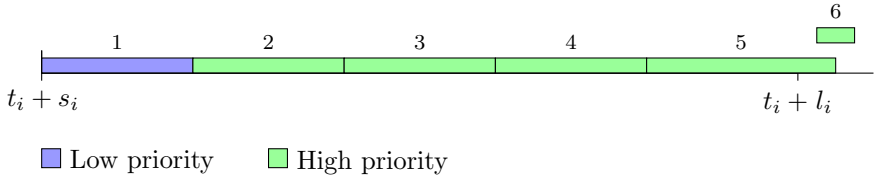

Figure 3.1: Resulting schedule for Example 3.1.1

that wish to start earlier our decision may not be available in time. Furthermore we have to consider each job that wishes to start before $t_{i+1} + s_{i+1}$, because in the next scheduling run it may be too late to make a decision for these jobs. All jobs we consider that start after $t_{i+1} + s_{i+1}$ may have to be reconsidered in the following scheduling run. Let l_i denote the amount of time we *look ahead*, that is, we consider in the i^{th} run jobs whose starting time window overlaps with $[t_i + s_i, t_i + l_i]$. As we have seen l_i has to be chosen in such a way that jobs that wish to start before $t_{i+1} + s_{i+1}$ are considered. This is represented by the following constraint: $l_i \geq t_{i+1} + s_{i+1} - t_i$. It remains to determine appropriate values for the l_i. Before we deal with this problem, we first note that for every finite value of l_i we can construct an example where a better solution can be found by increasing l_i. The following example illustrates this fact:

Example 3.1.1 Consider 6 jobs, each consisting of a single operation. The parameters for these operations are specified in Table 3.1. The priority listed is the priority of the corresponding job. In this example we consider $t_i + s_i$ to be 0. The last job (number 6) won't be considered in the first scheduling run because its starting time window does not overlap with $[0, l_i]$. In the next scheduling run (somewhere before l_i, but after 0) we do consider job 6, at this point job 1 is fixed though. This means we have to reject job 6. In the optimal schedule we would have rejected job 1, which has a lower priority than job 6. Figure 3.1 shows the resulting schedule, job 6 is shown hovering above it's desired location.

For some systems choosing l_i sufficiently large that all jobs are always considered may be appropriate, e.g. if all requests are made shortly before they wish to start. We should note that the scenario from the example can still occur; if job 6 is requested after the scheduling run, the outcome is the same. In a system where all requests for a long period are made at the start of the period, setting l_i in the above mentioned way may cause the processing time required for scheduling all of the operations to become too large to finish in time for

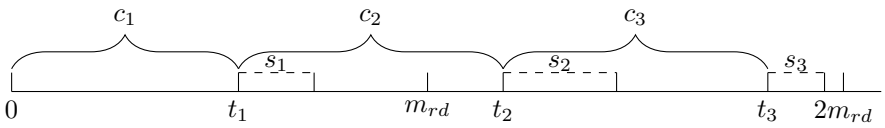

Figure 3.2: Scheduling intervals

the schedule to be executed. In a system where requests arrive over time the primary focus should be on obtaining a good schedule for the immediate future (until the next scheduling run). The operations that make up the schedule over this timespan are the operations that have to start processing shortly and can't be adjusted anymore. Furthermore, the more distant future is uncertain; that is to say, there may be additional requests which cause us to change the schedule in the future. This has the potential to render decisions we make now, based on our current information, obsolete and even counter-productive. Take for instance our example (Example 3.1.1); if we had considered every job in the first scheduling run, we would have rejected job 1 and accepted job 6. If a request then came in afterwards for a job of priority 10 that had to start at exactly $l_i + 2\epsilon$, we would have to reject job 6 in favour of this new job and we could have just scheduled job 1, but now both jobs are rejected. This leads us to conclude that the concrete value for the l_i parameters should be determined based on the system the scheduler is used in.

If we have chosen the l_i values, in each scheduling run, we have to solve an off-line version of the problem which we discussed in Section 2.2.1. In the off-line version of the problem we have a set of requested jobs and a current schedule to consider. In this context, we should note the relation between c_i and s_i. If we assume the amount of requests that wish to start in an interval is proportional to the size of that interval, then the larger we choose c_i, the longer it takes to perform the scheduling and the greater s_i will be. However, $c_i + s_i$ should be smaller than or equal to the minimum request delay m_{rd}, as the following example shows.

Example 3.1.2 Let $c_i + s_i > m_{rd}$ and consider a job request arriving immediately after t_{i-1} which wishes to start at $t_{i-1} + m_{rd}$. As the job is only considered in the next scheduling run starting at t_i, this means that the decision on whether or not to schedule this job is not made until $t_i + s_i$ which equals $t_{i-1} + c_i + s_i > t_{i-1} + m_{rd}$, which may be too late.

This places an upper bound on the value of c_i. On the other hand we do not want to choose c_i too small. In the $(i-1)^{\text{th}}$ scheduling run we consider each job that has a starting time window that overlaps with the interval $[t_{i-1} + s_{i-1}, t_{i-1} + l_{i-1}]$. If we choose a small value for $c_i = t_i - t_{i-1}$ then that implies we are solving roughly the same problem many times. When c_i is small compared to l_i the amount of *new* jobs in each scheduling run is relatively small. The majority of the jobs were already available in the previous scheduling run. Figure 3.2 illustrates and summarizes the different aspects discussed in this subsection.

3.1.2 General Solution Approach

In the previous chapter we have proven the scheduling problem to be NP-Hard in the strong sense (Theorem 2.3.1). In combination with the fact that we have a very limited amount of time to solve our scheduling problem, this leads us to conclude that an exact solution method is infeasible. This conclusion is backed up by the findings in [1]. Therefore, in the following we focus on heuristic solution methods to find a best possible solution in the time available.

As we have seen in the previous subsection, our scheduling problem consists of solving many small instances of the off-line version of the problem. Thus, to solve our scheduling problem we have to solve a series of these off-line scheduling problems. In this subsection we discuss the general outline of a solution approach to these off-line scheduling problems.

We can roughly divide the set of possible approaches in two: constructive approaches and local search approaches. Constructive approaches build a solution by starting with nothing and adding jobs one at a time, until no further jobs can be added. Local search approaches work by starting with some initial solution upon which they improve by making small changes to the solution to obtain different (better) solutions. For most problems finding an initial solution for a local search procedure is quite simple, however, for our problem even finding an initial feasible solution is not simple, since we can't simply take a random ordering of the operations (that will likely be infeasible). Therefore, we need to use a constructive approach to obtain an initial solution, after which we can apply a local search approach to improve on this solution as long as we have time left before the schedule has to start. In this manner we ensure we always have a feasible solution available (assuming there is always enough time to find a solution using a constructive approach). This way we can guarantee the machine does not become idle due to no schedule being available. At the point in time where we have to start executing our schedule, we can use the best schedule found so far. In the next subsections we discuss the various options for constructive and local search approaches.

3.1.3 Constructive Approaches

In this subsection we discuss different methods to find an initial solution using a constructive approach as well as their benefits and drawbacks.

Priority Rule

In Section 3.1.2 we explained that in a constructive approach we consider jobs one by one. Our primary objective with regards to solution quality is the amount of jobs scheduled, respecting priorities, as explained in Section 2.2.1. This objective implies that we never wish to reject a higher priority job in favour of any combination of lower priority jobs. The simplest manner to guarantee this behaviour is to consider the jobs in order of descending priority. So when considering a job, the only jobs already scheduled are of equal or higher priority and any job with lower priority has no effect on whether or not the job under consideration is accepted.

To schedule a job we have to schedule all of its operations, in the given order. We do this by considering the operations one at a time, determining a

place for them in the ordering. If a given operation does not fit anywhere, we perform backtracking. That is, we attempt to give the preceding operation of this job a different place in the ordering. If no different places for this operation are available, we proceed with the operation before that, continuing until we either find a place for each operation, at which point we accept the job and continue scheduling with the next job, or we can't find a different place for the first operation, in which case we reject the job and continue scheduling with the next job. This procedure is performed by Algorithm 1: Priority Rule which uses Algorithm 2: Operations in window and Algorithm 3: Place before as subroutines. The overall algorithm works as follows: The priority rule starts

Algorithm 1 Priority Rule

Procedure PRIORITYRULE($J, \vec{O}$)
1: Sort(J) //*Descending by priority*
2: **for all** $j \in J$ **do**
3: $k \leftarrow 0$
4: before[$\cdot$] $\leftarrow \infty$
5: **while** $k < |O_j|$ **do**
6: $o \leftarrow O_j(k)$ //*o becomes the* k^{th} *operation of j*
7: $(a, b) \leftarrow$ operationsInWindow($o, \vec{O}$)
8: $i \leftarrow$ before[k]
9: $i \leftarrow$ placeBefore($o, \vec{O}, a, b, i$)
10: **if** $i = -1$ **then**
11: **if** $k > 0$ **then**
12: $k \leftarrow k - 1$
13: **else**
14: reject(j)
15: $k \leftarrow |O_j|$
16: **end if**
17: **else**
18: before[k] $\leftarrow i$
19: $k \leftarrow k + 1$
20: **end if**
21: **end while**
22: **if** j was not rejected **then**
23: accept(j)
24: **end if**
25: **end for**

by sorting the jobs in descending order of priority (Alg.1 l.1) then the jobs are taken one at a time. For each job j we let k denotes the index of the operation we are currently considering (Alg.1 l.3). We initialize a value `before` for each operation, this value is used when backtracking to ensure the operation is put in a different place (Alg.1 l.4). Now we take the operations of job j one at a time in the given order and for each operation o we begin by determining the operations in the current ordering whose starting windows overlap with the starting window of o. This is performed by calling the `operationsInWindow` procedure (Alg.1 l.7).

The procedure `operationsInWindow` determines the index (in the ordering)

Algorithm 2 Operations in Window

Procedure OPERATIONSINWINDOW($o, \vec{O}$)

$i \leftarrow 0$
$a, b \leftarrow -1$
while $i < |\vec{O}|$ **do**
  $r \leftarrow \vec{O}(i)$
  if there exists x such that $x \in [\alpha_o, \beta_o]$ and $x \in [\alpha_r, \beta_r]$ **then**
    $b \leftarrow i$
  else
    if $b < 0$ **then**
      if $\beta_o < \alpha_r$ **then**
        $a \leftarrow i$
      else
        $b \leftarrow i$
      end if
    else
      break
    end if
  end if
  $i \leftarrow i + 1$
end while
if $b < a$ **then**
  $b \leftarrow a$
end if
return (a, b)

Algorithm 3 Place Before

Procedure PLACEBEFORE($o, \vec{O}, a, b, i$)

$i \leftarrow \min(i, b)$
while $i \geq a$ **do**
  $\vec{O} \leftarrow (o_1, o_2, \ldots o_i, o, o_{i+1}, o_{i+2}, \ldots, o_n)$
  if BellmanFord(graphRepresentation($\vec{O}$)) **then**
    return i
  else
    $\vec{O} \leftarrow \vec{O} \setminus \{o\}$
    $i \leftarrow i - 1$
  end if
end while
return -1

of the last operation whose starting window ends before the starting window of o begins and the index of the last operation whose starting window overlaps with the starting window of o. It does this by initializing the indices to -1 (Alg.2 l.2) and then iterating over the ordering. For the i^{th} operation r from the ordering we check if r's starting window overlaps with that of o (Alg.2 l.5). If it does, we set the end index, b, to r's index i. If it does not, we check if $b < 0$, because if $b < 0$ then b has not been changed since it has been initialized, meaning we are either still before the first operation whose starting window overlaps with that of o or none of the operations in the ordering has a window that overlaps with that of o and this is the first one after the window of o. If the former is the case, we set the begin index, a, to r's index i (Alg.2 l.10). If the latter is the case, we set the end index, b, to r's index i (Alg.2 l.12). If $b \geq 0$ then we have encountered an operation whose starting window does not overlap with that of o even though in the past we have found such operations. This means that at this point a and b are set as the begin and end indices correctly and so we stop iterating over the operations and return the indices we found (Alg.2 l.23). If we have iterated over all of the operations in the order without this occurring, we check if $b < a$, i.e. if the end index is smaller than the begin index. This only happens if the begin index, a, has been increased while the end index, b, has not. This means all of the operations have starting windows that occur completely before that of o. In this case we set the end index equal to the start index and return the pair.

We continue in the `PriorityRule` procedure. Now that we have the begin and end index of the "overlapping operations" the place for the operation can be determined. For this purpose the procedure `placeBefore` is used. It is given the current operation, the current ordering, the begin and end indices as computed by `operationsInWindow` and the value `before` discussed above as an argument.

The `placeBefore` procedure determines a place for the operation in the ordering before the index stored in `before` for this operation. It first tries to insert the operation at the last possible place, i.e. the minimum of the value in `before` and the end index computed by `operationsInWindow` (Alg.3 l.1). It does this by adding the operation in the ordering (Alg.3 l.3), creating a graph representation for this ordering and then use the Bellman-Ford algorithm to determine whether a feasible solution exists (Alg.3 l.4). If a feasible solution exists, the place where the operation was inserted is returned (Alg.3 l.5). If no feasible solution exists we attempt to insert the operation in an earlier place. If we can't find any place to insert the operation, a value of -1 is returned (Alg.3 l.11).

Back in the `PriorityRule` procedure, we check if a place was found for the operation (Alg.1 l.10). If a place was found, we store the index we found for the operation as the new value for `before` for this operation and move on to the next operation (Alg.1 l.18-19). If we have not found a place for the operation, we check if this is the first operation for this job (Alg.1 l.11). If it is the first operation, we reject the job and move on to the next job (Alg.1 l.14-15). If this is not the first operation, we track back to the previous operation for this job (Alg.1 l.12). If all operations have been processed and the job was not rejected, we accept the job and move on to the next job (Alg.1 l.22-23).

To analyze the complexity of the priority rule algorithm discussed above, we first consider the complexity of the `operationsInWindow` and `placeBefore` procedures, discussed in Algorithms 2 and 3 respectively. The `operationsInWindow`

procedure has a runtime complexity of $O(n)$ (where n denotes the amount of operations), the while loop is executed $O(n)$ times and all of the statements inside of the loop can be performed in constant time. The `placeBefore` procedure has a runtime complexity of $O(n^4)$, the while loop is executed $O(n)$ times, creating the graph representation requires $O(n^2)$ time and the Bellman-Ford algorithm runs in $O(n^3)$ time. The other operations only require constant time, making the total running time $O(n^4)$. The runtime complexity of the `PriorityRule` procedure is $O(n^5)$. Sorting requires $O(|J| \log |J|)$ time, everything inside of *both* loops is executed $O(n)$ times. We have just seen `operationsInWindow` requires $O(n)$ time and `placeBefore` requires $O(n^4)$ time. The rest of the statements require constant time to be executed. This brings the total to $O(n^5)$.

Shifting Bottleneck Heuristic

One popular heuristic in scheduling is the shifting bottleneck heuristic[6]. Before we describe how this heuristic can be applied to our problem, we first introduce a problem to which the shifting bottleneck heuristic is often applied, the job shop scheduling problem. An instance of the job shop scheduling problem consists of a set of machines and a set of jobs each of which has to be processed on a subset of the set of machines in a certain order. A job can only be processed on one machine simultaneously and the machines can only process one job simultaneously. The aim is to find a schedule which obtains the optimal value with respect to a given objective function. This problem is covered extensively in literature, because it has many practical applications and turns out to be a very hard problem.

The general idea of the shifting bottleneck heuristic is to find a sequence of jobs for the machines one by one. The order in which the machines are considered is determined by finding the *bottleneck machine* in each iteration. The manner in which this machine is found depends on the objective function. Finding an optimal (locally) sequence for this single machine is a much simpler problem than solving the overall problem. After a sequence has been found for the bottleneck machine, the other machines are re-optimized, i.e. an optimal sequence is determined for them again, one at a time while keeping the sequences of the other machines fixed. These steps are repeated until a sequence for all machines is found.

In our problem we have only one machine, we have many jobs though, all consisting of operations. Instead of applying the heuristic to machines we apply it to the jobs. In each iteration we take the job with the highest priority (corresponding to the bottleneck machine) and insert it into the schedule. That is, we find a place in the ordering for the operations that the job consists of instead of finding a sequence for a machine. If it is possible to add the job to the ordering, we perform a local search procedure on each of the jobs that were already scheduled before this iteration. We note that the shifting bottleneck heuristic is a combination of the priority rule and a local search approach. It builds a solution by adding jobs in the same fashion as the priority rule does, but it also applies local search after adding each job. So instead of applying local search to the complete solution, it is applied each time after a job is added to the solution up to that point. This makes the algorithm the same as that for the priority rule as given in Algorithm 1, except that after we accept a job on line 23 local search is applied.

The local search is performed by *reinserting* jobs into the schedule. Let $S = (A, s)$ denote the schedule after a job j is added. We then iteratively, for each job $a \in J^A \setminus \{j\}$, remove all the operations $o \in O_a$ from the ordering $\vec{O}$. Then we reinsert the job using the same mechanism as in the priority rule. We note that this does not lead to any rejections, since the situation before removal always remains feasible. Additionally we try to insert every already rejected job $r \in J \setminus J^A$ into the schedule using the same mechanism as in the priority rule. The complexity of this iterative improvement procedure is $O(n^5)$, due to the fact that for every operation belonging to a job preceding j (in descending order of priority) `placeBefore` is invoked. There are $O(n)$ such operations and each invocation of `placeBefore` requires $O(n^4)$. The iterative improvement procedure is executed once for every job, therefore the total added running time, relative to the priority rule, is $O(|J| \cdot n^5)$. We have seen in the previous subsection that the priority rule requires $O(n^5)$, therefore the shifting bottleneck heuristic requires $O(|J| \cdot n^5 + n^5) = O(|J| \cdot n^5)$.

Comparison

It depends greatly on the relative performance of the local search approach employed after finding an initial solution which method should be used for finding the initial solution. If the local search approaches are able to quickly find superior solutions the best option would be to just use the priority rule, since this provides a solution faster and leaves more time for local search. If on the other hand the improvement made by the local search approach is small, it is better to spend a little more time finding a better initial solution. How well the local search approaches perform depends on the solution space and the initial solution found. It is therefore very hard to decide which choice is best based purely on theory. Therefore we test different combinations of initial solution methods and local search approaches on a variety of scenarios to obtain a good comparison. These test results are presented in Chapter 6.

We should also note that the runtime complexities obtained in this section paint a somewhat distorted picture. In practical situations the size of the intervals discussed in Section 3.1.1 are small, this also makes it likely that the amount of operations is small. Thus the smaller order terms and coefficients we are ignoring in the asymptotic analysis can actually have a great impact on the running time.

3.1.4 Local Search Approaches

In this section we discuss different local search approaches. Recall from Section 3.1.2 that a local search approach takes a solution and changes it slightly to obtain a different solution. We call the set of solutions that can be obtained by slightly changing solution S the neighbourhood of S. Note, that the neighbourhood depends on the type of change made to the schedule. Thus, the same solution S can have different neighbourhoods depending on how we choose to change the solutions. The following example clarifies this concept of a neighbourhood.

Example 3.1.3 We consider a sequencing problem with 3 jobs: j_1, j_2 and j_3 with $p_1 = 8$, $p_2 = 5$ and $p_3 = 3$. We wish to minimize the sum of the completion times. Given an ordering of the jobs, we can easily find the start times for each job by starting the first job at 0 and consecutive jobs as soon as the preceding job finishes. Consider the ordering j_1, j_2, j_3. The sum of completion times for this solution is $p_1 + p_1 + p_2 + p_1 + p_2 + p_3 = 37$. Let us consider the local search heuristic where we manipulate solutions by swapping two adjacent jobs. This leads us to a neighbourhood consisting of two solutions, obtained by swapping j_1 and j_2, and j_2 and j_3 respectively. The first swap leads to solution j_2, j_1, j_3 with sum of completion times $p_2 + p_2 + p_1 + p_2 + p_1 + p_3 = 34$. The second swap leads to solution j_1, j_3, j_2 with sum of completion times $p_1 + p_1 + p_3 + p_1 + p_3 + p_2 = 35$. For both of these solutions we could then define a neighbourhood again, attempting to find better solutions. If we had instead chosen to manipulate solutions by only swapping non-adjacent jobs, we would have found a different neighbourhood for our initial solution, consisting solely of j_3, j_2, j_1 with a sum of completion times of $p_3 + p_3 + p_2 + p_3 + p_2 + p_1 = 27$.

We call the method of manipulating solutions the transfer function. The difference between local search heuristics is in the way the next solution is chosen from the neighbourhood. In the next subsections we discuss different local search approaches, afterwards we discuss possible transfer functions for our scheduling problem.

Descent methods

A descent method starts at a given feasible initial solution and iteratively chooses the first neighbour it encounters with better objective function, until the objective no longer can be improved. It can also be modified to choose the best among its neighbours instead of the first it finds or to take the best alternative from some random sample of neighbours. The problem with this heuristic in general is that it stops at the first local optimum, which can be far away from the global optimum and which depends greatly on the initial solution. The main advantage is that in general it converges to this local optimum fairly fast.

Simulated Annealing

In simulated annealing[7] we move from the current solution to a neighbouring solution with some probability that depends on the objective value of the current solution, the objective value of the neighbouring solution we are considering and a parameter which decreases over time known as the temperature. A neighbour is randomly selected from the neighbourhood. If it's objective value is better than that of the current solution, it becomes the new solution. If it's objective value is worse than that of the current solution, it becomes the new solution with some probability that depends on the temperature and the difference in objective value. The basic idea is that the temperature decreases over time and influences the probabilities in such a way that when the temperature is large neighbours with a worse objective value are selected with higher probability but this probability decreases as the temperature decreases. So over time the

algorithm becomes more and more likely to choose neighbours with a better objective value. Allowing the algorithm to select neighbours with worse objective values helps to avoid getting stuck in a local optimum. Even when the temperature gets close to 0 the probability of moving to a neighbour with a worse objective value stays positive (even though it tends to 0). Usually the algorithm is terminated when we are within some reasonable bound of the optimum or we reach some self-imposed bound on computational effort. Since we usually have a very small amount of time available to perform a local search heuristic, we need to choose our parameters very carefully to minimize the probability of ending up with a worse solution.

Genetic Algorithm

A genetic algorithm[8] starts with a random set of feasible solutions called the first or initial population. It then selects a small subset of this initial population (typically those with good objective values). This subset is then transferred to the next population. The rest of this new population is filled with offspring from this subset. The offspring is created using some function that combines two solutions into a new one which has attributes of both. Finally some additional solutions are selected from the previous population and added after they have been mutated. Mutation is achieved by randomly changing the solution, still obtaining a feasible solution. This process is repeated until a suitable solution is found or a bound on computational effort is reached. The introduction of mutation ensures the algorithm does not immediately converge to a local minimum. This method differs from our description of a local search heuristic in the sense that it works with sets of solutions and that there are multiple transfer functions. Mutation can be seen as a regular transfer function, combination is different since it requires two solutions as input. The main concept of finding new solutions by making small changes to (a set of) other solutions remains the same though. The main difficulty in this approach is maintaining feasibility through combination and mutation.

Transfer Functions

The choice of the transfer function has a great impact on the performance of the local search method. It defines which solutions are in the neighbourhood. The options for a transfer function are limited for our scheduling problem. Changes on the level of operations do not seem to be a proper choice, due to the requirement that operations within a job have to be ordered and the possibility of relations between operations. Therefore it seems we have to focus on the job level. We can't swap jobs like we did in the example in the beginning of this subsection, since they consist of operations. Two jobs may have a different amount of operations, making it impossible to really swap them. The operations of the jobs may be interleaved with each other and operations of other jobs, so swapping only the order is impossible as well. This leaves us with taking a job out of the schedule and inserting it again in a different place, like in the shifting bottleneck heuristic discussed in Section 3.1.3, and attempting to add the rejected jobs. Alternatively, if we have an accepted job j_1 and a rejected job j_2 with the same priority $\bar{p}_i$, we can take j_1 out of the schedule and insert j_2 instead. Whenever we change the solution in some way, we can attempt to

add previously rejected jobs to the schedule.

Comparison

It depends greatly on the amount of time we have available to run our local search heuristic, which local search heuristic is most suited for our problem. The descent algorithm likely performs better in the initial stages than the simulated annealing or genetic approach. Later on the latter two will catch up though. Tests have to show which heuristic is most suited. A parameter could be used to characterize which heuristic should be used for which specific system. This could be performed automatically, for instance by determining the quality of local optima or depending on the amount of time left. Alternatively the parameter could be determined for a system when it is designed. A search radar is much more predictable and easier to schedule than a complicated multi-function radar for instance. So it is likely that in the search radar there is a greater amount of time available to perform a local search heuristic, thus favouring the simulated annealing or genetic approach. Whereas for the multi-function radar finding an initial solution might take longer, leaving less time for local search, thus favouring the descent approach.

3.2 Duration Change Problem

In this section we discuss approaches to solving the duration change problem as defined in Section 2.3.3. In Section 2.3.2 we defined robustness in terms of float. These concepts also allow us to decide upon an appropriate response to a certain change in duration. For the following we consider the case that operation o_i has a change in duration from p_{o_i} to $p'_{o_i} = p_{o_i} + \Delta_{o_i}$.

The first three cases discussed in Section 2.3.2 can be handled by adjusting the starting times of operations. We can maintain the same ordering and only need to determine a new set of starting times. In the fourth case there is no feasible schedule using the current order with the new durations. This leaves us with the following options: *a*) change the order; *b*) reject a job or some jobs; or *c*) leave the schedule infeasible. There is no guarantee that changing the order works, since there may not be an ordering of these operations with these durations which supports a feasible schedule. We may also attempt to reschedule the operations after the affected operation, which may lead to a new schedule without requiring rejections. Depending on how quickly we can eventually perform the scheduling procedure, this may or may not be feasible in the time available. If we need to reject a job, this means that the feasible float of the changed operation o_i is not large enough to allow the change. As the feasible float is defined by $PF(o_i) = LFT(o_{i+1}) - ET(o_i) - p_(o_i)$ and since we can't influence $ET(o_i)$ anymore (at this point the past is already fixed), we are left with $LFT(o_{i+1})$, the latest feasible starting time of the operation following o_i. This latest feasible starting time is determined by the operations on the longest path $\tilde{l}_{o_{i+1}}$ between o_{i+1} and d in the graph used to determine the latest feasible starting times. For changing the schedule, we can restrict our attention to jobs that have operations on this longest path $\tilde{l}_{o_{i+1}}$, since removing an operation that is not on this longest path does not influence $LFT(o_{i+1})$ and thus does not make the schedule feasible. We have to find a collection of

operations C_i on the longest path such that their removal increases $LFT(o_{i+1})$ enough for the schedule to become feasible, i.e., $PF(o_i)$ has to be increased such that $\Delta_{o_i} \leq PF(o_i)$. Furthermore, we want to find the collection of operations with this property whose corresponding jobs have the lowest maximum priority. Let J_{C_i} denote the set of jobs to which the operations in C_i belong. That is, $J_{C_i} = \{j \in J | O_j \cap C_i \neq \emptyset\}$. Then among all the collections of operations C_i with this property we wish to find the collection for which $max_{j \in J_{C_i}}(\bar{p}_j)$ is lowest. If there are multiple collections with the same maximum priority, we choose the one with the smallest amount of jobs in the corresponding set of jobs, i.e., the one for which $|J_{C_i}|$ is minimal. If the maximum priority of the collection we find is larger than that of the job that o_i belongs to, we reject the job that o_i belongs to. Otherwise we reject the set of jobs J_{C_i} that the operations we have found belong to.

The last mentioned option is to leave the schedule infeasible. This can be done if the amount $\Delta_o - PF(o)$ is small and we expect future operations may have sufficiently lower durations than expected to compensate for this. Whether or not we want to use this option depends on how much risk we are willing to take and on the priority of the job the operation belongs to. If the priority of the job this operation belongs to is low, we risk having to reject a higher priority job later on if the duration of future operations does not decrease sufficiently. On the other hand, if the priority of the job is high the risk of rejecting a higher priority job later on is smaller (since the probability of a higher priority job occurring is lower). To make a decision we need additional information on the probability distribution of the duration of an operation and the amount of risk we are willing to take. These are things that are not necessarily the same for different radar systems though. Therefore, we can't give a general answer here and merely mention the possibility to implement it in specific systems.

3.3 Architecture

In Section 2.3.1 we discussed the place of the scheduler in the system. In this section we discuss the system architecture more thoroughly. The most important difference is that we split the scheduler up into several parts, representing the different functions the scheduler performs. First off we separate duration change handling from the scheduling. Second we split the scheduler in a generic and a project specific part. The generic part of the scheduler does the actual scheduling. It determines which jobs are accepted, in which order the operations belonging to those jobs are scheduled and when those operations start. Furthermore it makes sure these start times satisfy the constraints corresponding to starting time windows and relations between operations. If a specific project requires additional constraints or has additional demands on the schedule, beyond those identified in the requirements, they are to be handled by the project specific part of the scheduler. This handling is limited to modifying the jobs and operations before scheduling commences and changing the duration of operations after a schedule has been created, in order to employ the duration change mechanism. The actual scheduling is still performed by the generic part of the scheduler. Figure 3.3 depicts the system architecture. In this figure we see the requests arrive at the project specific part of the scheduler. This allows the project specific part to make adjustments to the requests before sending

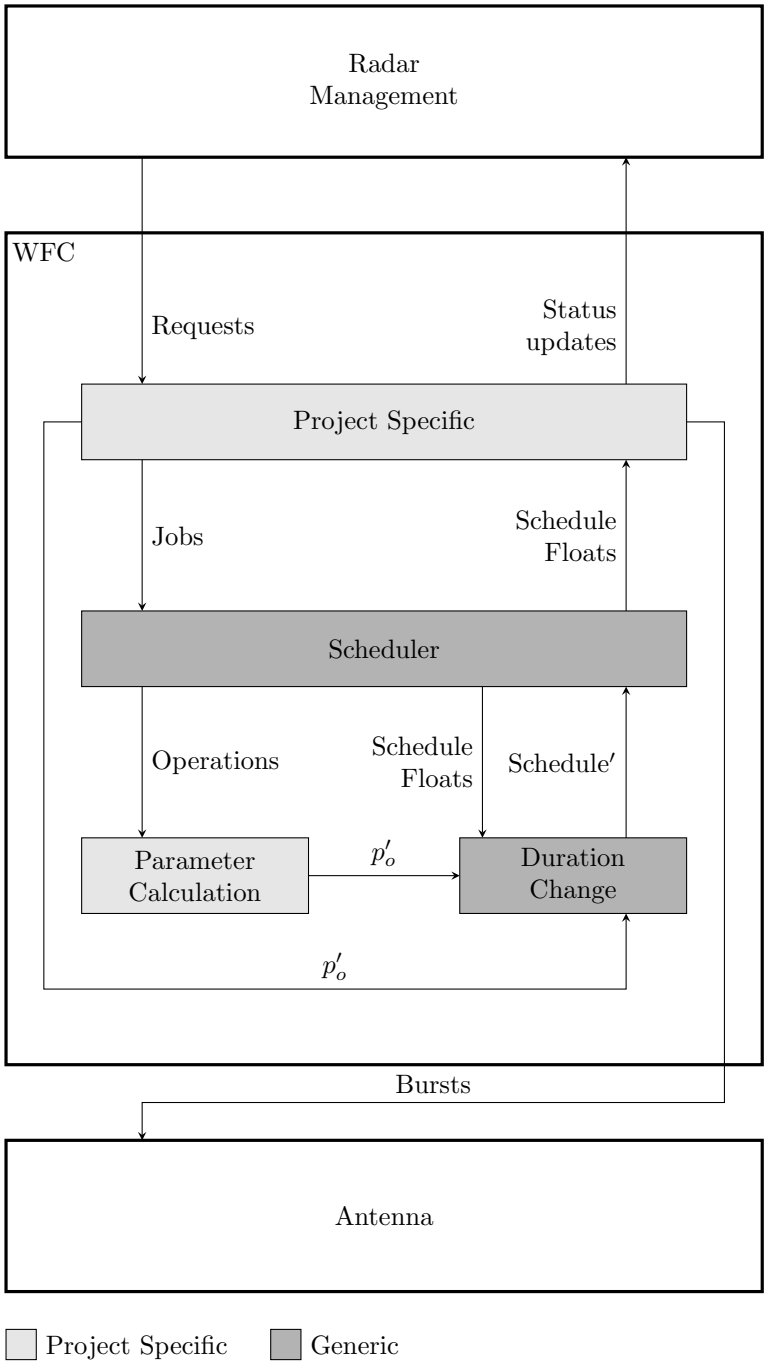

Figure 3.3: System Architecture

them along to the scheduler. The following example illustrates why this setup may be useful.

Example 3.3.1 We consider a radar system that has a certain job for which it asks that all operations have to be processed consecutively i.e. each operation has to start when the previous one finishes. The project specific part can modify the operations to enforce this constraint by creating relations between consecutive operations. That is, by creating a relation between operation o_2 that directly follows operation o_1, that states the minimum time between their starting times is equal to the maximum time between their starting times is equal to the duration of operation o_1.

Once these (possibly modified) requests arrive at the scheduler, the actual scheduling occurs. The result is a schedule and a set of floats for each operation. The schedule and float information is sent to the duration change handler, so that it can deal with possible changes in duration. The schedule and float information is also sent to the project specific part of the scheduler. This allows the project specific part to influence the schedule after it has been created by adjusting the duration of operations. The following example illustrates a scenario where this is used.

Example 3.3.2 Consider a radar system that performs a task which includes one operation that the system would like to perform for as long as possible. This operation still has a limited start window though and there are requests for jobs to start after it with higher priority. Now the requesting party has to choose a certain duration for the operation for it to be scheduled properly. If this is chosen too large, the operation does not fit and the job is rejected. Therefore the duration can't be chosen as large as the system might like it to be. Now let us assume the scheduler has created a schedule and the job the operation belongs to has been accepted. This schedule, including the float information is sent to the project specific part of the scheduler. The project specific part can now use the floats to determine how much the operation in question can be enlarged without making the schedule infeasible. It can then signal the duration change handler that the duration of the operation has changed by the determined value, which leads to a modified schedule in which the duration of the operation is as large as possible.

The scheduler also sends operations to the parameter calculation component shortly before they are scheduled to be processed. If parameter calculation leads to modified durations, these durations are communicated to the duration change handler. The duration change handler sends an updated schedule to the scheduler after having dealt with the change in duration. Status updates are sent back to Radar Management by the project specific part of the scheduler, since the project specific part may have changed the requests in such a way that there is no one to one correspondence between requests and jobs anymore. For the same reason, the bursts are also sent out by the project specific part of the scheduler.

Approach

In the previous chapter we discussed different methods that could be applied to our scheduling and duration change problem. In this chapter we discuss which methods are actually applicable to our specific problem. Furthermore we discuss the approach we take with regard to testing the performance of the methods we have chosen, both in quality and runtime.

4.1 Method Selection

We discussed different possible methods for both the scheduling and duration change problem in the previous chapter. In this section we determine which of those methods are suitable for our specific problems. We start with the methods for the constructive approach to the scheduling problem.

We only have to consider two options for the constructive approach, the priority rule and the shifting bottleneck heuristic. Both of these methods seem to be applicable to our problem. Computational experiments have to show whether or not one is to be preferred to the other.

There are more local search methods to choose from. We have the descent methods, simulated annealing and genetic algorithms. In our scheduling problem the time we have to perform the local search method is very limited. Furthermore, consistently finding the same solution given the same input is important (for testing purposes). Therefore simulated annealing and genetic algorithms aren't a good choice for our problem, since there is no guarantee that these will always find the same solution if presented with the same input. This leaves the descent methods. Summarizing, we are left with two options for the local search and they are the two different variants of the descent method: choose the first alternative we find which has better objective value(first fit), or consider all possible alternatives and take the best(best fit). Whether the latter alternative is feasible within the time available has to be determined through computational experiments.

We presented a single method to deal with the duration change problem. In Chapter 3 we discussed alternatives to handle the case where the increased processing time causes the current ordering to become infeasible. However, the choice for a certain alternative is not one that can be made globally. It is very dependent on the characteristics of the system it is used on. For instance, some systems might be set up in a way that makes it very likely that actual durations are shorter than expected. In that case leaving the schedule temporarily

infeasible might be a viable option, while in other systems this might not work. Therefore we only mention the different alternatives, without making a choice.

4.2 Performance Testing

To test the performance of the different methods, we have to implement them and design a framework to simulate requests and parameter calculations. The concrete implementation of these methods and the surrounding framework is discussed in the next chapter. For the testing of the performance of the methods, we then use two realistic scenarios. We use a search scenario and a tracking scenario, because these scenarios allows us to focus on specific parts of the scheduling problem. We run the scenarios 100 times and average the resulting running times to minimize the possible influence of other tasks active on the system.

Search Scenario The first scenario is a search scenario. In this scenario the ordering of the operations is pre-determined. The jobs have to be processed in the given order, and of course the operations that constitute a job always have to be processed in their given order. This means we only have to determine the exact start times for the operations. However, the durations of the operations are subject to change in this scenario. The requests in this scenario are engineered in such a way that everything should fit without problems. Therefore, this scenario tests the ability of the scheduling algorithm to handle changes in duration and it allows us to easily check whether or not the solution the scheduling procedure produces is correct, since we know the solution we want to achieve in advance.

Tracking Scenario The second scenario is a tracking scenario. We assume the scenario is executed on a phased-array system. In this scenario there are 5 different parties making requests. They are unaware of each other and can therefore each claim up to 100% of the radar timeline. This means a lot of jobs have to be rejected. There are no constraints on the ordering of jobs though and the duration of operations is known in advance. Therefore, this scenario tests the scheduler's ability to decide which jobs to accept and which jobs to reject. It should also prove to be computationally more complex, since the order is not known in advance. The fact that the order is not known in advance does make it more difficult to determine the quality of the solution the scheduling algorithm finds. Ideally we would like to compare the solution to the global optimal solution. This is very hard to compute though and we don't have a direct way of doing so at the moment. Therefore we use the degree to which the radar timeline is filled as a measure of quality.

To compare the approaches, we measure the quality of the solution produced by the scheduling algorithms as well as the processing time that is required to find the solution. To compare two solutions we use the optimization criteria discussed in Chapter 2. To measure the processing time we simply time how long the scheduling algorithms take to perform their task.

Implementation

In order for us to be able to draw conclusions regarding the methods outlined in Chapter 3, we need to perform computational experiments to determine the feasibility and performance of these methods. In order to carry out these experiments, we require an implementation of the methods as well as a framework to simulate requests being made and parameter calculation. In the following sections we discuss this implementation in greater detail. Java is used for the implementation, mainly based on the familiarity with that language.

5.1 Algorithms and Optimizations

In this section we discuss the different algorithms and some optimizations that were applied. We first discuss the constructive approaches and afterwards the local search algorithm.

5.1.1 Constructive Approaches

We consider only two constructive approaches, the priority rule and the shifting bottleneck heuristic. We begin by discussing the priority rule, since only a small modification is required to turn the priority rule into the shifting bottleneck heuristic.

Priority Rule

The pseudocode for the priority rule can be found in Chapter 3, Algorithm 1. The actual implementation is very similar to the pseudocode. The main difference is in the fact that we actually build the graph representation for the ordering in an incremental matter. Instead of generating the graph representation each time, we maintain a graph representation for the current ordering by adding and removing edges and vertices as necessary. This saves the cost of building the entire graph each time we have to determine starting times. Furthermore we aggregate vertices when the operations they correspond to become fixed (can no longer be changed, because they are already transmitted or are very close to transmission), into one single vertex. This vertex is given starting time 0. All edges between fixed operations are removed and the length of every edge from a fixed operation to a non-fixed one is increased by the starting time of the fixed operation. Figure 5.1 demonstrates how the fixing influences the graph. As can be seen in the figure, the amount of vertices and edges is reduced

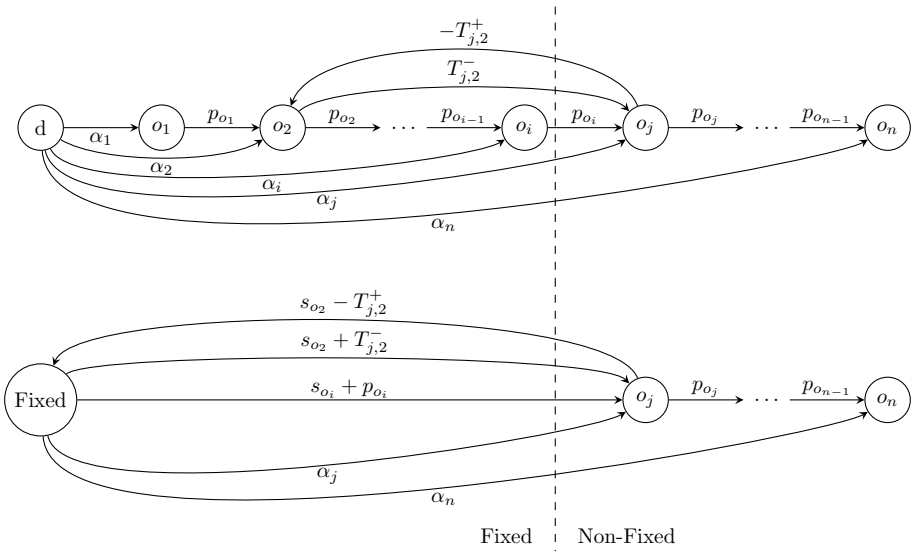

Figure 5.1: Fixing operations in graph representation

drastically since for the majority of the time, the amount of fixed operations will be far greater than the amount of operations under consideration for scheduling. Each of these vertices and edges contributes to the running time of the algorithm. The priority rule has a running time of $O(n^5)$ as we have seen in Chapter 3, where n denotes the number of vertices, so reducing these values has a great impact on the processing time required to find a solution.

Because we don't know whether or not future operations might have some relation to already processed operations, we do store the information of all past operations. For most systems this will not be necessary, because there will never be relations to operations in the distant past. It would be far better to stop storing information about operations after a certain amount of time, due to the fact that in the actual implementation we iterate over the list of operations quite often. In general we can't exclude the possibility of a relation to an operation that was executed long ago though, therefore we store all the information.

We stated in Chapter 3 that we first try to place an operation after the last operation which overlaps with it's window. In fact, we first iterate over the list once, to see if there are gaps between consecutive operations that this operation could fit in without having to move any of the operations that are already there. If no such place exists or placing the operation in any of these gaps does not lead to a feasible schedule, we continue by attempting to place the operation after the last, continuing as discussed in Chapter 3.

In Chapter 3 we stated that the priority rule schedules complete jobs one at a time. In reality, we sometimes only schedule part of a job. That is, some jobs are very long and stretch far into the future. As we have discussed in Chapter 2 it is not very useful to schedule far into the future. Therefore we only consider operations whose start time window overlaps with the period between the current time and some time in the near future. The actual amount of time

used should be determined for each system separately. In order to be able to use the algorithm as we specified it in Chapter 3, we create *dummy jobs* consisting only of those operations we consider at this point.

Shifting Bottleneck Heuristic

To obtain the shifting bottleneck heuristic from the priority rule, we perform an extra step after adding each job. This step works as follows. For each job that was already in the schedule which has operations that have not yet been fixed, we remove the operations from this job from the ordering and add them again. Similarly to the procedure in the priority rule, we only reinsert those operations that are currently under consideration. After adding a job again, we attempt to add previously rejected jobs (if there are any) to the schedule as well, using the same mechanism used for adding jobs in the priority rule.

5.1.2 Local Search approaches

As discussed in the previous chapter, only two descent methods are applicable to our problem. We either choose the first solution that is an improvement to our current solution, or we consider all possible solutions and choose the best among them. The implementation is basically the same, except in the latter case we have to keep track of the best solution found so far. The most important part of the local search method is the transfer function. In Chapter 3 we discussed two alternative transfer functions. We can either perform the same step as we discussed in the previous subsection, regarding the shifting bottleneck heuristic where we remove and reinsert jobs, or we can swap an accepted job of priority $\bar{p}_i$ with a rejected job of the same priority. This is implemented in a straightforward manner, the operations of a job are taken out of the ordering and attempted to be added again by using the same mechanism as in the priority rule. As was the case in the previous section, only the operations currently under consideration are reinserted. Except of course when we exchange a job that is currently scheduled for a rejected one, in which case the entire job is removed from the ordering.

5.2 Framework

In order to test the methods discussed in the previous section, we require a framework that simulates the rest of the radar system. It needs to feed the scheduler with requested jobs, needs to simulate parameter calculation, a place for the scheduler to send status updates to and ensure the scheduler performs a scheduling run every m_{rd} time units. Figure 3.3 in Chapter 3 shows the architecture of the system, as it would be in an actual radar system. In our case we have to simulate large parts of the system though. That is, there is no antenna and bursts are not actually transmitted, there is no *Radar Management* that makes requests, and there are no actual parameters being calculated. The antenna is simply ignored, it is not crucial to the scheduling problem. The parts of the system we have to simulate are discussed in the following subsections.

5.2.1 Project Specific

In the architecture overview in Figure 3.3 we see a project specific part and a scheduler. These are implemented as two separate classes, both of which are abstractions of something we call a *heuristic scheduler*. The rest of the system just interfaces with a heuristic scheduler. Therefore, in systems that don't require a project specific part this part can be left out. In that case all of the other parts of the system communicate directly with the scheduler. In systems that do require a project specific part, all other parts of the system communicate with this project specific part. The project specific part itself then communicates with the actual scheduler. As the name suggests the project specific part is specific to a certain project, whereas the scheduler contains generic scheduling methods. That is, different systems use different project specific parts, but they all use the same scheduler.

5.2.2 Request Simulation

In a real radar system requests originate from Radar Management. In our system the requests are generated according to the scenarios. The scenarios are specified in files, which are read by an initializing class. This class then translates the data from the file to a sequence of jobs and their operations and sends the jobs on to a class representing Radar Management. Here a request is generated for each job considering the starting time window and possibly some project specific settings (for instance, how far in advance to request the job). Each request consists of a job that should be requested and the time at which the request should be made. These requests are then send to the Request Generator. This class maintains a list of requests sorted ascending by request time. When a request is made, it is added to this list and a timer is set to go off at the time at which the request is to be made. When a timer goes of, the request method is invoked. This method checks the list of requests and handles every request on the list which has a request time that is before the current time. We choose to handle every request with this property and not just the request that the timer that invoked the request method corresponds to, because the timing mechanism of the test setup is imprecise and therefore requests might be delayed if we have to wait for the timer. To avoid problems, a mechanism is in place to make sure only one timer can invoke the request method at any given time. That is, the next invocation of the request method is always after the last has finished and so requests are not made more than once.

5.2.3 Parameter Calculation

To simulate parameters being calculated, shortly before a job is to be transmitted by the antenna, the scheduler invokes parameter calculations. Currently this is implemented by a mapping from operations to actual processing times. The parameter calculation method simply checks the mapping, if the time listed there is different than the duration of the operation, the duration change handler is invoked. The mapping from operation to actual duration is usually created when the request for the job containing the operation is created.

5.2.4 Status Updates

As mentioned earlier the system consists of a project specific part and a scheduler. The request generator sends its requests to the heuristic scheduler it is initialized with. If this is a project specific part, it can modify the requests before sending them on to the scheduler. The scheduler only performs the scheduling itself and ensures parameter calculation is invoked. The scheduler then reports back to the project specific part whether each job has been accepted or rejected. The project specific part then has the option to take further action (such as changing the duration of some of the operations, knowing the float values (see Section 2.3) for each operation) or relay the information directly to the party that made the request.

5.2.5 Scheduler Invocation

In Chapter 3 we discussed the fact that our scheduling problem is an on-line problem. This implies we have to invoke the scheduling procedure periodically. This is accomplished by using a repeated timer mechanism from Java, that allows a certain method to be invoked every t time units. This method creates a Thread that invokes the method every t time units. There are some issues with this, mainly that there are no guarantees that the timing is entirely accurate. For our testing purposes this system suffices though. Since the scheduler might not be fast enough to handle the requests in real time, we introduce a time-factor. This effectively acts as if it slows down time. That is, it reduces the speed at which requests are made as well as the frequency at which the scheduling procedure is invoked, allowing the procedure to take more time to find a solution.

5.3 Architecture

Figure 5.2 shows how the classes we have discussed in the previous section fit together to form our system.

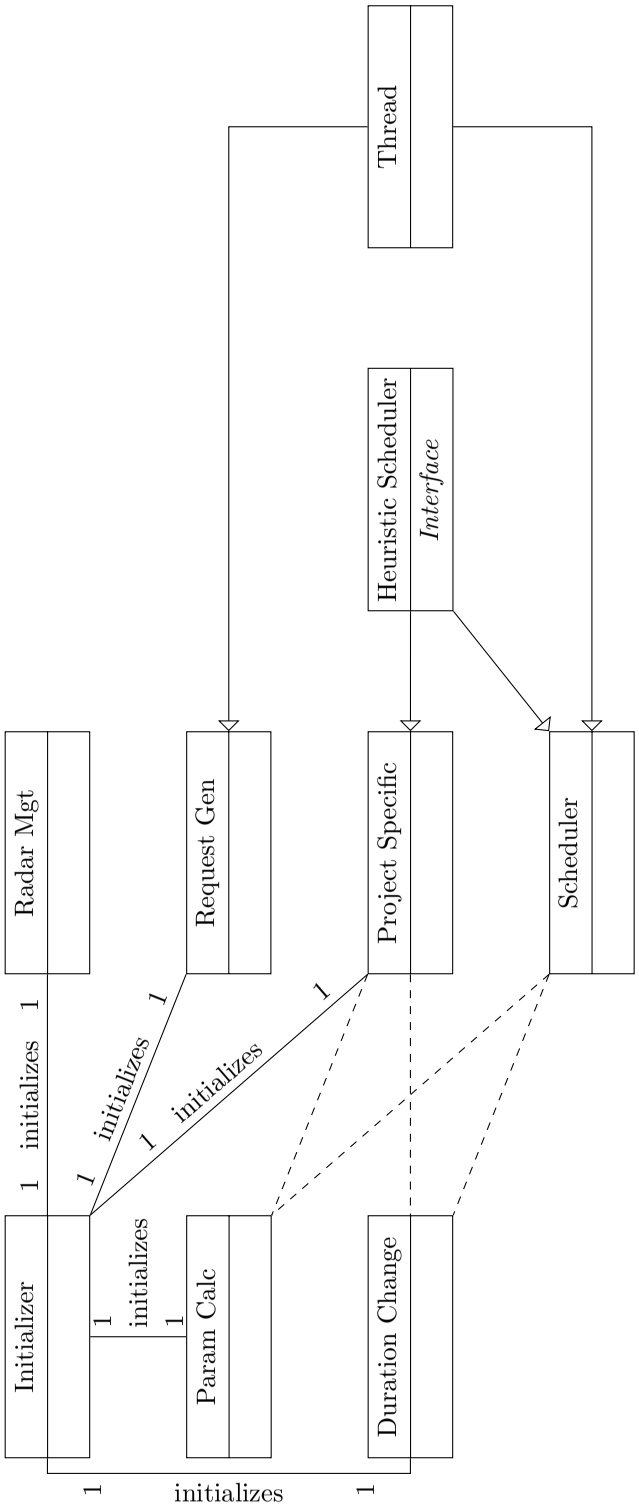

Figure 5.2: Class Diagram

Computational Experiments

In this chapter we discuss the computational experiments we performed in order to test the methods discussed in Chapter 3. We have already discussed the scenarios we intend to use in Chapter 4. In the following section we discuss them in greater detail. Finally we present the results of our experiments.

6.1 Data

As discussed in Chapter 4 we use three scenarios to test our implementation. In the following subsections we discuss the scenarios in greater detail. These scenarios are hard-coded in advance, that is, there are data files describing the operations and their parameters; the jobs, their parameters and which operations they consist of; and the scenario, describing the sequence of jobs that constitutes the scenario. These files are then read in by an initializing class, which translates the data to operations and jobs, initializes the components of the system and starts the scheduler. The initializing class then sends the jobs to radar management (as described in the previous chapter) which adds requests based on the jobs' desired starting times to the request generator.

6.1.1 Search Scenario

The search scenario is based on a SMILE system. SMILE is a non-rotating phased array radar which can electronically direct its transmissions. Its primary task is to search large volumes of air. It also supports surface surveillance and fire control though. Fire control is a waveform that can be used to get better accuracy readings on a small surface area, furthermore it is used as a pre-action calibration. That is, due to its better accuracy it is able to detect a splash from something hitting the ocean. This makes it possible to perform calibration of the weapons system by firing a projectile and using fire control to detect exactly where the shell hits. Each task, consisting of bursts, is represented by a job, consisting of operations. The operations in SMILE are subject to changes in duration. Therefore this scenario requires a solution to the scheduling problem as well as the duration change problem as discussed in Chapter 2. We begin by discussing the different jobs that make up the scenario and the operations that they consist of. Then we continue to discuss the sequence of requests that makes up the scenario along with the parameters for each of the jobs. Table 6.1 shows the different job acronyms and their descriptions. Each of these jobs consists of a sequence of operations. These are specified in greater detail in Appendix A

Job	Description
TECH	Calibration job
LRF	Long Range Frame, for long range air searching
MRF	Mid Range Frame, for middle range air searching
SRF	Short Range Frame, for short range air searching
FC	Fire Control
SURF	Surface surveillance

Table 6.1: Search scenario Job descriptions

The sequence of jobs along with their parameters that together make up the scenario is given in Table A.1. The expected duration is the expected time that is needed to process the job. The next job is requested to start when the current job is expected to finish. The only exception to this rule are the TECH jobs, which have to start at a pre-determined time, independent of the jobs preceding it. Furthermore, note that even though the expected durations for LRF (listed in Table A.1) are different each time, the actual processing times are the same each time. Therefore the actual times for this job are listed only once in Appendix A. The expected duration is a value that is supplied by the requesting party and is equal to the sum of the expected durations of the operations the job consists of. In SMILE there is a constraint that states that a job can never take longer than the expected duration. That is, the sum of the actual durations of the operation belonging to a job can never be greater than the expected duration of that job. However, it is allowed that a subset of these operations have a greater sum of actual durations than expected durations. If we let p_o denote the expected duration and $\bar{p}_o$ denote the actual duration of an operation o, then for some set $\bar{O} \subsetneq O_j$, $\sum_{o\in\bar{O}} \bar{p}_o > \sum_{o\in\bar{O}} p_o$ may hold. However, $\sum_{o\in O_j} \bar{p}_o \leq \sum_{o\in O_j} p_o$ must hold.

In Chapter 3 we discussed three possibilities to deal with a change in duration that makes a schedule on the current ordering infeasible. For this scenario option a (rescheduling) is infeasible due to the amount of time available, since we only obtain the change in duration by performing parameter calculation very shortly before transmions. The results of our experiments, which we discuss in Section 6.3, show that a larger amount of time is required to perform scheduling. We also don't wish to leave the schedule infeasible (option c). This leaves us with option b (rejecting a job), which is what we have implemented. However, as will be argued in the next paragraph, in principle it is not necessary to reject jobs.

We have not defined any priorities over the jobs in this scenario. This is because the requests in SMILE are engineered such that everything should fit without problems. This scenario is mainly intended to test the scheduling algorithm's ability to find correct starting times for the operations and the duration change algorithm to deal with changing durations. When the constraint, that no job can have greater actual duration than expected duration, is met, there should be no rejections. The data we have used for this scenario ensures that this is the case.

Job	Description
STANDBY	Calibration during standby
S_SEARCH_K	Sector search on K band
S_SEARCH_I	Sector search on I band
SEARCH_SR_K	Short range search on K band
SEARCH_SR_I	Short range search on I band
SEARCH_LR	Long range search
ACQUIRE	Target acquiring
TRACK_1T	Single target tracking
PAC	Projectile measurement

Table 6.2: Tracking scenario Job descriptions

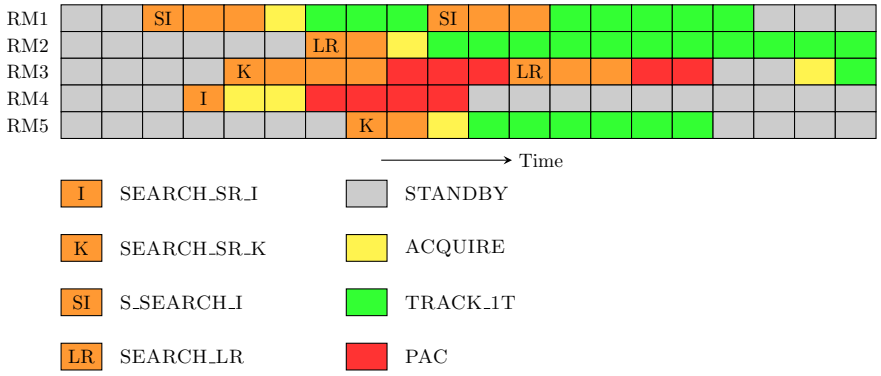

Figure 6.1: Tracking scenario request sequences

6.1.2 Tracking Scenario

The tracking scenario is based on a STING system. STING is a servo controlled radar system designed for tracking. The scenario is only partly based on STING though, i.e., the jobs along with their parameters are taken from STING, but the request sequence is not. In this scenario we have five requesters that each have their own sequence of requests. They all make requests as if they were the only requester, that is to say, they all request 100% of the timeline. Contrary to the search scenario, this scenario is aimed at deciding which jobs should be accepted and which should be rejected. Furthermore it places a far greater load on the scheduler due to the large amount of requests. Table 6.2 shows the different job acronyms and their description. Each of these jobs consists of a sequence of operations. These sequences are listed in Table B.1 in Appendix B. In the remainder of that appendix, a more detailed specification of the operations is provided. The priority of the STANDBY job is set to 1, while the other jobs all have priority 2. Figure 6.1 shows the request sequence for each of the five requesters. Each of the rectangles in the diagram represents 80 ms of time e.g., a gray rectangle, representing STANDBY jobs, represents a maximal amount of STANDBY jobs such that the sum of the processing times of these jobs does not exceed 80 ms.

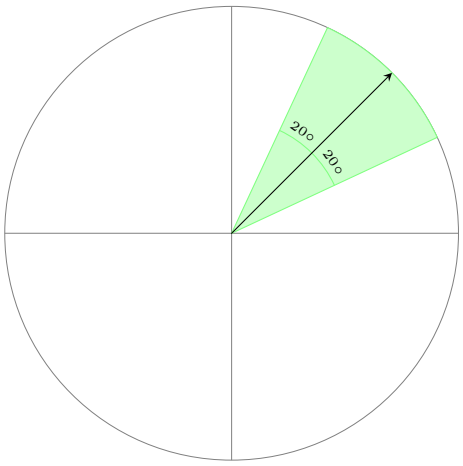

Figure 6.2: Rotating search radar coverage

6.1.3 Rotating Search Scenario

During the research, a practical case came up within Thales, a rotating search radar was under development and needed a scheduling mechanism. The third scanario is a test scenario that is loosely based on this system. The system consists of a single phased array antenna that rotates at a fixed speed. This means we can't transmit in every direction at all times. However, since it is a phased array antenna, we can transmit at an angle to the direction the antenna is pointing in. That is, normally we would transmit at an angle of 90 degrees to the face of the antenna, but with this antenna we can vary that angle. However, transmitting at too large angles drastically reduces the performance of the system. Therefore, we limit the angle at which we are allowed to schedule, to 20 degrees. Figure 6.2 depicts the directions the radar can transmit in, where the arrow points at an angle of 90 degrees to the face of the antenna.

The system has two operational modes, surveillance mode and defense mode. The operational mode determines what tasks the radar performs and the allocation of time to each of these tasks. We only tested the defense mode, since that was more resource intensive and therefore more interesting. The system divides the space around it in three parts: surface, low elevation air, and high elevation air. While in the defense mode, the system is required to search 360 degrees of low and high elevation air on every rotation and 360 degrees of surface once every two rotations. Furthermore, there is a video task which generates video output of the surrounding environment for the controller (the well-known green radar screen), which also has to cover 360 degrees every rotation. There is a task for measuring noise as well, which is also required to cover 360 degrees every rotation. Additionally there is a fire control task, which works slightly different. The operator can place up to three areas on the surface within the radar's range where he wants fire control to take place and these areas are allowed to overlap. Finally there are some calibration tasks for transmission and reception which don't have to cover a certain azimuth(direction) but have to occur a given number of times per rotation. Transmission calibration is performed ten times per rotation and reception calibration six times. There is a constraint on

the calibration tasks stating that two tasks of the same type (transmission or reception) have to be at least 50 ms apart.

All of the tasks discussed above correspond to a single job containing a single operation. The calibration jobs have priority 1 while the other jobs have priority 5. The requests are made in batches some time before the first job of that batch has to start. The requesting party checks the operational mode and determines which requests have to be made. A corresponding set of jobs is generated and at this point some additional relations are created. All of the tasks discussed above (except for the calibration and noise tasks) want to use the information gathered by a noise burst transmitted in the (roughly) same direction. That is, when the jobs are created they all have a target azimuth, the direction they are to be transmitted in, for each of these jobs a relation is then created to the noise job whose target azimuth is closest to theirs. This means they can also have a relation to a noise job that has a greater target azimuth. In traditional rotating systems this would not be possible, because a greater target azimuth would mean a later transmission time. Due to the fact that this system uses a phased array antenna, the noise burst can be transmitted earlier than the corresponding job, even though it has a greater target azimuth. There is a constraint stating there has to be at least 5 ms between the noise job and the job that has a relation with it, to allow for processing. This of course means that there can be multiple jobs that have a relation to the same noise job. Once all of the jobs and relations are created, they are sent as requests to the scheduler. In our test setup they are written to a file which is later read in by the scheduler, as explained in Chapter 5.

The durations of the operations are distributed uniformly between some given bounds. The exact data of the scenario we used can be found in Appendix C. Even though an expected and actual duration are listed, there are no changes in duration in this scenario. That is, the expected and actual durations are equal for each operation.

This scenario is mainly aimed at experimenting with applying the scheduling mechanism to a different system. The search and tracking scenarios are based on existing radar systems, meaning that the requirements from[1] and the formulation of our problem are in part based on those systems. This scenario is based on a new system though, which is slightly different. Therefore, it allows us to gain insight in what is required to apply the mechanism to a new system.

6.2 System

In this section we discuss the system that is used to perform the measurements, to provide some additional background concerning the running times we discuss in the upcoming section. The CPU in the system is a dual core Intel E2140 clocked at 1.6GHz and the system has 2GB of RAM. The scheduling algorithm only uses one thread though, so only one core is used for the actual scheduling. Tasks like request generation do run in a separate thread. The system is running version 5 of CentOS and uses Java version 1.6 (OpenJDK (IcedTea6 1.9.8)). To minimize the risk of other tasks that might be active on the system interfering with the experiments the tests were ran at night and the java process was run with a nice value of -20. Which means the process has the highest priority.

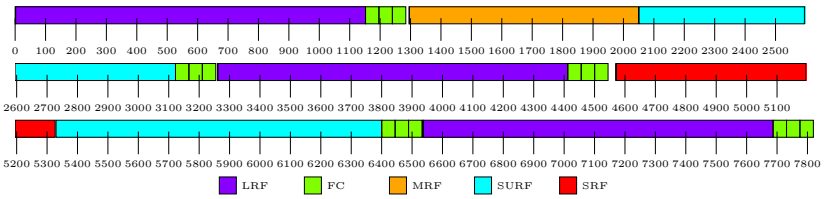

Figure 6.3: Search scenario result

6.3 Results

In this section we present the results for each scenario separately. We start by stating the expectations we have for the solution, in regard to quality. We then present the results obtained by the different methods described in Chapter 4. The results presented in this section are further discussed in Section 6.4.

6.3.1 Search Scenario

For this scenario we know exactly what we expect as output. The input is engineered in such a way that we know in advance what the schedule should look like: all jobs should be scheduled and we know the exact times at which each operation should start. The question then remains to what degree the approaches we discussed are capable of achieving this goal and what amount of time is required to do so.

We start by discussing the values we chose for the parameters introduced in Chapter 3. We decide to schedule once every 20 ms (i.e. we set $c_i = 20$ for all i) and every time we look ahead 50 ms (i.e. we set $l_i = 50$ for all i). That is to say, we consider operations (that are not yet fixed) whose starting time window overlaps with the window starting at the current time and ending 50 ms in the future. We chose these values because we ensure the requests are made 40 ms before they have to start, this makes 20 ms a good choice for the scheduling interval, since we then have 20 ms to perform the scheduling itself. If we run the scheduling algorithm more often, say every 10 ms, we solve the same (more or less) problem more often, but have less time to perform the scheduling algorithm each of these times. We already noted in Chapter 3 that we can't choose a larger value without reducing the amount of time we can use to perform the scheduling routine. The lookahead was chosen as 50 ms because some of the operations take over 15 ms and we always want to consider a decently sized group of operations.

The scenario was scheduled using the priority rule 100 times, because we wish to decrease the effect of variation in the running times we find on our results. The resulting schedule was the same each time, every job was scheduled and each operation started on the predicted time. Figure 6.3 gives a schematic view of the schedule.

We note there are some small gaps in the resulting schedule. This is by design and has to do with synchronization between the difference faces of the radar system. The exact implications are beyond the scope of the scheduler though and it suffices to say that the gaps are supposed to be there and the resulting schedule is exactly what we would expect. Furthermore, the TECH

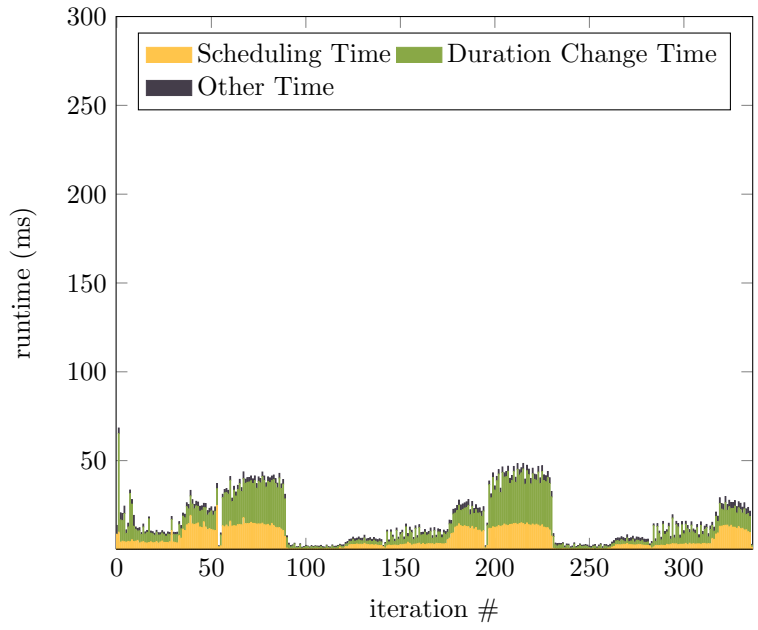

Figure 6.4: Search scenario running times per iteration

jobs are so small that they can't be seen in the figure, this is also the reason they are not shown in the legend.

The running times required for scheduling, handling duration changes, and additional time used for tasks like bookkeeping are plotted in Figure 6.4. An iteration in the figure denotes a scheduling invocation, i.e., the scheduling of a single scheduling window of 20 ms. The value for each iteration is obtained by taking the average value over the 100 times the scheduling routine was performed. The scheduling time covers only the time required to find the actual schedule, that is, the time required to determine where to insert the operations and when their processing should be started. The time required for duration change handling includes the time used to simulate parameter calculation and handling the change in duration itself, including the calculation of floats. Recall from Chapter 5 that parameter calculation is simulated by a simple lookup. Finally, the rest of the time covers everything else that is performed during an iteration. This is mainly bookkeeping to determine which operations need to be taken under consideration for scheduling, to determine which operations should be fixed and to track which jobs are rejected, which are already scheduled and which are not yet determined.

Even though the results after applying the priority rule can not be improved, we also processed the scenario using a local search method (the first fit descent method) after the priority rule on each iteration. The resulting schedule was the same, the additional processing time required can be seen in Figure 6.5. These figures are again averaged over 100 runs.

We also processed the scenario using the shifting bottleneck heuristic, again using first fit in the local search step. The resulting schedule was the same, except for the FC jobs that might be in a different order. More precisely, the

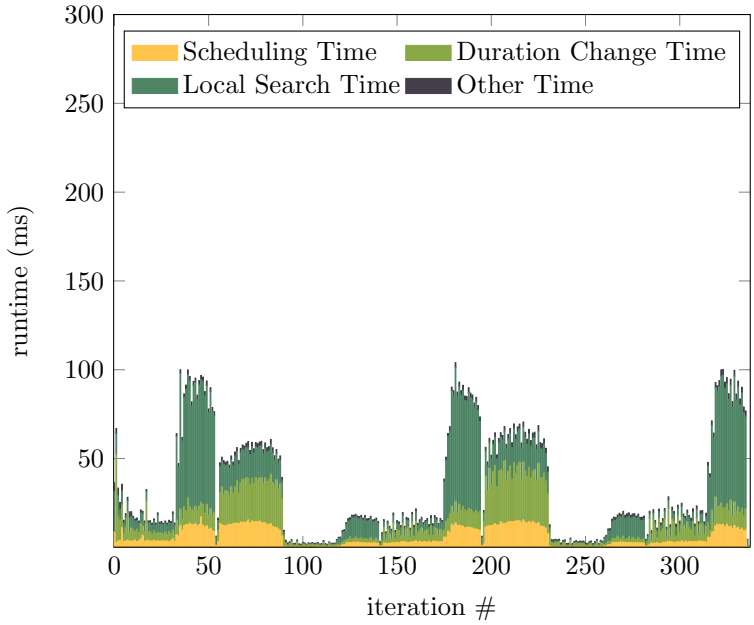

Figure 6.5: Search scenario running times per iteration with local search (first fit)

FC jobs occur in groups of three and they can be reordered by the shifting bottleneck heuristic within that small group. This can easily be forbidden by creating a relation between the last and first operations of two FC jobs that should be scheduled in order. Performing the shifting bottleneck heuristic does have an impact on the running time of the scheduling algorithm though. It greatly increases the running time required to find a solution (approximately sixfold), in iterations where an above average amount of operations are considered. Figure 6.6 shows the time required per iteration averaged over 100 invocations of the scheduling algorithm.

6.3.2 Tracking Scenario

For this scenario we don't have the same level of knowledge about the solution beforehand as we do for the search scenario. The goal is to schedule the greatest possible amount of jobs, respecting priorities.

We again discuss the values we chose for the parameters introduced in Chapter 3 first. We again chose a value of 20 ms as the scheduling interval (i.e. we set $c_i = 20$ for all i) and 50 ms as the amount of time to look ahead (i.e. we set $l_i = 50$ for all i) for the same reason as discussed in the previous section.

The tracking scenario was also scheduled using the priority rule 100 times. The resulting schedule was the same each time and can be seen in Figure 6.7. We expect the schedule to be the same each time, since every step in the priority rule algorithm is deterministic. The timing used to perform the experiments is not though, so small variations could in theory occur, although we have not found any. In the figure, the colour of a certain area specifies the sort of job

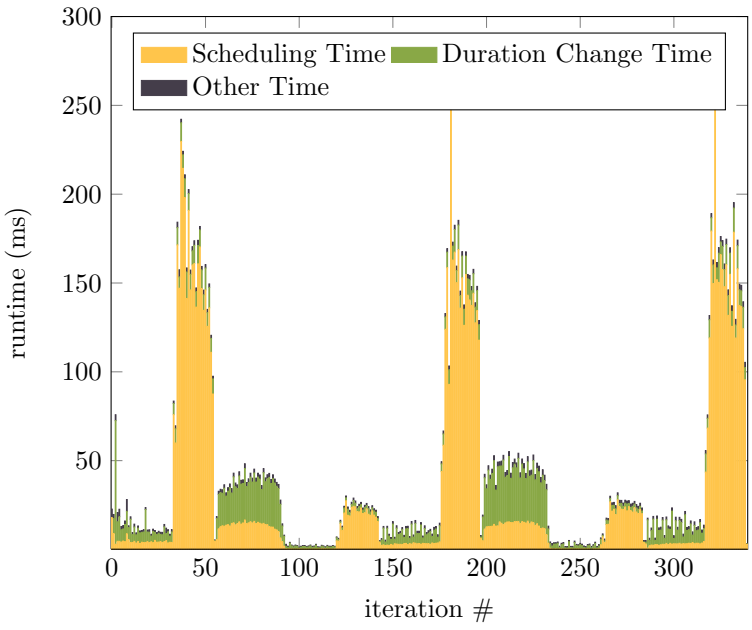

Figure 6.6: search scenario running times per iteration with shifting bottleneck

being performed (where the colours match those used in Figure 6.1). The height of this area corresponds to a number on the left of the figure, identifying the party that requested this job.

We can see that the entire timeline is filled (100%), a total of 132 jobs are scheduled and 670 are rejected. Of these 132 jobs, 22 are of low priority (STANDBY jobs) and the remaining 110 are of higher priority. Note that the first 160 ms are filled with STANDBY jobs. During this time these are the only requests that are made, as can be seen in Figure 6.1. We also see two STANDBY jobs at around 224 ms. This is due to the fact that the single other request that is active at that time, a S_SEARCH_I job, has a processing time of a little more than 64 ms. Thus, only one of these jobs fits in the 80 ms window we use to partition the timeline, leaving a little under 16 ms of time. This is filled by 2 STANDBY jobs, which take exactly 16 ms of time. Summarizing, we can conclude that at no point a higher priority job is rejected in favour of a lower priority one. This fact, combined with the fact that the timeline is completely filled, allows us to conclude that the schedule is of decent quality. It can be improved upon though, for instance, the three S_SEARCH_I jobs scheduled at 720, 800, and 880 ms could be replaced by a larger number of track jobs that are also requested at that moment. This would increase the amount of jobs of priority 2 that are scheduled. Due to the way the jobs are sorted, jobs of requester 1 are always scheduled first. We only see jobs of other requesters getting a turn when requester 1 is requesting jobs that leave the 80 ms window partly empty. In a real system the other requesters could dynamically increase the priority of their requests in order to ensure subsequent requests are considered earlier, so this is not a problem in practical situations.

The time required for the actual scheduling and the time used per iteration

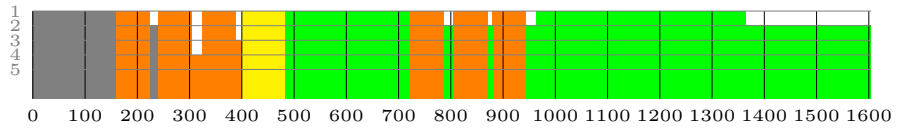

Figure 6.7: Tracking scenario result

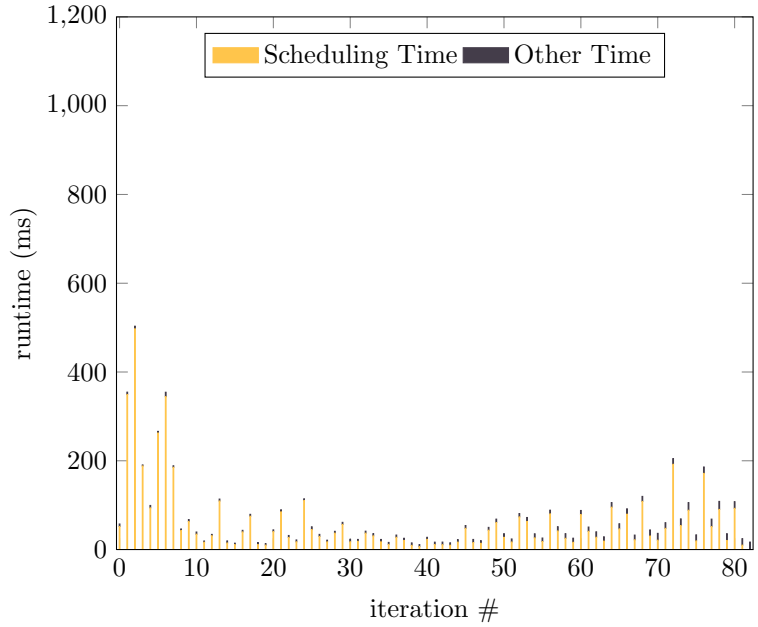

Figure 6.8: Tracking scenario running times per iteration

are plotted in Figure 6.8. The values are obtained in the same manner as for the search scenario.

We also used a local search approach to schedule this scenario, the first fit descent method. The resulting schedule is depicted in Figure 6.9 and is almost exactly the same as the one we obtain with just the priority rule (shown in Figure 6.7). As a matter of fact, the amount of jobs scheduled is exactly the same, 132, and so is the amount of STANDBY jobs and other jobs, 22 and 110 respectively. Only the order of some of the operations has changed. Since there are no gaps, this does not influence the makespan of the schedule. This means that performing local search has no effect whatsoever, in terms of quality of the solution.

We again ran the algorithm 100 times and the average results can be found in Figure 6.10. We see a significant increase in the time required to find a solution. On average the local search algorithm takes about the same time as the priority rule, effectively doubling the processing time required for scheduling.

Besides the first fit method, we also used the best fit descent method to schedule the scenario. The resulting schedule is shown in Figure 6.11. The result is exactly the same as that of the first fit descent method. This method was also performed 100 times, the results of which can be found in Figure 6.12.

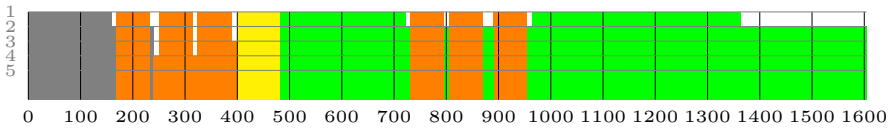

Figure 6.9: Tracking scenario result after local search (first fit)

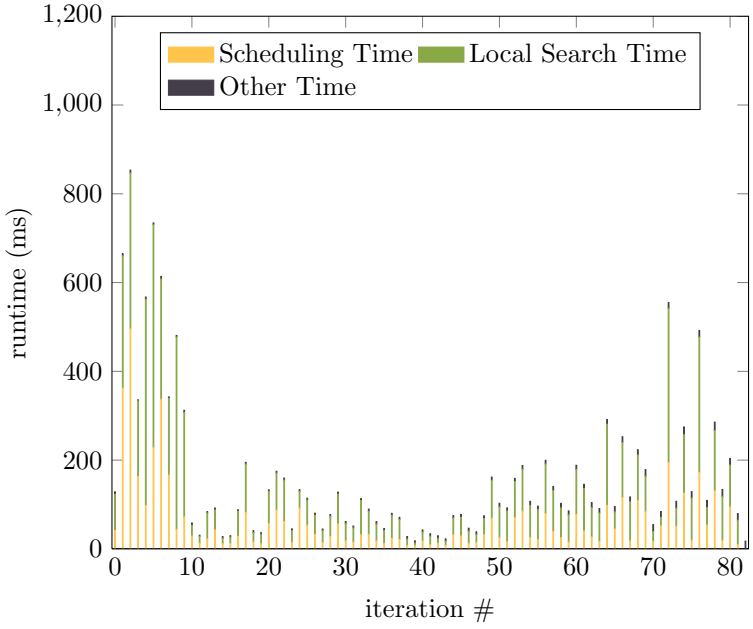

Figure 6.10: Tracking scenario running times per iteration with local search (first fit)

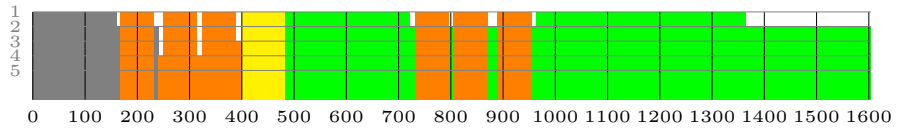

Figure 6.11: Tracking scenario result after local search (best fit)

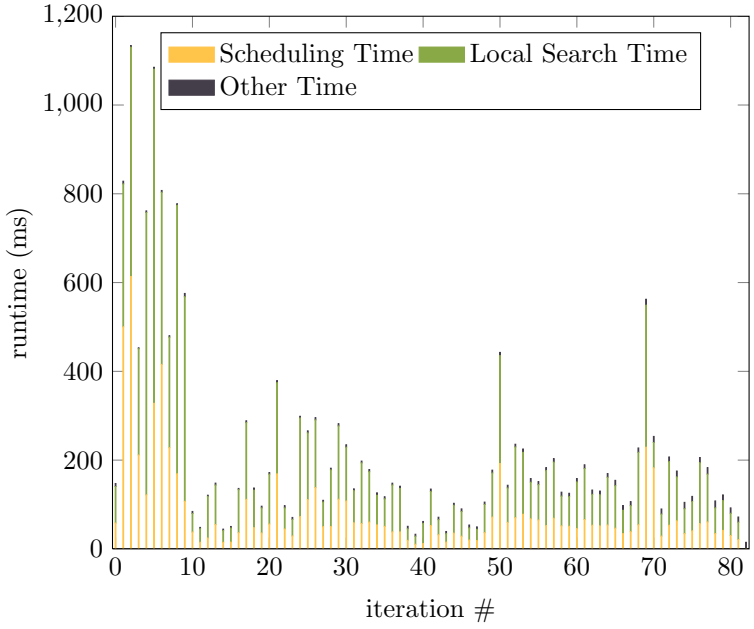

Figure 6.12: Tracking scenario running times per iteration with local search (best fit)

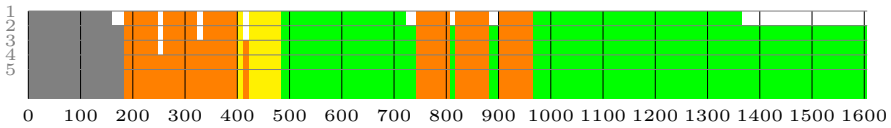

Figure 6.13: Tracking scenario result for shifting bottleneck heuristic

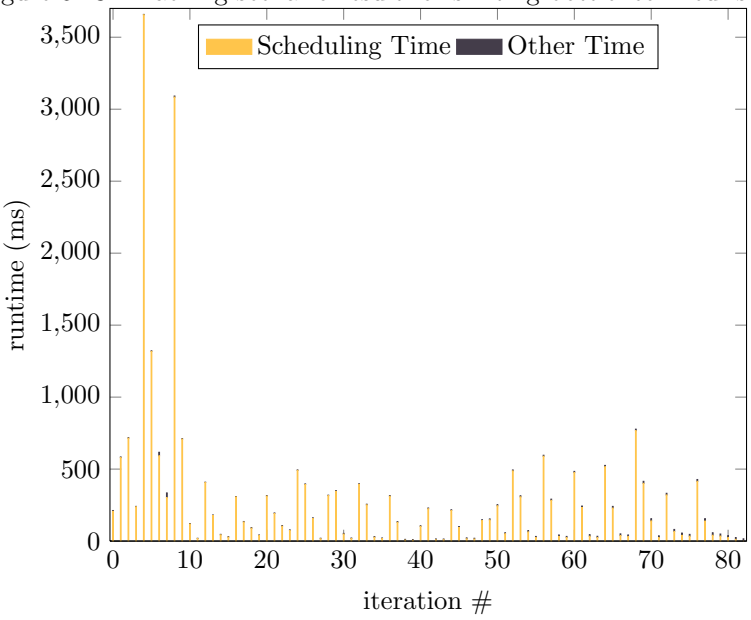

Figure 6.14: Tracking scenario running times per iteration with shifting bottleneck heuristic

Finally, we used the shifting bottleneck heuristic to schedule the tracking scenario, using the first fit descent method in the local search step. The resulting schedule is shown in Figure 6.13. In this schedule 132 jobs were scheduled, of which 22 were STANDBY jobs and 110 were higher priority jobs, exactly the same as after just the priority rule. The shifting bottleneck heuristic was also performed 100 times and the resulting running times can be found in Figure 6.14. We see a considerable amount of extra time is required compared to the approach using only the priority rule as well as a little additional time compared to the priority rule combined with local search.

6.3.3 Rotating Search Scenario

The requests in the rotating search scenario are determined by the operational mode, in this case the defense mode. These modes are designed in such a way that the load to the system is (slightly) less than 100%. Without relations, the operations would fit without problems. The relations possibly can cause problems though. The data we used consisted of approximately four rotations of the radar, corresponding to 8 seconds on the radar timeline. As explained in Section 6.1, in defense mode the radar wishes to perform several tasks in a 360 degree area, once every one or two rotations.

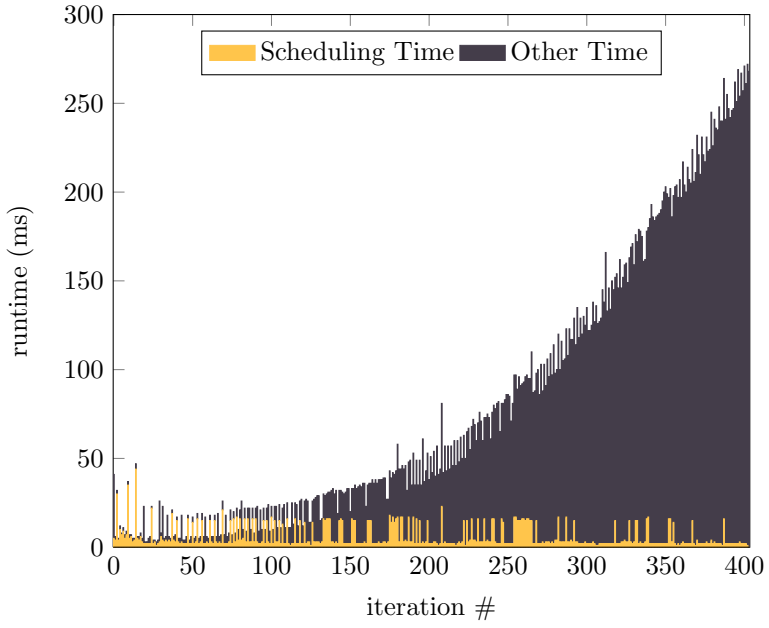

Figure 6.15: Rotating search scenario running times per iteration

We scheduled this scenario 100 times using the priority rule. The resulting schedule is the same every time. However, it is rather hard to visualize, since it consists of a great number of operations. All jobs are scheduled and there are some small gaps on the timeline. These gaps are to be expected, since the load is less than 100% and due to the fact that the operations have to be transmitted in a certain direction, the gaps resulting from the less than full load can't be moved to the end of the schedule. The average running times can be found in Figure 6.15.

Initial experiments showed that for this scenario local search approaches offered no improvements, as was the case for the other two scenarios. Therefore we did not perform extensive experiments with local search methods for the rotating search scenario.

6.4 Discussion

Here we discuss the results presented in the previous section. We discuss the search, tracking, and rotating search scenarios separately in the following subsections.

6.4.1 Search Scenario

For the search scenario, the results have shown that the priority rule finds the solution without requiring the help of local search. Furthermore, the priority rule does this within 25 ms in the worst case. The combination of scheduling and handling duration change takes less than 50 ms on every iteration, except one. The reason that duration change handling takes so long is that the procedure is

not optimized at all. That is, every time duration changes need to be handled, two complete graph representations are built to determine latest event times and latest feasible event times respectively. To be able to use the mechanism in real time, it would have to finish scheduling and handle duration changes within 20 ms. Considering the implementation was made in Java and the main focus was a proof of concept and not optimizing the efficiency of the implementation, we believe the mechanism can be implemented in such a way that real-time operation becomes feasible.

6.4.2 Tracking Scenario

The results for the tracking scenario show that a decent solution is found by the priority rule. That is, at no point a higher priority job is rejected in favor of a lower priority one and the timeline is completely filled. The results also show that local search and the shifting bottleneck heuristic do not improve the solution. This is caused by the fact that there are only two levels of priority and no relations between jobs. Furthermore, the requests are fairly well structured, i.e., each requester on its own requests a sequence of jobs. If there were no priorities we could simply take all the jobs requested by one requester and schedule them all in the order they were requested in. Because the priority rule considers the jobs in ascending order of starting time (after first sorting on priority), it benefits from this structure.

As far as running times go, we see increased running times at the start and end of the schedule. This is due to the fact that there are a larger amount of STANDBY jobs requested there. The running time of the algorithm depends greatly on the amount of operations that have to be scheduled. A STANDBY job has a duration of 8 ms, which means there are 10 in an 80 ms request block as shown in Figure 6.1. Each of these STANDBY jobs consists of four operations. At the start all five requesters are requesting STANDBY jobs. Which means that in the first 80 ms there are 200 operations that have to be considered. If we compare this to some of the blocks in the middle of the schedule, we see that in these blocks only about 60 operations have to be considered. When we recall from Chapter 3 that the running time of the priority rule is $O(n^5)$, it is clear that the difference between 60 and 200 operations is quite large. For comparison, the average number of operations in an 80 ms window in the search scenario is 30. In a practical situation the scheduler would only have to consider the standby requests from one of the requesters, since they are all the same. This behaviour could be implemented in the project specific part of the scheduler for such a system. For the scenario in its current form we find it unlikely that the priority rule mechanism can be sped up sufficiently for it to become feasible. However, this is not a realistic scenario, in the sense that in practice we would not have five requesters that each claim 100% of the timeline. The main purpose of this scenario was to see if the scheduling mechanism could deal with a large number of jobs, many of which needing to be rejected. The results show that the scheduler handles that challenge well, though not in a feasible amount of time.

6.4.3 Rotating Search Scenario

The results for the rotating search scenario show that the scheduler finds a very good solution to the problem. All jobs are scheduled, which is our primary goal. Furthermore, the scheduling mechanism is almost always able to find these solutions within 20 ms (per iteration), which means that it needs only a very minor improvement in order to be able to run it in real-time. In this scenario an average of 23 operations are requested in each 80 ms time window, which explains why the running times are again lower as those in the other scenarios.

When we introduced this scenario, we stated the main aim was to test how difficult it was to apply the mechanism to a new system. This has turned out to be simple. All of the constraints present in the scenario could be handled by the model we outlined in Chapter 2. Furthermore, translating tasks into jobs and operations and generating requests for them was easily accomplished as well. This shows that it is easy to apply the mechanism we proposed in this research to new scenarios.

Conclusions

We consider a single-machine scheduling problem in a radar context. Our aim is to determine whether or not schedules of acceptable quality can be found in a feasible amount of time and in a generic fashion. To answer this question, we consider: the quality of the solutions, the amount of time required to find them and whether the mechanism is generic.

First, we consider the question of generics. We decided to use the same basic model as in the internship[1], since it covers all the requirements that were found at that point. During this research we have not encountered anything that leads to additional requirements. Furthermore, the scenarios that we have used to evaluate the mechanism can be expressed using this model without problems. Based on this we believe the model is generic enough to express any scheduling problem we may encounter in a radar context.

Second, we consider the quality of the solutions. We have seen in Chapter 6 that the scheduler finds good solutions to the scenarios we used to evaluate it. Furthermore, it does this using only the priority rule. We saw no improvement when a local search approach was used along with the priority rule. The structured nature of the instances of the problem we consider benefits the priority rule. We believe that most practical instances have this structured nature.

Finally, we consider the time required to find a solution. For the search scenario, we see that the scheduling mechanism needs to be sped up by approximately a factor five. The tracking scenario requires a far more significant speed-up, approximately a factor fifty. Since our implementation is merely a proof of concept and there was only a very limited amount of optimization performed, we believe the implementation can be improved significantly. At least to the point where the search scenario can be scheduled in real-time. In the following section we offer some recommendations on how the mechanism can be optimized. Whether the tracking scenario in its current form can be scheduled in real-time using the mechanism discussed here, remains an open issue.

Summarizing, the mechanism described in this report finds solutions to instances of the described problem of a satisfactory quality. It does this in an amount of time that leads us to believe that this mechanism can be successfully used in real-time applications to solve these scheduling problems.

7.1 Recommendations

Here we present recommendations for further research.

7.1.1 Optimization

The most important recommendation is to determine with certainty whether the priority rule can be implemented in such a way that the running times are low enough for it to be used in practical situations. For this purpose a proper implementation has to be made and some optimization has to be performed. In the following we discuss some possible optimizations.

We recommend replacing the explicit graph calculations with simply iterating over the relations contained therein. That is, instead of explicitly building a graph representation, with vertices representing nodes and edges representing constraints, these values should be fetched from the existing data structure used to store the operations when a longest path problem is to be solved. This would eliminate the overhead created by translating the sequence of operations into a graph representing the underlying relations as well as simplifying the system, because keeping the graph up to date requires a large amount of bookkeeping.

We also recommend removing operations from the schedule at some point. That is, currently operations remain in the scheduled list at all times. However, at some point they are so far in the past that they really have no influence on the schedule anymore. Keeping these operations in the sequence serves no real purpose and slows the system down considerably.

Furthermore, floats could be used more extensively in the scheduling phase. Currently floats are used to handle duration changes, they could also be used to determine whether inserting an operation in some place ever can lead to a feasible schedule. That is, if the gap between two consecutive operations plus the feasible float of the latter operation is less than the duration of the operation we want to insert, it will never fit in that position. This would incur more float calculations though, if these calculations can be sped up, then looking into this possibility may prove to be useful.

7.1.2 Local Search

In the considered scenarios the scheduler using local search proved to be ineffective. It would be interesting to investigate whether or not situations arise in practical scenarios where local search does offer improvements on the solutions found. Additionally, research into other transfer functions and local search approaches could be performed. We only tested the most obvious possibilities, perhaps there are other approaches which yield better solutions. We experimented briefly with a transfer function that replaces scheduled jobs with rejected jobs of the same priority, but this required far too much time to be considered feasible. However, perhaps a limited version of this approach can be used, for instance by targeting only jobs with a long duration for replacement.

Bibliography

[1] R. Spijker, "Generic scheduler." Internship at Thales, 2011.

[2] R. Bellman, "On a routing problem," *Quart. Appl. Math.*, vol. 16, pp. 87–90, 1958.

[3] L. Ford and D. Fulkerson, "Flows in networks," vol. 3, pp. 61–65, 1962.

[4] W. L. Winston, *Operations Research: Applications and Algorithms*. Brooks/Cole, 4 ed., 2004.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[6] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Management Science*, vol. 34, no. 3, pp. pp. 391–401, 1988.

[7] I. Osman and C. Potts, "Simulated annealing for permutation flow-shop scheduling," *Omega*, vol. 17, no. 6, pp. 551 – 557, 1989.

[8] J. F. Goncalves, J. J. de Magalhaes Mendes, and M. G. C. Resende, "A hybrid genetic algorithm for the job shop scheduling problem," *European Journal of Operational Research*, vol. 167, pp. 77–95, November 2005.