A DYNAMIC RESOURCE SCHEDULING FRAMEWORK APPLIED TO RANDOM DATASETS IN THE SEARCH AND RESCUE DOMAIN

by

Wolfgang Haas

B.Sc., Brock University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Fall 2007

APPROVAL

Name:	Wolfgang Haas
Degree:	Master of Science
Title of thesis:	A Dynamic Resource Scheduling Framework applied to Random Datasets in the Search and Rescue Domain

Examining Committee: Dr. David Mitchell, Professor, Computing Science
Simon Fraser University
Chair

Dr. Bill Havens, Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Fred Popowich, Professor, Computing Science
Simon Fraser University
Supervisor

Dr. Dirk Beyer, Professor, Computing Science
Simon Fraser University
SFU Examiner

Date Approved: September 13, 2007

Abstract

Dynamic scheduling refers to a class of scheduling problems in which dynamic events, such as delaying of a task, occur throughout execution. We develop a framework for dynamic resource scheduling implemented in Java with a random problem generator, a dynamic simulator and a scheduler. The problem generator is used to generate benchmark datasets that are read by the simulator, whose purpose is to notify the scheduler of the dynamic events when they occur. We perform a case-study on the CoastWatch problem which is an over-subscribed dynamic resource scheduling problem in which we assign unit resources to tasks subject to temporal and precedence constraints. Tabu search is implemented as a uniform platform to test various heuristics and neighbourhoods. We evaluate their performance on the generated benchmark dataset and also measure schedule disruption.

Keywords: dynamic scheduling; scheduling framework; problem generator; tabu search; scheduling algorithms

To my family

A problem worthy of attacks proves it's worth by hitting back!

— Paul Erdös (Hungarian mathematician, 1913-1996)

Acknowledgments

I would like to thank my Senior Supervisor Dr. Bill Havens for introducing me to the world of scheduling and constraint programming by teaching such a fun and challenging course. I am also grateful to him for allowing me to work on the CoastWatch project and his assistance and encouragement throughout the whole thesis process.

I would like to thank my Supervisor, Dr. Fred Popowich, for his numerous insights and my Examiner, Dr. Dirk Beyer, for agreeing to examine my thesis. I would like to thank the members of the Intelligent Systems Laboratory, especially Saba Sajjadian, and the members of the Natural Language Laboratory for fruitful discussions. I am also grateful to the Computing Science department for *forcing* me to attend seminars of the different labs, which helped me tremendously in finding the right area of research for myself.

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for the generous financial support they have provided in the form of Undergraduate and Postgraduate Scholarships.

I would like to thank the Computer Science department at Brock University for providing such a wonderful undergraduate education. I am especially grateful to Dr. Sheridan Houghten for seeing potential in me and letting me work as a research assistant. I am also grateful to Dr. Brian Ross and Dr. Tom Jenkyns for their excellent teaching and encouragement to undertake graduate studies.

Last but definitely not least, I would like to thank my family and friends. I am very grateful to my parents Edith and Helmuth and my siblings Claudia, Elisabeth, Christian, Michael and Alexander for their endless love, support and encouragement. I am very grateful to Wendy for her continuous love and support throughout my graduate studies and for many more years to come.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Scheduling is the science of allocating limited resources to competing tasks over time [13]. It is a very important class of combinatorial search problems with many different real-world applications. In general, scheduling problems can be described as follows: Given a set of resources and a set of tasks, find a schedule that satisfies a set of constraints and optimizes some objective function. A schedule is a mapping of tasks to time intervals on resources [11]. Examples of constraints include precedence constraints, which control the amount of time that has to elapse between the starting times of two different activities, or temporal constraints, which ensure that an activity is executed within a given time frame.

A lot of research has been done in the area of scheduling. Probably the most famous class are job-shop scheduling problems [21], in which one has to assign a set of jobs to a set of machines in order to minimize the time at which the last job is completed. A real-world example for that would be a factory which develops a number of different products each day. Each product is developed by performing a sequence of tasks which are totally ordered and can only be executed on designated machines. The goal is to minimize the latest completion time among all products in order to send factory workers home as early as possible to minimize total wages. The drawback with job-shop and many other well-studied scheduling problems is that they do not account for machine failures or other events that might occur while tasks are being executed. Instead, everything is known ahead of time and nothing unpredictable will ever happen. But in the real-world there exists no such guarantee. The research described in this document takes a different approach: we deal with so-called dynamic scheduling problems in which many different kinds of unexpected events will occur throughout execution of the tasks. Consequently, these problems are much more realistic and are closely related to scheduling problems encountered in the real-world.

We study the Canadian CoastWatch Dynamic Resource Scheduling Problem, in short CoastWatch. It is an oversubscribed dynamic multi-mode scheduling problem with unit resources and lies in the Search & Rescue domain. Missions are composed of tasks which have to be executed during a specified time interval. Tasks are semi pre-emptive [12], meaning they can be interrupted but must be restarted completely instead of resuming the remaining workload. CoastWatch datasets simulate a typical day for the Canadian Coast Guard, where officers assign resources (planes, helicopters, ships, ...) to execute several different kinds of missions (patrol, transport, ...). The most important kind of missions are Search & Rescue when a human being needs to be rescued due to some unfortunate incidents experienced by a crew onboard a ship or aircraft out in the ocean. The objective of the CoastWatch problem is to maximize the sum of priorities of all accomplished missions.

Unfortunately, there was no actual data available for this dynamic scheduling problem and consequently our first task was to generate our own datasets. We studied the scheduling literature in an attempt to find a dynamic problem generator that we can run in order to create a CoastWatch benchmark dataset. However, we were not able to find one that we could adapt easily for our specific dynamic scheduling problem. As a result, our research goal is two-fold in that we first develop a problem generator that can be used to generate datasets for the CoastWatch problem, in order to achieve our second research goal of testing various scheduling algorithms and compare their performance.

In dynamic scheduling problems, the addition of a new mission or the occurrence of a dynamic event may cause a lot of disruption in the schedule. On a busy resource, it is very likely that a delay of a mission propagates to other missions that are assigned to be executed afterwards. Even worse than that, if we are dealing with time window constraints (as in CoastWatch), then it is possible that a mission can no longer be executed and might be reassigned to another resource. Although schedule disruption is not our main objective, we would still prefer a scheduling algorithm that minimizes this objective. In this way, the scheduling algorithm wouldn't completely reassign all missions to other resources every time a dynamic event occurs.

1.2 Approach

We develop a dynamic resource scheduling framework which can be applied to many different kinds of dynamic resource scheduling problems. It is implemented in Java and has three components: a random problem generator, a dynamic simulator and a scheduler.

The problem generator is a stand-alone component and can be used to create instances of the problem. It may be turned off at any time in order to use the framework simply for solving datasets and running scheduling algorithms on them. The problem generator provides great flexibility in generating datasets for dynamic resource scheduling problems. The specification of the problem and the parameters for all missions and events are passed into the problem generator. This makes it as general as possible in order to allow for generating benchmark datasets with very different kinds of characteristics. Changing a single parameter value for an event might cause the dynamic event to have a very different influence on the whole scheduling problem. By inputting the problem specification we ensure that the problem generator can be applied easily to different dynamic scheduling problems. This is achieved by simply making the appropriate changes in the specification file. Dynamic events that are generated by the problem generator include:

- resources can be either added or deleted from the problem
- new missions and tasks can be added to the problem
- tasks can be completed earlier or later than anticipated
- time window constraints on tasks can be altered

A resulting dataset is parsed by the dynamic simulator, which creates all tasks and events at the appropriate time. This simulator is necessary to hide all future events from the scheduler. Every time an event occurs, the scheduler is invoked in order to make adjustments to the schedule to accommodate the new event. A special feature of the simulator is its visualization tool that creates an animation of the scheduling problem on Google Earth. One can watch resources as they are moving around to execute missions and see the decisions made by the scheduler.

The scheduler is a platform for scheduling algorithms and communicates with the simulator. To test an algorithm on the generated datasets, one would simply implement the new algorithm in a class. The scheduler component also provides several useful methods.

We first run the random problem generator to create a benchmark dataset for the CoastWatch problem. We use Tabu search as a uniform platform to test various scheduling heuristics on the generated problem instances, because stochastic local search algorithms are very well suited for hard scheduling problems [11]. In addition to that, we experiment with different search neighbourhoods. We evaluate the performance of our runs on the generated datasets and also measure their resulting schedule disruption.

1.3 Contributions

We develop a dynamic resource scheduling framework which is composed of three components: a random problem generator, a dynamic simulator and a scheduler. It can be applied to many different kinds of dynamic resource scheduling problems.

We develop a random problem generator which generates benchmark datasets for dynamic resource scheduling problems. It is very easy to add new mission types and dynamic events, because it only requires minor changes in its input files. The use of parameters provides great flexibility in changing the characteristics of the generated instances.

We develop a dynamic simulator which is used to run dynamic scheduling datasets. It hides future events from the scheduler and contains a visualization tool to create an animation of the executing schedule on Google Earth.

We implement Tabu search as a uniform platform to test various scheduling heuristics on the CoastWatch problem. Additionally, we experiment with different search neighbourhoods.

1.4 Thesis Outline

The remainder of this thesis is outlined as follows. In Chapter 2, we give a literature review of topics related to our work. First, we introduce other problem generators that exist within the community. We then talk about several scheduling algorithms that have proven to be successful for similar oversubscribed problems. Chapter 3 defines a general model for dynamic resource scheduling problems, which acts as the basis for our framework. In Chapter 4, we first give an overview of our dynamic resource scheduling framework and then explain all the components in detail. We point out challenges that we faced during implementation of our framework and explain our solutions. Chapter 5 defines the CoastWatch scheduling problem and lists all the different entities. We then give a description of various heuristics and neighbourhoods that we run on our Tabu search in Chapter 6. The next chapter analyzes the performance of the different variations of the algorithm and includes discussion of experiment results. In Chapter 8 we give concluding remarks and suggest possible future work.

Chapter 2

Literature Review

2.1 Problem Generators

For combinatorial problems, performing a complete evaluation of the entire search space is only feasible for very small and often uninteresting datasets. Typically a stochastic local search technique such as *Tabu search* or *Iterative improvement* is used instead [20] [22] [24]. More sophisticated so-called hybrid algorithms combine the systematic approach of constructive search algorithms while incorporating the heuristic guidance of local search [8] [19]. When deciding on a local search method to use for a combinatorial search problem, it is never easy to select a specific algorithm, because there are so many different local search methods that have been applied very successfully before. If we could understand why certain algorithms work very well on specific problems but not on others, then the choice would be much easier. This is one of the main reasons for the importance of problem generators. By generating datasets according to desired characteristics we would be able to characterize local search algorithms and determine when they should be used. Recently, Kramer et al. [16] have used this idea in an attempt to understand when algorithms using permutation-based representations perform better than schedule-based ones for oversubscribed scheduling problems (see Section 2.2.3 for more information). The other main motivation for developing problem generators comes from the fact that it can be very difficult to obtain real-world instances for combinatorial search problems.

Many problem generators are written as part of a research project and are tailored specifically towards a studied problem. For example, Jang [14] as well as Barbulescu et al. [1] developed their own instances for the Air Force Satellite Control Network (AFSCN) problem. Both approaches analyzed existing real-world datasets in order to determine common characteristics of the given problem. This included computing mean durations and time windows for missions or studying different customers and learning their preferences regarding what type of missions they request. Unfortunately, it is very unlikely that such a problem generator can be applied to a different problem with minimal effort. While there exist more general problem generators that can be used to generate static datasets for an entire class of problems (e.g. project scheduling problems [5] [15]), no-one, to the best of our knowledge, has tried to come up with a problem generator for dynamic scheduling problems until recently [23].

Elkhyari et al. [6] studied the class of dynamic resource constraint project scheduling problems (RCPSP), in particular the university timetable problem. Since there existed no such publicly available dynamic datasets, the authors took existing static benchmark datasets for the RCPSP problems. They used explanation-based constraint programming, where nogoods[1] are derived during search when the assignment of a subset of the variables leads to a contradiction. Erasing these explanations when they are no longer relevant to the current variable assignment, guarantees polynomial space complexity. Nogoods are very powerful, not only do they enable Systematic Local Search [8] to have the systematicity of complete search methods, they also allowed Elkhyari et al. to solve the dynamic RCPSP in a more simplified manner.

If an unexpected event leads to an addition or a modification of a constraint in the system, then the explanations allow them to identify other constraints that were responsible for the contradiction. As a result, the process of repairing the solution is much faster than scheduling the whole problem from scratch again. Repairing is achieved by removing at least one constraint, preferably an assignment of a variable, and adding its negation. If an unexpected event leads to a removal of a constraint in the system one needs to reset values by undoing past events with the help of the recorded explanations, and re-propagate to get back to a consistent state.

Elkhyari et al. considered a large variety of different dynamic events: temporal events such as precedence constraints, activity events such as addition of a new activity and

[1]A nogood is a set of partial assignments that are not part of any consistent solution [8].

resource-related events such as the removal of a resource. These possible events are very similar to what we consider for the CoastWatch problem, however in their experiments the probability of such events occurring was very small. In their first experiment, Elkhyari et al. picked one event purely at random and compared the performance of their dynamic rescheduling technique with scheduling the whole problem again from scratch. They claim that their dynamic technique always obtained better performance, while improving efficiency up to 98.8%. In their second experiment different datasets were used and four events were created before their algorithm was run. For the majority of test cases their new dynamic algorithm performed better, although sometimes scheduling the dataset from scratch completed faster.

2.1.1 Policella & Rasconi

The work of Policella & Rasconi [23] deals with project scheduling problems, which are defined by the following components [3]:

- **Activities**: Every activity a_i is defined by a processing time p_i and requires a certain number of units, denoted req_{ik}, of a resource for execution. Depending on the selected resource, the required number of units may differ.
- **Resources**: The set of resources required to execute the activities. There may exist different types of resources such as renewable or nonrenewable ones.
- **Constraints**: The two types of constraints that exist within project scheduling problems are resource and temporal constraints. The former limit the maximum capacity of each resource while the latter restrict the possible start times for an activity. It is also possible to impose a binary constraint between two activities, a precedence constraint, in order to express the finish time of an activity in terms of another activity's starting time.

Because of unexpected events, a good solution to an instance of a project scheduling problem doesn't necessarily turn out to be well suited for execution in real-world environments. Policella & Rasconi stated that a scheduling problem can be divided into two sub-problems:

- **Static sub-problem**: Given the problem definition, find a schedule that optimizes the objective function. This is equivalent to the commonly known scheduling problem.

- **Dynamic sub-problem**: Given the solution to the static sub-problem, monitor the execution of the schedule. Should a dynamic event invalidate the current schedule, then repair it while trying to maintain the quality of the current solution and continue execution.

Policella & Rasconi were concerned with developing a testset generator for the dynamic sub-problem. A model was defined to allow for the following dynamic events:

- Delay of an activity
- Change of an activity processing time
- Change of a resource availability
- Change of the set of activities to be served
- Insertion or removal of a causal constraint between two activities

An absolute event time is associated with each dynamic event in order to determine when an event occurs. Policella & Rasconi did not use relative event times because they claim that the use of such might lead to invalid events during execution of the dynamic sub-problem. A simple example was provided where the execution of one event causes the next event to have an event time that lies in the past.

A relaxed version of the scheduling problem, the so-called simple temporal problem [4], was applied to allow Policella & Rasconi to compute the feasible range for the starting times of all activities. This information can be used to guarantee valid absolute event times for each dynamic event. However, to ensure that the event times remain valid throughout execution, it is necessary to make the following restrictions to their dynamic events:

- Activity delays can only be positive. That is, it is not possible that an activity can be executed earlier than first anticipated (i.e. it's time window cannot be shifted backwards).
- Activity processing times can only increase.
- Only resource availability reductions are allowed.
- No causal constraint removals are allowed.

We overcome these limitations by using dynamic event times while ensuring that no invalid events are created. The detailed description of our approach can be found in Chapter 3.

Finally, Policella & Rasconi also introduced several metrics that measure the difficulty of the sets of the generated events. A dynamic event may have enormous consequences on one specific schedule and little or no consequence on another schedule. In general, the closer two dynamic events are spaced to each other, the more critical the situation will be. Using absolute event times allowed them to use these metrics, since the generated event times were independent of the considered schedule.

2.2 Scheduling Algorithms

The algorithms explained in the following sections were all developed for oversubscribed scheduling problems. These are scheduling problems for which not all tasks can be scheduled and the algorithm needs to select the best subset of these tasks that can be completed while obeying all problem constraints. Typically, priorities are assigned to each activity and the objective function is used to select the best subset.

Local search algorithms have proven to be very effective for scheduling and a wide range of other combinatorial optimization problems [11]. In local search one typically starts with an initial solution and continuously makes adjustments to it in an attempt to find a better solution. The set of all possible moves during one iteration is referred to as the neighbourhood. After evaluation, the algorithm will select one such move to change the current schedule. The choice of neighbourhood can have a huge impact on the overall performance of an algorithm.

2.2.1 Roberts et al.

Roberts, Whitley, Howe & Barbulescu [24] examined the effect of neighbourhood choice on the performance of local search for the Air Force Satellite Control Network (AFSCN) scheduling problem. This problem consists of scheduling communication requests for earth orbiting satellites from a set of 16 antennas at 9 ground-based tracking stations. Tasks have time windows during which they have to be executed and the objective is to minimize the number of late jobs. At the beginning, the AFSCN problem is oversubscribed, but through negotiating relaxed task requirements, all jobs are eventually scheduled.

Roberts et al. examined the bias found in four variations of the shift neighbourhood under next-descent local search. Potential solutions were encoded using a permutation of all tasks and a schedule builder was used to generate solutions from the permutation. The permutation acts as a priority queue and each task is assigned to the first available resource at the earliest possible starting time. The shift operator selects a task from the permutation and moves it to another position. Consequently, assuming there are n tasks in the permutation, the size of the complete shift neighbourhood is $O(n^2)$ which can be very costly depending on the chosen underlying search method. The four neighbourhood variations differed in two binary characteristics: size (full or restricted) and order (structured or unstructured).

The first neighbourhood (N1) was structured and full: it randomly chooses a task and systematically shifts it into each of the other possible positions. Should none of these positions be acceptable, another task is selected at random. The N1 neighbourhood performed very poorly, because it induced a significant negative bias against improving or equal moves (80% of all considered moves resulted in worse evaluations).

The second neighbourhood (N2) attempted to overcome this bias by randomly selecting the task as well as its new position in the permutation. This neighbourhood was full but unstructured, since it did not systematically explore the entire shift neighbourhood. N2 resulted in a major performance improvement, and was competitive with the best previous solutions.

Roberts et al. reported that 40% of shifts resulted in no change to the schedule in AFSCN. Typically, restricting the search neighbourhood to only the tasks that induce a change produces more efficient search. Therefore, the remaining two variations of the shift neighbourhood, N3 and N4, were restricted. Given a task x to be moved, the move operator was restricted to only tasks that interact with x. Task x is said to interact with another task y, if the release or due date of task x lies within the time window of y. It is important to note that this measurement overestimates the actual amount of contention in the schedule, since it considers the entire time window of task y and disregards its duration. Additionally, after running the schedule builder, one of the tasks might actually end up being scheduled on another resource.

Roberts et al. calculated pair-wise task interaction for all tasks to build an undirected, unweighted graph where vertices are the tasks and existing edges indicate interaction. This dramatically reduced the neighbourhood size from $O(n^2)$ to the average degree per vertex, which ranged between 6 and 8 compared to the hundreds of tasks given in the problem. The N4 neighbourhood created a random graph with the same degree per vertex as in the interaction graph. This was achieved by connecting an edge to two different randomly selected tasks. Similarly to the outcome of their experiments with full neighbourhoods, the unstructured neighbourhood (N4) showed no performance decrease from N3. In fact, for almost half of the problems N4 significantly outperformed structured restricted search. Most surprisingly however, the restricted neighbourhoods (N3 and N4) showed no performance improvement over N2. Usually reducing the search space using problem specific structure leads to better results, but it seems that this is not the case for the AFSCN problem.

Similarly to the work of Roberts et al., we experiment with different neighbourhoods for our Tabu search algorithm. CoastWatch lies in the search & rescue domain and as a result it is a necessity to have a very efficient algorithm. Any time that is saved by decreasing the size of the chosen neighbourhood, can be used in other parts of the algorithm.

2.2.2 Kramer & Smith

Kramer & Smith [18] invented a task swapping algorithm for improving schedules in over-subscribed problem domains. Their goal was to solve the USAF Air Mobility Command (AMC) mission scheduling problem [2]. It can be characterized as follows:

- A set of tasks (or missions): A task is defined by its earliest pickup time, its latest delivery time, a pickup location, a dropoff location, a duration and a priority.
- A set of resources (or air wings): A resource with capacity ≥ 1. The resources used in this problem are individual planes with the capacity equaling the total number of planes available of that type at a specific base.
- Each task is associated with a subset of feasible resources which can be assigned to carry out the task. Every task always requires exactly 1 unit of capacity of the selected resource.
- Each resource has a designated home location (or base) to which it has to return after accomplishing a mission. Therefore when a task is to be executed by a resource, the resource requires a positioning leg to travel to the start location of the mission and a de-positioning leg to travel back to the base.

Typically, the problem is oversubscribed and only a subset of tasks can be feasibly accommodated. The objective of the AMC problem is to schedule as many missions as possible while adhering to the global constraint that higher priority missions must take precedence over lower priority missions. This constraint makes the problem very unusual as here it is more important to schedule one high priority task rather than an infinite number of lower priority tasks.

Their task swapping algorithm, called *MissionSwap*, starts with an initial schedule which is created by considering the missions from highest to lowest priority and assigning them to one of their candidate resources. It then considers all unassignable missions in order of their priority and tries to insert them one-by-one by temporarily bumping some of the tasks in the schedule. If the new task can be scheduled and all bumped tasks can be rescheduled then the schedule is accepted and the next unassignable task is considered. However, should it not be possible to reschedule all of the bumped tasks then *MissionSwap* restores the old schedule, since it is guaranteed from the construction of the initial schedule that these have higher or equal priority. Kramer & Smith used several heuristics to decide which tasks should be bumped. Max-flexibility estimated the flexibility for rescheduling of a task by looking at the size of its time window and the utilization of all its feasible resources. Min-conflicts measured the number of conflicts a task faces within its feasible execution interval, while Min-contention determined the portion of a task's time window that is in conflict.

In their original experiments [17], Kramer & Smith were able to demonstrate the efficiency of *MissionSwap* for the AMC problem. Out of the three different heuristics, using the Max-flexibility heuristic resulted in superior performance. In their follow-up experiments the authors re-evaluated their design decisions and considered several variations of their algorithm. Their pruning techniques make use of the conflict set of a task and for illustration we will re-use one of their diagrams. Figure 2.1 shows a set of tasks assigned to a resource (with capacity 2) that prevent a new task from being executed. A conflict interval is defined as the range during which the same set of tasks is being executed on the given resource. The conflict set then consists of the set of all conflict intervals that exist from the earliest time a resource leaves its home location (earliest starting time est_u - positioning time $pos_{r,u}$) to the latest time it will return to the base (latest finish time lft_u + depositioning time $depos_{r,u}$). Consequently, for the given example, the conflict set is {{a,b},{b,c},{d,e}}.

In their original experiments, *MissionSwap* retracted one task out of each conflict interval in order to make space for the new task. This task was chosen by their heuristic without

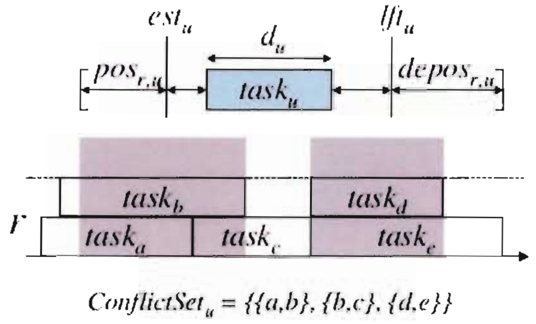

Figure 2.1: Conflict Sets in MissionSwap

considering the previous choices already made. As a result sometimes a task was retracted unnecessarily. For example in Figure 2.1, if task b is selected out of the first interval, the heuristic shouldn't have to make a choice for the second one, since one of its tasks had already been unscheduled. Kramer & Smith called this improvement task pruning, which led to a very significant increase in efficiency as well as some improvement in solution quality.

The second pruning technique is concerned with inserting missions that have slack, that is, their time window is larger than the required duration. For these tasks it may not be necessary to remove a task from each conflict interval, because the new task might already fit into the schedule after removing some of the tasks. Because Kramer & Smith make the assumption that all tasks were scheduled as early as possible, interval pruning retracts the tasks from left-to-right and stops when there is enough space for the new mission. This pruning technique resulted in another significant improvement in execution time, however, the max-flexibility heuristic actually performed worse.

Kramer & Smith improved the efficiency of their *MissionSwap* algorithm even more by limiting the maximum depth during the search. Their experiments showed that after 8 to 10 recursive calls the algorithm is basically doomed to fail and therefore they stop the search at that depth. All this gain in speed allowed them to experiment with techniques for expanding the task-swapping repair search that is performed in the hope of obtaining better solutions in circumstances when extended computation is possible.

Up to this point, the *MissionSwap* algorithm considered each of the unassignable tasks exactly once. However, when their algorithm is successful in inserting another task, circumstances change and it may now be possible to insert a task that didn't succeed earlier. Hence, their improved version of the algorithm cycles the set of unassignable tasks until no additional mission has been inserted for a whole iteration. Another variation of their algorithm added randomness to their heuristics. Instead of letting the heuristic select a task to be retracted, a task was retracted randomly from the set of all choices whose heuristic value fall within a certain percentage of the highest rated choice. In another variation, the probability of selecting a task was tied directly to the difference of its heuristic value and those of the competing choices. The advantage of the latter version is that no task is ever excluded from being selected. Instead, it would just be very unlikely. This technique obtained the best solutions in the end.

In CoastWatch, high-priority tasks, such as search and rescue, are added dynamically. Consequently some of the highest priority tasks are not known at the beginning of execution and the algorithm should therefore consider unscheduling lower priority tasks that have already been scheduled before. As a result it would be a very bad idea to only insert a new task if all retracted tasks were rescheduled. Regardless of this difference, *MissionSwap* includes very good ideas that might be applicable for any other scheduling problem as well, such as adding randomness to deterministic heuristics.

2.2.3 Kramer et al.

The algorithms introduced in the two previous sections performed very well for their respective applications. The main difference between these algorithms is that *MissionSwap* searches directly in the space of possible schedules, while the former searches in an alternative space of permutations and uses a schedule builder to create the mapping to schedule space. Kramer et al. [16] state that for some problems schedule-space search methods outperform permutation-based search methods and for some problems the opposite holds. They were interested in analyzing whether problem characteristics exist under which one technique can be expected to dominate the other.

Kramer et al. study two different problems, the AFSCN problem and the AMC problem. In the former, permutation-space scheduling algorithms dominate schedule-space methods, while in the latter, the opposite holds true. The main differences between these two problems are:

- **Task priority**: In AMC task priorities must be respected at all times and higher priority tasks must always be scheduled if possible. In AFSCN there is no priority and all tasks are considered to be of equal importance.
- **Number of tasks**: The benchmark datasets for the AMC problem have more than twice as many tasks.
- **Resource capacity**: AFSCN varies between 1 and 3 while AMC varies between 4 and 37.
- **Slack**: In AFSCN almost half of all tasks have no slack, while in AMC all tasks have temporal flexibility.

Despite the differences, these problems share many commonalities:

- A problem instance consists of n tasks.
- Each task specifies a required processing duration.
- There exists a set of resources that are available for executing tasks and each individual resource has a capacity ≥ 1.
- Every task has a set of feasible resources that can be used to execute it and every task requires exactly one unit of resource.
- Each of the feasible resources for a task specifies a time window during which execution has to happen.
- The basic objective is to minimize the number of unassignable tasks.

The advantage of permutation representation is that general-purpose algorithms can be used easily since all the problem-specific work is performed by the schedule builder. On the other hand, using such a technique might disconnect the search space from the optimal solution. The advantage of schedule representation is that usually many powerful heuristics, such as resource contention, are available to guide the search. However, it can be very challenging to find the right search operator.

Squeaky Wheel Optimization (SWO) was implemented for the permutation-based method, which repeatedly iterated through a 3-step cycle until a termination condition was met:

1. The schedule builder produces an actual schedule using max-availability heuristic.
2. The unscheduled tasks are ranked according to their contribution to the objective function.
3. The schedule is modified by moving some of the unscheduled tasks forward in the permutation.

The algorithm used for the schedule-based method was exactly the variation of the *MissionSwap* method explained in the previous section, where the probability of retracting a task was tied directly to the difference of its heuristic value and those of the competing choices.

Kramer et al. defined a series of problem sets that generalized from the AFSCN problem and increasingly incorporated characteristics of the AMC problem. They used the AFSCN benchmark datasets to produce new problem instances based on several parameters:

- **Problem size**: Kept constant, doubled or tripled the size of the initial AFSCN datasets.
- **Slack**: A duration factor *df* is used to determine the durations for each new task. The new duration is computed by multiplying the initial duration with (1-random(0,*df*)), where random(0,*df*) generates a random number between 0 and *df*.
- **Resource capacity**: A capacity factor *cf* is used to determine the capacities for each resource. The new capacity is computed by adding the initial capacity to a random number between 0 and *cf*.
- **Priority**: A priority flag determines whether priorities are present or not. When the flag is set, task priorities are uniformly distributed from (1...5).

Kramer et al. generated 36 problem sets with 50 instances each. The first 18 problem sets were identical to the second set with the exception that there were no priorities associated with the tasks. Their experiments showed that in terms of the number of unassignable tasks for the datasets without priorities the two algorithms performed very similarly. The authors claim that there was some evidence that the performance of *MissionSwap* improved when slack was held constant and the capacity was increased. In terms of penalty scores for the datasets with priority their results showed that for moderate levels of oversubscription the permutation-based SWO algorithm performed very similar to the other method.

From their experiments, Kramer et al. concluded that for problems that do not incorporate task priority, the search space was less constrained and since *MissionSwap* performs more localized search than SWO, it was not as effective. For problems where every task is assigned a priority, the performance of the algorithm depended on the level of oversubscription. The permutation-based search algorithm performed very similar to the schedule-based method on moderately oversubscribed problems, but as problems became more oversubscribed the situation was different. *MissionSwap* outperformed SWO, because rearrangement of task permutations became less productive.

The study of Kramer et al. attempted to classify scheduling problems for which it is best to use a specific type of local search algorithm. This is very different from the usual approach of developing and applying an algorithm to a given problem. The Tabu search algorithm we selected for CoastWatch differs significantly from both studied algorithms. More information can be found in Chapter 6.

Chapter 3

Dynamic Resource Scheduling Model

We describe a general model for dynamic multi-mode resource scheduling problems with unit resources subject to temporal and resource constraints. For a task we assume that there are multiple modes of execution and its duration depends on the assigned resource.

We extend static resource scheduling problems to include dynamic events where tasks and resources can be added, modified and deleted from the schedule during execution thereby possibly interrupting some already scheduled tasks. Unlike Policella & Rasconi [23], we use relative event times and guarantee that by doing so no dynamic event will occur at an illegal event time. In the following sections, we will describe each entity in detail along with a simple BNF syntax that we have implemented into our dynamic resource scheduling framework. A supplemented reading on our data model can be found at [9].

3.1 Constraints

3.1.1 Time window constraints

Given a task t, its time window specifies the earliest possible starting time est_t and the latest possible finish time lft_t. A task must not be executed earlier than the given est_t nor later than the given lft_t.

3.1.2 Precedence Constraints

Precedence constraints express the starting time of an activity in terms of another activity's starting time. They control the amount of time that has to elapse between them. Precedence constraints can be specified between two tasks belonging to the same mission and between a task and the mission itself. The syntax for specifying a precedence constraint is as follows:

```
<precedence> --> precedence <var1> <var2> <offset>
```

which defines the inequality: $<\text{var}_1> \leq <\text{var}_2>$ - <offset>

In other words, the start time of some task or mission, $<\text{var}_1>$, must precede the start time of some other task, $<\text{var}_2>$, by the specified offset. The value of <offset> is some integer.

For example, suppose a task B must start at least 10 minutes but at most 20 minutes after a task A starts. We would have two precedence constraints as follows:

```
precedence A B 10
precedence B A -20
```

It is also possible to specify simple precedence constraints such as a task A cannot start after task B:

```
precedence A B 0
```

To represent a precedence constraint requiring that a task A has to start at least 30 minutes after the beginning of its mission *M1*, we would write:

```
precedence A M1 30
```

3.1.3 Resource constraints

Resources are renewable, meaning that they can serve another task as soon as their current task is completed. Additionally, resources are capacitated and may only execute one task at a time. Similarly, a task always requires just one resource for execution which will execute it from the very beginning to the very end.

3.2 Scheduling problems

A scheduling problem contains the following entities:

- **Bases**: Bases are the home locations of resources. This includes air bases for aircrafts and ports for ships.
- **Resources**: Resources such as aircrafts, helicopters and ships, execute tasks and have a designated home base.
- **Capabilities**: A mapping of task requirements to resource capabilities. Tasks specify the capability that is necessary to execute them and every resource has a predetermined list of capabilities they can perform.
- **Tasks**: The activities that are scheduled and executed according to their time window and resource requirements.
- **Missions**: A partially ordered set of tasks that have to be completed to achieve some mission with specified priority.

We propose a simple regular language in BNF to specify a dynamic resource scheduling problem. We "borrow" the characters '*' and '+' from regular expressions [7] to specify quantities. The former denotes zero or more of the preceding element while the latter character denotes one or more. Note that the various sections of the problem definition must appear in the order shown.

```
<capability>*
<base>*
<resource>*
problem <horizon>
<event>*
```

First we specify all capabilities, bases and resources that exist within the problem. We then set the scheduling horizon by providing a start time and an end time for the schedule followed by a list of events, which include the creation of new missions. The syntax for the horizon is:

```
<horizon> --> (<start-time>, <end-time>)
```

where <start-time> and <end-time> are integers such that <start-time> $\leq$ <end-time>. Typically, <start-time> = 0. We assume that time is measured in minutes but any other integral time unit should also be acceptable. For instance, a scheduling problem spanning 24 hours would be specified in minutes as:

```
problem (0,1440)
```

3.2.1 Bases

Bases give a physical start location for resources at the start of the scheduling horizon. They are defined by a unique base name and a location, which is specified by its latitude and longitude values. We assume that bases are at sea level. The syntax for defining a base is:

```
<base> --> base <id> <location>
```

where the location is defined as follows:

```
<location> --> (<lat>,<long>)
```

For example, CFB Comox is an airbase on Vancouver Island, BC at latitude = N49.72052 and longitude = W124.89249. This is expressed as:

```
base CFB_Comox (49.72052,-124.89249)
```

3.2.2 Resources

Resources are defined by a unique identifier and a resource type. They move at a predetermined speed (in km/h) and are assigned a home base which is the starting location at the beginning of the scheduling horizon. Their syntax is:

```
<resource> --> resource <resource-type> <id> <base> <speed>
```

Depending on the types of resources in the scheduling problem, <resource-type> may be very general such as (plane, ship, ...) or very specific (F/A-18 Hornet, Boeing 747, ...). For example, a plane belonging to Austrian Airlines stationed at Vienna Airport might be described as:

```
resource airbus340 OS10001 VIE 1030
```

3.2.3 Missions & Tasks

We needed a general data model for dynamic resource scheduling problems that would provide enough flexibility to generate datasets for problems such as CoastWatch. Activities in some domains are made up of a several tasks, all of which have to be executed in order to complete the activity. For example, in Search & Rescue (SAR), a search task has to be completed before the rescue task can be executed. Here, only executing the search task doesn't accomplish the mission of rescuing a person and the activity should not be marked as finished. To be able to deal with these kinds of activities, we differentiate between missions and tasks.

We define a mission to be a collection of tasks that need to be executed. The mission is only considered accomplished if all of its tasks have been completed successfully. As a consequence, priorities are specified with missions rather than the tasks themselves. On the other hand, execution time windows are associated with tasks. This is because not all tasks of a mission are known at beginning of execution and some tasks are created dynamically. For example, for the SAR mission, the search task can be executed right away, but the rescue task is not created until search has been completed successfully by finding the missing person. If the time window were large enough to accommodate both search and rescue tasks, then the scheduling algorithm may delay the search activity until the end of the time window and there wouldn't be enough time to execute the rescue task when it is created. Therefore, the dynamic creation of new tasks requires either dynamic time windows for missions or associating time windows with tasks instead. We chose the latter approach for logical reasons: dynamically adding a rescue task to the SAR mission shouldn't extend the time window of the search task.

It is important to note that these definitions for missions and tasks do not exclude in any way activities that are only composed of one task.

Missions

The syntax for defining a mission is:

```
<mission> --> mission <id> <priority> {<new-task> <body>} {<precedence>*}
```

where <id> is a unique identifier and <priority> is a positive integer specifying the priority of the mission with value 1 representing the lowest possible priority. A mission must have at least one "New Task" event that creates a new task. The body of a mission contains a set of dynamic events, possibly including more new tasks, and is executed once the mission is introduced into the scheduling problem. For more information, see Section 3.3. Finally, <precedence>* is the set of precedence constraints that must be obeyed. There may exist at most 1 precedence constraint between any ordered pair of tasks belonging to this mission.

Tasks

Each mission contains a partially ordered set of tasks that need to be executed to complete the mission. Tasks are defined by a unique identifier, a task type and a time window for execution. Their syntax is:

```
<task> --> task <time-window> <task-type> <id> {<body>} {<precendence>*}
```

where the body of a task is a set of dynamic events that is parsed once the execution of this task has started. A task is deemed to execute when the time of the simulator reaches its start time. Similarly to missions, there may exist at most 1 precedence constraint between any ordered pair of subtasks. The field <time-window> specifies the earliest start time (EST) and latest finish time for a task (LFT). The syntax for time windows is:

```
<time-window> --> (<EST>,<LFT>)
```

where both fields are positive integers such that <EST> $\leq$ <LFT>.

Body

In dynamic scheduling problems, unexpected events can occur during the execution of tasks. We model this behaviour by associating a set of statements, called the *body*, to tasks and missions.

The <body> field defines the set of changes to the scheduling problem which can occur as a result of scheduling and executing a task. The syntax is as follows:

```
<body> --> <event>*
```

where <event>* is the set of unexpected events. For missions, these statements are evaluated when the mission is created, which may contain the creation of new tasks as well as mission events. For tasks, evaluation happens when the execution of the task commences and possible event types include new subtasks as well as task events. These events are explained in detail in Section 3.3.

3.2.4 Capabilities

The <task-type> of a task field specifies the capability a resource requires in order to be able to execute it. A capability is a mapping of a task type to a set of resource types which are able to perform it. The syntax for this relation is the following:

```
<capability> --> capability <task-type> (<resource-type>*)
```

For example, a rescue task out in the ocean could be specified as follows:

```
capability rescue (helicopter ship)
```

For resources that have specialized capabilities, we can define subclasses by name which possess those capabilities. For example, for an aurora aircraft which has a specialized radar onboard, we can add the following resource statement:

```
resource aurora_w/radar CP-140411 CFB_Comox 750
```

The capability relation will now have an added line which says:

```
capability surveillance_w/radar (aurora_w/radar)
```

3.3 Dynamic Scheduling Events

Static scheduling problem models are not concerned with simulating the execution of tasks in their schedules. The scheduling problem is given initially and the execution of the schedule cannot affect the problem dataset itself. This is not true for dynamic scheduling. We assume that executing a task at a particular time affects the world and introduces changes to the scheduling problem itself.

Policella & Rasconi [23] address the needs for dynamic resource scheduling benchmark datasets by coming up with a similar model. They use a relaxed version of the scheduling problem to compute the feasible range for the starting times of all activities. This information can be used to guarantee valid absolute event times for each dynamic event. However, to ensure that the event times remain valid throughout execution, they introduce several restrictions to their dynamic events. For example, it is not possible for a task that its time windows is shifted forward, i.e. that it can be executed earlier than first anticipated. Suppose there is no such restriction and an event happening at time t shifts a task's time window forward such that the earliest possible starting time $est < t$. Then, the shift specified in the event cannot be completed, it has to be adjusted such that $est = t$. However, as a result, the schedule produced by the scheduling algorithm affects the set of dynamic events specified in the benchmark dataset, therefore changing the difficulty of the problem instance.

Unfortunately, the restrictions on unexpected events adopted by Policella & Rasconi are inadequate for our purposes. For example, scheduling a SAR mission may involve an initial search task with an expected duration to find the target to rescue. The duration of this task may be reduced thus violating their restrictions.

We propose an alternative scheme to obey causality: we differentiate between regular events, task events and mission events and explain our solution for each of them.

3.3.1 Regular Events

Regular events are events that do not directly influence tasks or missions. Examples are the addition or removal of a resource. For these kinds of events, obeying causality is very straight-forward, because we are not concerned with the schedule produced by running the scheduling algorithm. Removing a resource can happen anytime and under any circumstances. One still has to be careful, though, because, for instance, if the same resource breaks down twice during the scheduling horizon, the second event should happen after the resource has been fixed. The syntax for regular events is as follows:

```
<event> --> <event-time> <event-type> <additional-parameters>*
```

where <event-time> is an absolute time within the scheduling horizon and <event-type> is a name that uniquely identifies the type of the event. This is followed by an optional set of additional parameters. For example, to remove a resource *heli1* at time 500 we would write:

```
500 remove_resource heli1
```

3.3.2 Task Events

Task events are events that directly influence a task such as the change of its duration or the addition of a new sub task. These events differ significantly from regular events, because their event time t_{rel} is relative. We impose the restriction that $0 \leq t_{rel} \leq 1$ and treat the event time as a percentage of the task duration. For a task a, the absolute event time t_{abs} can be computed by the following formula:

$$t_{abs} = \textit{start-time}_a + t_{rel} * \textit{duration}_a$$

It is guaranteed that all events will occur during execution of the task since the relative event time is a fraction of the total task duration. When a SAR mission is being executed we can create the rescue task anywhere during the execution of the search task. For instance, an event time of 0.9 would signal that the missing person is found after completing 90% of the search path. Assigning various resources with various speeds to the same rescue task, will result in different absolute event times. However, the rescue location will always be the same.

To guarantee that no invalid events will ever occur, we need to make sure that all dynamic events also obey causality. In particular, we need to ensure that neither start nor end time of a task can shift into the past. Luckily, the use of relative event times simplifies this issue significantly. Suppose there is a dynamic event which lowers the duration of a task. Due to some unexpected circumstances, the resource is able to execute the task quicker than first anticipated. If the event happens at relative event time t_{rel}, then the delay $delay_e$ must obey the constraint $delay_e \geq (t_{rel}$ - 1). This ensures that the event obeys causality; the updated end time of the task cannot be in the past after execution of the event. We can use the same argument for any other task event: we are aware exactly how far into the task execution the event happens and consequently, we know the maximum shifts that are possible.

Task events can be defined as follows:

```
<event> --> <event-time> <event-type> <task-id> <additional-parameters>*
```

where <event-type> must be the name of a task event and <task-id> the unique identifier of a previously defined task.

3.3.3 Mission Events

Mission events are events that influence a mission or one of its tasks. Examples include adjusting the mission priority or adding a new task. Additionally, delaying a task (i.e. shifting it's time window) should also be considered a mission task. This is because the delay has to happen before the start of a task, but task events only get executed once execution has commenced.

For mission events, the event time t_{rel} is also relative. But here it is relative to the creation of the mission and t_{rel} doesn't represent a fraction because it is independent from resources. For a mission m, the absolute event time t_{abs} can be computed by the following formula:

$$t_{abs} = \textit{creation-time}_m + t_{rel}.$$

For instance, suppose there is a delay task event which delays the main task of a mission by 5 minutes and it has an event time of 10. If the mission was created 300 time units into the scheduling problem, then the absolute time of the event is 310 minutes.

Obeying causality is very straight-forward for mission events. We need to ensure that the task doesn't start executing before the delay task event is executed. This is achieved, by setting t to be smaller than the task's earliest starting time *est*. In addition, if the delay d is negative, that is a task can be started earlier than first anticipated, we need to ensure that the event doesn't move *est* into the past. This can be achieved by choosing a value for delay during event generation such that it obeys the constraint $d \geq (t - est)$. We can give a similar argument for any possible mission event.

Chapter 4

Dynamic Resource Scheduling Framework

We develop a dynamic resource scheduling framework which can be applied to many different kinds of dynamic resource scheduling problems. We assume the existence of renewable unit resources and schedule tasks subject to temporal and resource constraints. Since this framework has been designed for tasks with multiple modes of execution, it can also be used for single-mode problems by simply specifying only one mode. Furthermore, we assume semi pre-emption [12] meaning the execution of a task may be interrupted but must be restarted completely. However, the framework can be easily extended to cover non-preemptive scheduling[1] by implementing only scheduling algorithms that will not consider retracting a task that is currently executing. Additionally, this scheduling framework can be applied to oversubscribed as well as undersubscribed dynamic resource scheduling problems.

Scheduling problems are very popular among many scientific communities because they have so many real-world applications. However, the drawback with most work up to this point is that they make the assumption that nothing unexpected will ever happen during execution of the schedule. In the real-world there exists no such guarantee and consequently there exists a need for developing scheduling algorithms that take into account future dynamic events. We are interested in studying CoastWatch, a dynamic resource scheduling problem, but were faced with the dilemma that we had to create our own datasets. A detailed description of the problem can be found in Chapter 5.

[1]In non-preemptive scheduling, tasks must be executed to completion and may never be interrupted.

The dynamic resource scheduling framework is implemented in Java and has three components: a random problem generator, a dynamic simulator and a scheduler.

The random problem generator is a stand-alone component and can be used to create instances of the problem. It may be turned off at anytime in order to use the framework simply for solving datasets and running scheduling algorithms on them. The problem generator provides great flexibility in generating datasets for dynamic resource scheduling problems. The specification of the problem and the parameters for all missions and events are passed into the problem generator. This makes it as general as possible in order to allow for generating benchmark datasets with very different kind of characteristics. Changing a single parameter value for an event might cause the dynamic event to have a very different influence on the whole scheduling problem. By inputting the problem specification we ensure that the problem generator can be applied easily to different dynamic scheduling problems. This is achieved by making the appropriate changes in the specification file.
Dynamic events that are generated by the problem generator include:

- resources can be either added or deleted from the problem
- new missions and tasks can be added to the problem
- tasks can be completed earlier or later than anticipated
- a task can be delayed which alters the tasks time window

The dynamic simulator parses the resulting dataset and creates all tasks and events at the appropriate time. This simulator is necessary to hide all future events from the scheduler. Every time an event occurs, the scheduler is invoked in order to make adjustments to the schedule to accommodate the new event. A special feature of the simulator is its visualization tool which creates an animation of the scheduling problem on Google Earth. One can watch resources as they are moving around to execute missions based on the decisions made by the scheduler.

The scheduler is a platform for scheduling algorithms and it communicates with the simulator. To test an algorithm on the generated datasets, one would simply implement the new algorithm in a class. The scheduler component also provides several useful methods.

Figure 4.1 gives an overview of the dynamic resource scheduling framework. The following sections explain each of the components in detail.

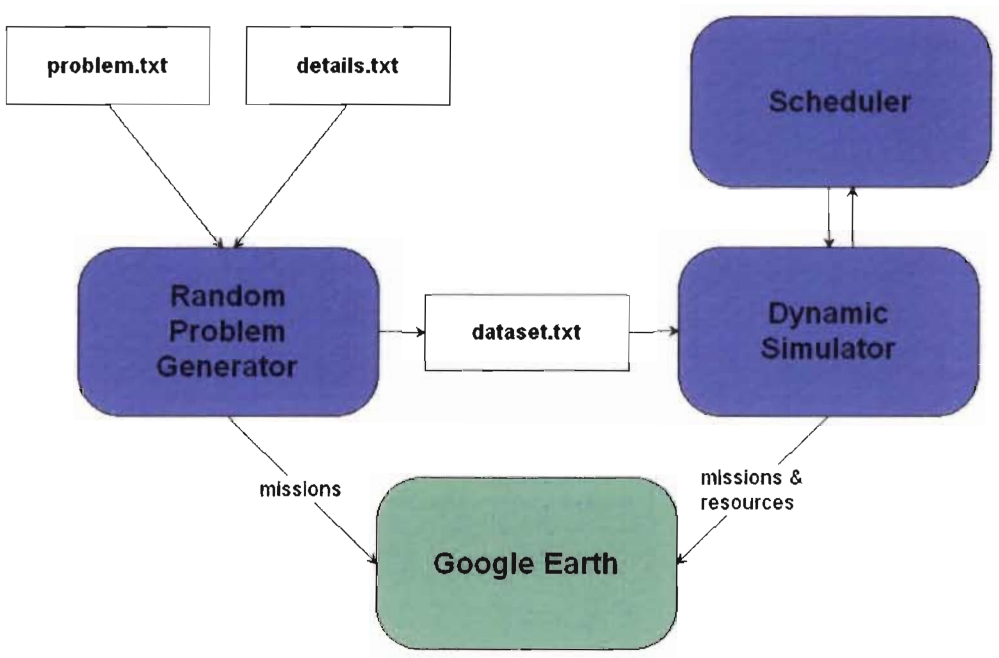

Figure 4.1: Overview of the Dynamic Resource Scheduling Framework

4.1 Random Problem Generator

Many problem generators are written as part of a research project and are tailored specifically towards a studied problem. Unfortunately, it is very unlikely that such a problem generator can be applied to a different problem with minimal effort. While there exist more general problem generators that can be used to generate datasets for many different static scheduling problems, the same cannot be said for their dynamic counterparts. We address the need for such random problem generators by creating one for dynamic resource scheduling problems as part of a larger framework.

4.1.1 Motivation

Dynamic scheduling problems model real-world environments very closely because they take into account that the execution of a schedule will not always go as planned. The lack of problem generators for dynamic resource scheduling problems prompted us to develop our own benchmark generator. The main goal is to keep it very general to allow for adaptation to many different kinds of dynamic resource scheduling problems very quickly. We provide great flexibility by passing the specification of the problem and the parameters for all missions and events into the problem generator. This makes it as general as possible in order to allow for generating benchmark datasets with very different kind of characteristics. Changing a single parameter value for an event might cause the dynamic event to have a very different influence on the whole scheduling problem. Similarly, the problem generator can be applied to different dynamic scheduling problems, by making the appropriate changes in the specification file.

We develop a random problem generator as part of the dynamic resource scheduling framework in order to be able to generate benchmark datasets. Such a tool can be very useful even if such datasets already exist for the given problem. Typically, a stochastic local search method is used for solving combinatorial search problems. However, it is never easy to select a specific algorithm, because there are so many different local search methods that have been applied very successfully before. If we could understand why certain algorithms work very well on specific problems but not on others, then the choice would be much easier. We could use a problem generator to develop datasets with certain characteristics in an attempt to understand when a given algorithm should be used. Similar research has been done by Kramer et al. and is discussed in Section 2.2.3.

4.1.2 Input Files

The random problem generator requires two input files, the Scheduling Problem file and the Mission and Event file. Appendix A contains an example for each input file.

Scheduling Problem file

This file lists all capabilities, bases and resources that exist within the scheduling problem, respectively. These items have to be specified exactly as stated in the description of our model in the previous chapter. The following is a simple example:

```
resourceTypes (helicopter plane ship)
capability    search     (helicopter plane ship)
capability    rescue     (helicopter ship)
base          Victoria   (49.7,-124.9)
base          Vancouver  (49.19388,-123.18444)
resource      plane      plane1   Vancouver  750
resource      helicopter heli1    Victoria   278
resource      ship       ship1    Vancouver  54
```

At the beginning we add a list of all resource types that exist within the problem, since it allows us to catch typing errors within the two input files. A capability is described by its name and a list of resource types capable of performing it. For example, a search task may be executed by helicopters, planes and ships. A base is specified by listing its name and location in latitude & longitude notation. Vancouver airport might be defined using the city name as its identifier and setting its location to (49.19388,-123.18444). A resource is described by its resource type, followed by a unique identifier, its assigned home base and its speed (in km/h). A plane might be able to fly up to 750 km/h and have Vancouver as its home base.

Mission and Event file

This file contains all parameter values for any mission, task or dynamic event type that is defined in the dynamic resource scheduling problem. Changing just a single parameter might have a strong effect on the characteristics of the generated datasets. For instance, a detailed mission and event file might look as follows:

```
horizon      0 1440
numBases     2
numResources 3
events
delay        probability=0.1          time=normal(10,4)
tasks
transport    numStatic=poisson(0.9)   numDynamic=5
             priority=random(5,10)    relativeTime=normal(60,10)
```

The first line specifies the scheduling horizon of the problem. Static tasks are always created at the given start time, while dynamic tasks are created some time before the end of the scheduling horizon. The next two lines determine the number of bases and resources that will be included in the generated dataset. Both entities are chosen randomly from the set defined in the scheduling problem file.

Section 4.1.3 lists all the dynamic events that may occur during execution. Each line contains the parameters for one specific event type. As outlined in our model, we differentiate between regular events, task events and mission events. It is very important to note that all task events are considered for every single task. In other words, should a task event have a probability of 1 then it will be applied to every single task in the problem. Similarly for missions, all mission events are considered for every single mission.

The same event might be applicable to different tasks, but with different characteristics. For instance, a search task is more likely to be delayed than a transport task. Therefore all the event parameters are merely default values. It is possible to overwrite an event parameter value by specifying it as a task parameter using the following syntax:

```
[name-of-event]_[name-of-parameter]=value
```

Suppose there is an event which changes the duration of a task and should occur in approximately 10% of all tasks and delay them by at least 5 but at most 20 minutes. The event could be specified as follows:

```
change_duration probability=0.1 time=random(5,20)
```

Now assume that 50% of all search tasks should change their duration. Then the probability parameter can be overwritten using:

```
search change_duration_probability=0.5
```

Similarly, setting the probability parameter of an event to 0 will result in no such dynamic events for tasks where this value was specified.

The remainder of the Mission and Event file lists all the different task types that exist within the scheduling problem. For each task type, it lists all its parameters in one line. Recall that our model differentiates between missions and tasks. A mission is a collection of tasks that need to be executed in order to achieve some goal. It is only accomplished when all its tasks have been completed and only then will it contribute towards the objective value. Typically, missions have one main task, which may create one or more subtasks which in turn may create even more subtasks.

Every task needs to specify at least two parameters:

1. **releaseDate**: the amount of time that needs to elapse after the task has been created and before execution can begin. Suppose at 1pm a transport task is created and is required to move some goods to another base no earlier than 2pm. Assuming that time is measured in minutes, the value of this parameter would be 60.

2. **relativeTime**: specifies how much time needs to elapse in the execution of the parent task before this task is created. Recall from Section 3.3, that this parameter is treated as a fraction of the task duration. It will be set to 0 automatically for main tasks of a mission, since these tasks do not have a parent task.

Task types which may be the main task of a mission must also specify three additional parameters:

1. **numStatic**: the number of static missions of that type. A static mission is known at the beginning of the scheduling horizon.

2. **numDynamic**: the number of dynamic missions of that type. Dynamic missions are created anytime throughout the simulation.

3. **priority**: the mission priority specified as a positive integer with larger numbers representing higher priority.

In order to provide more flexibility for the created problem instances, we allow parameter values to be generated using one of these common distributions:

- **Uniform distribution**: This generates one uniformly distributed random integer within the specified lower and upper bounds. The syntax for defining such a value is:

  ```
  [parameter-name]=random(lower bound, upper bound)
  ```

- **Normal distribution**: This generates one normally distributed double with the specified mean and standard deviation. A normally distributed value can be generated as follows:

  ```
  [parameter-name]=normal(mean, sigma)
  ```

- **Poisson distribution**: This generates one poisson distributed integer with the specified lambda value. Such a value can be defined using the following syntax:

  ```
  [parameter-name]=poisson(lambda)
  ```

This input file contains all the information that is necessary for the random problem generator to create benchmark datasets. In fact, by specifying no dynamic events and setting the *numDynamic* parameter of all task types to zero, we could use this framework for static resource scheduling problems. By modifying the input files, we can introduce additional flexibility. We associate a priority with each mission and assume that the objective function is to maximize the sum of priorities of completed missions. This objective can easily be modified to maximize the number of completed missions by setting all priorities to 1. Similarly, we can remove time window constraints from the problem by setting the time windows for each task to $(-\infty, \infty)$.

4.1.3 Dynamic Events

In this section, we provide a detailed description of the dynamic events that have already been implemented into our framework. These events are very common and apply to most dynamic scheduling problems.

New Mission

This dynamic event introduces a new mission during execution of the problem. The mission has a priority and must consist of at least one task that needs to be executed in order to achieve some goal. The syntax for defining a new mission event is as follows:

```
<event-time> <mission>
```

where <event-time> is an absolute event time within the scheduling horizon and <mission> is the description of a mission as defined in our model.

New Task

The new task event adds a task to the dynamic scheduling problem. It must be created in the body of a mission or a task. To define a new task event, the following syntax is used:

```
<event-time> <task>
```

where <task> defines the task as described in Chapter 3. Since a new task is added to the problem and is initially unscheduled, the mission it belongs to will not be considered completed even if it all its other tasks have been executed successfully. The meaning of <event-time> depends on the object that created the task:

- If the object is a mission, then the new task is one of its main tasks and the event time specifies the number of minutes that have to elapse after the creation of the mission before the task will be added to the scheduling problem. This is not to be confused with the task parameter *releaseDate*, which determines the earliest possible starting time of the task. Typically, missions have only one main task and as a result the event time should be 0. It makes no sense to create a mission without specifying any task.

- If the new task event was created in the body of another task p, then it is created as a subtask of p belonging to the same mission. In this case, the event time is relative and it determines the percentage of task completion of p when the new task is added to the problem. Suppose the starting time of p is 100 and its duration is 50. Further assume the new task event appeared in the body of p and its event time is 0.8. Then the actual time when this event occurs is $100 + 0.8*50 = 140$.

Add Resource

This event dynamically adds a resource to the scheduling problem. To be consistent with our model, we assume that the new resource is also a renewable unit resource. Additionally, the type of the new resource has to be from the set of possible resource types specified initially in the problem instance. The syntax for such an event is:

```
<event-time> add_resource <resource>
```

where <event-time> is an absolute time within the scheduling horizon and <resource> is the resource to be added. If the added resource has been part of the scheduling problem before, then it is sufficient to specify solely its unique identifier.

Remove Resource

This event removes a resource from the scheduling problem. If, at the moment of removal, the resource was currently executing a task, it will be unscheduled. Similarly, all tasks that were assigned to the resource to be executed in the future will also be unassigned. It is the task of the scheduler to reschedule them on one of the remaining resources. The remove resource event can be defined by:

```
<event-time> remove_resource <resource-id>
```

where <event-time> is an absolute time within the scheduling horizon and <resource-id> is the unique identifier of the resource to be removed.

Disable Resource

This dynamic event disables a resource for a period of time. The purpose of this event is to simulate that a resource encounters a mechanical problem which needs to be fixed. The selected resource is removed from the scheduling problem and added again after the specified delay. We assume the resource will remain at the same location. As a consequence of the disable resource event, all the tasks that were assigned to it, will be unscheduled. This includes the currently executing task as well as all its future tasks. The dynamic event obeys the following syntax:

```
<event-time> disable_resource <resource-id> <duration>
```

where <event-time> is an absolute time within the scheduling horizon, <resource-id> is the unique identifier of the resource to be disabled and <duration> is a positive integer representing the time it takes to perform the necessary repairs.

Delay Task

This dynamic event shifts time window of a task by a given delay. This delay can be positive or negative. The syntax for this event is:

```
<event-time> delay_task <task-id> <delay>
```

where <event-time> is relative to the creation of the mission, <task-id> is the unique identifier of the task to be delayed and <delay> is some integer.

We impose the constraint that *delay* <> 0, otherwise the delay task event would have no effect on the underlying scheduling problem. Additionally, if the given *delay* is negative, the earliest possible starting time *est* of the delayed task should not be not shifted into the past. Consequently, we introduce the additional constraint

$$delay \geq (time - est)$$

where *time* is the actual time of the dynamic event.

Change Duration

The change duration event modifies the required execution time of a task for the assigned resource. The purpose of this event is to simulate unexpected events that might occur during execution which have an effect on the duration. For instance, a flight from Vienna to Vancouver might arrive an hour early because of strong tailwind. Additionally, sometimes it can be used instead of the disable resource event, for example when a vehicle runs out of gas. In such an event, we know that the problem can be resolved very quickly, and we can simulate the resulting delay without having to unschedule all assigned tasks. Its syntax is as follows:

```
<event-time> change_duration <task-id> <delay>
```

where <event-time> is a relative event time, <task-id> is the unique identifier of the task whose duration needs to be adjusted and <delay> is the amount of change measured as the percentage of total duration.

For example, the change duration event

```
0.7 change_duration task1 -0.1
```

will have the following meaning: After successfully completing 70% of *task1*, the duration is reduced by -10%. Assuming the duration of *task1* using the assigned resource is 100 minutes, there will only be 20 minutes remaining after the event.

To guarantee that no invalid events will ever occur, we need to make sure that the updated end time of the task is not shifted into the past. Similarly as before, we impose two constraints:

$$\mathit{delay} <> 0$$

$$\mathit{delay} >= (\mathit{event\text{-}time} - 1)$$

The first constraint ensures that the dynamic event affects the underlying scheduling problem, while the second constraint limits the shift of the end time such that it cannot be in the past.

4.1.4 Mission Event Times

Suppose we were to generate event times such that the latest finish time of all tasks lies within the scheduling horizon. This way every task can be completed before the end of the scheduling horizon. Suppose further, we schedule tasks which typically require 500 minutes to execute and assume that the end of the scheduling horizon is set to 800. Then, all missions have earliest starting times ≤ 300 and as a result there will be 500 more minutes during which no additional mission is created.

Instead, the random problem generator tries to spread out the generated tasks. We uniformly distribute the absolute event times for the creation of missions. As a result, there might be several tasks which cannot be executed completely before the simulator halts. We deal with that problem by considering these tasks as completed as long as the scheduling algorithm was able to assign them to a resource such that they can be executed within their respective time windows.

4.1.5 Time Windows

An important consideration for dynamic resource scheduling datasets with task time windows is the size of the generated time windows. On the one hand they should be large enough so that they can be completed successfully. But on the other hand the generated problem instances become too easy if task time windows are too large.

We require that any dataset with a single task and a single resource should be solved optimally. Therefore, in computing the size of the time window we need to include the time it takes the assigned resource to get to the starting location of the task. For a given task type, we differentiate between two cases:

1. If the considered task is a subtask of some other task t, and t can be performed by the same resource type, we compute the time it takes to get from the end location of t to the start location of the considered subtask.

2. Otherwise, we compute the positioning time of a resource from its assigned home base. Of course during the simulation there is no guarantee that the resource will still be at that location. But we cannot do any better, since the problem generator cannot predict what the scheduling algorithm will do.

Initially, we set the size of task time windows to the sum of the average positioning time and the worst task duration. However, after investigating several runs of some generated datasets, we found that the scheduling algorithm interrupted a large number of tasks. Some of them were rescheduled up to 10 times! As a result we modified our problem generator to create smaller time windows. In our current version the size of the time window equals the sum of the best positioning time and the average duration.

4.2 Dynamic Simulator

This section describes the dynamic simulator component of the dynamic resource scheduling framework. Its main purpose is to hide all future events from the scheduling algorithm, since these events should occur unexpectedly as they do in the real-world. This is achieved by creating all missions, tasks and events at the appropriate time without ever releasing any information prematurely. Every time an event occurs, the scheduler is invoked in order to make adjustments to the schedule to accommodate the new event. A special feature of the simulator is its visualization tool which creates an animation of the scheduling problem on Google Earth. A supplemented reading on our simulator model can be found at [10].

Traditional scheduling models assume that the scheduler has no effect on the scheduling problem which it solves. Tasks are scheduled and eventually executed but the original problem does not change. This is not true in many real-world situations where the world is invariably affected by the execution of tasks and therefore the subsequent scheduling problem altered. We model these interactions by incorporating this dynamic simulator into our framework.

We define a dynamic simulator which forms a feedback loop with the underlying scheduling problem. Given a problem instance to solve, the scheduler produces a new schedule. The execution of this schedule produces a stream of events which are interpreted over time by the simulator. The results of these events are a sequence of incremental changes to the scheduling problem which are then iteratively re-solved by the scheduling algorithm. Each new schedule may produce more events in the future as scheduled tasks are being executed. This process is driven by a simulation clock which iterates through the scheduling horizon.

4.2.1 Simulator Model

An overview of the dynamic simulator model is shown in Figure 4.2. A scheduling problem P is inputted into the simulator. It specifies the set of missions, tasks and resources which are known initially. The scheduling problem is repeatedly modified by incremental changes ΔP to P as produced by the Event Executor. The scheduler accepts the modified problem P' and returns a new schedule S'. The addition, deletion or rescheduling of tasks will cause changes to their start and end times. The set of all changes, denoted ΔA, are input to the Start/Stop Generator. This module is responsible for creating a set of new events ΔE which will modify the start and end times accordingly.

The Event Queue stores all future events E to be processed by the Event Executor. Events are added to the queue from three sources:

1. Initially, a data-file containing one problem instance of the scheduling problem is parsed. In addition to missions, tasks and resources, this file contains a set of regular events E_0 which will be executed at specified times throughout the scheduling horizon.

2. Event changes ΔE are created by the Start/Stop Generator as follows. For each newly scheduled task, two new events are added: one representing the start time of

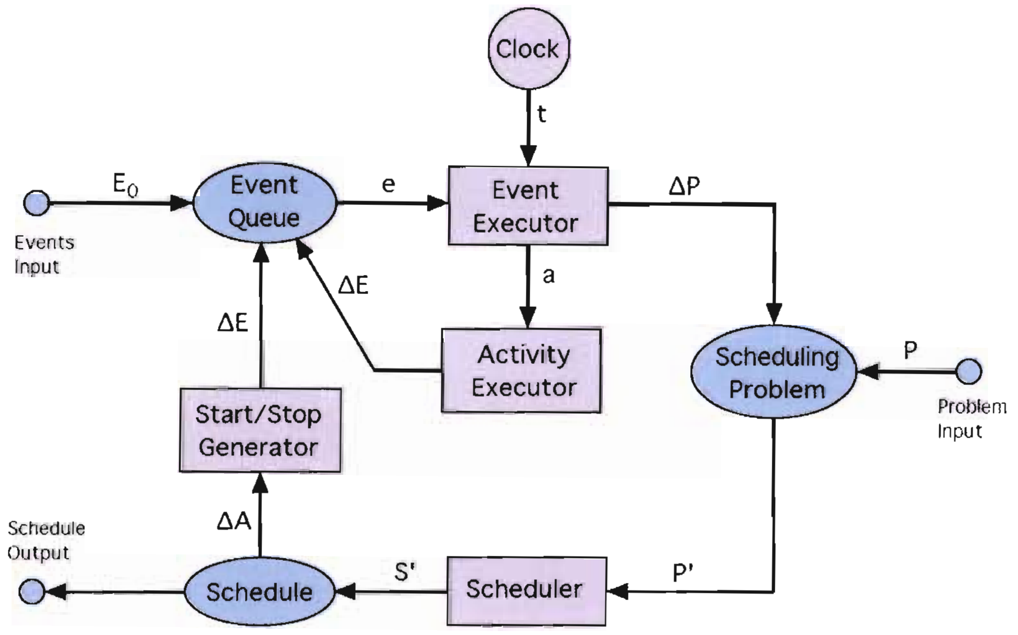

Figure 4.2: Dynamic simulator Model

the task and one representing its end time. Likewise, for an unscheduled task which had been scheduled before invoking the scheduler, a corresponding deletion of the start and end time events is added. Finally, rescheduled tasks have corresponding changes represented in ΔE.

3. Event changes are also produced by the Activity Executor by executing the body of missions and tasks. A mission's body is executed when a mission is created and added to the scheduling problem. The body of a task is executed when the simulation clock has reached its starting time. In the event that a task has been completed or terminated abnormally, the Activity Executor is also responsible for removing the corresponding future events from the queue.

The Event Executor module uses the Event Queue to organize events in temporal order and a simulation clock is used to advance the simulation. The simulation starts at the beginning of the scheduling horizon and stops when the clock reaches the end. The Event Executor repeatedly removes the first event e from the queue and advances the simulation clock to its event time. If this event is a dynamic event which introduces a modification ΔP to the scheduling problem (see Section 3.3), it is executed and the scheduler is invoked. Otherwise, e deals with the execution of a task. We define three additional types of events which are delegated to the Activity Executor. The remainder of this section gives a short description of these events. We omit their syntax as they are created automatically by the framework.

- **Send Resource**: This dynamic event signals that the assigned resource of a task has initiated the positioning leg in order to execute the task. A positioning leg is an activity which moves the assigned resource to the starting location of the task. Hence, event parameters include the event time, the id of the task as well as the unique identifier of the resource.
- **Start Task**: The start task event signals the start of execution. Parameters for this event include the event time and the unique identifiers of the task. It is not necessary to include the id of the resource, since this information is already known from the send resource event.
- **End Task**: This dynamic event simulates the end of execution of a task. Event parameters are the event time and the id of the completed task. For same reason as

for the start task event, it is not necessary to include the id of the resource. Since all resources in the problem are renewable, the assigned resource is free to start execution of another task immediately.

4.2.2 Visualization Tool

The dynamic simulator includes a visualization tool which creates an animation of the scheduling problem on Google Earth. We have chosen this application, because we can make use of all their current features, such as changing camera angles, and additional features from future updates. Interfacing with this application is achieved by using KML files[2] to display geographic data in an Earth Browser. Models for resources were obtained from an online database[3]. The animation steps through the scheduling horizon and visualizes the different entities. It is even possible to halt the simulation clock anytime in order to investigate some state in detail. The animation is made up of the following three entities:

1. **Missions**: An instance of a dynamic scheduling problem might contain many missions and displaying all of them simultaneously might overcrowd the screen. We group the scheduled tasks based on the mission they belong to so that entire missions can be hidden at once. It is even possible to visualize a single mission only and watch the animation in an attempt to understand the decisions made by the scheduler. This can be a very useful tool for analyzing and understanding the pros and cons of a scheduling algorithm.

2. **Tasks**: A task is visualized in Google Earth using points and paths to indicate its function. For instance, a search task can be visualized by drawing the reported location as a point and the flight path of the resource as a sequence of connected lines. Every task has to implement a method *writeKML* which is called by the scheduler at the end of the scheduling horizon. A task is included in the animation as soon as it has been added to the scheduling problem and it is removed when the simulation clock reaches the end of its execution window. Tasks which are completed successfully will be hidden the moment they finish execution, since they are no longer interesting.

[2]KML - Documentation, http://code.google.com/apis/kml/documentation/docIndex.html
[3]3D Warehouse, http://sketchup.google.com/3dwarehouse/

3. **Resources**: A resource is visualized during the entire scheduling horizon and its location is updated every minute. Resources are hidden during time intervals in which they are removed from the problem due to dynamic events. We associate a 3D model with every resource type so that it is very easy to differentiate between them on the Earth Browser.

Figures 4.3 and 4.4 give sample screenshots of the visualization tool.

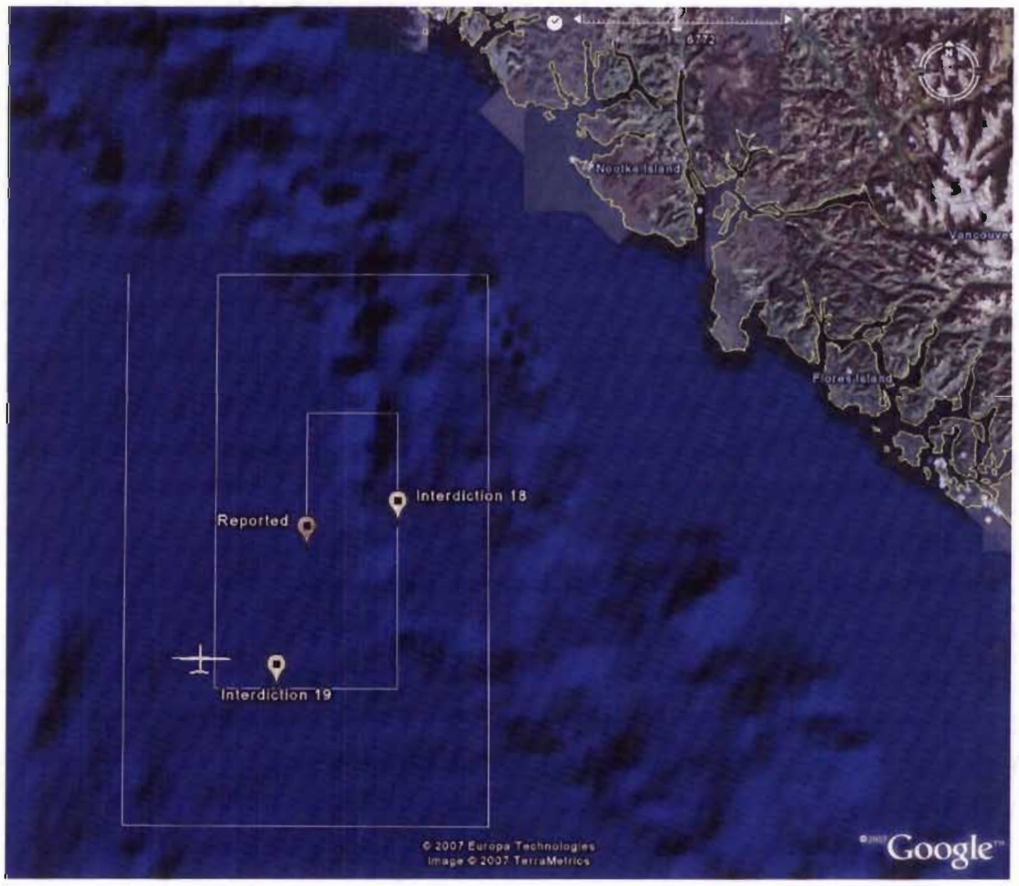

Figure 4.3: Screenshot from Visualization Tool - View from top

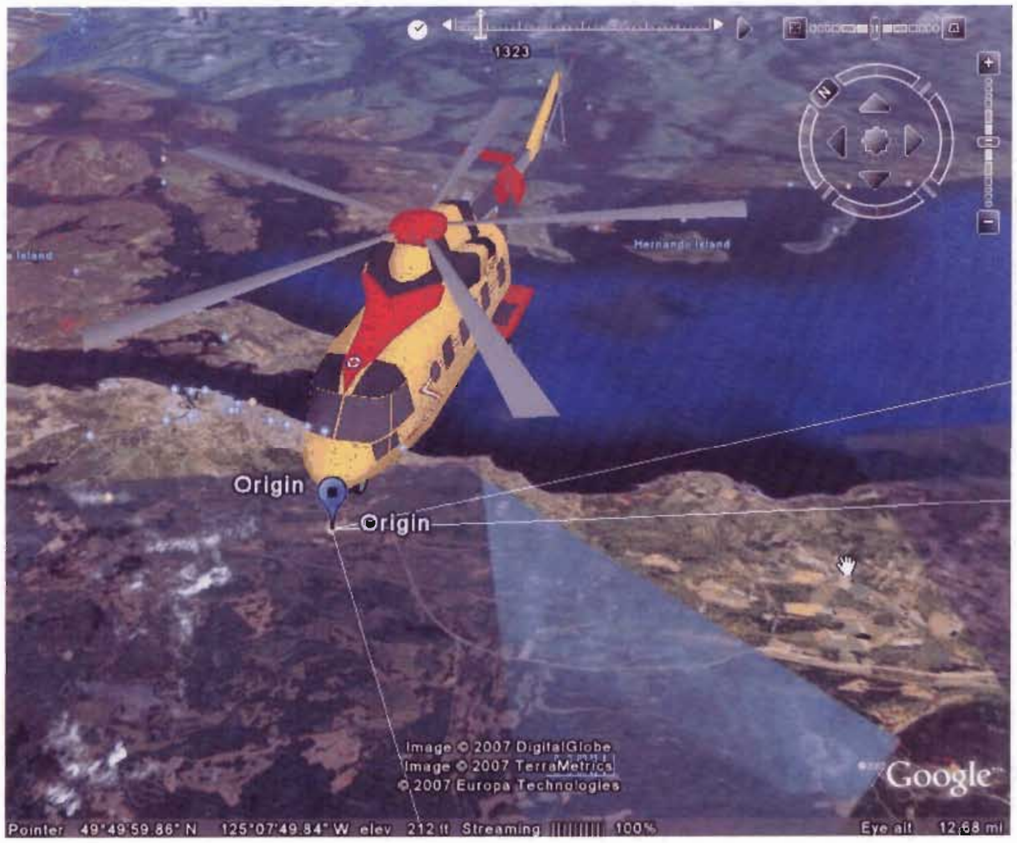
1323
Hernando Island
Origin
Origin
Image © 2007 DigitalGlobe
Image © 2007 TerraMetrics
© 2007 Europa Technologies
Google
Pointer 49°49'59.86" N 125°07'49.84" W elev 212 ft Streaming 100%
Eye alt 12.68 mi

Figure 4.4: Screenshot from Visualization Tool - View from other aircraft

4.3 Scheduler

The scheduler is a platform for scheduling algorithms and it communicates with the simulator. It is invoked every time a dynamic event occurs which changes the underlying scheduling problem. To test an algorithm on the generated datasets, one would simply implement the new algorithm in a method called *run* as a subclass of *Scheduler.* The scheduler component also provides several useful methods:

- **setUp**: This method makes a backup of the current schedule by storing the scheduling information for every task and resource in their respective classes. It also locks the Event Queue in order to guarantee that the scheduling algorithm cannot modify the scheduling problem.

- **recordNewBest**: This method records the current schedule as the best one encountered so far. It is important to update the best schedule found during the run of the scheduling algorithm, because in the end it will be used to updated the previous schedule.

- **finish**: This method restores the backup of the schedule before the scheduler was invoked. It then determines the changes that are necessary to obtain the best schedule encountered during the run and sends them to the Start/Stop Generator.

- **findEST**: Given a resource and a task, this method finds the earliest possible starting time of the task on the given resource. This method returns $-\infty$ if the task cannot be executed during it's time window.

- **rescheduleEST**: Given a resource, this method reschedules all its assigned tasks by scheduling them as early as possible. The tasks are considered according to the order in which they are scheduled before the method is called. This method can be used to repair a schedule that was made invalid by a dynamic event.

Consider the Gantt chart[4] in Figure 4.5 of tasks scheduled on the same resource along with their time windows for execution.

Assume task *T1* is currently executing and its duration is increased by 5 minutes due to some unexpected circumstances. The rescheduleEST method would try to schedule task

[4]A Gantt chart is a graphical representation of a schedule in which the horizontal axis represents time[11].

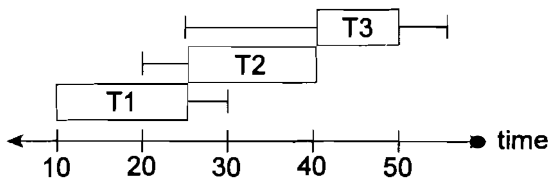

Figure 4.5: Gantt chart of tasks scheduled on same resource

T2 as early as possible and immediately recognize that there is no suitable time, since the execution would finish after its latest finish time. Consequently, since *T3* is the next task to be rescheduled, it will now start at time 30. The Gantt chart of the modified schedule can be seen in Figure 4.6.

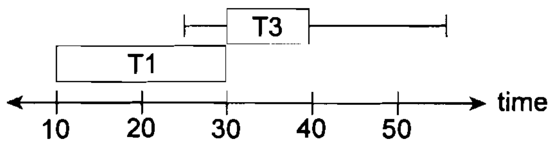

Figure 4.6: Gantt chart after change duration event

4.4 Other Issues

4.4.1 Precedence constraints

As defined in Section 3.1.2, precedence constraints express the starting time of an activity in terms of another activity's starting time. Precedence constraints can be specified between two tasks belonging to the same mission and between a task and the mission itself. It turns out that implementing these kinds of constraints into our dynamic scheduling framework is not that simple.

It is trivial to guarantee that a precedence constraint between a mission and one of its tasks is obeyed. Suppose in the given scheduling problem the following precedence constraint exists:

```
precedence mission1 task1 d
```

As a result, *task1* has to start at least d minutes after the creation of *mission1*. The earliest time for the creation of a task is when its mission is added to the scheduling problem. As long as we can guarantee that the task will not execute before d minutes are elapsed, the given precedence constraint is obeyed. This can be achieved by selecting a value for the release date rd of *task1* such that $rd \geq d$. Hence, there is no need to include this precedence constraint in the scheduling problem.

However, it is not that simple for precedence constraints between two tasks. Assume there exists a task A which adds two subtasks to the problem instance sometime during its execution. We denote the subtasks of A as B and C.

Suppose there is a precedence constraint between tasks A and B. Using the same argument as before, we do not need to add the precedence constraint to the problem since it can be enforced using the release date parameter of task B. In general, this argument holds for any precedence constraint between a task and one of its subtasks. However, it doesn't hold for a precedence constraint between B and C with delay d.

Assume that B is created earlier than task C. After running the scheduling algorithm, task B might start execution right away. Eventually task C will be created, but what if the other task has already been executing for d or more minutes? Then the constraint has been violated without any fault of the scheduler. It is not possible to guarantee that this precedence constraint is obeyed at all times.

Consider the following solution to the problem: Instead of adding the given precedence constraint to the scheduling problem, modify the dataset by moving the task C into the body of task B. Setting the release date of task C to a value $\geq d$ enforces the constraint. Hence, we can completely ignore the use of precedence constraints if we generate datasets such that they are implied automatically by the definition of our dynamic scheduling model.

4.4.2 Rescheduling of Tasks

Semi-preemptive scheduling problems allow interruption of tasks in order to reschedule them, possibly on a different resource. Our model uses the concept of bodies which contain dynamic events that will be put into the event queue when the execution of a task commences. However, what should be done with the dynamic events if a task is interrupted and scheduled to execute a second time?

Our solution is very simple: During simulation of the problem instance, we keep track of the dynamic events contained in task bodies and allow every event to occur only once. Any further execution of the body will ignore those events which have already happened. This is especially important for the creation of subtasks. Suppose a new task event has already occurred. The second time it is executed, the Event Executor will try to create another task with the same id and violate the problem definition since task identifiers must be unique. The side-effect of our solution is that if the execution of a task is delayed on a resource by such a large amount that it needs to be interrupted, it will not be delayed again.

Chapter 5

Case-Study: CoastWatch

We study the Canadian CoastWatch Dynamic Resource Scheduling Problem, in short *CoastWatch.* It is an oversubscribed dynamic multi-mode scheduling problem with unit resources. The task is to schedule both routine and emergency missions within a Search & Rescue (SAR) operational command. CoastWatch datasets simulate a typical day for the Canadian Coast Guard, where officers assign resources (planes, helicopters, ships, ...) to execute several different kinds of missions (patrol, transport, ...). There are more routine patrol missions than can be flown by the available resources. Unexpected SAR missions are of highest priority and must be accommodated in the schedule if possible.

5.1 Problem definition

CoastWatch can be defined as follows:

- **Missions**. $M = \{m_1, m_2, ..., m_n\}$ is a set of missions which have to be completed. A mission has a priority p_i and is composed of a set of tasks T_i, all of which have to be completed in order for the mission to be considered accomplished. Every mission has an associated set of dynamic events called its *body.* These events occur at specified times after the creation of the mission.

- **Tasks**. $T = \{T_1, T_2, ..., T_n\}$ is a set of set of tasks which have to be scheduled. T_i contains all tasks that belong to mission m_i. Similar to missions, every task has a *body* which contains a set of dynamic events that occur at specified times after the start of execution of this task. A task is characterized by the following parameters:

  - *rd*: the release date of the task. It must be scheduled at this time or later.
  - *dd*: the due date of the task. Execution must terminate on or before this time in any schedule.
  - *CR*: the set of resources which can service the task. We also refer to them as capable resources.
  - *type*: the type of the task. This parameter determines the set of capable resources.
  - *D*: the set of durations containing the execution times of the task depending on the assigned resources. The duration may be altered during execution by dynamic events.

- **Body**. A set of dynamic events associated with a task or mission. These events, which may affect the underlying scheduling problem, occur at specified times after the creation of the mission or the execution of the task.

- **Resources**. $R = \{r_1, r_2, ..., r_k\}$ is a set of renewable unit resources which are scheduled to perform tasks. C is the set of capabilities, or task types, a resource is able to perform.

The scheduling problem is semi-preemptive, meaning the execution of a task may be interrupted and restarted from the beginning at a later time.

5.2 Description of Tasks and Resources

5.2.1 Tasks

Search

The purpose of the search task is to find a missing person. After receiving an S.O.S. signal, the search task is performed by starting at the reported location and spiraling around it, while constantly increasing the distance to the starting point. Figure 4.3 shows an example for such a search path. Every time it is believed that the missing person has been sighted, an interdiction task is generated at that location.

Interdiction

The interdiction task is generated to identify whether the object that was found during the search task, is the missing person or not. Unfortunately, since resources fly high above sea level in order to have a larger area of sight, it is possible that the object turns out not to be the missing person. If the missing person is identified successfully, a rescue task is generated at the same location. Regardless of the outcome, the interdiction task is completed when identification of the object has completed.

Rescue

The rescue task is generated when the missing person has been located. The task involves rescuing the person and transporting him back to the nearest base. Rescuing the person may be very tricky and special equipment may be necessary to perform the task. Consequently, not every resource type is able to perform a rescue task.

Patrol

The patrol task is generated when it is necessary to fly over a specific area to provide protection or simply to be aware of movements of other ships or aircrafts. Typically, there exist more such tasks that can be executed by the given resources. The pilot is given a sequence of points which make up the flight path. After completing a specified number of rounds, the task is considered accomplished.

Transport

The purpose of the transport task is to transport humans and/or goods from one base to another. This task is the simplest and can be executed by virtually any resource.

5.2.2 Resources

Aurora

An Aurora aircraft can fly up to 750 km/h and is able to perform search, interdiction, patrol and transport tasks. These kinds of resources are the fastest ones that are included in the CoastWatch problem. Figure 5.1 shows a picture of the aurora aircraft along with the model used in the Visualization Tool of the Dynamic Simulator.

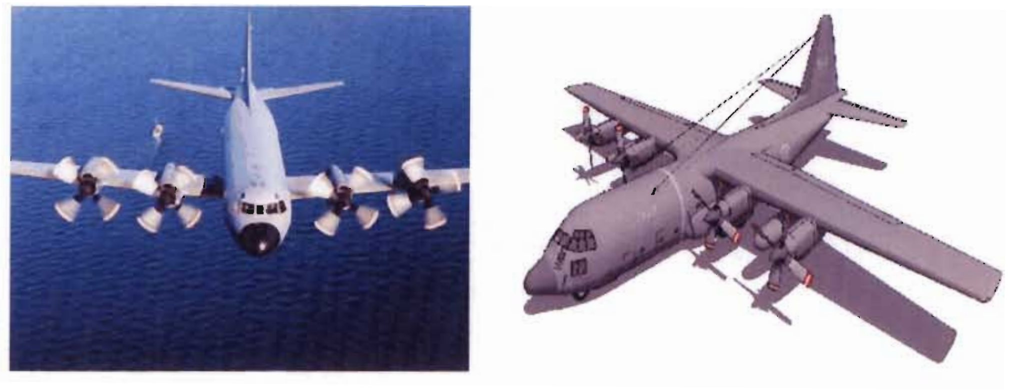

Figure 5.1: Picture and model of an Aurora aircraft

Cormorant

A Cormorant is the helicopter typically used for Search & Rescue. It is very flexible and is able to perform all types of tasks. The Cormorant flies at a speed of 278 km/h and is shown in Figure 5.2.

Figure 5.2: Picture and model of a Cormorant helicopter

Cyclone

The Cyclone helicopter has very similar characteristics compared to the Cormorant. It is also able to perform all task types, but moves a little faster at a speed of 305 km/h. Figure 5.3 shows a Cyclone helicopter in action and the corresponding Google Earth model.

Figure 5.3: Picture and model of a Cyclone helicopter

Eagle-UAV

The Eagle aircraft is an Unmanned Aerial Vehicle which flies up to 207 km/h. As the name implies it flies by itself without the need for a pilot. Planes are remotely controlled and equipped with cameras allowing the Canadian Coast Guard to use them for finding missing persons. Eagle aircrafts can be used for search and interdiction tasks and are shown in Figure 5.4.

Figure 5.4: Picture and model of an Eagle Unmanned Aerial Vehicle

Frigate

The Frigate ship is the slowest resource available in the CoastWatch dynamic resource scheduling problem. It travels at a speed of 54 km/h, but can perform all types of tasks with the exception of transport tasks. As a consequence of its low speed, Frigates typically are not involved in tasks which require a lot of travel. But they may be close to the task locations and act as first responders. Figure 5.5 shows the model used in the Earth Browser as well as a picture of a Frigate out in the ocean.

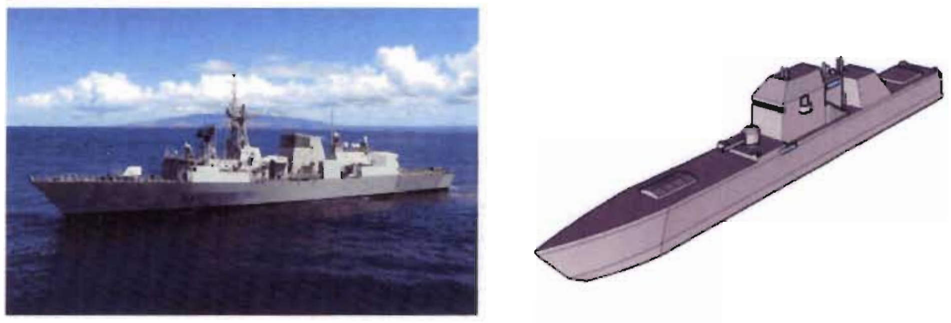

Figure 5.5: Picture and model of a Frigate ship

5.3 Benchmark Datasets Generation

This section contains a listing of all parameters values that we have selected for generating benchmark datasets for the CoastWatch dynamic resource scheduling problem. By changing parameter values, generated datasets may have very different characteristics.

- **Scheduling horizon**: 0 to 1440. We measure time in minutes and set the size of the scheduling horizon to equal a whole day.
- **numBases**: 4. We define a set of 4 real-world bases and include them in every dataset. The selected bases are CFB-Comox, CYBL-CampbellRiver, YAZ-Tofino and YVR-Vancouver.
- **numResources**: 10. We specify 18 different resources in the scheduling problem file for CoastWatch: 2 Aurora aircrafts, 4 Cormorant and 4 Cyclone helicopters, 4 Eagle Unmanned Aerial Vehicles and 4 Frigate ships. Aurora aircrafts are kept very scarce, since they are much faster and would otherwise dominate the other resources by performing most of the tasks. Every problem instances contains 1 resource for every type and randomly selects the remaining 5.

5.3.1 Dynamic Events

Delay Task

probability: 0.1, *delay*: random(-10,60).

This dynamic event shifts the time window of a task by a given delay. This delay can be positive or negative. The latter case refers to a task that can start execution ahead of schedule. This event has to occur at least 1 minute ahead of the earliest possible starting time of a task. If the execution time window of a task starts at the time of its creation, no delay task will be created.

We assume that 10% of all tasks will be delayed. However, we allow no such event for interdiction and rescue tasks. Additionally, the execution of search tasks should never be delayed. But this is already guaranteed because we set its earliest possible starting time to be the time at which the task was created. Additionally, we set the maximum possible delay to be one hour and allow a task to be started at most 10 minutes earlier than first anticipated.

Change Duration

probability: 0.2, *relativeTime*: random(1,99), *delay*: random(-10,25).

The change duration event causes the current execution of a task to be delayed due to some unforeseen circumstances. Similarly to the delay task event, the delay can be positive or negative.

20% of all tasks, excluding interdiction tasks, experience a change in its duration. The relative time is set to a value between 1 and 99 meaning this event can occur anywhere during the execution of a task. A task can be executed up to 10% faster than first anticipated, which equals a delay of -10%, but its duration may be increased by up to 25%.

Disable Resource

numResources: 2, *time*: random(30,120).

This disable resource event temporarily removes a resource from the problem instance. After a specified amount of time has elapsed, the resource is added back into the scheduling problem. We modify the definition of this event as given in the model by adding another parameter which determines the number of resources that will be disabled throughout the scheduling horizon. We assume that a resource cannot be disabled more than once.

In our experiments, we simulate a typical day for the Canadian Coast Guard by assuming that two resources will experience technical difficulties and be temporarily disabled. We assume that repair will take anywhere from 30 minutes to 2 hours.

5.3.2 Tasks

We give a brief description of the different types of tasks that are included in our generated benchmark datasets. In total we generate 60 missions for each dataset: 30 patrol, 20 transport and 10 search and rescue. Since search & rescue involves several tasks that need to be completed to accomplish one mission, they are the most difficult missions, but it would be unrealistic to generate a large number of them. We create two types of transport tasks: static and dynamic. We assume that half of the transport tasks are known the previous day and therefore should be known at the beginning of the time horizon. Their release date is normally distributed around the middle of the day, so that theses missions typically occur during day time. The other half of transport tasks are spontaneous. They are generated throughout the day with much their time window of execution starting within the next 2 hours.

Transport-static

numStatic: 10, *numDynamic*: 0, *priority*: random(1,10), *relativeTime*: 0, *releaseDate*: normal(800,100).

Ten static transport missions are generated in our problem instances. They are known at the beginning of the scheduling horizon with their release dates being normally distributed around the early afternoon.

Transport-dynamic

numStatic: 0, *numDynamic*: 10, *priority*: random(1,10), *relativeTime*: 0, *releaseDate*: random(0,120).

Dynamic transport tasks have different characteristics than static ones. They are generated throughout the day, with their earliest possible starting time being within 2 hours of their creation time. All transport tasks have low priority values.

Patrol

numStatic: 30, *numDynamic*: 0, *priority*: random(20,30), *relativeTime*: 0, *releaseDate*: normal(0,1440), *maxDistance*: random(120,150), *numPoints*: random(5,10), *numRounds*: random(3,4).

The patrol task is generated when it is necessary to fly over a specific area to provide protection or simply to be aware of movements of other ships or aircrafts. Patrol missions have medium priority and are all known at the beginning of the scheduling horizon. To execute such a mission, the pilot is given a sequence of points which make up the flight path. After completing a specified number of rounds, which we have chosen to be either 3 or 4, the task is considered accomplished. We generate the sequence of points by first generating a starting point. The remaining 4 to 9 points are created randomly, within 120 to 150 kilometers from the starting point. We select the closest of these points to be the second point along the flight path. From the remaining randomly generated points we select the third point to be the closest one as measure from the second point. We continue this process until there exists a total order of all points.

After experimenting with different parameter values, we found that it is very important to limit the flexibility in length that exists for patrol missions. For instance, allowing a maximum distance of 50 km or less and a small number of points and rounds, we sometimes generated patrol tasks that took an Aurora aircraft only several minutes to execute.

Search

numStatic: 0, *numDynamic*: 10, *priority*: random(150,250), *relativeTime*: 0, *releaseDate*: 0, *numInterdiction*: random(1,5), *radius*: random(50,75).

We generate 10 Search & Rescue missions during the course of an entire day. These have a very large priority because human life is involved. The search path to be flown by the pilot is pre-determined: starting at the reported location, the resource will spiral around it while constantly increasing its distance. We set the search radius to be between 50 and 75 kilometers from the reported location. During execution of this task, 1 to 5 interdiction tasks may be created.

Interdiction

relativeTime: random(1,100), *releaseDate*: 0, *duration*: random(1,5).

The purpose of an interdiction task is to identify whether the object sighted during the search task is the missing person. The relative time is chosen such that these tasks can occur anywhere along the search path. They can be executed immediately and have very short durations.

Rescue

relativeTime: 100, *releaseDate*: 0, *duration*: random(3,15).

Once it has been verified that the missing person has been found, the rescue task needs to be performed. Relative time is set to 100%, since the interdiction task needs to complete in order to know for sure whether the object is the missing person. Rescuing the human takes anywhere from 3 to 15 minutes, but the actual duration is much longer, because the person needs to be transported to the nearest base in order to get him to a hospital.

Chapter 6

Algorithms

In this chapter, we describe various scheduling algorithms that we have implemented for the CoastWatch Dynamic Resource Scheduling problem case-study. We use Tabu search as a uniform platform, because stochastic local search algorithms have been proven to be successful for hard scheduling problems [11]. We use this platform to test various heuristics and neighbourhood strategies.

We encode potential solutions using a permutation of the tasks and write a schedule builder which generates a schedule from the permutation. This approach is similar to Barbulescu et al. [1]. The permutation acts as a priority queue and each task is assigned to the first available resource at the earliest possible starting time.

In dynamic scheduling problems, the addition of a new mission or the occurrence of a dynamic event may cause a lot of disruption in the schedule. On a busy resource, it is very likely that a delay of a task propagates to other tasks that are assigned to be executed afterwards. Even worse than that, if we are dealing with time window constraints, then it is possible that a task can no longer be executed and might be reassigned to another resource. Although schedule disruption is not our main objective, we would still prefer a scheduling algorithm that minimizes this objective.

As a result, we modify our encoding to be a set of permutations of the tasks, one for each resource. This is done to help minimize schedule disruption since changes made on a single resource should effect fewer other tasks. We can assume there exists a dummy permutation which contains all the tasks that are not assigned to any resource.

6.1 Tabu search platform

Tabu search is a very simple stochastic local search algorithm that has been applied very successfully to many different kind of scheduling problems. In general, Tabu search algorithms work as follows [11]:

1. Determine initial candidate solution *s*
2. While termination criterion is not satisfied
   (a) Determine set *N* of non-tabu neighbours of *s*
   (b) Choose a best improving solution *s'* in *N*
   (c) Update tabu attributes bases on *s'*
   (d) Set *s* equal to *s'*

A neighbour of the current solution is another candidate solution that can be reached by making a single modification to it. The set of all possible neighbours during an iteration is called *neighbourhood*. Tabu search remembers the selected moves because for a certain number of iterations, called the *tabu tenure*, it will disallow moves that reverse a previous move. Typically, an exception is made for moves which improve the best solution encountered during the run.

6.2 Neighbourhoods

In a local search strategy, the neighbourhood is defined as the set of all possible moves to modify the current solution. For the CoastWatch problem we experiment with different neighbourhoods by running them on the Tabu search platform.

First we will explain the different move operators that we consider. Then we will give a brief description of the various neighbourhoods.

Recall that we encode candidate solutions using a permutation of tasks for each resource. After applying the move operator a schedule builder will traverse through all resources and try to schedule the tasks on the given resource as early as possible. The permutations act as a priority queue which determines the order in which the tasks are considered.

For our experiments we allow the following move operators:

- **Switch Resource**: switches a given task to another resource
- **Move Task**: moves a task to another position within the permutation
- **Add Task**: adds a currently unscheduled task to one of the permutations

The three neighbourhood variations we test on the CoastWatch datasets are:

1. **Full**: Permits all three types of moves and exhausts all possible moves during every iteration. This neighbourhood is computationally expensive, however, it will provide the greatest flexibility since it can explore the whole search space.
2. **AddOnly**: Allows only *add task* moves. The idea behind this neighbourhood is to try to insert more tasks into the schedule, while leaving scheduled tasks alone. This method differs from the *MissionSwap* algorithm [17] as described in chapter 2, because we do not require retracted tasks to be rescheduled.
3. **Restricted**: This neighbourhood combines features of both, *Full* and *AddOnly*. It has a parameter *max moves* which limits the number of moves per iteration that Tabu search is allowed to consider. The neighbourhood will first consider all *add task* moves and then randomly select *switch resource* or *move task* moves until the maximum number of moves has been reached.

6.3 Scheduling Heuristics

The three possible neighbourhood moves, *switch resource*, *move task* and *add task*, all need to select new positions within the permutation. We define several scheduling heuristics which will decide where in the permutation the task is inserted:

- **Random**: This algorithm selects the new position within the permutation purely at random. In the worst case, it might schedule a task such that no other assigned tasks of the resource may still be completed in time. Its performance won't be very good, but we use this strategy as a baseline.
- **MaximizeObjective**: This heuristic places the task in the permutation such that the given resource itself contributes as much to the objective value as possible. This

technique should perform very well since its optimization goal is exactly the objective function of the scheduling problem. However, optimizing this objective on each resource doesn't necessarily produce the best overall results.

- **MinimizePositioningTime**: For a given task, this algorithms selects the new position within the permutation such that in the resulting schedule, its positioning times are minimized. In other words, this strategy will attempt to schedule the task such that the distance from the previous task and the distance to the next task are as small as possible. For the first task to be completed by a resource, we consider its positioning time from the home base, while for the last task we do not include any positioning leg after it has been completed. This strategy attempts to minimize the time that resources spend on positioning legs in order to maximize utilization.

- **SpreadOutResources**: This heuristic places the task in the permutation such that resources are spread out as much as possible during the remainder of the simulation. By doing so we provide more flexibility to the Canadian Coast Guard, since we ensure that resources are operating in very different locations. As a result, we increase the probability that a new task can be executed very quickly. We generate a specified number of absolute times spaced evenly throughout the remainder of the simulation and determine the position of every resource according to the current schedule. We sum up the square distances of the assigned resource to all others, which will give us a good measure of how spread out resources are during the remainder of the scheduling horizon.

6.4 Discussion

Before discussion of experimental results, we would like to give a short comparison of our Tabu search algorithm to the *MissionSwap* and SWO algorithms from Kramer et al. [16]. Their study attempts to classify scheduling problems for which it is best to use a specific type of local search algorithm. The Tabu search algorithm we selected for CoastWatch differs significantly from either of the studied algorithms. The difference to the studied SWO method is that our algorithm contains a permutation of assigned tasks for each resource. This is done to help minimize schedule disruption since changes made on the permutation of a subset of tasks should effect fewer other tasks. Additionally, after each iteration of Tabu search we update the permutations to reflect the actual order that tasks will be executed on the different resources. This limits the explored search space since all permutations that lead to the same schedule will cause an implicit jump to the same region of the search space. There are two main differences between our Tabu algorithm and *MissionSwap*: first, we do not require that all retracted tasks be rescheduled when inserting a new task and secondly, we move through the search space by changing the permutations rather then operating on the schedule themselves.

Chapter 7

Evaluation & Experimental Results

7.1 Evaluation Criteria and Methodology

The algorithms described in Chapter 6 were all developed for oversubscribed scheduling problems. These are scheduling problems for which not all tasks can be scheduled and the algorithm needs to select the best subset of these tasks that can be completed while obeying all problem constraints.

The CoastWatch scheduling problem is such an oversubscribed scheduling problem and consequently, we use the problem generator to generate more missions than can be executed by the available resources. We estimate the value of oversubscription (ov) by dividing the sum of average durations of all tasks by the size of the scheduling horizon multiplied by the number of resources:

$$ov = \frac{\sum_{t \in T}^{T} \frac{worst_t + best_t}{2}}{r * (t_{end} - t_{start} + 1)}$$

where: ov is the oversubscription value of a problem instance, T is the set of all tasks, $worst_t$ and $best_t$ refer to the worst and best task durations, respectively, r equals the number of resources in the scheduling problem and t_{start} & t_{end} refer to the beginning and end of the scheduling horizon, respectively.

The oversubscription value estimates the number of tasks a resource has to execute concurrently in order to complete all tasks during the simulation of the scheduling problem. If $ov > 1$, then most likely the given problem instance is oversubscribed. Typically, the sum of average durations underestimates the actual time it requires resources to execute the given tasks. This is because resources also need to travel to the starting location of the task, and the durations of these positioning legs are not included. We generate 100 datasets with oversubscription values between 1 and 5 and use them to test various algorithms.

We select our objective to be to maximize the sum of priorities of all accomplished missions:

$$obj = \sum_{t \in T}^{t} c_t * priority_t$$

where T is the set of all tasks, $priority_t$ is the priority of task t, and c_t equals 1 if the task has been completed within the given execution time window, and 0 otherwise.

This objective function is a simple solution quality measurement for oversubscribed scheduling problems. Other objectives, such as minimizing the lateness of tasks, introduce some difficulties, since only a subset of the tasks will actually be executed.

Additionally, we collect statistics about schedule disruption, because we would prefer an algorithm that reschedules fewer tasks. We measure disruption by comparing the schedules before and after running the scheduler and summing up the number of tasks that:

- have been previously been unassigned, but are now scheduled on a resource, and
- have been assigned previously, but are now assigned to a different resource, and
- have been rescheduled. We do not include tasks which have been assigned a different time slot or resource as a result of rescheduling some other task. In other words, we only count tasks that have been rescheduled because they have been selected by the algorithm.

7.2 Implementation

As specified in Section 3.3, we use relative event times for dynamic task events created during the execution of tasks. In our implementation we restrict these relative times to any integer i, such that $0 \leq i \leq 100$. When the body of a task is executed we divide these event times by 100 to get the percentage of completed execution after which the event occurs. The reason for this difference in implementation is that it is more convenient to store all event times as integers, regardless of their types.

As explained in Section 4.2.1, the initial scheduling problem must be parsed by the dynamic simulator before the simulator can start. We achieve the same effect by inputting the initial problem instance as a set of events with event times equal to 0. Additionally, we do not implement a Start/Stop Generator module, because we provide the desired functionality in both Task and Resource classes.

In our implementation of the event model, we further divide every task into a sequence of actions. As a consequence we are able to calculate the duration of tasks by summing up the durations of their actions. This minimizes the required effort to introduce new types of tasks. There exist only two different types of actions: *Go* and *Wait*. The *Go* action requires a resource to move from one location to another, while the *Wait* action instructs a resource to remain at a specified location. Every task can be translated into a sequence of these actions.

7.3 Experimental Results

We perform two different experiments on the generated CoastWatch datasets. Both experiments are run on a 3.0 GHz computer with 1GB of RAM. We measure run-time in seconds per scheduler call in order to determine how long it would take an algorithm to respond to the occurence of one dynamic event. Total run-time does not give much information about the efficiency of an algorithm since the execution of more tasks will result in more dynamic events, and consequently in more scheduler calls.

In the first experiment we investigate the effect of neighbourhood choice on the studied dynamic scheduling problem. We exclusively use the random scheduling heuristic for this experiment, in order to have no bias regarding the new position in the permutation.

The second experiment compares the performance of the various scheduling heuristics on the CoastWatch benchmark datasets.

7.3.1 Experiment 1 - Neighbourhoods

Experiment 1 compares the performance of *Full*, *Restricted* and *AddOnly* neighbourhoods on a set of oversubscribed problem instances, consisting of problems 1 to 50 from the CoastWatch benchmark datasets. The random scheduling heuristic is used exclusively in this experiment, and hence, all results are averaged over 10 runs. Figure 7.1 shows typical run-time behaviour of the Tabu search algorithm on one of the datasets. In particular, it shows the *AddOnly* neighbourhood as run on dataset #1 with 100 iterations.

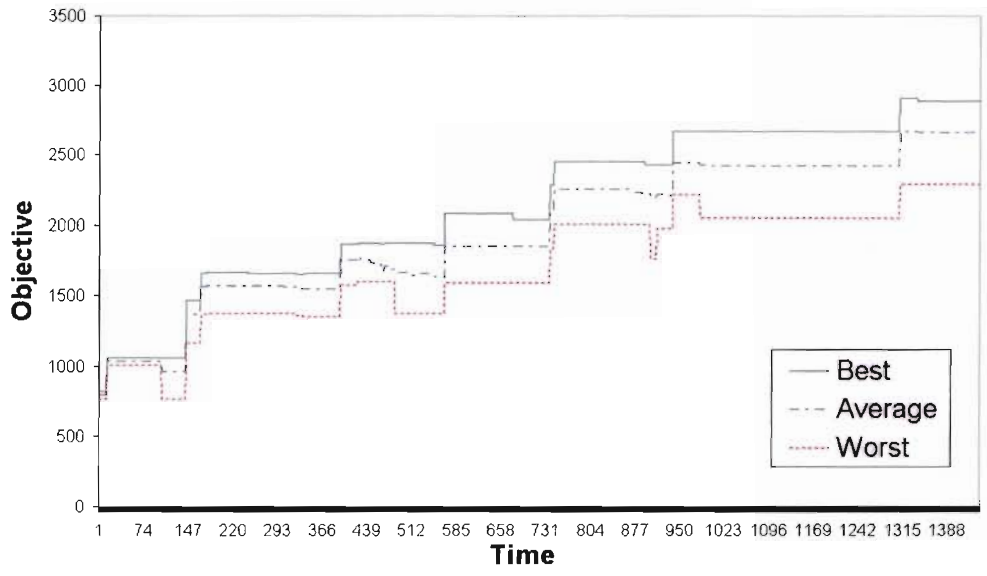

Figure 7.1: Runtime behaviour of *AddOnly* on Dataset #1 using 100 iterations

The graph shows a steady objective increase as time progresses in the simulation. This is because more and more missions are added dynamically and the objective value increases every time one of those missions is completed. The objective function is not monotonically increasing because sometimes subtasks are created that cannot be executed and consequently, the mission is not considered accomplished anymore.

Table 7.1 summarizes the performance of the *AddOnly* neighbourhood on all 50 datasets for 50, 75 and 100 iterations. Results are nearly identical: 100 iterations provide very little improvement over 75 iterations, which in turn performs almost identical to 50. This suggests that this particular neighbourhood has already reached its performance level after a small number of iterations and further improvements are due to luck.

Figure 7.2 shows a comparison of these runs for each dataset. All performance graphs in this chapter show the relative performance of the various algorithms. In this type of graph, we use the worst performing algorithm as a baseline and plot the difference between its average and worst performance (out of 10 runs). As a result, we can deduce the performance

Neighbourhood	Run-time	Objective	Disruption	Best
AddOnly 50	1.21 s	2442.60	144.31	11
AddOnly 75	2.13 s	2453.47	145.36	23
AddOnly 100	2.62 s	2461.84	147.45	16

Table 7.1: Average performance of *AddOnly* for 50, 75 and 100 iterations

variation that exists within different runs. For all other algorithms, we compare its average performance with the average baseline performance and visualize their difference for each dataset. Consequently, larger bars representer better performance.

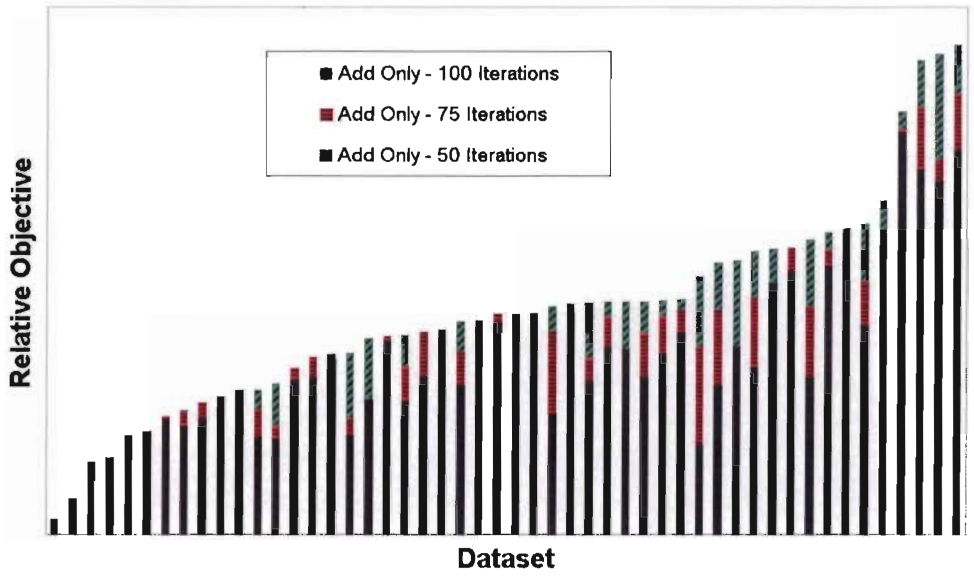

Figure 7.2: Performance comparison of *AddOnly* for 50, 75 and 100 iterations

The performance graphs for the *AddOnly* neighbourhood show nearly identical results when terminating Tabu search after various different number of iterations. 100 iterations provide very little improvement over 75 iterations, which in turn performs almost identical to 50. This suggests that this particular neighbourhood has already reached its performance level after a small number of iterations and further improvements are due to luck.

Figure 7.3 shows the performance comparison of *AddOnly*, *Restricted* and *Full* neighbourhoods on the benchmark datasets for 50 iterations with average run-times of 1.21, 2.53, 2.93 and 10.33 seconds, respectively.

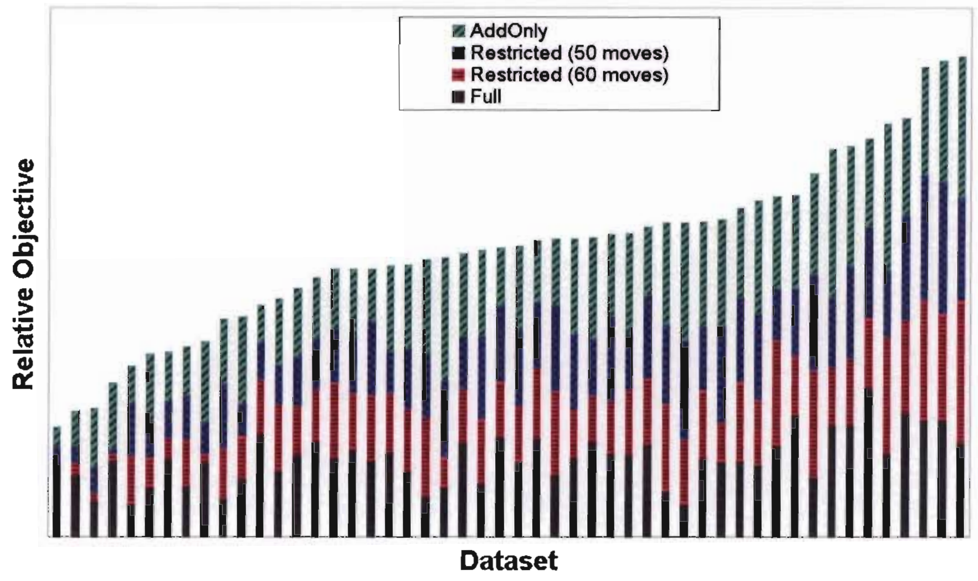

Figure 7.3: Performance comparison of *AddOnly*, *Restricted* and *Full* for 50 iterations

The *AddOnly* neighbourhood dominates all other neighbourhoods after 50 iterations. Additionally, it seems that using neighbourhoods which allow more moves during an iteration, leads to worse results. Although we have shown in figure 7.2 that when terminating the Tabu search after more iterations, performance improvement for *AddOnly* is minimal, this may not be the case for the other neighbourhoods. We repeat the experiment using 100 iterations and show the resulting performance graphs of all three neighbourhoods in figure 7.4. We summarize their performance in Table 7.2.

The relative performance of the different neighbourhoods using 100 iterations is identical to before. In an attempt to understand why these larger neighbourhoods lead to inferior solution quality, we modified the *Restricted* neighbourhood so that it does not try to reschedule tasks which are currently executing on any of the given resources. Since our scheduling problem is semi-preemptive, meaning that an interrupted task must be restarted

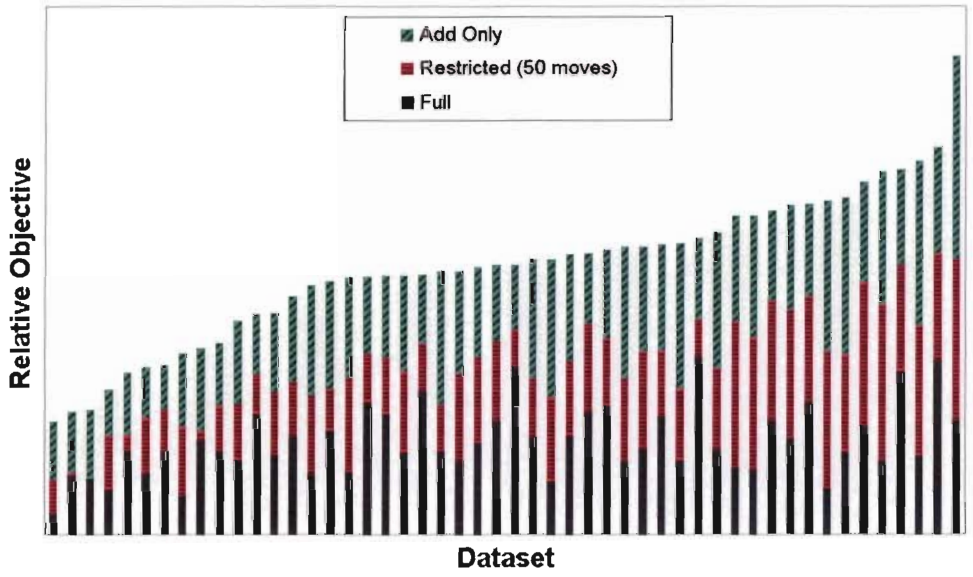

Figure 7.4: Performance comparison of *AddOnly*, *Restricted* and *Full* for 100 iterations

Neighbourhood	Run-time	Objective	Disruption	Best
AddOnly	2.62 s	2461.84	147.45	37
Restricted	5.30 s	2381.01	676.10	13
Full	32.09 s	2094.41	859.20	0

Table 7.2: Average performance of *AddOnly*, *Restricted* and *Full* for 100 iterations

Neighbourhood	Run-time	Objective	Disruption	Best
AddOnly	1.21 s	2442.60	144.31	29
Restricted	2.53 s	2355.91	500.50	9
Modified Restricted	1.57 s	2394.99	563.72	12

Table 7.3: Average performance of *AddOnly*, *Restricted* and *modified Restricted* for 50 iterations

from the beginning, it may not be a good idea to reschedule such tasks. Figure 7.5 shows the resulting performance graphs, while table 7.3 summarizes their overall performance.

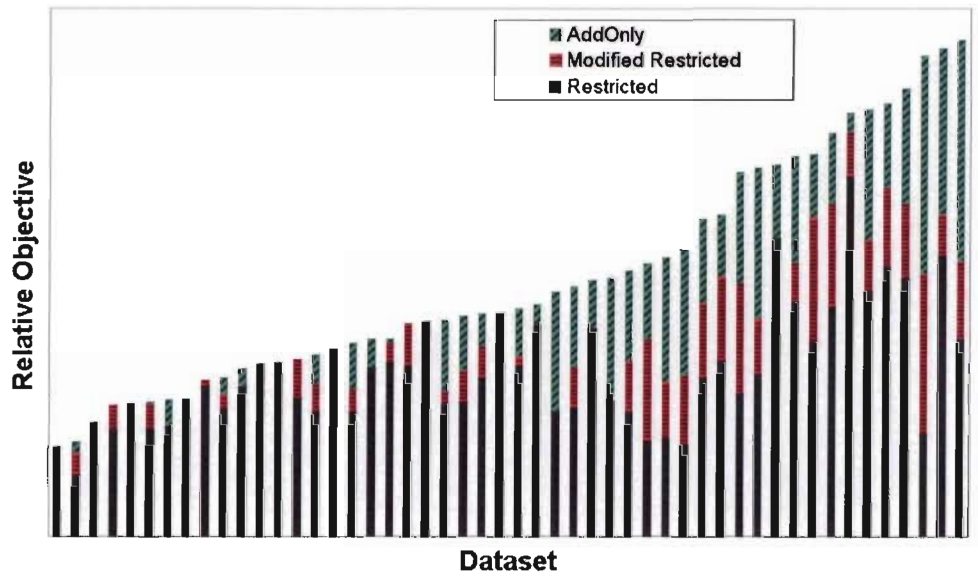

Figure 7.5: Performance comparison of *AddOnly*, *Restricted* and *modified Restricted* for 50 iterations

The modified *Restricted* neighbourhood resulted in better solution quality, however, it was not able to match the performance of *AddOnly*. This indicates that although rescheduling currently executing tasks contributes to the poor performance of large neighbourhoods, it is not the only contributing factor.

Neighbourhood 1	Neighbourhood 2	Result	Significant?
AddOnly	Restricted	0.000	YES
AddOnly	modified Restricted	0.001	YES
AddOnly	Full	0.000	YES
modified Restricted	Restricted	0.004	YES
modified Restricted	Full	0.000	YES
Restricted	Full	0.000	YES

Table 7.4: Paired sample t-test for *AddOnly*, *Restricted*, *modified Restricted* and *Full* for 50 iterations

To validate our results, we carry out paired sample t-test using SPSS[1] (Statistical Package for the Social Sciences). This test provides evaluation of the performance difference of two algorithms. A resulting value < 0.05 indicates significant difference in performance. The signficance tests between various neighbourhoods for 50 iterations are summarized in table 7.4.

We conclude that the *AddOnly* neighbourhood dominates the other two neighbourhoods in terms of solution quality and schedule disruption. As a result, we use it exclusively in experiment 2 to test various scheduling heuristics. *AddOnly* provides the least flexibility in moving from one candidate solution to another, but its advantage is that it concentrates on scheduling tasks which are currently unscheduled and does not re-arrange tasks which have already been scheduled. The *Full* neighbourhood provides the greatest flexibility, but requires the largest amount of computation time and actually performed the worst.

7.3.2 Experiment 2 - Scheduling heuristics

Experiment 2 compares the performance of the *Random*, *MaximizeObjective*, *MinimizePositioningTime* and *SpreadOutResources* heuristics on the CoastWatch benchmark datasets which consists of 100 randomly generated oversubscribed problem instances. These different strategies determine which position in the permutation a given task is assigned to. We exclusively use the *AddOnly* neighbourhood for our Tabu search algorithm because it outperformed all other neighbourhoods from experiment 1. We average the performance of *Random* over 10 runs for each dataset and perform only one run for all other scheduling

[1]More information can be found at http://www.spss.com/spss/

heuristics since they are deterministic.

Figure 7.6 shows the performance graphs of the various scheduling heuristics on the CoastWatch benchmark datasets and table 7.5 summarizes their average performance. The resulting schedule disruption of these strategies is shown in figure 7.7.

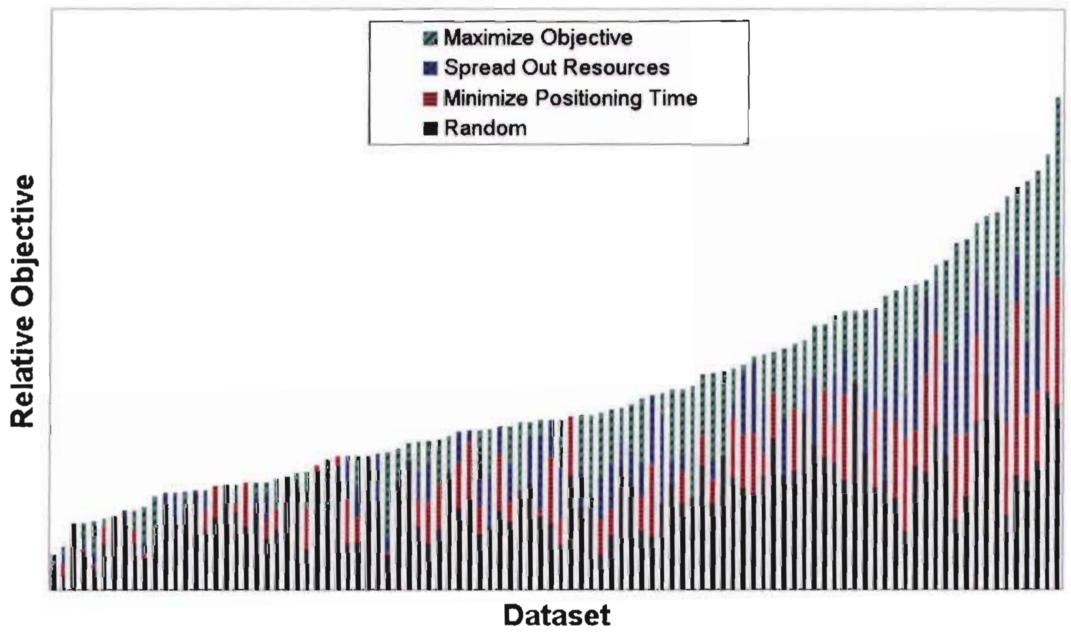

Figure 7.6: Performance comparison of *Random*, *MaximizeObjective*, *MinimizePositioning-Time* and *SpreadOutResources* for 100 iterations

The results show that *MaximizeObjective* outperforms all other heuristics for most datasets and performing paired sample t-test verifies that these performance differences are significant. Although, *SpreadOutResources* leads to better overall solution quality than *MinimizePositioningTime*, differences are minimal and insignificant. As expected, all heuristics outperform *Random* in terms of objective value. *MaximizeObjective* and *SpreadOutResources* do not lead to as much schedule disruption as *MinimizePositioningTime*. Similarly, as for solution quality, the *Random* heuristic performs the worst in terms of disruption. We conclude that *MaximizeObjective* outperforms all other scheduling heuristics since it provides superior solution quality with relatively little schedule disruption.

Scheduling heuristic	Run-time	Objective	Disruption	Best
Random	2.64 s	2452.47	145.33	4
Maximize Objective	7.84 s	2543.14	123.37	48
Minimize Positioning Time	11.57 s	2483.47	130.79	25
Spread Out Resources	18.13 s	2502.90	123.27	26

Table 7.5: Average performance of *Random*, *MaximizeObjective*, *MinimizePositioningTime* and *SpreadOutResources* for 100 iterations

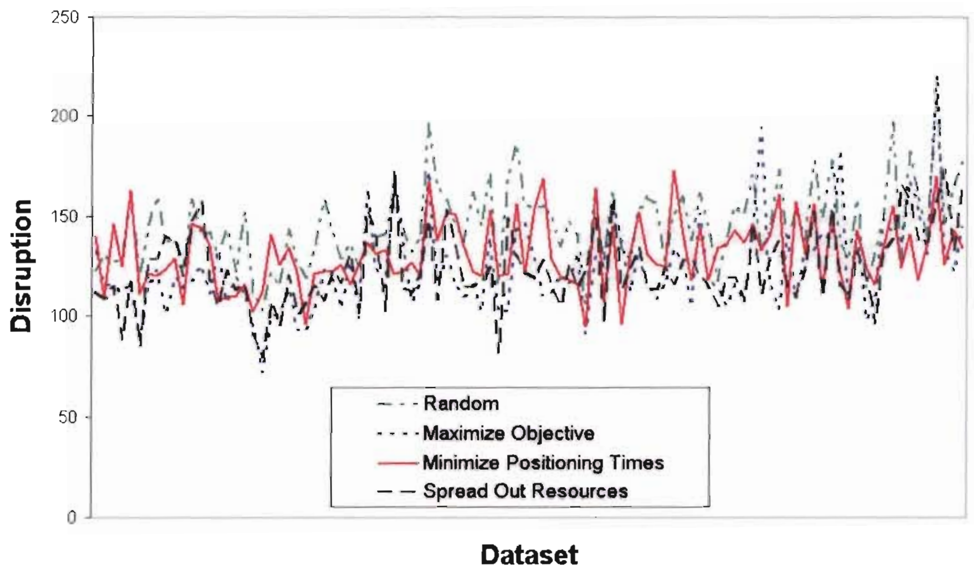

Figure 7.7: Schedule disruption of *Random*, *MaximizeObjective*, *MinimizePositioningTime* and *SpreadOutResources* for 100 iterations

Scheduling heuristic	Run-time	Objective	Disruption	Best
Random	2.64 s	2452.47	145.33	3
Maximize Objective	7.84 s	2543.14	123.37	35
Minimize Positioning Time	11.57 s	2483.47	130.79	19
Spread Out Resources	18.13 s	2502.90	123.27	17
MaxObj + Minimize Response Time	11.91 s	2555.71	123.51	46

Table 7.6: Average performance of *Random*, *MaximizeObjective*, *MinimizePositioningTime*, *SpreadOutResources* and *MaxObj+MinimizeResponseTime* for 100 iterations

As a result of the superior performance of *MaximizeObjective* we extend our experiment to include one more heuristic: we combine *MaximizeObjective* and a modified version of *SpreadOutResources* into one heuristic that we call *MaxObj+MinimizeResponseTime*. *MinimizeResponseTime* differs from *SpreadOutResources* in that it computes the minimum time it takes a resource to get to randomly generated points, rather than maximizing distances between all resources. Since resources operate at different speeds, maximizing distances between them is not enough to guarantee short response times. Additionally, the new heuristic computes this response time for each task type, since not all tasks can be executed by every resource. *MaxObj+MinimizeResponseTime* uses the *MaximizeObjective* heuristic and breaks ties using the new *MinimizeResponseTime* strategy. The performance summary of all five scheduling heuristics are shown in table 7.6. Although *MaxObj+MinimizeResponseTime* results in the best overall performance in terms of solution quality, it turns out that the difference to *MaximizeObjective* is insignificant. Table 7.4 shows the results of paired sample t-test between the different scheduling heuristics.

Heuristic 1	Heuristic 2	Result	Significant?
MaxObj+MinimizeResponseTime	MaximizeObjective	0.353	NO
MaxObj+MinimizeResponseTime	SpreadOutResources	0.005	YES
MaxObj+MinimizeResponseTime	MinimizePositioningTime	0.001	YES
MaxObj+MinimizeResponseTime	Random	0.000	YES
MaximizeObjective	SpreadOutResources	0.023	YES
MaximizeObjective	MinimizePositioningTime	0.004	YES
MaximizeObjective	Random	0.000	YES
SpreadOutResources	MinimizePositioningTime	0.294	NO
SpreadOutResources	Random	0.001	YES
MinimizePositioningTime	Random	0.054	NO

Table 7.7: Paired sample t-test for *Random*, *MaximizeObjective*, *MinimizePositioningTime*, *SpreadOutResources* and *MaxObj+MinimizeResponseTime* for 100 iterations

Chapter 8

Conclusion

In this thesis, we described a framework for dynamic resource scheduling problems with unit resources subject to temporal and resource constraints. It is composed of three components: a random problem generator, a dynamic simulator and a scheduler. We proposed a model for dynamic resource scheduling problems and incorporated it into our framework. We performed a case-study on the CoastWatch problem whose goal is to schedule both routine and emergency missions within a Search & Rescue operational command. We tested different heuristic scheduling strategies and various neighbourhoods on our Tabu search platform.

In Section 8.1, we first summarize the approach taken by our work. Then we review the key contributions to the field of dynamic resource scheduling in Section 8.2. Section 8.3 speculates on possible future research directions and Section 8.4 gives some concluding remarks.

8.1 Thesis Summary

Chapter 2 presented a survey of problem generators and scheduling algorithms for resource scheduling problems. Unfortunately, past research on problem generators has concentrated almost exclusively on static scheduling problems. Recently, Policella & Rasconi [23] developed a problem generator model for dynamic project scheduling problems. However, their dynamic events were very restrictive and couldn't be used to create realistic datasets for our dynamic resource scheduling problem.

Chapter 3 defined a general dynamic resource scheduling model which we incorporated into our framework. We differentiate between three different types of dynamic events: regular events, task events and mission events. Regular events have an absolute event time anywhere within the scheduling horizon. The event time for mission events is relative to the creation of the mission. For task events, the event time is also relative, but unlike mission events it represents a percentage. The dynamic event is created after the assigned resource has completed the specified percentage of the parent task.

We described our dynamic resource scheduling framework in Chapter 4. We explained all its components in detail: the random problem generator, the dynamic simulator and the scheduler. During development of the problem generator, we tried to keep it as general as possible so that it could be applied to similar scheduling problems with very little effort. The dynamic simulator hides future events from the scheduling algorithm and contains a visualization tool which creates animations on Google Earth. We implemented a Tabu search platform as part of our scheduler component and used it to carry out several experiments.

Chapter 5 describes the CoastWatch Dynamic Resource Scheduling problem. It is an oversubscribed dynamic multi-mode scheduling problem with unit resources and lies in the Search & Rescue domain. CoastWatch datasets simulate a typical day for the Canadian Coast Guard, where officers assign resources (planes, helicopters, ships, ...) to execute several different kinds of missions (patrol, transport, search & rescue).

We described our Tabu search algorithm in Chapter 6. It was used to run various algorithms on CoastWatch datasets. We experimented with different scheduling strategies and tested different neighbourhoods.

In Chapter 7 we stated our research goals and discussed experimental results. The objective function for our evaluation is to maximize the sum of priorities of all accomplished missions. Ignoring algorithm efficiency, we expected the Full neighbourhood to achieve the best results, as it provides the greatest flexibility in adjusting a candidate solution. However, our results showed that the simplest neighbourhood with the most restricted moves, *AddOnly*, resulted in superior performance. The advantage of this simple neighbourhood is that it concentrates on scheduling tasks that are currently unscheduled and leaves all other tasks alone. In another experiment, we used this neighbourhood to test various scheduling heuristics. The *MaximizeObjective* strategy, which positioned a given task in the permutation such that the assigned resource contributes to the objective value as much as possible, outperformed the other techniques. Combining this heuristic with another strategy to break ties, resulted in insignificantly superior performance.

8.2 Contributions

The main contribution of this thesis is a dynamic resource scheduling framework that can be applied to many different kinds of dynamic resource scheduling problems. We defined a model for such problems which allows a large set of unexpected dynamic events.
Contributions in this thesis include:

- We developed a dynamic resource scheduling framework which is composed of three components: a random problem generator, a dynamic simulator and a scheduler. It can be applied to many different kinds of dynamic resource scheduling problems.
- We developed a random problem generator which generates benchmark datasets for dynamic resource scheduling problems. It is very easy to adapt to add new mission types and dynamic events, because it only requires minor changes in its input files. The use of parameters provides great flexibility in changing the characteristics of the generated instances.
- We developed a dynamic simulator which is used to run dynamic scheduling datasets. It hides future events from the scheduler and contains a visualization tool to create an animation of the executing schedule on Google Earth.
- We implemented Tabu search as a uniform platform to test various scheduling heuristics on the CoastWatch problem. Additionally, we experimented with different search neighbourhoods.

8.3 Future Research

Although we have tried several different variations of a Tabu search algorithm, in the future, we could test more algorithms to determine how good our results really are. Since we have implemented the algorithm using a permutation-based method, it would be very interesting to try a scheduling algorithm which modifies schedules directly. We might be able to adapt the *MissionSwap* algorithm so that it doesn't require that all retracted tasks be rescheduled.

Similarly, finding other dynamic heuristics or neighbourhoods might improve our experimental results. We could combine several heuristics into one algorithm and run them concurrently by selecting one heuristic at random during each iteration. Another alternative would be to run two heuristics iteratively: The second heuristic restarts Tabu search from the best schedule found during the run using the first heuristic.

We used the random problem generator to create problem instances for the CoastWatch Dynamic Resource Scheduling problem. We made these datasets publicly available in order to spark more interest in studying dynamic scheduling problems. Possible future work could include generating more datasets for the CoastWatch problem. It would be interesting to see if changing some of these parameters changes the outcome of our experiments significantly. Additionally, it would be beneficial to identify a subset of parameters that significantly influences the difficulty of the generated datasets. We can achieve this by trying various combinations of these parameters.

Precedence constraints are a very common type of temporal constraints and are included in many different scheduling problems. In Chapter 4 we explained how difficult it is to obey precedence constraints in dynamic problem instances. Although our solution of enforcing such a constraint by means of generating an appropriate release date for the task, works for our problem, it may not be sufficient for other problems. As part of our future research, we could find a better solution in order to introduce more complicated precedence constraints to the dynamic resource scheduling framework.

The size of the execution time windows for tasks can have a significant impact on the difficulty of the resulting problem instances. Consequently, we need to test several strategies for determining their size and analyze the resulting datasets. Currently, we generate the execution time window for a task by considering the positioning times and durations of the capable resources. However, there is a disadvantage to this approach: resources that are either much faster or much slower than other ones, influence the resulting size significantly. In the future, we could look for alternative ways such that time window sizes are not dependent on the available resources.

8.4 Concluding Remarks

As more and more researchers are working on dynamic scheduling problems, the need for good problem generators will only increase over time. We have taken one step towards this direction: developing a random problem generator that is flexible enough to be used for many different kinds of dynamic resource scheduling problems.

We have achieved the two goals that we set out before starting our research. We developed a general random problem generator as part of a larger framework for dynamic resource scheduling problems and we were able to get decent results by running several variations of a Tabu search algorithm. In addition to that, we hope to have achieved two additional goals:

1. To spark interest in other researchers to try their dynamic scheduling algorithms on our benchmark datasets for the CoastWatch scheduling problem.
2. To spark interest in the scheduling community to attack even more dynamic scheduling problems in the future.

Appendix A

Problem Generator Input Files

A.1 Sample Scheduling Problem file:

```
resourceTypes (aurora cormorant cyclone frigate eagle_uav)
capability search (aurora cormorant cyclone eagle_uav frigate)
capability interdiction (aurora cormorant cyclone eagle_uav frigate)
capability rescue (cormorant cyclone frigate)
capability patrol (aurora cormorant cyclone frigate)
capability transport-static (aurora cormorant cyclone)
capability transport-dynamic (aurora cormorant cyclone)
base CFB_Comox (49.72052,-124.89249)
base CYBL_CampbellRiver (49.95054,-125.27070)
base YVR_Vancouver (49.19388,-123.18444)
base YAZ_Tofino (49.13106,-125.89075)
resource aurora CP-140411 YVR_Vancouver 750
resource aurora CP-140412 YVR_Vancouver 750
resource cormorant CH-149901 CFB_Comox 278
resource cormorant CH-149902 CYBL_CampbellRiver 278
resource cormorant CH-149903 YVR_Vancouver 278
resource cormorant CH-149904 YAZ_Tofino 278
resource cyclone CH-148001 CFB_Comox 305
resource cyclone CH-148002 CYBL_CampbellRiver 305
resource cyclone CH-148003 YVR_Vancouver 305
resource cyclone CH-148004 YAZ_Tofino 305
resource eagle_uav CE-147001 CFB_Comox 207
resource eagle_uav CE-147002 CYBL_CampbellRiver 207
resource eagle_uav CE-147003 YVR_Vancouver 207
resource eagle_uav CE-147004 YAZ_Tofino 207
resource frigate CF-141001 YAZ_Tofino 54
resource frigate CF-141002 YAZ_Tofino 54
resource frigate CF-141003 YAZ_Tofino 54
resource frigate CF-141004 YAZ_Tofino 54
```

A.2 Sample Mission and Event file:

```
horizon       0   1440
numBases      4
numResources  10
events
mission_task_delay    probability=0.1  delay=random(-10,60)
task_change_duration  probability=0.2  relativeTime=random(1,99)
                      delay=random(-10,25)
disable_resource      numResources=2   time=random(30,120)
tasks
transport-static      numStatic=10          numDynamic=0
                      priority=random(1,10) relativeTime=0
                      releaseDate=normal(800,100)
transport-dynamic     numStatic=0           numDynamic=10
                      priority=random(1,10) relativeTime=0
                      releaseDate=random(0,120)
patrol                numStatic=30           numDynamic=0
                      priority=random(20,30) relativeTime=0
                      numRounds=random(3,4)  releaseDate=random(0,1440)
                      numPoints=random(5,10) maxDistance=random(120,150)
search                numStatic=0            numDynamic=10
                      radius=random(50,75)   relativeTime=0
```

```
                    releaseDate=0         numInterdiction=random(1,5)
                    priority=random(150,250)
rescue              relativeTime=100      releaseDate=0
                    duration=random(3,15)
                    mission_task_delay_probability=0
interdiction        releaseDate=0         duration=random(1,5)
                    relativeTime=random(1,100)
                    mission_task_delay_probability=0
                    task_change_duration_probability=0
```

Appendix B

Experimental Results

Set	Rand	MaxObjective	MinPosTime	SpreadOut	MaxObj+MinResponse
1	2907	2980	2523	2955	2980
2	2766	2619	2764	2764	2619
3	2440	2393	2367	2521	2393
4	2677	2692	2660	2674	2692
5	2574	2405	2577	2491	2405
6	2482	2458	2277	2401	2458
7	2605	2625	2545	2560	2625
8	2488	2527	2274	2467	2527
9	2637	2473	2349	2281	2473
10	2514	2349	2494	2455	2349
11	2797	2784	2771	2791	2766
12	2817	2789	2740	2796	2787
13	2711	2723	2719	2488	2739
14	2520	2259	2140	2385	2522
15	2707	2493	2675	2538	2707
16	2751	2811	2742	2754	2826
17	2614	2644	2171	2528	2619
18	2287	2366	2283	2139	1942
19	2703	2731	2464	2109	2590
20	2750	2750	2741	2542	2801

Set	Rand	MaxObjective	MinPosTime	SpreadOut	MaxObj+MinResponse
21	2638	2624	2612	2560	2624
22	2605	2624	2421	2551	2624
23	2653	2701	2617	2636	2701
24	2820	2781	2610	2595	2781
25	2659	2687	2694	2641	2687
26	2694	2680	2667	2680	2680
27	2636	2370	2377	2395	2370
28	2826	2825	2362	2430	2825
29	2864	2625	2857	2851	2625
30	2615	2667	2575	2536	2667
31	2978	2946	2931	2926	2952
32	2522	2336	2542	2412	2509
33	2481	2168	2140	2291	2461
34	2729	2581	2561	2748	2765
35	2561	2573	2345	2522	2424
36	2837	2832	2703	2793	2683
37	2663	2400	2337	2385	2193
38	2539	2507	2300	2472	2514
39	2610	2564	2197	2642	2470
40	2291	2203	2497	2152	2342
41	2565	2337	2551	2469	2337
42	2583	2530	2557	2405	2530
43	2692	2537	2649	2588	2537
44	2592	2327	2535	2407	2327
45	2760	2757	2705	2800	2757
46	2706	2318	2505	2272	2318
47	2558	2428	2381	2531	2428
48	2596	2523	2540	2546	2523
49	2443	2537	2402	2491	2537
50	2743	2677	2415	2700	2677

Set	Rand	MaxObjective	MinPosTime	SpreadOut	MaxObj+MinResponse
51	2776	2744	2690	2762	2744
52	2796	2388	2683	2746	2388
53	2872	2821	2841	2573	2821
54	2392	2250	2074	1871	2250
55	2694	2236	2683	2454	2236
56	2509	2525	2127	2488	2560
57	2816	2743	2794	2793	2852
58	2599	2370	2375	2227	2612
59	2439	2037	2244	2217	2233
60	2554	2516	2516	2291	2544
61	2629	2763	2704	2708	2686
62	2536	2221	2160	2321	2589
63	2617	2695	2615	2622	2684
64	2572	2456	2577	2600	2654
65	2629	2461	2590	2481	2288
66	2634	2433	2657	2603	2500
67	2716	2372	2321	2525	2211
68	2768	2571	2540	2716	2730
69	2614	2048	1991	1966	2068
70	2747	2228	2272	2431	2285
71	2730	2708	2679	2390	2719
72	2294	2418	2162	2185	2428
73	2577	2540	2266	2154	2563
74	2578	2503	1934	2551	2447
75	2392	2251	2400	2431	2469
76	2493	2377	2328	2341	2363
77	2753	2659	2567	2490	2388
78	2774	2789	2721	2795	2809
79	2814	2575	2139	2330	2793
80	2671	2456	2709	2699	2525

Set	Rand	MaxObjective	MinPosTime	SpreadOut	MaxObj+MinResponse
81	2679	2288	2627	2349	2720
82	2491	2547	2323	2355	2364
83	2664	2384	2523	2231	2642
84	2801	2590	2562	2778	2641
85	2784	2815	2510	2785	2809
86	2780	2800	2408	2602	2800
87	2626	2903	2755	2363	2903
88	2459	2492	2358	2340	2492
89	2788	2588	2436	2781	2588
90	2613	2636	2445	2637	2636
91	2419	2222	2294	2173	2306
92	2442	2216	2207	2279	2421
93	2623	2472	2424	2096	2478
94	2820	2772	2720	2758	2652
95	2715	2723	2342	2385	2493
96	2615	2654	2577	2635	2664
97	2558	2582	1868	2175	2369
98	2590	2331	2168	2307	2328
99	2672	2672	2636	2495	2205
100	2965	2937	2944	2939	2933

Bibliography

[1] Laura Barbulescu, Jean-Paul Watson, L. Darrell Whitley, and Adele E. Howe. Scheduling spaceground communications for the air force satellite control network. *J. of Scheduling*, 7(1):7–34, 2004.

[2] Marcel Becker and Stephen Smith. Mixed-initiative resource management: The amc barrel allocator. In *Proceedings 5th International Conference on AI Planning and Scheduling*, April 2000.

[3] Peter Brucker, Andreas Drexl, Rolf Mohring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112:3–41 30 31 32 33 34, January 1999.

[4] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Knowledge Representation*, pages 61–95. MIT Press, London, 1991.

[5] Andreas Drexl, Ruediger Nissen, James H. Patterson, and Frank Salewski. Progen/pix - an instance generator for resource-constrained project scheduling problems with partially renewable resources and further extensions. *European Journal of Operational Research*, 125:59–72(14), 2000.

[6] Abdallah Elkhyari, Christelle Guéret, and Narendra Jussien. Solving dynamic resource constraint project scheduling problems using new constraint programming tools. In *PATAT*, pages 39–62, 2002.

[7] Jeffrey E. F. Friedl. *Mastering Regular Expressions.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[8] William S. Havens and Bistra N. Dilkina. A hybrid schema for systematic local search. In *Canadian Conference on AI*, pages 248–260, 2004.

[9] William S. Havens and Wolfgang Haas. Coastwatch benchmark datasets. Technical report, Simon Fraser University, 2007.

[10] William S. Havens and Wolfgang Haas. Coastwatch scheduling simulator event model. Technical report, Simon Fraser University, 2007.

[11] Holger Hoos and Thomas Stuetzle. *Stochastic Local Search: Foundations & Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[12] Shaoxiong Hua and Gang Qu. A new quality of service metric for hard/soft real-time applications. In *ITCC '03: Proceedings of the International Conference on Information Technology: Computers and Communications*, page 347, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Laura E. Jackson and George N. Rouskas. Deterministic preemptive scheduling of real-time tasks. *Computer*, 35(5):72–79, 2002.

[14] Kwangho Jang. The capacity of the air force satellite control network. Master's thesis, Air Force Insititute of Technology, 1996.

[15] Rainer Kolisch, Arno Sprecher, and Andreas Drexl. Characterization and generation of a general class of resource-constrained project scheduling problems. *Manage. Sci.*, 41(10):1693–1703, 1995.

[16] Laurence Kramer, Laura Barbulescu, and Stephen Smith. Understanding performance tradeoffs in algorithms for solving oversubscribed scheduling. In *Proceedings 22nd Conference on Artificial Intelligence (AAAI-07)*, July 2007.

[17] Laurence Kramer and Stephen Smith. Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, August 2003.

[18] Laurence Kramer and Stephen Smith. Task swapping for schedule improvement: A broader analysis. In *Proceedings 14th International Conference on Automated Planning and Scheduling*, June 2004.

[19] Helena R. Lorenco, Olivier Martin, and Thomas Sttzle. Iterated local search. *ISORMS*, 57:321, 2002.

[20] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[21] John F. Muth and Gerald L. Thompson. *Industrial scheduling.* Prentice-Hall, 1963.

[22] Eugeniusz Nowicki and Czeslaw Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

[23] Nicola Policella and Riccardo Rasconi. Designing a testset generator for reactive scheduling. *Intelligenza Artificiale*, 3:29–36, 2005.

[24] Mark Roberts, L. Darrell Whitley, Adele E. Howe, and Laura Barbulescu. Random walks and neighborhood bias in oversubscribed scheduling. In *Multidisciplinary International Conference on Scheduling (MISTA-05)*, 2005.