WINDOW-BASED PROJECT SCHEDULING ALGORITHMS

by

TRISTAN SMITH

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2004

"Window-Based Project Scheduling Algorithms," a dissertation prepared by Tristan Smith in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Christopher Wilson

Dr. Christopher Wilson, Chair of the Examining Committee

June 1, 2004

Date

Committee in charge:

Dr. Christopher Wilson, Chair
Dr. Matthew Ginsberg
Dr. David Etherington
Dr. Andrzej Proskurowski
Dr. John Goodale

Accepted by:

Dean of the Graduate School

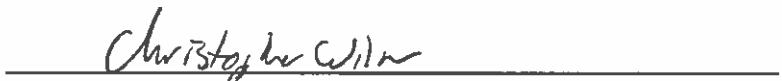
An Abstract of the Dissertation of

Tristan Smith for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken June 2004

Title: WINDOW-BASED PROJECT SCHEDULING ALGORITHMS

Approved: 

Dr. Christopher Wilson

The goal in project scheduling is to assign start times to activities so that all constraints are satisfied and some objective function is optimized. A wide range of real-world scheduling applications is matched by a variety of theoretical and experimental results in Artificial Intelligence and Operations Research.

In this dissertation, we suggest a window-based approach to project scheduling. Underlying this approach is the use of a generalized simple temporal problem (GSTP) framework that allows us to maintain for each activity an interval of temporally feasible start times, called a time window. We describe how windows can be maintained during both schedule construction and local search and discuss the properties of time windows for problems with both cyclic and acyclic constraint graphs.

We show that window-based search is effective for two very different problem domains. The first is the resource constrained project scheduling problem with arbitrary temporal constraints (RCPSP/max). We present a new heuristic algorithm that

combines the benefits of squeaky wheel optimization with an effective conflict resolution mechanism, called bulldozing, to address RCPSP/max problems. On a range of benchmark problems, the algorithm is competitive with state-of-the-art systematic and non-systematic methods and scales well.

The second problem for which we use window-based search is the labor cost optimization problem (LCOP) where the objective is to minimize the total labor costs (including wages, overtime, undertime, hire and fire costs). For the LCOP, simply computing the objective function is time-consuming; we show how this computation can be done quickly enough to be incorporated into search. We then describe the ARGOS optimization tool, a collection of algorithms for the LCOP, and show how it can produce significant cost savings over other available approaches on a number of real-world problems. Finally, we describe SimYard, a simulation tool for a shipyard environment, and use it to show that the theoretical cost savings of ARGOS schedules should be matched by actual savings in a real-world setting.

This dissertation includes my co-authored materials.

CURRICULUM VITA

NAME OF AUTHOR: Tristan Smith

PLACE OF BIRTH: Newton, MA, U.S.A.

DATE OF BIRTH: September 23, 1975

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Williams College

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 2004
University of Oregon

Bachelor of Arts in Mathematics, 1998, Williams College

AREAS OF SPECIAL INTEREST:

Project scheduling
Search
Optimization

PROFESSIONAL EXPERIENCE:

Software Engineer, On Time Systems, Inc., 2000 - present
Mathematics Teacher, Casablanca American School, 1998 -2000
Teaching Assistant, Williams College Mathematics Dept., 1997-1998
Researcher, SMALL Undergraduate Math Research Program, 1997

AWARDS AND HONORS:

Sigma Xi Associate Membership, 1998 - present
Phi Beta Kappa Society Membership, 1998 - present
Sam Goldberg Prize in Mathematics, Williams College, 1998
Witte Problem Solving Award, Williams College, 1998
Governor General's Award, 1994
Matheson Cup for Leadership, 1993

PUBLICATIONS:

Tristan B. Smith and John M. Pyle. An effective algorithm for project scheduling with arbitrary temporal constraints. In *Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI-2004) (to appear)*, San Jose, July 2004. AAAI Press.

Tristan B. Smith. *The Planarity of Subgraphs in Outer-facial Drawings.* Senior honors thesis, Williams College, 1998.

ACKNOWLEDGMENTS

I am grateful for the contributions that so many people have made to this project. I would particularly like to thank Matt Ginsberg. I wouldn't have considered tackling a Ph.D. without his encouragement and couldn't have completed it without his guidance and support. I have been very fortunate to have Matt as an employer, adviser, and friend.

Thanks to everyone on my committee. Andrzej Proskurowski and John Goodale brought different perspectives to my research area. Chris Wilson served on all three of my committees and agreed to be the Chair of my dissertation committee; I enjoyed spending time discussing my research with him. David Etherington was also on all three committees and his thoughtful and detailed suggestions helped me improve the final version considerably.

Thank you to all of the people, past and present, at CIRL and OTS who have contributed, each in their own way, to my work. Discussions, both in Friday meetings and in hallways, have been invaluable. Andrew Baker and John Pyle directly contributed to the work in this dissertation while Bryan Smith and Najam Ul-Haq each worked on the ARGOS project at one time or another. Bud Keith helped me figure out object oriented programming and get the ARGOS project started on the right foot. Richard Jones read an early draft and gave me suggestions. Matthew Austin, Laurie Buchanan and Pat Sullivan keep the whole place running. I owe a tremendous amount to Heidi Dixon and Andrew Parkes for their guidance and friendship throughout my graduate school experience.

My life over the past three years has had the biggest impact on, and has been most improved by, Kate. I thank her for everything.

To family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

We suggest a window-based approach to project scheduling. By maintaining a window of feasible start times for each activity, search techniques can directly and efficiently navigate and explore the space consisting of only time-feasible schedules.

For problems without resource constraints, this is exactly the space we wish to explore. For problems with resource constraints, resource conflicts can often be resolved by moving directly to other time-feasible schedules in this space.

Using a window-based approach, a variety of objective functions can be addressed and a variety of algorithms can be used. Constructive search is easy to implement and can be augmented with local repair. Local search can explore the space directly as the windows identify possible neighboring schedules that are achievable via small or large-scale moves in the space.

Windows can be maintained efficiently, even when real-world temporal constraints are involved. This allows for scalable algorithms that can be effective on large-scale real-world problems.

1.1 Overview

A project consists of a number of activities, each of which uses resources during its execution. The goal in project scheduling is to decide start times for the activities so that all constraints are satisfied and a given objective function is optimized. Because

there are so many real-world applications, scheduling problems have been well-studied by the Artificial Intelligence (AI) and Operations Research (OR) communities.

The scheduling algorithms in the literature can roughly be grouped into the following categories:

- Chronological scheduling approaches that step forward in time; at each time point, a subset of activities is scheduled.
- Order-based scheduling where activities are prioritized and scheduled one at a time in a greedy fashion.
- Disjunctive scheduling where activities are scheduled as early as possible and conflicts are resolved by the addition of temporal constraints that force conflicts to be avoided on subsequent iterations.
- Constraint-based scheduling where the focus is on search space reduction using constraint propagation.

The above approaches have been applied successfully on the problems usually addressed by the research community: problems of small size, straightforward constraints and regular objective functions. Unfortunately, real-world problems are often much larger, involve more complex constraints and have nonregular objective functions, and it is not clear how the above approaches will generalize to real-world settings.

In this dissertation, we suggest a window-based approach to scheduling. It relies on a generalized simple temporal problem (GSTP) framework that allows us to maintain an interval of temporally feasible start times for each activity, called a time window. Our GSTP framework differs from current approaches in the types of constraints that can be modeled. One important benefit is the ability to handle the calendar issues of commercial project management systems. We describe how to maintain windows during schedule construction, schedule deconstruction and local search and discuss the properties of time windows and how those properties differ between problems with cyclic and acyclic constraint graphs.

Resource-constrained project scheduling problems with arbitrary temporal constraints (RCPSP/max) are an important generalization of standard job-shop problems. Approaches for RCPSP/max problems with the goal of minimizing project

duration (minimizing makespan) have been developed by both the AI and OR communities. We describe a non-systematic hybrid algorithm called SWO(B,R) that, on a number of RCPSP/max benchmark suites, is competitive with state-of-the-art approaches and scales well. SWO(B,R) combines squeaky wheel optimization, an order-based algorithm, with an effective window-based conflict resolution mechanism, called bulldozing.

While makespan minimization is the most commonly studied scheduling objective, a more important objective in many real-world projects is the minimization of cost (of which makespan might be a component). When labor is an important resource, extra costs are incurred for overtime, undertime (workers paid when no work is available), hiring and firing. We define the labor cost optimization problem (LCOP), whose goal is the minimization of these excess costs. The LCOP subsumes a number of recently considered problems with resource-based objective functions.

For an LCOP instance, evaluating the objective function for a schedule requires finding the optimal staffing levels for that schedule; calculating these levels can be done with dynamic programming. We formally present the work of Andrew Baker who observed properties of this dynamic programming approach that allow for an efficient implementation of it. We then present the ARGOS algorithms that take advantage of this implementation to heuristically solve LCOP problems. The ARGOS algorithms are window-based and built upon a GSTP foundation. We show how ARGOS can produce less expensive schedules than existing techniques for a number of real-world problems.

Although ARGOS produces schedules that are significantly cheaper in theory, it is important to understand how the theoretical costs will be affected by reality. A real-world project differs significantly from its theoretical representation, both because the details will not all be represented correctly and because unpredictable disturbances invariably occur in the real world that require schedule modifications. To help understand these effects, we describe SimYard, a simulation tool that is an attempt to incorporate real-world issues into a shipyard model. SimYard takes schedules as input and outputs their expected costs. We use SimYard to show that, despite real-world complications, ARGOS schedules can be expected to reduce shipyard costs.

1.2 Outline

Chapter 2 provides background concerning project scheduling problems, objective functions, and complexity. A survey of scheduling algorithms is provided and the proposed window-based alternative is introduced.

Chapter 3 introduces the GSTP framework for maintaining time windows, describes the associated algorithms and shows how the GSTP framework can be used to model the temporal issues of standard commercial project management systems.

Chapter 4 describes the SWO(B,R) algorithm for RCPSP/max problems and compares experimental results with the best results reported in the literature. This chapter is based on material co-authored with John Pyle that is in press [90].

Chapter 5 formalizes the LCOP, describes Andrew Baker's dynamic programming algorithms for effectively computing labor costs and shows how this dynamic programming approach can be used within search algorithms.

Chapter 6 describes the ARGOS algorithms that take advantage of the dynamic programming techniques discussed in Chapter 5. Experimental results from applying ARGOS to a number of real-world problems are then presented in Chapter 7.

Chapter 8 outlines real-world issues that arise in scheduling, presents the SimYard simulation system, and discusses how real-world issues are handled in SimYard. Experimental results are given using SimYard to evaluate the schedules produced by ARGOS.

Finally, chapter 9 reviews the contributions of this dissertation and suggests possible extensions and future work.

CHAPTER 2

Background

2.1 Introduction

Project scheduling is the allocation of resources and times to activities in a plan. Scheduling problems arise in an enormous range of fields, including manufacturing, construction, distribution, software development, transportation and a number of space applications. Research has been done in a wide variety of settings including manufacturing, government, scheduling software companies and academia (in departments that include computer science, mathematics, business, management science, operations research and engineering).

The approaches taken to tackle scheduling problems in both Operations Research (OR) and Artificial Intelligence (AI) are as varied as the problems themselves. Exact or systematic approaches search for optimal schedules. While they achieve the highest solution quality, they rarely scale well and are limited to very small problems. Non-systematic or heuristic algorithms sacrifice optimality but attempt to find good solutions quickly. Finally, local search is a heuristic approach that improves solutions by iteratively considering nearby solutions in the search space and selecting ones that are better.

In Section 2.2, we define the scheduling problem and discuss problem classes, objective functions and complexity. In Section 2.3, we overview the history and current state of scheduling research by describing the main approaches used in the

literature.[1] Finally, Section 2.4 contains a brief description and discussion of the window-based approach we use in this dissertation.

2.2 Problem Definition

The input to a scheduling algorithm includes information about activities, resources and constraints. Given this input, the goal of scheduling is to produce a schedule that specifies when activities will be started so that all constraints are satisfied and some objective function is maximized or minimized.

For the purpose of this dissertation, we can break a scheduling problem into the following pieces:

Activities: $A = \{A_1, ..., A_n\}$ Also called jobs, operations or tasks, these are the pieces that must be done to complete the schedule.

Resources: $R = \{R_1, ..., R_m\}$ These are the resources required to complete activities. Execution of each activity A_i requires an amount r_{ik} of resource k for each time unit of execution.

We consider only renewable resources; when an activity finishes, the resources it used become available for other activities. Common examples of renewable resources include machines, facilities, equipment, space and people.

Constraints: These are rules or restrictions that limit the possible arrangements of the activities and can be divided into two types:

1. **Resource constraints** limit the capacity of resources. For example, there may only be a certain number of machines or people available to work on activities at any given time.
2. **Temporal constraints** restrict the times at which activities can be scheduled. A unary constraint restricts a single activity, usually with a release time (the earliest possible start time) or a deadline (the latest possible finish). A binary constraint (or precedence constraint) restricts the relative start times of a pair of activities. For example, we would probably require that an activity that paints a room be scheduled after an activity that installs the walls. A minimum time lag between A_i and A_j imposes a minimum amount of time that must pass between execution of A_i and

[1] In addition to the general background provided here, later chapters discuss details concerning background in particular areas relevant to those chapters.

A_j. A maximum time lag is the opposite; it imposes a maximum amount of time that can pass between A_i and A_j.

A schedule S is an assignment of a value to the start time $start_{A_i}$ of each activity, A_i. A schedule is **time-feasible** if it satisfies all temporal constraints and is **resource-feasible** if it satisfies all resource constraints. A schedule that satisfies all constraints is **feasible**.

2.2.1 Problem Classes

A number of problem types and problem classification schemes appear frequently in the literature. In the OR community, the classification scheme of Brucker [15] has been widely used. Others are described by Herroelen et al. [50] and Tzafestas and Triantafyllakis [95].

The problems that have received the lion's share of attention over the past 50 years are **job shop problems**. An $n * m$ problem contains n jobs (activities) and m machines (resources). Each job j consists of n_j operations $\{o_{j1}, ..., o_{jn_j}\}$. Precedence relations form a chain between all of the operations of each job. So, for job j, the precedence relations will be $o_{j1} \rightarrow o_{j2} \rightarrow ... \rightarrow o_{jn_j}$. Consecutive operations of a job are processed by different resources. Each resource R_i has a capacity $c_i = 1$ (a machine can perform a single operation at a time).

Job shop problems have a huge number of variants including general shop, flow shop, open shop and mixed shop problems.

The **resource constrained project scheduling problem (RCPSP)** is a proper generalization of job shop problems. Each R_i has some capacity c_i for the duration of the schedule (where we no longer require $c_i = 1$). In addition, each activity can require any quantity of any number of resources (unlike job shop problems where each activity requires a single unit of a single resource).

Finally, the **resource constrained project scheduling problem with arbitrary temporal constraints (RCPSP/max)** is a generalization of the RCPSP in which the temporal constraints include both minimum and maximum time delays between activities. The addition of maximum time delays makes a problem much more

difficult to handle and solve. In Chapter 3, we show that allowing maximum time delays is equivalent to allowing cycles in a problem's constraint graph and discuss the important differences between cyclic and acyclic constraint graphs.

2.2.2 Objective Functions

Recall that scheduling is an optimization problem. Therefore, there is some objective function $f(S)$ that serves to measure the quality of any schedule S. The goal is then to find a schedule that minimizes or maximizes $f(S)$.

Definition 1. *An objective function $f(S)$ is **regular** if for any feasible schedules S and S' in which each activity A_i has start times $start_{A_i} \leq start'_{A_i}$, $f(S) \leq f(S')$ (or $f(S) \geq f(S')$ if the objective is to maximize). That is, if all activities start at least as early in S as they do in S', S will be at least as good a schedule. A **nonregular** objective function is one that is not regular.*

The majority of scheduling research has focused on regular objectives. By far the most common of these is makespan minimization: shortening the schedule length or duration. One reason for this is that project makespans are simple to calculate and compare. Another is momentum in the scheduling community; if most work has focused on makespan, the only way for a new approach to be comparable is for it also to consider makespan.

This focus has been somewhat misguided as makespan minimization is an appropriate objective in only a small number of real-world scenarios. This problem has not gone unnoticed. In his 1974 book, K. R. Baker brought up "the disproportionate attention that researchers have paid to the makespan problem"[5]. Almost a quarter century later, Beck and others echoed that concern: "It is unclear whether our preoccupation with makespan has allowed us to make many in-roads into the realities of scheduling problems that have existed for decades in industrial settings"[11].

Here are a few other common regular objectives:

- Maximize throughput: a factory may want to maximize the amount it can produce in any given time period.

- Minimize tardiness: if deadlines must be missed, the number missed should be minimized. Weighted tardiness is an extension where some deadlines are more important than others.

- Minimize maximum lateness: in make-to-order production, no delivery date should be missed by too much.

While regular objectives have received most of the attention, some nonregular objectives are mentioned in the literature. Here are a few:

- Minimize work in progress (WIP): for a variety of reasons, factories may not want a lot of partially finished jobs lying around.

- Minimize waiting times: in a steel making plant, for example, it is extremely costly to keep molten steel heated between operations [80]. Similar issues arise in semiconductor manufacturing [34].

- Maximize net present value (NPV) [48, 60, 75]: if a project has significant cash flow during the project, in terms of expenses for beginning certain activities and income for completing others, it may be important to maximize the amount of cash on hand at various times during the project.

Notice that the above nonregular objectives are all based on the start times of activities. It is possible for objectives to be based instead on how resources are used. Recently, three types of resource-based nonregular objective functions have been tackled by the OR scheduling community (see Neumann et al. [74] for a good survey[2]). For these problems, the objective functions no longer depend on how individual activities are scheduled but on how they combine to use resources.[3]

The first is the **resource investment problem** (RIP) where the goal is to minimize the sum (possibly weighted) of the maximum work levels over all resources. The idea is that an investment in each resource has to be made that is proportional to the maximum capacity that will be required.

The second is the **resource leveling problem** (RLP) of which there are three varieties:

---

[2]It is interesting that even here, half of the book is devoted to makespan minimization.

[3]Specifically, the contributions of individual activities to $f(S)$ cannot be computed separately.

- Minimize the *total squared utilization cost.* This is the sum of the squares of the work profile values where the work profile of a resource is the amount used per time unit.
- Minimize the *total overload cost.* Here each resource R_k is given a threshold c_k above which a weighted cost is incurred per unit used.
- Minimize the *total adjustment cost.* Here costs are incurred for each increase w_k^+ and decrease w_k^- in the work profile.

The third is the **resource renting problem** (RRP) [78]. Here, each resource R_k is assigned a procurement cost $cost_k^p$ for each increase in capacity and a renting cost $cost_k^r$ for each time unit of capacity (imagine, for example, a backhoe that costs 100 dollars to procure plus a rental cost of 20 dollars per day). This problem subsumes the RIP since the RIP can be represented by setting $cost_k^r = 0$ for each resource k.

If labor is a significant resource, minimizing labor costs is a reasonable goal. Horowitz [51] discusses three variations of this goal, all of which are addressed by the above three problem classes. In fact, the goal of the RLP is often expressed as the indirect minimization of labor costs.

In this dissertation, we suggest a fourth resource-based objective whose goal is to minimize labor costs directly. Labor costs include standard wages and overtime costs as well as the costs to increase or decrease staffing levels. The **labor cost optimization problem** (LCOP) has as a goal the minimization of the sum of these costs and properly subsumes the above resource-based objectives. See Section 5.3 for a formal definition of the LCOP and Section 5.9 for a comparison of the LCOP with other resource-based objectives.

2.2.3 Complexity

There has been considerable research on the complexity of scheduling problems [13, 15, 41, 55, 74]. Garey and Johnson [41] list a number of scheduling problems known to be NP-complete.[4] Brucker's book [15] contains tables of known complexity

[4]The authors, of course, are referring to the decision version of a problem whereas we usually discuss the optimization version. The decision variant of a scheduling problem asks whether, given some value v, there is a schedule S for which $f(S) \leq v$ (or $f(S) \geq v$ for a maximization problem).

results for scheduling problems and includes references to those who showed them. For polynomially solvable problems, Brucker gives known upper bounds for the asymptotic complexity. The problems listed include many cases where small changes (a different optimization goal, for example) will push a problem class from polynomially solvable to NP-hard.

Only a tiny subset of scheduling problems are actually polynomially solvable.[5] Good overviews of such problems and polynomial algorithms are given by Tzafestas [95] and Karger [55]. These two papers also discuss pseudo-polynomial algorithms and approximation algorithms (algorithms that are guaranteed to arrive within a certain percentage of the optimal value) for some problems that are NP-hard.

Scheduling problems tend to be difficult, not just in theory, but in practice as well. Applegate and Cook note that the job shop problem is "not only NP-hard, it also has the well-earned reputation of being one of the most computationally stubborn combinatorial problems to date"[1]. In 1963, a book by Muth and Thompson [72] introduced a ten machine, ten job problem that took the OR community more than 20 years of hard work to solve.[6]

The two specific scheduling problems we consider in this dissertation are both NP-hard. RCPSP/max problems properly subsume a number of job shop problems that are known to be NP-hard [41] while LCOP problems subsume the other three problems with resource-based objective functions (the RIP, RLP and RRP) that are also known to be NP-hard [74].

Finally, it is worth noting that, for the RCPSP/max, the problem of simply finding a feasible solution is NP-hard [23].[7]

---

[5]For example, the job shop problem with at most 2 resources where each activity has duration 1 and the goal is makespan minimization.

[6]It has since become clear that this problem isn't even a particularly difficult one!

[7]This differs from many scheduling problems, for which feasible schedules are easy to find but optimal ones are not.

2.3 Classification of Scheduling Algorithms

Most search algorithms fit cleanly into a tree search framework.[8] Each node in the tree corresponds to a branch point; at each node a decision is made that determines which branch is selected. The leaves of the tree correspond to potential schedules.

In this section, we classify scheduling algorithms into four groups according to general approach. For each, we describe how it fits into the tree search framework. In addition, we highlight some specific algorithms that have been developed using that approach.

2.3.1 Chronological Scheduling

An iteration of a chronological scheduling algorithm takes the following form:

CHRONOLOGICAL-SCHEDULING
1 U = set of unscheduled activities
2 **while** $U \neq \emptyset$
3 **do** select the next relevant time t
4 schedule a subset of U at t and reduce U accordingly

Each t is a branch point and the decision at each branch point is the subset of activities to schedule at t. This is also referred to as a parallel schedule generation scheme[59, 74].

The most famous such algorithm is that of Giffler and Thompson [43] for job shop makespan minimization problems. They begin with all activities scheduled as early as possible. They then step through the relevant time points (given by the completion times of activities) and at each such t determine conflicts due to resource constraints. In their search tree, each branch decision corresponds to the choice of which activity involved in the conflict to leave at t; the rest are postponed. They discuss variations of their algorithm for other problems (besides the job shop) and different objective functions. Their algorithm has formed the basis for local search using genetic algorithms [27, 97, 98].

[8] Local search is an important exception.

Chronological scheduling continues to be used in many algorithms. In fact, Baptiste et al. claim that "Most search procedures for the RCPSP chronologically build a schedule" [8, page 152]. Obviously, chronological scheduling can easily be used in both systematic and non-systematic algorithms depending on how the search is structured.

2.3.2 Order-Based Scheduling

In an iteration of order-based scheduling (OBS), activities are greedily scheduled one at a time according to their priority.

ORDER-BASED-SCHEDULING
1 **for** $i = 1$ **to** n
2 **do** select unscheduled A_i with highest priority
3 schedule A_i greedily

There is a branch point after each activity is scheduled and the decision at each point is which activity to schedule next. OBS is also called priority-based scheduling, list scheduling and a serial schedule generation scheme [59, 74].

In some cases, priorities are determined dynamically; in this case the next activity to schedule depends on how other activities have been scheduled. Most often, however, a priority order (or queue) P is determined before scheduling begins. In this case, P completely determines the resulting schedule since activities are scheduled greedily. Therefore, P can serve as a stand-in for the schedule it will produce.

One of the oldest forms of OBS approaches is single-pass dispatch scheduling. Here a single iteration of the ORDER-BASED-SCHEDULING procedure is run to construct a schedule. Dispatch algorithms are easy to understand and implement and scale well. They are also the algorithms used in most commercial project management systems. The key to successful dispatch scheduling is the quality of the heuristics used to determine priorities. Many such dispatch rules have been tried [5, 53, 55]. Karger and others [55] consider a few dispatch rules that are optimal for very simple problems, and provably within a certain fraction of optimal for some others.

The problem with single-pass dispatch approaches, of course, is that they are not particularly effective at optimization. For most problems, there is no priority rule

that is guaranteed to outperform others and the schedule quality produced by any one heuristic is often quite problem dependent.

An obvious way to improve dispatching approaches is to run more than a single pass. If the heuristic incorporates some randomness, multiple iterations with the same heuristic will yield different schedules.

Multi-pass scheduling leads to other non-systematic or heuristic approaches that search for orderings that produce high quality schedules. An example is squeaky wheel optimization (SWO), where on each iteration, activities that are scheduled poorly are moved toward the front of the priority order under the assumption that they will be handled better when handled earlier [24, 54].

Most local search algorithms for scheduling use OBS techniques. In local search, operators are defined that modify an existing schedule to produce neighbor schedules in the search space. Because each priority order P can serve as a stand-in for a schedule, a natural approach is to define neighborhood operators that modify P. For example, in job shop scheduling, a common local search operator randomly exchanges the order of two operations that use the same machine.

There are a number of local search approaches that have been applied to scheduling problems. The simplest approach is to hill-climb in the search space by moving to neighbors with improved objective values until no such improvements can be made [53].

Simulated annealing is a variation of hill-climbing where neighbors with less good objective values are sometimes selected to keep the search from getting stuck in local optima [13, 85].[9]

In Tabu search [46], memory is used to both guide search toward potentially fruitful parts of the search space and to avoid previously visited areas [56, 74, 77, 100]. Blazewicz et al. [13] provide a good review of Tabu approaches in job shop scheduling and describe six different neighborhood operators, all of which are different ways to modify a priority order.

[9]In addition to swapping the order of activities, Sadeh et al. [85] also consider shifting activities left or right as a neighborhood operator. This can affect the objective value because they consider a nonregular objective function. Therefore, this approach is not purely an order-based one.

In genetic algorithms, populations of priority orders are evolved and mutated over time [27, 86, 92, 97, 98].

Finally, OBS approaches are used in many of the branch and bound algorithms of the OR community [74].

2.3.3 Disjunctive Scheduling

Each step in a disjunctive scheduling algorithm creates an early start schedule (ESS).[10] Before each step, temporal constraints are added to the problem to help eliminate resource conflicts and another ESS schedule is produced. The process continues until a feasible[11] schedule is produced. The constraints added are usually precedence constraints between activities.

An iteration of disjunctive scheduling takes the following form:

```
DISJUNCTIVE-SCHEDULING
1  construct ESS S
2  while S is not feasible
3        do add temporal constraints to the problem
4           reconstruct ESS S
```

The branch point for this approach is a current set of constraints and the decision at the branch point concerns which constraint(s) to add. This has been called a precedence constraint posting approach [21, 25, 26].

Disjunctive scheduling originates in the job shop scheduling environment where each machine can work on a single activity at a time. In this setting, for any pair of activities in conflict, a branch decision corresponds to choosing which one will go first. This approach is currently the most effective for systematic job shop algorithms [1, 18].

These ideas have been extended to other scheduling problems in a variety of ways. For RCPSP and RCPSP/max problems, a common approach is to analyze the set of activities that contribute to a resource over-allocation. A minimal critical set (MCS)

[10] Every activity is put at its earliest time-feasible start time.

[11] Even for problems where it is easy to find feasible schedules, this approach can be used to find *good* feasible schedules.

X is a subset of the conflicting activities for which the combined requirements of any $X' \subset X$ is less than the resource capacity. Therefore, given any MCS, a single precedence constraint between two of the activities in it will eliminate the resource contention among the activities in the MCS. Notice that for job shop problems, an MCS is simply a pair of activities and this approach is equivalent to basic disjunctive scheduling.

MCSs are introduced by Cesta et al. and used within non-systematic search [21, 23, 22]. Earlier work in the OR community developed similar ideas using the notion of forbidden sets; sets of activities that cannot all overlap [9, 70]. Neumann et al. [74] develop a number of branch and bound algorithms based on these ideas. Another similar approach is the use of schedule schemes [14, 16]. The Tabu algorithm of Baar [3] using schedule schemes is one of the few local search algorithms we have seen that fits into the disjunctive scheduling framework.

Finally, a variation on disjunctive scheduling is to add unary temporal constraints instead of binary ones. Fest et al. [35] impose temporary precedence constraints by delaying activities involved in resource conflicts (the constraints are temporary because the predecessor might be delayed in a subsequent iteration and the two may end up overlapping again).

2.3.4 CSPs and Constraint Propagation

It quickly became apparent that constraints play a key role in scheduling. In AI, a number of early approaches designed for real world problems were described as constraint directed [36, 37, 83, 89]. However, this early real-world focus soon became limiting. The burden of working with complex problems made implementation of, and experimentation with, new ideas difficult.

At the same time, the importance of constraints was becoming apparent in a wide range of other AI domains. Therefore, work in scheduling merged with work in other areas under the umbrella of the constraint satisfaction problem (CSP) framework. Within this framework, it became apparent that many general AI methods could compete with and sometimes outperform specialized approaches to specific problems.

A CSP is described by a set of variables, each with a finite domain, and a set of constraints that allow only certain combinations of values. A more formal definition of a CSP is [45]:

Definition 2. *A CSP* (V, D, C) *consists of a set* V *of variables, a set* D *of variable domains and a set* C *of constraints. Each variable* $v_i \in V$ *has a finite domain* $D_i \in D$ $(D_i = \{D_{i1}, ..., D_{ik}\})$ *of possible values. The set of constraints* C *consists of pairs* (J, P) *where* $J = (j_1, ..., j_k)$ *is an ordered subset of* V *and* P *is a subset of* $D_{j_1} \times ... \times D_{j_k}$.

A solution contains, for each $v_i \in V$, *a value* $d \in D_i$ *such that for every constraint* (J, P) *of the above form in* C, $(d_{j_1}, ..., d_{j_k}) \in P$.

The standard CSP approach for scheduling is to let variables represent the activity start times; the domain of each variable is then its set of possible start times.

While the CSP definition doesn't include an optimization function (it is designed for decision problems), it can easily be extended with one, resulting in the constraint optimization problem (COP).

A constraint graph can be associated with a CSP. In the constraint graph, each node represents a variable and the edges represent constraints between variables. Binary constraints are plain edges. For example, each precedence constraint would induce an edge in the graph. Unary resource constraints (as in job shop scheduling) induce an edge between each pair of activities that requires that resource (indicating that no pair can be scheduled at the same time).

More general scheduling problems require more complicated constraints. For example, in an RCPSP, a resource bound would be represented by a constraint restricting the sum of the resource usage over all activities using that resource. The resulting constraint graph is a hyper-graph with hyper-edges that join groups of nodes.

The CSP framework provides a way to cleanly separate a problem's representation from the algorithms and heuristics used to solve it [81]. This allows a system to have a flexible and extensible representation since any constraints are expressible and can easily be included. Systems that have a CSP foundation include the commercially available ILOG suite of tools [52] and the ODO framework that was designed as a

laboratory for studying and comparing scheduling algorithms [10].

Constraint Propagation

In general, a search algorithm for a CSP might consider every possible domain value for every variable. Constraint propagation techniques allow many such values to be ignored by recognizing implicit and explicit constraints in a problem. Constraint propagation can be viewed as either the reduction of variable domains or as the addition of extra constraints.

The goal of constraint propagation is to maintain a certain level of consistency among the variable domains. We describe the notion of consistency with respect to the constraint graph:

Definition 3. *A constraint graph is* ***node consistent*** *if and only if all values in each variable's domain are consistent with unary constraints on that variable.*

Definition 4. *A constraint graph is* ***arc consistent*** *if and only if it is node consistent*[12] *and, in addition, all values in each variable's domain are consistent with the domains of all variables with which it shares an edge.*

In scheduling, node consistency is achieved by limiting an activity's start times to those that satisfy unary constraints such as release times (earliest starts) or deadlines. Arc consistency is achieved by the propagation of precedence constraints between activities.

Example 2.3.1. *Consider a CSP with two variables and constraints,* $v_1 \neq 0$, $v_2 \neq 0$ *and* $v_1 \neq v_2$. *The domains* $D_1 = D_2 = \{0\}$ *are not node consistent. Domains* $D_1 = D_2 = \{1\}$ *are node consistent but not arc consistent. Domains* $D_1 = D_2 = \emptyset$ *are both node consistent and arc consistent; this shows that a constraint graph can be arc consistent but not solvable.*

Notice also that domains $D_1 = D_2 = \{1, 2\}$ *are arc consistent while domains* $D_1 = \{1, 2\}$ *and* $D_2 = \{1\}$ *are not (*$v_1 = 1$ *is not consistent with any values in* D_2*).*

[12] Some definitions of arc consistency do not require node consistency.

These concepts can be generalized using the following definitions:

Definition 5. *A constraint graph is* ***k-consistent*** *if and only if we can choose any value for $k-1$ variables so that all constraints among those variables are satisfied, and be guaranteed a value for any kth variable that satisfies all constraints among the k variables.*

Definition 6. *A graph is* ***strongly k-consistent*** *if and only if it is j-consistent for all $j \leq k$.*[13]

Note that node consistency is 1-consistency while arc consistency is strong 2-consistency. There is one further form of consistency commonly maintained in practice:

Definition 7. *A constraint graph is* ***path consistent*** *if and only if for any path $v_1, ..., v_m$ in the graph and any values for v_1 and v_m that satisfy the constraints between v_1 and v_m, there will be values for v_2 through v_{m-1} that satisfy the binary constraints between each pair (v_i, v_{i+1}) in the path.*

Many other varieties of consistency exist and a number of generic consistency maintenance algorithms have been developed [4, 32, 62, 94].

If variable domains are ordered (as they are for scheduling problems when the domain is a set of possible start times), an alternative to arc consistency is the following:

Definition 8. *A constraint graph is* ***arc-B consistent*** *if and only if the highest and lowest values in each variable's domain are pairwise-consistent with those values in the domains of all variables with which it shares an edge.*

Maintaining arc-B consistency can be done more efficiently since internal domain values can be ignored. In addition, this allows time to be continuous in problem representation even though the technical definition of a CSP requires discrete domains.

[13] It seems counterintuitive that a graph could be k-consistent but not $(k-1)$-consistent, but Tsang [94] cites the following simple example in which this occurs for $k = 3$: $V = \{v_1, v_2, v_3\}$, $D_1 = \{r\}$, $D_2 = \{r, b\}$, $D_3 = \{r\}$, and constraints require that $v_1 \neq v_2$ and $v_2 \neq v_3$.

In most scheduling approaches, a domain of start times is therefore maintained as an interval.

In the standard CSP approach, propagation achieves the desired level of consistency before search begins. In addition, lookahead strategies can be used [62]; these perform propagation at each node in the search (every time a variable is valued). This results in further domain reductions due to valued variables. In Chapter 3, we describe lookahead algorithms that maintain arc-B consistency for real-world problems.

Resource Constraint Propagation

In resource constrained problems, resource constraint propagation can significantly reduce the search space. Within the past decade, much work has been done to find algorithms that efficiently propagate resource constraints.

The simplest form of resource constraint propagation is timetabling, where intervals of time are tested to see if the minimum resource requirements over that interval eliminate possible start times from some activity domains [7, 8].

Another common form of resource constraint propagation is edge-finding, or energetic reasoning, where the ways that activities can be scheduled with respect to one another are reduced [1, 7, 8, 18, 33]. The amount of resources used over various intervals (the energy) is calculated in order to find activities that must be scheduled before or after other subsets of activities. Notice that this is similar to the disjunctive scheduling approaches mentioned above; the crucial difference is that here the constraints are added only when they are necessary rather than as a branch decision.

The above propagation techniques, although much better for some problems, suffer from high computation costs and limited applicability. Recently, Laborie [63] has attempted to deal with the latter disadvantage by introducing resource constraint propagation that depends on the relative positions of activities rather than absolute position; while his approaches appear promising, it is unclear how effective they will be in practice.

2.4 Window-Based Search

A central contribution of this dissertation is the suggestion that time windows can be used as a basis for search. For each activity, A_i, we maintain two time windows during search:

1. The *hard window*, $[hes_{A_i}, hls_{A_i}]$, contains the interval of time points t for which there exists a time-feasible schedule with A_i starting at t. The hard window will not change during schedule construction.
2. Given a partially constructed schedule, the *soft window*, $[ses_{A_i}, sls_{A_i}]$, contains time points t for which there exists a feasible schedule with each scheduled activity A_j starting at the current value of $start_{A_j}$ and A_i starting at t.

The hard window corresponds to the domain achieved by temporal arc-B consistency prior to search while the soft window corresponds to that domain during search. In Chapter 3, we discuss these windows in detail and show how and when they can be efficiently maintained.

We use the window-based approach to tackle two very different types of scheduling problems:

1. **Type I:** RCPSP/max problems with makespan minimization (a regular objective function) as the objective.
2. **Type II:** LCOP problems with cost minimization (a nonregular objective function) as the objective.

All of our algorithms schedule activities one at at time by selecting for each A_i a start time from A_i's soft window. Here is the pseudocode for a constructive window-based algorithm.

WINDOW-BASED-SCHEDULING
1 **for** $i = 1$ **to** n
2 **do** select unscheduled A_i
3 Choose a start time for A_i from $[ses_{A_i}, sls_{A_i}]$

Notice that this is similar to the order-based scheduling approach described earlier. However, in the order-based approach, the emphasis is wholly on the selection of the

next A_i. We shift the focus to the selection of an appropriate start time for A_i. For Type I problems, we enhance an order-based approach with a window-based conflict resolution mechanism called bulldozing. For Type II problems, we focus entirely on the choice of start time as experimental evidence suggests that priority order makes little difference.

In addition to the shift in focus, the use of both hard and soft windows during search is new. This allows mechanisms like bulldozing to select start times that are outside an activity's soft window but within its hard window. The resulting schedule can be made feasible by 'bulldozing' activities involved in temporal constraints that have been broken.

For Type II problems, we use window-based local search. From any schedule we can obtain neighbor schedules by changing the start time of a subset of activities.[14] While most local search algorithms search over priority orders, we show that we can effectively search schedule space directly.

A majority of scheduling algorithms do maintain time windows either implicitly or explicitly. However, the windows themselves are rarely viewed as the search space. There are a number of possible explanations for this:

1. In the job-shop environment, where most scheduling theory originates, the objective is to find an order of operations on each machine to optimize a given criteria. This focuses attention on the order of operations and hides the fact that windows exist at all.

2. From the point of view of systematic search, branching on possible start times for an activity may not work well. The branching factor can be large and is affected by factors such as time granularity that arguably should not affect the problem difficulty.

3. Most algorithms have been designed to optimize regular objective functions with an emphasis on makespan minimization. When minimizing makespan, it makes sense to simply choose the earliest possible start for an activity and there appears little reason to consider other possible start times. Recent attempts to optimize nonregular objective functions seem to be extensions of approaches that have been used for regular objectives.

[14] Usually we move a single activity. However, with bulldozing, this may force additional activities to move.

We believe that standard scheduling approaches will not be effective for nonregular objective functions or problems with unlimited or variable resource capacities for the following reasons:

1. When the objective function is nonregular, an algorithm cannot simply schedule activities at their earliest possible start times. Therefore, chronological scheduling is unlikely to be effective since it will be hard to tell at time point t which activities should be scheduled at t and which should be delayed (making this decision requires the consideration of other time points as well).

   Similarly, order-based approaches will no longer be as effective since there may be no way to achieve an optimal schedule by greedily scheduling the activities in *any* priority order. As a result, a priority order P can no longer be a stand-in for a schedule S. Therefore, most local search algorithms will not carry over to the case of nonregular objective functions.

2. If resource capacities are non-existent or highly variable over time, resource constraint propagation will be impossible. Disjunctive scheduling will be difficult to guide since it can no longer focus on the resolution of resource conflicts.

A few window-based approaches have been suggested in the literature. Some of the original CSP approaches are based on start time selection; recall that this is the most obvious way to solve a scheduling problem within the CSP framework. Nuitjen and Aarts use a CSP approach for 'multiple capacitated' job shop problems [79] but focus on constraint propagation and not on start time selection. Javier and Larrosa [64] define two CSP approaches for job shop problems, one window-based and the other disjunctive.

Some work has been done on texture-based heuristics, where expected resource contention over time is used to guide variable and value selection in CSP approaches [10, 12, 36, 83, 84]. Probabilities of resource contention can be used to select activities that are likely to be difficult to schedule and to select times for those activities that are most likely to make other activities easy to schedule.

In research produced by the OR group at Universitat Karlsruhe, there are examples of scheduling algorithms that use a window-based approach. For nonregular resource-based objective functions, Neumann and Zimmerman [75] consider activities one at a time and search for the start time that minimizes the decrease in quality of the overall objective function.

Similarly, the text by Neumann et al. [74] has "Time Windows" in the title and discusses many of the issues we address in this dissertation. They also describe a repair algorithm that is very similar to the bulldozing we describe. However, despite their focus on time windows, the algorithms they discuss are almost entirely order based.

In their effective algorithm for RCPSP/max problems, Ulrich et al. [33] describe a 'time-oriented' approach. However, the branch decision in their search is to either schedule an activity or delay it by a certain amount of time. Therefore, they are never really selecting a start time from a set of possibilities.

Finally, we are not aware of any approaches that do window-based local search.

CHAPTER 3

Maintaining Time Windows for Real-World Problems

Solving a scheduling problem involves the assignment of a start time to each activity so that constraints are satisfied while an optimality criterion is optimized. To facilitate start time assignments, we wish to maintain a domain of feasible start times, called a time window, for each activity.

In Section 3.1, we define the generalized simple temporal problem (GSTP) framework that can be used to maintain time windows so that schedule construction can proceed backtrack-free with respect to temporal constraints. We distinguish between the acyclic and cyclic version; propagation can be done more efficiently and the resulting windows have more desirable properties in the acyclic case.

In Section 3.2 we consider the temporal issues that arise in project management systems. Sections 3.3 through 3.6 show how the GSTP framework can be used to represent them. Section 3.7 takes a brief look at other work on temporal constraint management.

3.1 Solving the GSTP

To handle temporal constraints in many contexts, we define a *generalized simple temporal problem* (GSTP). An n-variable GSTP consists of a set $V = \{V_1, .., V_n\}$ of

variables and a set E of binary constraints $\{E_{V_i,V_j}\}$ between variables.[1]

Each V_i represents a time point and can have a continuous or discrete domain. Two fixed times, t_{begin} and t_{end}, are added as dummy variables to represent the earliest and latest relevant time points in the problem (we usually set $t_{begin} = 0$).

Each binary constraint, E_{V_i,V_j}, represents the minimum time lag between V_i and V_j. While time lags are assumed to be constant in the literature, we will see that calendar issues create situations where the lag depends on the value of V_i. For example, suppose V_i and V_j are meetings and V_j is to be scheduled at least one business day after V_i. If V_i is on Friday, there will be a minimum time lag of 3 days before V_j (because the weekend cannot be used). However, if V_i is on any other business day the lag is only 1.

To handle situations like this, we represent a binary constraint as a function $e_{V_i,V_j}(t)$ that computes the minimum time lag between V_i and V_j when V_i has value t. We would like the function $e_{V_i,V_j}(t)$ to have the following two properties:

$$\textit{If } V_i = t \textit{ then any value of } V_j \geq e_{V_i,V_j}(t) \textit{ satisfies the constraint, and} \tag{3.1}$$

$$\textit{If } V_j = e_{V_i,V_j}(t) \textit{ then any value of } V_i \leq t \textit{ satisfies the constraint.} \tag{3.2}$$

A maximum time lag between V_i and V_j can be represented by a minimum lag $e_{V_j,V_i}(t)$ in the other direction. Therefore, although maximum time lags are discussed in later chapters (they are the reason for the 'max' in RCPSP/max, for example), we do not need to handle them separately in our framework.

We can also represent unary constraints on V_i with binary ones that use our dummy variables t_{begin} and t_{end}:

- To represent $V_i \geq d$, use $e_{t_{begin},V_i}(t) = d$.
- To represent $V_i \leq d$, use either $e_{V_i,t_{end}}(t) = t_{end} - d$ or $e_{V_i,t_{begin}}(t) = -d$.

[1]Our definition of a GSTP generalizes the simple temporal problem (STP) of Dechter et al. [32]. They generalize the STP to a TCSP (temporal constraint satisfaction problem) but the two generalizations are orthogonal. Their general case includes disjunctive constraints that we do not allow here; it is known that allowing disjunctive constraints makes it NP-hard even to determine problem consistency [32]. On the other hand, our general case allows edge weights to be functions rather than constants, something not allowed in their framework.

To reason in the opposite direction (to find a maximum time for V_i given a time for V_j), we note that each constraint function defines an implicit function representing the minimum time lag in the other direction:[2]

Definition 9. $e^{-1}_{V_i,V_j}(t) = max\{t' \mid e_{V_i,V_j}(t') \leq t\}$

The following example shows that to satisfy both (3.1) and (3.2), we must require each $e_{V_i,V_j}(t)$ be a non-decreasing function.

Example 3.1.1. *Suppose we have a constraint function given by* $e_{V_i,V_j}(0) = 2$ *and* $e_{V_i,V_j}(1) = 1$. *According to* (3.2), *if* $V_j = 1$, *any* $V_i \leq 1$ *should satisfy the constraint. However,* $V_i = 0$ *does not.*

It should be clear that if we require constraint functions be non-decreasing, (3.1) and (3.2) will be consistent with each other. Therefore, we require:

$$t_m < t_n \Rightarrow e_{V_i,V_j}(t_m) \leq e_{V_i,V_j}(t_n) \tag{3.3}$$

Lemma 3.1.1 shows that the same property holds for $e^{-1}_{V_i,V_j}(t)$:

Lemma 3.1.1. $t_m < t_n \Rightarrow e^{-1}_{V_i,V_j}(t_m) \leq e^{-1}_{V_i,V_j}(t_n)$

Proof: Let $t'_m = e^{-1}_{V_i,V_j}(t_m)$ and $t'_n = e^{-1}_{V_i,V_j}(t_n)$. By Definition 9, $e_{V_i,V_j}(t'_m) \leq t_m$. Then, since $t_m < t_n$, we have $e_{V_i,V_j}(t'_m) < t_n$. Therefore t'_m satisfies the requirement of Definition 9 for t'_n so $t'_m \leq t'_n$ as required.

■

A GSTP is *feasible* if there is an assignment $\{V_1 = v_1, ..., V_n = v_n\}$ that satisfies all binary constraints. We say any such assignment is *feasible*. Similarly, a partial assignment is *feasible* if it can be extended to a feasible assignment. For any feasible partial assignment, each unvalued variable has a set of *feasible times* that can be assigned to the variable so that the resulting partial assignment is feasible. The maintenance of these sets of feasible times is the goal of this chapter.

Finally, associated with a GSTP is a *directed constraint graph* $G = (V, E)$ where nodes represent the variables and each binary constraint defines a directed edge from

[2]Although not technically an inverse function, we use the notation $e^{-1}_{V_i,V_j}(t)$ for clarity later.

V_i to V_j labeled with $e_{V_i,V_j}(t)$. We say the GSTP is *acyclic* if its constraint graph is acyclic.

3.1.1 Time Windows: The General Case

While solving a GSTP, we would like to efficiently maintain a set of feasible times for each variable so that search can proceed backtrack free. To do so, we define for each V_i an interval $[lb_{V_i}, ub_{V_i}]$, called a *time window*. We say a window is *empty* if $lb_{V_i} > ub_{V_i}$. If possible, we would like our windows to have these two properties:

- A window is *sound* if it contains no infeasible times (every t such that $lb_{V_i} \leq t \leq ub_{V_i}$ is feasible).

- A window is *complete* if all feasible times are included ($lb_{V_i} \leq t \leq ub_{V_i}$ for every feasible t).

In some cases, unfortunately, it is not possible to maintain a single interval that is both sound and complete because the set of feasible times is not contiguous. We will see such an example in Section 3.5.1. In such cases, we aim for the following weaker property:

- A window is *minimally complete* if it is the smallest possible complete window. That is, lb_{V_i} and ub_{V_i} are the lowest and highest feasible times for V_i. Notice that a sound and complete window is necessarily minimally complete.

The CREATE-WINDOWS procedure shows how we can use the constraint graph to find time windows for a GSTP given a feasible partial assignment. This is roughly adapted from the constraint propagation described by Meiri [69].

CREATE-WINDOWS(V, E)

```
1  for each V_i
2        do [lb_{V_i}, ub_{V_i}] = [t_begin, t_end]
3  Q = E
4  while Q ≠ ∅ and no empty window
5        do select and delete any edge (V_i, V_j) from Q
6           if REVISE-FORWARD(V_i, V_j)
7              then Q = Q ∪ {all edges (V_j, V_k) in E}
8  Q = E
9  while Q ≠ ∅ and no empty window
10       do select and delete any edge (V_i, V_j) from Q
11          if REVISE-BACKWARD(V_i, V_j)
12             then Q = Q ∪ {all edges (V_k, V_i) in E}
```

Lines 1 and 2 initialize each window to its largest possible domain $[t_{begin}, t_{end}]$. Lines 3 to 7 update the lower bounds of all windows while lines 8 to 12 update the upper bounds. CREATE-WINDOWS will stop if an empty window is created (in which case the problem is infeasible); otherwise it completes the temporal propagation.

The CREATE-WINDOWS procedure calls the two subroutines REVISE-FORWARD and REVISE-BACKWARD. The former is outlined below. It considers a constraint (V_i, V_j) and updates the window of V_j to be consistent with the current window of V_i. If V_i is valued, $[lb_{V_j}, ub_{V_j}]$ will be made consistent with that value. If not, $[lb_{V_j}, ub_{V_j}]$ will be made consistent with lb_{V_i}. REVISE-FORWARD returns TRUE if an update was made and FALSE otherwise. The procedure REVISE-BACKWARD (not shown here) is symmetric and reduces the upper bounds of a window.

REVISE-FORWARD$((V_i, V_j))$

```
1  if V_i is valued
2     then lower-limit = V_i
3     else lower-limit = lb_{V_i}
4  new-lower-bound = e_{V_i,V_j}(lower-limit)
5  if new-lower-bound > lb_{V_j}
6     then lb_{V_j} = new-lower-bound
7          return TRUE
8     else
9          return FALSE
```

If time is continuous, there is no guarantee that CREATE-WINDOWS will terminate; a counterexample is described below in Example 3.1.2. If time is discrete, it should be clear that CREATE-WINDOWS will terminate since edges are added to Q only when a window has been reduced. We bound the running time of CREATE-WINDOWS in Lemma 3.1.6.

Example 3.1.2. *Consider a GSTP with $V = \{V_1, V_2\}$ and two constraints:*

$$e_{V_1,V_2}(t) = e_{V_2,V_1}(t) = t + \frac{t_{end} - t}{2}$$

Even though this problem has a feasible solution ($V_1 = V_2 = t_{end}$), the forward propagation of CREATE-WINDOWS *will not terminate; the* REVISE-FORWARD *procedure will continually update lb_{V_1} and lb_{V_2} by smaller and smaller amounts but never reach $lb_{V_1} = lb_{V_2} = t_{end}$.*

For example, suppose that $t_{end} = 16$ and we begin with $lb_{V_1} = lb_{V_2} = 0$. Propagating the first constraint will update lb_{V_2} to 8. Propagating the second will yield $lb_{V_1} = 12$ and add the first constraint back into Q. Propagating this will yield $lb_{V_2} = 14$ and add the second constraint to Q. This loop will continue increasing lb_{V_1} and lb_{V_2} by smaller and smaller amounts but never end.

Example 3.1.3 describes how windows are created by CREATE-WINDOWS in a more straightforward case.

Example 3.1.3. *Consider a GSTP with $V = \{V_1, V_2, V_3\}$ and two constraints given by $e_{V_1,V_2}(t) = t + 4$ and $e_{V_1,V_3}(t) = t + 2$ (in other words, V_2 must be at least 4 time units after V_1 and V_3 must be at least 2 time units after V_1). In addition, suppose that $V_3 = 5$ (while V_1 and V_2 are unvalued). If $t_{begin} = 0$ and $t_{end} = 10$,* CREATE-WINDOWS *will initialize the three windows to*

$$[lb_{V_1}, ub_{V_1}] = [lb_{V_2}, ub_{V_2}] = [lb_{V_3}, ub_{V_3}] = [0, 10].$$

During forward propagation (using REVISE-FORWARD*) the two constraints will be used to reduce the lower bounds of V_2 and V_3 resulting in windows:*

$$[lb_{V_2}, ub_{V_2}] = [4, 10] \qquad [lb_{V_3}, ub_{V_3}] = [2, 10]$$

During backward propagation (using REVISE-BACKWARD*), the constraint between* V_1 *and* V_2 *will force* $lb_{V_1} \leq 6$. *In addition, since* $V_3 = 5$, *the constraint between* V_1 *and* V_3 *will force* $lb_{V_1} \leq 3$.[3] *Therefore, we will get window:*

$$[lb_{V_1}, ub_{V_1}] = [0, 3].$$

As noted above, we will see examples where, in the general case, windows cannot be both sound and complete. In these cases, we would rather include infeasible time points than exclude feasible ones because we wish to know that it is never worth trying times outside a window.

Therefore, we want minimally complete windows. Theorem 3.1.2 shows that CREATE-WINDOWS results in complete windows. Lemma 3.1.3 shows that the endpoints of those windows are themselves feasible. These two facts guarantee that the windows are minimally complete.

Theorem 3.1.2. *For a GSTP with a feasible partial assignment,* CREATE-WINDOWS *results in complete windows if it terminates.*

Proof: Suppose the resulting windows are not complete. Since windows are clearly complete on initialization, there is a feasible t removed from some window. Assume that t is the first such time removed during CREATE-WINDOWS and is removed from the window of V_j. There are four ways t could be removed (where the first two are symmetric with the last two).

1. t is removed by a valued predecessor $V_i = v_i$ because $t < e_{V_i,V_j}(v_i)$. This is a contradiction; t would not be feasible if it did not satisfy this constraint.

2. t is removed by an unvalued predecessor V_i because

$$t < e_{V_i,V_j}(lb_{V_i}). \tag{3.4}$$

   If t is feasible for V_j then there is a feasible solution with $V_j = t$ and $V_i = t'$. At the point t is removed, $lb_{V_i} \leq t'$ since t is the first feasible time removed. Then by (3.3),

$$e_{V_i,V_j}(lb_{V_i}) \leq e_{V_i,V_j}(t').$$

   By combining this with (3.4), $t < e_{V_i,V_j}(t')$ and we again have a contradiction because the constraint is not satisfied.

[3]Notice that if V_3 were unvalued, the constraint between V_1 and V_3 would only force $lb_{V_1} \leq 8$.

3. t is removed by a valued successor V_k because $t > e^{-1}_{V_j,V_k}(v_k)$. This is a contradiction; t would not be feasible if it did not satisfy this constraint.

4. t is removed by an unvalued successor V_k because

$$e^{-1}_{V_j,V_k}(ub_{V_k}) < t. \tag{3.5}$$

If t is feasible for V_j then there is a feasible solution with $V_j = t$ and $V_k = t'$. When t is removed, $t' \leq ub_{V_k}$ since t is the first feasible time removed. Then by Lemma 3.1.1,

$$e^{-1}_{V_j,V_k}(t') \leq e^{-1}_{V_j,V_k}(ub_{V_k}).$$

By combining this with (3.5), $e^{-1}_{V_j,V_k}(t') < t$ and we again have a contradiction because the constraint cannot be satisfied.

Since each case leads to a contradiction, there can be no such feasible t and CREATE-WINDOWS must produce complete windows.

■

Completeness by itself does not imply much; notice each initial window $[t_{begin}, t_{end}]$ is complete as well. We now show that the windows are as small as possible by showing that the endpoints of each window are feasible. This is shown directly: we show how to extend a feasible partial assignment to feasible full assignments that use the endpoints.

Lemma 3.1.3. *If* CREATE-WINDOWS *has been invoked for any GSTP with a feasible partial assignment, two extensions of that partial assignment to a full feasible assignment are:*

1. *Any unassigned variable V_i is assigned lb_{V_i}.*
2. *Any unassigned variable V_i is assigned ub_{V_i}.*

Proof: Consider the first assignment. We must show that each constraint is satisfied. For any constraint $E_{(V_i,V_j)}$ there are four possibilities:

1. Both V_i and V_j are valued in the partial assignment. Since the partial assignment is feasible, the constraint must be satisfied.
2. V_j is unvalued and $V_i = t$. In this case, REVISE-FORWARD will have ensured that $e_{V_i,V_j}(t) \leq lb_{V_j}$ and the constraint will be satisfied.

3. V_i is unvalued and $V_j = t$. In this case, REVISE-BACKWARD will have ensured that $e_{V_i,V_j}(ub_{V_i}) \leq t$. Since Theorem 3.1.2 implies that no window can be empty, $lb_{V_i} \leq ub_{V_i}$ and the constraint will be satisfied.

4. Both V_i and V_j are unvalued in the partial assignment. Again, we know that REVISE-FORWARD will have ensured that $e_{V_i,V_j}(lb_{V_i}) \leq lb_{V_j}$ and the constraint will be satisfied.

Since all constraints must be satisfied, the assignment is feasible.

■

Example 3.1.4. *Consider the problem of Example 3.1.3 where $V_3 = 5$. A feasible schedule can be attained by assigning values $V_1 = lb_{V_1} = 0$ and $V_2 = lb_{V_2} = 4$. Notice that these values leave all constraints satisfied.*

Corollary 3.1.4. *For a GSTP with a feasible partial assignment,* CREATE-WINDOWS *results in minimally complete windows.*

Proof: Theorem 3.1.2 shows that the windows are complete and Lemma 3.1.3 shows that the endpoints are feasible.[4]

■

A side benefit of the above Lemmas is that CREATE-WINDOWS can be used to determine problem feasibility. It can be used before any variables are valued to determine problem feasibility and can be used at any point during schedule construction to ensure that a partial assignment is feasible.

Lemma 3.1.5. *For a GSTP with a partial assignment,* CREATE-WINDOWS *will result in an empty window if and only if the partial assignment is infeasible.*

Proof: If any V_i has an empty window, Theorem 3.1.2 implies there can be no feasible time for V_i so the partial assignment cannot be feasible.

On the other hand, if no V_i has an empty window the arguments of Lemma 3.1.3 can be used to show that the partial assignment is feasible.

■

[4]Technically, we did not show that the windows of valued variables are also minimally complete. However, the arguments of Lemma 3.1.3 can also be used to show that we could move each valued V_i to lb_{V_i} or ub_{V_i} without changing the feasibility of the schedule.

The point of using windows is that an algorithm that is selecting values for variables need not consider infeasible values for variables. Unfortunately, infeasible times cannot be completely avoided in the general case because windows are not sound. However, if an infeasible time is chosen for some variable, the subsequent call to CREATE-WINDOWS will recognize the problem. This follows directly from Lemma 3.1.5.

Therefore, although backtracking may be required, it will only ever need to undo the most recently valued variable. If the domain size of V_i is d, we will choose at most $d-2$ infeasible times for V_i from its window before finding a feasible one.[5] We saw in Example 3.1.2 that CREATE-WINDOWS might not terminate when time is continuous; the following lemma bounds the runtime for the case where time is discrete.

Lemma 3.1.6. *The time complexity of* CREATE-WINDOWS *is* $O(v+ed)$ *where* v *is the number of variables (vertices in the constraint graph),* e *is the number of binary constraints (edges in the constraint graph), and* d *is the maximum domain size (in our formulation, we have* $d = t_{end} - t_{begin} + 1$*. Note that for continuous time, domain sizes are infinite and this result does not bound the running time of* CREATE-WINDOWS.[6]

Proof: Setting up the initial windows is $O(v)$. Each loop (forward and backward) steps through an initial set containing all edges of the constraint graph and processes the constraints. An edge from V_i to V_j will be added to the set of constraints to process if and only if V_i's window was decreased. Each window can be decreased at most d times. Therefore, each edge can be added at most d times giving overall time complexity $O(v+ed)$.

■

[5] By Lemma 3.1.3, we know that lb_{V_i} and ub_{V_i} are feasible; they can safely be chosen if we wish.

[6] Using arguments like those of Franck et al. [39], we can bound the running time for the continuous case if we know there are a finite number of 'jumps' in the constraint function.

3.1.2 Time Windows: The Acyclic Case

In the case where there are no cycles in the constraint graph, the CREATE-WINDOWS procedure has better properties; it is able to maintain sound and complete windows and has a lower worst-case running time.

To show these properties, we take advantage of the fact that, for an acyclic problem, we can topologically sort V with respect to the constraint graph.[7] This gives us a one-to-one mapping $T : V \rightarrow \{1, ..., n\}$ such that $T(V_i) < T(V_j)$ implies that there is no path from V_j to V_i in the constraint graph (there might or might not be a path from V_i to V_j).

CREATE-WINDOWS maintains minimally complete time windows due to Corollary 3.1.4; we now show that it also maintains sound time windows for acyclic problems.

Lemma 3.1.7. *For an acyclic GSTP with a feasible partial assignment,* CREATE-WINDOWS *produces sound windows.*

Proof: We must show that for any variable V_j, valuing $V_j = t$ where $lb_{V_j} \leq t \leq ub_{V_j}$ will result in a feasible partial assignment.[8] We show the partial assignment is feasible by demonstrating how the partial assignment can be extended to a feasible solution.

First, use a topological sort to separate the variables (except V_j) into two sets: $S_1 = \{V_i \mid T(V_i) < T(V_j)\}$ and $S_2 = \{V_i \mid T(V_j) < T(V_i)\}$. Then, schedule the activities as follows:

$$V_i = \begin{cases} v_i & V_i \text{ already valued to } v_i, \\ t & i = j, \\ lb_{V_i} & V_i \in S_1, \\ ub_{V_i} & V_i \in S_2 \end{cases} \tag{3.6}$$

[7] A constraint graph may have many topological sorts; any one of them will work.

[8] Notice that we do not require V_j to be unvalued. Windows are also maintained for valued variables V_j.

We put any variables before V_j (in the topological sort) as early as possible and any variables after V_j as late as possible. We must show this schedule is feasible. Consider the constraints on V_j. Any predecessor V_i of V_j is in S_1. This means that $V_i = lb_{V_i}$ and the REVISE-FORWARD piece of CREATE-WINDOWS guaranteed that $e_{V_i,V_j}(lb_{V_i}) \leq lb_{V_j}$. Since $lb_{V_j} \leq t$, this constraint will be satisfied. A similar argument shows that constraints between V_j and its successors will be satisfied.

Now consider constraints not involving V_j. If a constraint is between two variables in either S_1 or S_2, the arguments of Lemma 3.1.3 show the constraint is satisfied. Finally, if there is a constraint between $V_i \in S_1$ and $V_k \in S_2$, V_i must be the predecessor by the topological sort. But then $e_{V_i,V_k}(lb_{V_i}) \leq lb_{V_k} \leq ub_{V_k}$ so this constraint will be satisfied.

■

Corollary 3.1.8. *For an acyclic GSTP with a feasible partial assignment,* CREATE-WINDOWS *produces sound and complete windows.*

Proof: This follows directly from Corollary 3.1.4 and Lemma 3.1.7.

■

Corollary 3.1.9. *For an acyclic GSTP with a feasible partial assignment, if all windows are minimally complete, they are also sound.*

Proof: The arguments of Lemma 3.1.7 can be used to show that, given minimally complete windows, any variable V_i can be valued to any t satisfying $lb_{V_i} \leq t \leq ub_{V_i}$ such that the new partial assignment can be extended to a feasible solution. Therefore, all such t are feasible and V_i's window is sound.

■

As a result of Corollary 3.1.8, search can proceed backtrack free in an acyclic problem. We can also improve its running time by taking advantage of a topological sort as shown below in the procedure CREATE-WINDOWS-ACYCLIC.

In CREATE-WINDOWS-ACYCLIC, a topological sort is obtained on line 3 (for the rest of the procedure, we assume for convenience that $T(V_i) = i$). TOPOLOGICAL-SORT is a straight-forward algorithm and is described by Cormen et al. [30]. Lines 4

to 6 perform a forward-pass through the topological sort to update each lb_{V_i}. In lines 7 to 9, a similar backward pass is performed to update each ub_{V_i}.

CREATE-WINDOWS-ACYCLIC(V, E)
1 **for** each V_i
2 **do** $[lb_{V_i}, ub_{V_i}] = [t_{begin}, t_{end}]$
3 $(V_1, ..., V_n)$ =TOPOLOGICAL-SORT(V)
4 **for** $i = 1$ **to** n
5 **do for** each successor V_j of V_i
6 **do** REVISE-FORWARD(V_i, V_j)
7 **for** $i = n$ **to** 1
8 **do for** each predecessor V_j of V_i
9 **do** REVISE-BACKWARD(V_j, V_i)

Theorem 3.1.10. *For an acyclic GSTP with a feasible partial assignment,* CREATE-WINDOWS-ACYCLIC *produces sound and complete windows.*

Proof: We will show that CREATE-WINDOWS-ACYCLIC results in the same windows as CREATE-WINDOWS and Corollary 3.1.8 confirms that those windows are sound and complete.

The first two lines of each procedure are identical. Now consider the forward propagation: lines 3 to 7 in CREATE-WINDOWS and lines 4 to 6 in CREATE-WINDOWS-ACYCLIC.

In CREATE-WINDOWS, REVISE-FORWARD is called for every edge (V_i, V_j) and if the window of V_j is updated, all edges (V_j, V_k) are added to Q. In CREATE-WINDOWS-ACYCLIC, REVISE-FORWARD is still called for every edge; the difference is that edges are never reconsidered. However, when (V_i, V_j) is revised, since $T(V_i) < T(V_j)$ (by the topological sort), V_j will be reached later in the for loop. Therefore all edges (V_j, V_k) will be subsequently considered anyway. Symmetric arguments can be made for the backward propagation. Therefore, the resulting windows are the same and CREATE-WINDOWS-ACYCLIC does produce sound and complete windows.

■

Because the topological sort is used, the running time of CREATE-WINDOWS-ACYCLIC has a much better theoretical bound. Most importantly, perhaps, the running time remains bounded when time is continuous.

Lemma 3.1.11. *The time complexity of* CREATE-WINDOWS-ACYCLIC *is* $O(v+e)$ *where* v *is the number of variables and* e *is the number of binary constraints.*

Proof: Setting up the initial windows is $O(v)$. TOPOLOGICAL-SORT is $O(v+e)$ [30]. The forward and backward passes can both be completed in $O(v+e)$ since each activity is visited once and each binary constraint is propagated once. Therefore, the overall running time is $O(v+e)$.

■

Although it doesn't affect the theoretical running time, it is worth noting that any implementation of CREATE-WINDOWS-ACYCLIC need only create the topological sort once and reuse it in subsequent iterations. We will see other practical improvements later in this chapter when we consider an implementation of the GSTP approach that is specific for project scheduling.

3.2 Time In Project Management Systems

In the real world, most scheduling is done within project management systems (PMSs). Examples include Microsoft Project, Primavera Enterprise, Open Plan and Artemis 9000/EX. These systems allow managers, supervisors, schedulers and all of those involved in large projects to coordinate and keep track of completed and projected work during a project. While scheduling is only a small piece of the focus, all project management systems include some basic schedule optimization, usually in the form of an order-based greedy construction algorithm.

Any attempt to provide scheduling algorithms to companies working within these frameworks must account for problems as represented in them. Otherwise, the schedules produced may not satisfy all of the real world constraints. Unfortunately, much of scheduling theory has been developed within the comfortable confines of academia and does not account for some of the complications that arise in the real world.

In this section, we begin by outlining the properties of time as maintained in PMSs and show how the GSTP framework of Section 3.1 can be used to tackle PMS scheduling problems.

3.2.1 Time

In PMSs, time is discrete and has some atomic base unit. Depending on the PMS and the particular problem, the size of this base unit can range anywhere from a minute to a year.

3.2.2 Temporal Constraints

PMSs include both unary and binary temporal constraints. Unary constraints restrict the start time, $start_{A_i}$ or finish times, $finish_{A_i}$ of an activity A_i in four possible ways:[9]

1. An **early start** $\mathbf{es_{A_i}}$ represents the constraint, $start_{A_i} \geq es_{A_i}$ (often called a release time).
2. An **early finish** $\mathbf{ef_{A_i}}$ represents the constraint, $finish_{A_i} \geq ef_{A_i}$.
3. A **late start** $\mathbf{ls_{A_i}}$ represents the constraint, $start_{A_i} \leq ls_{A_i}$.
4. A **late finish** $\mathbf{lf_{A_i}}$ represents the constraint, $finish_{A_i} \leq lf_{A_i}$ (often called a deadline).

Binary constraints are precedence constraints between pairs of activities.[10] A constraint $(A_i, A_j, type_{A_i,A_j}, lag_{A_i,A_j})$ represents a minimum time lag, given by lag_{A_i,A_j}, between A_i and A_j. It has four basic forms based on $type_{A_i,A_j}$:

1. If $\mathbf{type_{A_i,A_j} = FS}$ **(Finish-Start)** the constraint is $start_{A_j} \geq finish_{A_i} + lag_{A_i,A_j}$ (A_j cannot *start* until a certain number of units after A_i *finishes*).
2. If $\mathbf{type_{A_i,A_j} = FF}$ **(Finish-Finish)** the constraint is $finish_{A_j} \geq finish_{A_i} + lag_{A_i,A_j}$ (A_j cannot *finish* until a certain number of units after A_i *finishes*).
3. If $\mathbf{type_{A_i,A_j} = SS}$ **(Start-Start)** the constraint is $start_{A_j} \geq start_{A_i} + lag_{A_i,A_j}$ (A_j cannot *start* until a certain number of units after A_i *starts*).
4. If $\mathbf{type_{A_i,A_j} = SF}$ **(Start-Finish)** the constraint is $finish_{A_j} \geq start_{A_i} + lag_{A_i,A_j}$ (A_j cannot *finish* until a certain number of units after A_i *starts*).

---

[9] Any subset of these 4 constraints can be specified for an activity.

[10] It is possible for a pair of activities to have more than one binary constraint between them. We see below why it may not be possible to choose a strongest one and ignore the others.

Example 3.2.1. *The University of Oregon Graduate School enforces the following constraints on doctoral students:*

$$(\textit{submit committee}, \textit{defend thesis}, FS, \textit{6 months})$$

$$(\textit{enroll in 18 credits of dissertation}, \textit{defend thesis}, SF, 0)$$

In scheduling literature, including the vast majority of work done on job shop problem variants, constraints are usually limited to $type_{A_i,A_j} \in \{FS, SS\}$ and $lag_{A_i,A_j} = 0$.[11] Among the exceptions is some work of Steve Smith and his colleagues [23, 25].

3.2.3 Calendars

An important feature of project management systems is that users may define and use multiple calendars. A calendar splits the overall time units into disjoint working and non-working subsets. For example, one calendar might represent an 8 hour shift each day, another might represent 10 hour shifts and a third might represent shifts that only work on weekends.

One way to represent a calendar is to use a bit mask where 1 represents a working unit and 0 represents a non-working unit.[12] Consider these examples that we will return to in later examples (here, the atomic time unit is a day):

Example 3.2.2. *1. An absolute calendar (CALBASE):*

1111111111111111...

2. A five day a week calendar (CAL5):

0011111001111100...

3. A five day a week calendar with a holiday on a Wednesday (CAL5H):

0011111001101100...

[11] This is one of the complaints of Beck et al. [10].

[12] If the granularity of time units is small, it may be more efficient to use a different data structure where only the changes between working and non-working units are stored.

Calendars can be assigned to both activities and constraints. The calendar assigned to an activity A_i limits $start_{A_i}$ and $finish_{A_i}$ to fall on working units in that calendar and requires that the duration, dur_{A_i}, of A_i is the number of working units in the interval $[start_{Ai}, finish_{A_i})$.[13] In other words, the duration of an activity with respect to CALBASE may be longer than dur_{A_i} if A_i's calendar has some non-working units. For example, an activity requiring 10 business days will actually span more than 10 days.

The calendar assigned to a constraint is only relevant when $lag_{A_i,A_j} \neq 0$. In this case, just as for activity duration, lag_{A_i,A_j} is the number of working units in the span between the two times specified by the constraint type, $type_{A_i,A_j}$.

Example 3.2.3. *If $lag_{A_i,A_j} = 2$, $type_{A_i,A_j} = FS$ and $finish_{A_i} = Friday$, A_j may start Sunday if the constraint calendar is CALBASE but cannot start until Tuesday if the constraint calendar is CAL5.*

3.3 Adapting the GSTP Framework to PMS Problems

In this section, we see how a scheduling problem as represented by PMSs can be fit into the GSTP framework. Section 3.3.1 outlines the framework. In Section 3.3.2, we define a number of functions that arise from the PMS situation. In Section 3.3.3, we combine those into edge functions that make up the binary constraints of the GSTP. Finally, Section 3.3.4 shows our implementation-specific representation of time windows.

Once our implementation of a GSTP framework has been presented, Sections 3.4 and 3.5 discuss scheduling-specific implementations of the GSTP algorithms.

---

[13]Notice that we are using $finish_{A_i}$ to denote the first time unit on which A_i is not working. This allows us to say $finish_{A_i} = start_{A_i} + dur_{A_i}$.

3.3.1 The Basic Setup

We represent an n-activity scheduling problem with an n-variable GSTP where the variable V_i represents the start time of activity A_i (From now on, we use A_i to refer to the variable) and the binary constraints represent the binary temporal constraints of the problem. Although we have seen that a unary constraint on A_i can be represented by a binary constraint with either t_{begin} or t_{end}, we choose to treat unary constraints separately for our implementation.

Because time is discrete in all PMSs, we use the discrete version of a GSTP[14] and use the units used by the PMS in question. All time points discussed will be with respect to an absolute calendar (CALBASE); therefore every time point t will be t units after the base time $t_{begin} = 0$.[15]

3.3.2 The Building Blocks for Edge Functions

Calendars complicate temporal reasoning. Any time span associated with a calendar has no fixed span with respect to the absolute calendar.

Example 3.3.1. *Consider a two-day activity, A_i, using calendar CAL5. If $start_{A_i}$ = Monday then $finish_{A_i}$ = Wednesday = $start_{A_i} + 2$. However, if $start_{A_i}$ = Friday then $finish_{A_i}$ = Tuesday = $start_{A_i} + 4$.*

To handle calendars, we represent all time spans as functions of the time point on which they begin. We define the following functions that form the building blocks of the edge functions used by our GSTP algorithms.

1. $\mathbf{dur_{A_i}(t)}$ The duration in absolute time units of A_i when $start_{A_i} = t$.

2. $\mathbf{dur^{-1}_{A_i}(t)}$ The duration in absolute time units of A_i when $finish_{A_i} = t$.

---

[14]Since most of our temporal reasoning concerns intervals, much of it generalizes to a continuous representation of time. However, for cyclic problems, we have seen that the complexity of the GSTP algorithms is unbounded in the continuous case.

[15]Most PMSs have something like a project start date that is an obvious candidate for the beginning of time.

3. $\mathbf{lag_{A_i,A_j}(t)}$ The duration in absolute time units of the lag when the relevant time point of A_i is t.[16]

4. $\mathbf{lag^{-1}_{A_i,A_j}(t)}$ The duration in absolute time units of the lag when the relevant time point of A_j is t.

The following examples show cases where $lag_{A_i,A_j}(t)$ can be positive or negative even when $lag_{A_i,A_j} = 0$:

Example 3.3.2. *For constraint, $(A_i, A_j, FS, 0, CALBASE)$, the lag is zero. However, if A_i uses CALBASE and A_j uses CAL5, $lag_{A_i,A_j}(t)$ can be nonzero. For example, $lag_{A_i,A_j}(Saturday) = 2$ (if A_i finishes Saturday, A_j cannot start for 2 days).*

Example 3.3.3. *Suppose in Example 3.3.2 that A_i uses CAL5 and A_j uses CALBASE. If $finish_{A_i} = Monday$,[17] then A_i was completed at the end of Friday and A_j may start Saturday. Therefore, $lag_{A_i,A_j}(Monday) = -2$.*

Using the above functions, we can now define:

$$durFinish_{A_i}(t) = t + dur_{A_i}(t) \tag{3.7}$$

$$durStart_{A_i}(t) = t - dur^{-1}_{A_i}(t) \tag{3.8}$$

$$lagFinish_{A_i,A_j}(t) = t + lag_{A_i,A_j}(t) \tag{3.9}$$

$$lagStart_{A_i,A_j}(t) = t - lag^{-1}_{A_i,A_j}(t) \tag{3.10}$$

While dur_{A_i} and lag_{A_i,A_j} represent relative time, the above functions specify absolute positions in time. For example, given a start time t for A_i, $dur_{A_i}(t)$ tells us how many time units after t A_i will finish but $durFinish_{A_i}(t)$ tells us the specific time on which A_i will finish.

It is worth noting that, as we saw for generic edge functions e_{V_i,V_j} and $e^{-1}_{V_i,V_j}$ in the GSTP, $lagFinish_{A_i,A_j}(t)$ and $lagStart_{A_i,A_j}(t)$ are not strictly inverses of each other since they can each be many-to-one functions.[18] For example, in Example 3.3.2,

$$lagFinish_{A_i,A_j}(Saturday) = lagFinish_{A_i,A_j}(Sunday) = Monday$$

[16]The relevant time unit will be $start_{A_i}$ if $type_{A_i,A_j} \in \{SS, SF\}$ and $finish_{A_i}$ otherwise.

[17]Recall that $finish_{A_i}$ is really the first time unit that A_i does not work.

[18]This is not an issue for our duration functions since we can assume that both time points (input and output) and the duration itself share the same calendar.

The above functions are all well-defined and easily computed. It should be clear that the following inequalities hold (since, for example, starting an activity later will never allow it to finish earlier):

$$t_m < t_n \Rightarrow durFinish_{A_i}(t_m) < durFinish_{A_i}(t_n) \tag{3.11}$$

$$t_m < t_n \Rightarrow durStart_{A_i}(t_m) < durStart_{A_i}(t_n) \tag{3.12}$$

$$t_m < t_n \Rightarrow lagFinish_{A_i,A_j}(t_m) \leq lagFinish_{A_i,A_j}(t_n) \tag{3.13}$$

$$t_m < t_n \Rightarrow lagStart_{A_i,A_j}(t_m) \leq lagStart_{A_i,A_j}(t_n) \tag{3.14}$$

Because calendars can be defined arbitrarily, the above are the strongest assertions we can make. It is simple to construct examples where the functions return arbitrarily far apart values for any pair of non-equal t_m and t_n (Consider a calendar with a long string of holidays in it).

Finally, although the value of $dur_{A_i}(t)$ changes depending on t, we assume here that dur_{A_i} is constant.[19] This is what allows us to use variables only for start times and ignore finish times since the finish time of an activity can always be obtained from the start time:

$$\begin{aligned} finish_{A_i} &= start_{A_i} + dur_{A_i}(start_{A_i}) \\ &= durFinish_{A_i}(start_{A_i}) \end{aligned}$$

3.3.3 The Edge Functions of the GSTP

We now have the building blocks to define the edge functions that are the constraints in our scheduling GSTP. Each edge function $e_{i,j}(t)$ represents the constraint $(A_i, A_j, type_{A_i,A_j}, lag_{A_i,A_j})$. It computes the minimum successor start time $(start_{A_j})$ that satisfied the constraint given the predecessor start time $(start_{A_i})$. There are four cases:

- If the constraint type is SS:

$$\begin{aligned} e_{i,j}(start_{A_i}) &= start_{A_i} + lag_{A_i,A_j}(start_{A_i}) \\ &= lagFinish_{A_i,A_j}(start_{A_i}) \end{aligned}$$

[19] In some PMSs, activities can be allowed to stretch in which case dur_{A_i} is itself a variable to optimize. We mention this extension and others in Section 3.6.

- If the constraint type is SF:

$$\begin{aligned} e_{i,j}(start_{A_i}) &= start_{A_i} + lag_{A_i,A_j}(start_{A_i}) \\ &\quad - dur_{A_j}^{-1}(start_{A_i} + lag_{A_i,A_j}(start_{A_i})) \\ &= durStart_{A_j}(lagFinish_{A_i,A_j}(start_{A_i})) \end{aligned}$$

- If the constraint type is FS:

$$\begin{aligned} e_{i,j}(start_{A_i}) &= start_{A_i} + dur_{A_i}(start_{A_i}) \\ &\quad + lag_{A_i,A_j}(start_{A_i} + dur_{A_i}(start_{A_i})) \\ &= lagFinish_{A_i,A_j}(durFinish_{A_i}(start_{A_i})) \end{aligned}$$

- If the constraint type is FF:

$$\begin{aligned} e_{i,j}(start_{A_i}) &= start_{A_i} + dur_{A_i}(start_{A_i}) \\ &\quad + lag_{A_i,A_j}(start_{A_i} + dur_{A_i}(start_{A_i})) \\ &\quad - dur_{A_j}^{-1}(start_{A_i} + dur_{A_i}(start_{A_i}) \\ &\quad + lag_{A_i,A_j}(start_{A_i} + dur_{A_i}(start_{A_i}))) \\ &= durStart_{A_j}(lagFinish_{A_i,A_j}(durFinish_{A_i}(start_{A_i}))) \end{aligned}$$

Because the function $e_{i,j}(t)$ can be expressed as a combination of functions (3.7) through (3.10), we can use facts (3.11) through (3.14) to conclude that, as required for the GSTP (see (3.3) and Lemma 3.1.1):

$$t_m < t_n \Rightarrow e_{i,j}(t_m) \leq e_{i,j}(t_n) \tag{3.15}$$

$$t_m < t_n \Rightarrow e_{i,j}^{-1}(t_m) \leq e_{i,j}^{-1}(t_n) \tag{3.16}$$

3.3.4 Time Windows

For each activity A_i, we extend the GSTP notion of a window as follows. We define a window as a five-tuple of time values, $(hes_{A_i}, ses_{A_i}, start_{A_i}, sls_{A_i}, hls_{A_i})$ with intended meaning: (hard earliest start, soft earliest start, actual start time, soft latest start, and hard latest start). These five can be divided into three subsets that serve the following purposes during schedule construction:

- The start time, $start_{A_i}$, will be the start time of the activity if it is scheduled and DEFAULT-TIME otherwise.

- The *hard window*, $[hes_{A_i}, hls_{A_i}]$, represents the interval of time points t for which there is at least one feasible schedule with A_i starting at t. The hard window will not change during schedule construction.
- Given a partially constructed schedule, the *soft window*, $[ses_{A_i}, sls_{A_i}]$, contains time points t for which there might be a feasible schedule with each scheduled activity A_j starting at the current value of $start_{A_j}$ and A_i starting at t. The soft window corresponds to the window $[lb_{A_i}, ub_{A_i}]$ defined for the generic GSTP.

When no activities are scheduled, $ses_{A_i} = hes_{A_i}$ and $sls_{A_i} = hls_{A_i}$. Also, for any feasible partial schedule, each activity window will have the property that $hes_{A_i} \leq ses_{A_i} \leq sls_{A_i} \leq hls_{A_i}$. In the following sections, we show how the GSTP algorithms can be used and enhanced to ensure backtrack-free search to solutions that satisfy all temporal constraints.

3.4 Maintaining Windows for Acyclic Problems

Now that we have shown how the GSTP framework can be used for scheduling problems as represented in PMSs, we adapt the GSTP algorithms for this problem. While some of our changes simplify implementation, others improve the running time in practice, if not in theory. We begin with the less general case and consider acyclic scheduling problems.

3.4.1 Window Initialization

Before running a scheduling algorithm using time windows, we initialize the windows using the procedure INITIALIZE-WINDOWS, a variation of CREATE-WINDOWS-ACYCLIC. The two main differences are:

1. We separate unary constraints from binary ones so that the former only need to be considered during initialization and can be ignored afterward.
2. We assume here that no variables are valued (notice that we set $start_{A_i} =$ DEFAULT-TIME on line 14).

As was shown in Section 3.1.2, the resulting windows will be sound and complete. We also know the problem is feasible if and only if all hard windows are non-empty.

Finally, Lemma 3.1.11 showed that the complexity is $O(v+e)$ where v is the number of activities and e is the number of binary constraints.

INITIALIZE-WINDOWS(A)

```
1  for i = 1 to n
2        do hes_{A_i} =max(es_{A_i}, durStart_{A_i}(ef_{A_i}), t_begin)
3           hls_{A_i} =min(ls_{A_i}, durStart_{A_i}(lf_{A_i}), t_end)
4  (A_1, ..., A_n) = TOPOLOGICAL-SORT(A)
5  for i = 1 to n
6        do
7           for each successor A_j of A_i
8                 do hes_{A_j} =max(hes_{A_j}, e_{i,j}(hes_{A_i}))
9  for i = n to 1
10       do
11          for each predecessor A_j of A_i
12                do hls_{A_j} =min(hls_{A_j}, e^{-1}_{i,j}(hls_{A_i}))
13 for i = 1 to n
14       do start_{A_i} = DEFAULT-TIME
15          ses_{A_i} = hes_{A_i}
16          sls_{A_i} = hls_{A_i}
```

3.4.2 Windows During Schedule Construction

During schedule construction, an algorithm will assign start times to activities. If we maintain sound and complete windows at each step, the algorithm need never select infeasible start times. The procedure SET-START-TIME is a variation of CREATE-WINDOWS-ACYCLIC that can be used to assign a start time and update all soft windows at the same time. Although similar to the way that hard windows are initialized, there are two key differences:

1. Instead of starting at one end of a topological sort and propagating toward the other, the algorithm begins at the scheduled activity and propagates constraints outward in both directions. In the successor direction, soft earliest starts are moved later. In the predecessor direction, soft latest starts are moved earlier.

2. The propagation must be able to handle activities whether or not they have been scheduled. A scheduled activity only serves to cut off propagation; since any scheduled activity A_i will have already propagated constraints using $start_{A_i}$

when A_i was scheduled, propagating constraints use ses_{A_i} and sls_{A_i} is unnecessary since they are less constraining than $start_{A_i}$.[20]

Consider lines 2 to 5 of SET-START-TIME. Because A_i has been scheduled, each successor A_j may no longer be able to be scheduled as early as ses_{A_j}. A new time $ses_{A_j} = e_{i,j}(start_{A_i})$ is calculated on line 4. This is done for each successor and a list of (activity, new-earliest-start-time) pairs is created.

```
SET-START-TIME(A_i, t_i)
 1  start_{A_i} = t_i
 2  succs-to-update = ∅
 3  for each successor A_j of A_i
 4        do t_j = e_{i,j}(t_i)
 5           succs-to-update.insert(A_j, t_j)
 6  SHRINK-SUCCESSORS(succs-to-update)
 7  preds-to-update = ∅
 8  for each predecessor A_j of A_i
 9        do t_j = e^{-1}_{j,i}(t_i)
10           preds-to-update.insert(A_j, t_j)
11  SHRINK-PREDECESSORS(preds-to-update)
```

Once all successors of A_i have been added, SHRINK-SUCCESSORS is called. Pseudocode for this procedure is shown below; it propagates the constraints forward through the constraint graph. At each iteration, pair (A_j, t_j) is selected where A_j is the earliest activity in the topological sort appearing in *succs-to-update*.[21] If t_j forces the soft window of A_j to be smaller, the soft window is updated. If this occurs and A_j is unscheduled, this new time must be propagated to all successors of A_j. The process continues until the list of propagations to perform becomes empty.

The procedure SHRINK-PREDECESSORS is symmetric to SHRINK-SUCCESSORS. It propagates backward according to the topological sort and will update soft latest start times of activity predecessors.

---

[20] This assumes that $ses_{A_i} \leq start_{A_i} \leq sls_{A_i}$ which is guaranteed by Theorem 3.1.2.

[21] The topological sort can be kept around from the INITIALIZE-WINDOWS phase since it will never change.

SHRINK-SUCCESSORS(*succs-to-update*)

```
1  while succs-to-update ≠ ∅
2        do Choose (A_j, t_j) ∈ succs-to-update such that
                    ∀(A_k, t_k) ∈succs-to-update, T(A_j) < T(A_k)
                    and remove (A_j, t_j) from succs-to-update
3           if t_j > ses_{A_j}
4              then ses_{A_j} = t_j
5                   if start_{A_j} = DEFAULT-TIME
6                      then for each successor A_k of A_j
7                                 do if (A_k, t_k) ∈ succs-to-update
8                                       then set t_k = max(t_k, e_{j,k}(t_j))
9                                       else
10                                            succs-to-update.insert(A_k, e_{j,k}(t_j))
```

Notice that when an activity is scheduled, we do not set $ses_{A_i} = sls_{A_i} = start_{A_i}$ as one might expect. Instead, the soft window maintains the interval of start times that work for A_i given the current partially constructed schedule. This will be useful later for unscheduling (Section 3.4.3).

We use Lemmas 3.4.1 and 3.4.2 to show that windows will be minimally complete which implies that they are sound for acyclic problems. The arguments are similar to those used for in Section 3.1.1; this should be no surprise since SET-START-TIME is, in spirit, a reorganized version of CREATE-WINDOWS-ACYCLIC that avoids unnecessary computation.

Lemma 3.4.1. *For an acyclic problem, if all soft windows are sound and complete for a feasible partial schedule and* SET-START-TIME(A_i, t_i) *is called for a feasible time* t_i*, each* ses_{A_j} *in the resulting soft windows will be the smallest feasible start time for* A_j*.*

Proof: To show that there is no smaller $t < sesA_j$ that is feasible, we can make arguments similar to those of Theorem 3.1.2. We assume that there is such a t and that, without loss of generality, A_j is the first activity to have a feasible time removed from its window.

We know the windows were complete before the SET-START-TIME call therefore t must have been removed during an iteration of the SHRINK-SUCCESSORS while loop. But then the new value of ses_{A_j} is determined by a predecessor with whom all

$t' < ses_{A_j}$ will be inconsistent. Therefore, we have a contradiction and each ses_{A_j} is the smallest possible feasible start time for A_j.

We must also show that ses_{A_j} is in fact feasible. Let P and P' be the partial schedules before and after the call to SET-START-TIME. We now modify the arguments of Lemma 3.1.3 to show that we can extend P' to a feasible schedule S with each $start_{A_j} = ses_{A_j}$ for each unscheduled A_j.

To show that S is feasible, we must show that all constraints are satisfied. Clearly, any unary constraint will be satisfied due to the initialization of hard windows (and subsequently, soft windows). For any binary constraint between A_j and A_k, consider the possibilities for the values of A_j and A_k in P':

- A_j and A_k are both valued. If one of them is A_i then the constraint must be satisfied since we require that $start_{A_i}$ be feasible. Otherwise, the constraint must be satisfied since we assume P is feasible.
- A_j is valued. If $A_j = A_i$, then lines 2 to 5 guarantee that any ses_{A_k} will be consistent with this start time. Otherwise, A_j was valued in P and ses_{A_k} was already consistent with its value.
- A_j is unvalued. If ses_{A_j} is not changed during SET-START-TIME, the constraint will be satisfied since the old ses_{A_k} will work. If ses_{A_j} is changed, A_k will be updated on line 10 of SHRINK-SUCCESSORS and the constraint will be enforced in a subsequent iteration of the corresponding while loop.

Therefore, each ses_{A_j} is the smallest feasible start time for A_j.

■

Lemma 3.4.2. *For an acyclic problem, if all soft windows are sound and complete for a feasible partial schedule and* SET-START-TIME(A_i, t_i) *is called for a feasible time t_i, each sls_{A_j} in the resulting soft window will be the largest feasible start time for A_j.*

Proof: The proof is symmetric to that of Lemma 3.4.1.

■

Corollary 3.4.3. *For an acyclic problem, if all soft windows are sound and complete and* SET-START-TIME(A_i, t_i) *is called for a feasible time t_i, the resulting soft windows will be sound and complete.*

Proof: Together, Lemmas 3.4.1 and 3.4.2 imply that the resulting windows are minimally complete. By Corollary 3.1.9 they must also be sound.

■

If start times are always chosen from within soft windows and SET-START-TIME is invoked with each selection, schedule construction can proceed backtrack free.

Example 3.4.1. *Consider the GSTP of Example 3.1.3 as a 3 activity scheduling problem with SS constraints* $e_{1,2}(t) = t + 4$ *and* $e_{1,3}(t) = t + 2$. *Because we have* $start_{A_3} = 5$, *we have soft windows*

$$[ses_{A_1}, sls_{A_1}] = [0, 3], [ses_{A_2}, sls_{A_2}] = [4, 10], \text{ and } [ses_{A_3}, sls_{A_3}] = [2, 10].$$

If we decide to schedule A_1 *at time 3, the call to* SET-START-TIME$(A_1, 3)$ *will invoke* SHRINK-SUCCESSORS$(\{(A_2, 7), (A_3, 5)\})$. *This will reduce those two windows to*

$$[ses_{A_2}, sls_{A_2}] = [7, 10], \text{ and } [ses_{A_3}, sls_{A_3}] = [5, 10]$$

.

Lemma 3.1.11 means we can perform backtrack-free schedule construction in $O(v(v + e))$ time. There are a couple of reasons to believe the running time in practice will be much better than this. First, if a soft window is not shrunk during SHRINK-SUCCESSORS, the successors of that activity need not be considered at that time[22] (likewise for predecessors in SHRINK-PREDECESSORS). Second, when a scheduled activity is reached during propagation, the propagation stops. Therefore, in practice, each SET-START-TIME call will only lead to propagation through a small subset of the constraints rather than all of them.

3.4.3 Windows During Schedule Deconstruction

Some algorithms may not be purely constructive. For example, it is possible to do local search in schedule space by altering completed schedules. Such algorithms

[22]The successors might still be affected by another activity whose soft window is shrunk.

require the ability to unschedule and reschedule activities. We would like to be able to do so while maintaining soft windows that are sound and complete.

The procedure UNSET-START-TIME updates soft windows when an activity A_i is unscheduled. We noted above that the soft window of A_i itself does not need updating: it is dependent only on the predecessors and successors of A_i and not on $start_{A_i}$. However, the windows of the predecessors and successors of A_i must be suitably expanded.

```
UNSET-START-TIME(A_i)
1  start_{A_i} = DEFAULT-TIME
2  preds-to-update = ∅
3  for each predecessor A_j of A_i
4        do preds-to-update.insert(A_j)
5  EXPAND-PREDECESSORS(preds-to-update)
6  succs-to-update = ∅
7  for each successor A_j of A_i
8        do succs-to-update.insert(A_j)
9  EXPAND-SUCCESSORS(succs-to-update)
```

Consider the predecessors of A_i. These are added to a list that is processed in EXPAND-PREDECESSORS. When A_i is unscheduled, this affects sls_{A_j} for each predecessor A_j of A_i since it may be possible to schedule A_j later than before.

EXPAND-PREDECESSORS, outlined below, processes the *preds-to-update* list beginning with the largest A_j according to the topological sort. The maximum value for sls_{A_j} is recomputed from scratch beginning with its maximum value, hls_{A_j}, as set in line 3. In lines 4 to 8, sls_{A_j} is further reduced enough so that the constraint with each successor of A_j is satisfied.

In lines 9 to 13, sls_{A_j} is updated. If the window is expanded and if A_j is not scheduled, all predecessors of A_j are added to the *preds-to-update* list. Notice that the propagation does not continue if A_j is scheduled since $start_{A_j}$ ($\leq sls_{A_j}$) remains the relevant constraint on the predecessors.

EXPAND-PREDECESSORS(*preds-to-update*)

```
1  while preds-to-update ≠ ∅
2        do Choose (A_j) ∈ preds-to-update such that
                    ∀A_k ∈preds-to-update, T(A_j) > T(A_k)
                    and remove A_j from preds-to-update
3           latest-start = hls_{A_j}
4           for each successor A_k of A_j
5                do if start_{A_k} = DEFAULT-TIME
6                      then must-start-before = e^{-1}_{j,k}(sls_{A_k})
7                      else must-start-before = e^{-1}_{j,k}(start_{A_k})
8                   latest-start = min(latest-start,must-start-before)
9           if latest-start > sls_{A_j}
10             then sls_{A_j} = latest-start
11                  if start_{A_j} = DEFAULT-TIME
12                     then for each predecessor A_i of A_j
13                               do preds-to-update.insert(A_i)
```

Notice that UNSET-START-TIME is the reason we have explicitly separated hard and soft windows.[23]

Example 3.4.2. *Consider the problem of Example 3.4.1. We have two activities scheduled ($start_{A_1} = 3$ and $start_{A_3} = 5$) and windows*

$$[ses_{A_1}, sls_{A_1}] = [0, 3], [ses_{A_2}, sls_{A_2}] = [7, 10], \text{ and } [ses_{A_3}, sls_{A_3}] = [5, 10].$$

If we decide to unschedule A_3 (currently scheduled at time 5), UNSET-START-TIME *will invoke* EXPAND-PREDECESSORS($\{A_1\}$). *The window of A_1 will be expanded to*

$$[ses_{A_1}, sls_{A_1}] = [0, 6]$$

Notice that A_1 is now limited by A_2 rather than A_3.

Lemma 3.4.4. *For an acyclic problem, if all soft windows are sound and complete for a feasible partial schedule and* UNSET-START-TIME(A_i) *is called, each sls_{A_j} in the resulting soft windows will be the largest feasible start time for A_j.*

---

[23] Although it is possible to do the above without the hard window, hard windows also allow the problem to be quickly reinitialized without another call to INITIALIZE-WINDOWS.

Proof: This will be an inductive proof based on the topological sort and beginning at its end. It relies on the fact that the sls_{A_j} values are updated in reverse topological order. This fact can be seen by noting that line 2 chooses the activities in this order and an activity can only be added to *preds-to-update* if it is before the current activity in the topological sort (since only predecessors of the current activity are added).

For each such A_j, we must show the following two facts:

1. sls_{A_j} is feasible.
2. There is no feasible t where $t > sls_{A_j}$.

Base case: Consider the last activity A_j in the topological order. Since sls_{A_j} is only dependent on successors and A_j has none, $sls_{A_j} = hls_{A_j}$. Therefore sls_{A_j} must be the largest feasible start time for A_j.

Inductive case: We assume the above two facts hold for all A_k such that $T(A_k) > T(A_j)$. Therefore, for each successor A_k of A_j, sls_{A_k} is the largest feasible start time for A_k.[24]

The correct value of sls_{A_j} depends on the windows of its successors. Inspection of the code indicates that sls_{A_j} will be updated if and only if it is added to *preds-to-update* which implies either:

1. A_i (the unset activity) is a successor of A_j, or
2. An unvalued successor A_k of A_j has been updated.[25]

If either of these occur, sls_{A_j} will be updated in lines 3 to 10. Since this checks every constraint (A_j, A_k), the resulting sls_{A_j} will be feasible. Also, since each update uses sls_{A_k} which is the largest feasible time of A_k (by the inductive hypothesis), sls_{A_j} is only reduced in cases where it is necessary.

If neither of the above occur, sls_{A_j} will be unchanged. It is clear that the unchanged sls_{A_j} is feasible. To see that there is no larger feasible time, we use the

[24] Although our inductive hypothesis only states that this is true *after* the call to UNSET-START-TIME, it is clear that it must then also be true before A_j is considered due to the fact that windows are expanded in reverse-order according to the topological sort.

[25] If, on the other hand, a valued successor A_k is updated, A_j is unaffected by that update. This is because $start_{A_k}$ is more constraining than sls_{A_k}.

fact that windows were complete before the UNSET-START-TIME call; for any larger time to be feasible would therefore require at least one constraint between A_j and a successor to be weakened and this could not have occurred.

Therefore, our inductive arguments show that each sls_{A_j} after UNSET-START-TIME is the largest feasible start time for A_j.

■

Lemma 3.4.5. *For an acyclic problem, if all soft windows are sound and complete for a feasible partial schedule and* UNSET-START-TIME(A_i) *is called, each* ses_{A_j} *in the resulting soft windows will be the smallest feasible start time for* A_j.

Proof: The proof is symmetric to that of Lemma 3.4.4.

■

Corollary 3.4.6. *For an acyclic problem, if all soft window are sound and complete for a feasible partial schedule and* UNSET-START-TIME(A_i) *is called, the resulting time windows will be sound and complete.*

Proof: Together, Lemmas 3.4.4 and 3.4.5 imply that the resulting windows are minimally complete. By Corollary 3.1.9 they are also sound.

■

Like SET-START-TIME, the complexity of UNSET-START-TIME is $O(v+e)$ since all vertices and edges will be handled at most once. Notice that we could also invoke a version of CREATE-WINDOWS-ACYCLIC to update windows when an activity is unscheduled. Although the theoretical complexity would be the same, it should be clear that UNSET-START-TIME will be much faster in practice.

The above results now guarantee that sound and complete soft windows can be maintained throughout any amount of scheduling or unscheduling. This framework can therefore underlie many different search algorithms and ensure that search will be backtrack-free as far as temporal constraints are concerned. In later chapters, we build search algorithms using this framework.

3.5 Maintaining Windows for Cyclic Problems

In the above procedures for maintaining time windows, we assumed an acyclic constraint graph. Although most commercial tools disallow cycles, they are common in real-world problems. For example, consider a supervisor's inspection that must take place at some point while an electrician is wiring a house. We want to impose constraints $(wiring, inspection, SS, 0)$ and $(inspection, wiring, FF, 0)$. Including both of these constraint puts a cycle into the constraint graph. Other examples occur in situations like the 'hot ingot' problem where there is the need to impose maximum time lags as well as minimum time lags between activities [42]. This is the motivation for work considering RCPSP/max scheduling problems [14, 20, 23, 74].

In this section, we adapt the GSTP framework for PMS problems with cycles. The main difference between the results here and the previous section is that we cannot define and use a topological sort for a cyclic constraint graph. In the many places where that sort was used above, we can no longer do so.

3.5.1 Window Initialization

The procedure INITIALIZE-WINDOWS-WITH-CYCLES adapts procedure CREATE-WINDOWS for our problem formulation. As before, the main difference is that unary constraints are handled separately. Notice that a topological sort is no longer used. Also notice that on lines 5 and 13 we stop if a window has become empty. Without this, propagation could continue forever even when time is discrete (see Example 3.5.2, for example).

As we saw in Section 3.1.1, the resulting windows will be minimally complete. Lemma 3.1.3 showed two feasible schedules that can be obtained and Lemma 3.1.5 showed that the problem is feasible if and only if no empty window results. Example 3.5.1 shows why the resulting windows are not necessarily sound.

INITIALIZE-WINDOWS-WITH-CYCLES(A)

```
1  for i = 1 to n
2        do hes_{A_i} = max( es_{A_i}, dur_{A_i}^{-1}(ef_{A_i}), t_begin)
3           hls_{A_i} = min( ls_{A_i}, dur_{A_i}^{-1}(lf_{A_i}), t_end )
4  Q = (A_1, ..., A_n)
5  while Q ≠ ∅ and no empty hard window
6        do Choose any A_i in Q
7           for each successor A_j of A_i
8                 do new-earliest = e_{i,j}(hes_{A_i})
9                    if new-earliest > hes_{A_j}
10                      then hes_{A_j} = new-earliest
11                           add A_j to Q
12 Q = (A_1, ..., A_n)
13 while Q ≠ ∅ and no empty hard window
14       do Choose any A_i in Q
15          for each predecessor A_j of A_i
16                do new-latest = e_{j,i}^{-1}(hls_{A_i})
17                   if new-latest < hls_{A_j}
18                      then hls_{A_j} = new-latest
19                           add A_j to Q
20 for i = 1 to n
21       do start_{A_i} = DEFAULT-TIME
22          ses_{A_i} = hes_{A_i}
23          sls_{A_i} = hls_{A_i}
```

Example 3.5.1. *Consider constraints $(A_i, A_j, SS, 0)$ and $(A_j, A_i, FF, 0)$ between two 1 day activities where A_i uses CALBASE and A_j uses CAL5. It is possible for the constraint propagation to result in hard windows $[hes_{A_i}, hls_{A_i}] = [hes_{A_j}, hls_{A_j}] = [Friday, Monday]$.* INITIALIZE-WINDOWS-WITH-CYCLES *need not further reduce these windows, since scheduling both activities on either Friday or Monday gives a feasible schedule.*

However, the window for A_i is not sound. In particular, A_i cannot start on Saturday or Sunday even though these are work days in its calendar. In both cases A_j would have to be on Monday to satisfy the SS constraint and on Friday to satisfy the FF constraint which is impossible.

The next example shows a simple case where a single hard window can be updated a number of times before INITIALIZE-WINDOWS-WITH-CYCLES is finished. This

shows why the complexity for cyclic problems is $O(v + ed)$ (see Lemma 3.1.6).

Example 3.5.2. *Consider the two constraints of Example 3.5.1 and suppose that A_i works Monday, Wednesday and Friday and A_j works Tuesday, Thursday and Saturday. Suppose that line 2 of* INITIALIZE-WINDOWS-WITH-CYCLES *sets $hes_{A_i} = Monday$. The constraint propagation of* INITIALIZE-WINDOWS-WITH-CYCLES *will result in the following set of propagations:*

$$\begin{aligned} hes_{A_i} = Monday &\Rightarrow hes_{A_j} = Tuesday \\ &\Rightarrow hes_{A_i} = Wednesday \\ &\Rightarrow hes_{A_j} = Thursday...etc. \end{aligned}$$

This should make it clear that the cyclic nature of the precedence graph means that a single constraint can be propagated many times.

3.5.2 Windows During Schedule Construction

As for the acyclic problem, we maintain soft windows during schedule construction. The SET-START-TIME procedure can still be used because it does not rely on a topological sort although line 2 in SHRINK-SUCCESSORS must be replaced with one that arbitrarily picks an A_j from the *succs-to-update*.

Example 3.5.1 shows that soundness does not hold during schedule construction for a cyclic problem (since not all time points in the window are feasible). By the arguments of Lemma 3.1.5, however, if SET-START-TIME is called with an infeasible time, we immediately get an empty window. Therefore, backtracking only requires undoing the most recent decision.

Because of the possibility of backtracking, an attempt to feasibly schedule any activity might take up to d tries. Therefore, the time complexity of constructing a schedule will be $O(v(v + ed))$. However, we always know that ses_{A_i} and sls_{A_i} are feasible start times for an activity and we could construct the two feasible schedules of Lemma 3.1.3 in $O(v)$ time.

3.5.3 Windows During Schedule Deconstruction

The following example shows that UNSET-START-TIME will not work for cyclic problems.

Example 3.5.3. *Return to Example 3.5.1 and suppose that* $start_{A_i} = Friday$ *and* A_j *is unscheduled. We will have soft windows:*

$$[ses_{A_i}, sls_{A_i}] = [ses_{A_j}, sls_{A_j}] = [Friday, Friday]$$

If we then unschedule A_i*, we should get back the soft windows:*

$$[ses_{A_i}, sls_{A_i}] = [ses_{A_j}, sls_{A_j}] = [Friday, Monday]$$

However, UNSET-START-TIME *will not expand the soft windows. In particular, the attempt to expand the soft window of* A_i *will fail because the latest start of* A_j *(Friday) will continue to constrain it even though that latest start itself should be expanded.*

When cycles are allowed, a scheduled activity can constrain itself and this constraint will not be lifted when UNSET-START-TIME is used. Instead, to maintain minimally complete soft windows (and therefore get the most out of window expansion), we must recompute soft windows from scratch. This can be done with procedure UNSET-START-TIME-WITH-CYCLES as shown below. It is simply a version of INITIALIZE-WINDOWS-WITH-CYCLES that takes into account scheduled activities.

Theoretically, then, the complexity of unscheduling is roughly the same as scheduling. In practice, however, UNSET-START-TIME-WITH-CYCLES is likely to do a lot of unnecessary work as the following example shows.

Example 3.5.4. *Consider an activity* A_i *that is not involved in any binary constraints. When* A_i *is unscheduled,* UNSET-START-TIME-WITH-CYCLES(A_i) *will end up resetting all soft windows even though it is clear (to us) that no updates to the soft windows are necessary.*

To avoid the slower speed of UNSET-START-TIME-WITH-CYCLES, there are a couple of approaches that can be used:

1. A simple approach is to ignore the problem of Example 3.5.3 and use UNSET-START-TIME (modified to not use a topological sort). Because the only problem is that soft windows are not expanded as much as possible, search will be able to continue toward feasible solutions. The disadvantage, of course, is that some feasible schedules may not be considered once unscheduling has been done.

2. A better approach might be to keep track of which activities are part of cycles. Only when a window of one of these activities is expanded would extra work be necessary. Unfortunately, though, even an activity not involved in any cycle can affect activities that are involved in cycles. Nonetheless, unnecessary computation in situations like that of Example 3.5.4 can be avoided.

UNSET-START-TIME-WITH-CYCLES(A_i)

```
1  for i = 1 to n
2        do ses_{A_i} = hes_{A_i}
3           sls_{A_i} = hls_{A_i}
4  Q = (A_1, ..., A_n)
5  while Q ≠ ∅
6        do Choose any A_i in Q
7           for each successor A_j of A_i
8                 do if start_{A_i} = default-time
9                       then new-earliest = e_{i,j}(ses_{A_i})
10                      else  new-earliest = e_{i,j}(start_{A_i})
11                    if new-earliest > ses_{A_j}
12                      then ses_{A_j} = new-earliest
13                           add A_j to Q
14 Q = (A_1, ..., A_n)
15 while Q ≠ ∅
16       do Choose any A_i in Q
17          for each predecessor A_j of A_i
18                do if start_{A_i} = default-time
19                      then new-latest = e^{-1}_{j,i}(sls_{A_i})
20                      else  new-latest = e^{-1}_{j,i}(start_{A_i})
21                    if new-latest < sls_{A_j}
22                      then sls_{A_j} = new-latest
23                           add A_j to Q
```

Note that this second approach could also speed up window initialization and activity scheduling for cyclic problems. A topological sort could be maintained for all non-cycle activities. This could help keep to a minimum the number of times any binary constraint is re-propagated.

In conclusion, it is probably the above complications of cyclic problems that have led most PMS developers to disallow cycles.

3.6 Other Issues

We have not yet touched on all of the temporal issues that exist in PMSs. There are options available in some systems that affect how window management should work. While most of them do not affect the complexity of our approach, they do affect the implementation and ought to be mentioned here.

3.6.1 Fixed Start Times

In addition to the unary constraints we have considered, Artemis allows 'fixed start day' or 'fixed start period' unary constraints that further restrict when an activity can begin. For example, the user can require that activity A_i start on a particular day of the week or at a particular time of day. A variation of this type of constraint could force an activity to be completed during a single shift or during a single day.

The properties and complexity of our window management procedures remain unchanged as long as we ensure that the five time points in a window also satisfy the new constraint. For example, line 2 in the procedure INITIALIZE-WINDOWS should be followed by one that then moves hes_{A_i} to the next time point that also meets the fixed start time constraint.

3.6.2 Stretchable Activities

In Artemis, the user may allow activities to stretch. In this case, dur_{A_i} specifies a minimum duration for A_i. Because each activity A_i no longer has a set duration dur_{A_i}, we can no longer use only $start_{A_i}$ to define how A_i is scheduled. Our GSTP framework must include two variables for each activity: $start_{A_i}$ and $finish_{A_i}$. We will then have two time windows for each activity: $(hes_{A_i}, ses_{A_i}, start_{A_i}, sls_{A_i}, hls_{A_i})$ and $(hef_{A_i}, sef_{A_i}, finish_{A_i}, slf_{A_i}, hlf_{A_i})$.

If we represent an activity with two windows, we must also add a binary constraint between them that represents the minimum duration. With this representation, we can continue to use our procedures without affecting their properties or complexity.

It is worth noting that some cyclic problems become acyclic when stretch is allowed as the following example shows.[26]

Example 3.6.1. *Consider a problem with the two constraints* $(A_i, A_j, SS, 0)$ *and* $(A_j, A_i, FF, 0)$.[27]

There is a cycle since A_i *is a successor of itself. However, there is no cycle when stretch is allowed. This is because* $finish_{A_i}$ *is a successor of* $start_{A_i}$ *but not of itself. If we define separate windows (and separate variables) for the starts and finishes of activities, our constraint graph will have no cycles.*

3.6.3 Splittable and Elastic Activities

In Artemis and Open Plan, activities may be defined as splittable or as elastic. A splittable activity may have its duration split into a number of pieces. For example, a three day activity could have two days scheduled one week and the third day scheduled a month later.

An elastic activity is further relaxed so that it has no specified duration. Instead, a certain amount of total work must be accomplished between its start and finish.

In the literature, splittable activities correspond to preemption as often discussed with respect to job shop scheduling [14, 42, 47, 55]. Elastic activities are less studied but are considered by Baptiste et al. [8] and Caseau and Laburthe [19]. Although these theoretical studies consider fully preemptive schedules or the 'fully elastic problem,' PMS users are required to bound the flexibility of activities with extra constraints (maximum duration, for example).

---

[26] In fact, it appears that this is precisely the reason that stretch is allowed in Artemis. It allows some desirable cyclic constraints to be 'represented' without losing the computational advantages of an acyclic problem.

[27] This is the example used by Artemis to explain loops with respect to their STRETCH/NO STRETCH option [2].

As far as our window representation is concerned, it should be the case that the extension discussed above for stretchable activities can be used in both the splittable and elastic cases as well. The procedures will still work and their properties will hold as far as the start and finish of any activity are concerned. However, more information than simply a start and finish time will be required in order to know exactly how and when an activity is scheduled.

3.7 Related Work

Maintaining temporal consistency using time windows is common in scheduling systems. It has origins in the CPM and PERT project management techniques that have been around for half a century (the basic idea of performing a forward and backward pass to calculate windows was introduced in these techniques). In AI approaches, temporal consistency forms the foundation for solving problems within a CSP framework [23, 66, 96, 99] and in OR approaches, windows are used to prune possible choices, determine branching choices and propagate resource constraints [8, 13, 14, 19, 73].

Because temporal domains are ordered, it is standard for temporal constraint propagation algorithms, like those we have outlined above, to propagate the constraints using the domain bounds (earliest start and latest start). For example, our procedures explicitly maintain arc-B consistency since they make sure that ses_{A_i} and sls_{A_i} are kept feasible. For the acyclic case, we saw that arc consistency (which is stronger) is achieved as well since windows are sound (Lemma 3.1.7).

A common approach to maintaining temporal consistency is that described by Dechter et al. [32]. Dummy activities A_0 and A_{n+1} are used to represent the begin and end of the project and the Floyd-Warshall all pairs shortest path algorithm is used to calculate a *distance matrix* of minimum time lags d_{ij} between every pair of activities. Windows can be derived directly from the resulting distance matrix by setting $[lb_{A_i}, ub_{A_i}] = [d_{0i}, t_{end} - d_{i(n+1)}]$. Window updates during schedule construction or deconstruction can also be derived directly from the distance matrix. Path consistency can be achieved.

The work of Dechter et al. is generalized somewhat by Meiri [69]. The latter approach has many features in common with ours including the notion of sound domains, a constraint graph and the use of a topological sort for acyclic problems. Many of Meiri's algorithms are similar to the ones we have described here. For example, our INITIALIZE-WINDOWS algorithm is quite similar to Meiri's algorithm 2-DAC (two way directional arc consistency).

On the one hand, the work of both Dechter et al. and Meiri is more general than ours in the sense that they consider a wider variety of constraints including disjunctive temporal constraints (A_i and A_j cannot overlap, for example). We keep our definition of the GSTP narrower because otherwise the computational efficiency of maintaining time windows is lost.[28] For example, disjunctive constraints lead to domains that can no longer be represented by single intervals.

On the other hand, the work of Dechter et al. and Meiri is not as general as ours because it cannot handle calendars. Specifically, they assume that edge functions are constant. Because our constraint graph has variable edge weights (represented by functions), we cannot compute a distance matrix. There is no reasonable way to combine edge functions and to maintain implicit constraints. Even two constraints between a single pair of activities cannot be combined; the one that is more constraining may depend on the particular time point under consideration.

Therefore, it is impractical to maintain anything stronger than arc consistency in our framework. This suggests our procedures are doing as much as we can expect in terms of consistency maintenance in the general case.[29] If a problem has certain characteristics (many activities share a single calendar, for example), it might be reasonable to locally maintain stronger forms of consistency.

There is very little discussion in the literature concerning the maintenance of time windows during search but we suspect many people do it. It has the flavor of

---

[28] Also, other varieties of constraints do not typically show up in the scheduling problems considered in the literature or allowed in PMSs.

[29] Of course, if we used the general CSP approach and represented constraints as sets of feasible values, stronger forms of consistency would be possible but impractical. For each pair of variables, we would have to keep track of the specific combinations of domains values that are consistent with each other.

lookahead algorithms for maintaining arc consistency during search as discussed by Tsang [94]. There is even less discussion of how temporal issues are handled when scheduling decisions are retracted, probably because most algorithms are purely constructive. An exception to this is the ODO constraint-directed scheduling framework discussed by Beck et al. [10], in which retraction is a key component.

As for calendars, there is work on efficient and rational ways to represent time [29, 61, 76, for example]. These concern the representation of time with different granularities (seconds relative to years), how to understand relationships between temporal representations (how the first Monday of December relates to the first day of December, for example) and how to represent time compactly so that computation is efficient. This work is orthogonal to our notion of calendars. Here we are only really concerned with splitting time units into working units and non-working units.

What appears to be a first attempt to address the calendar issues we address here appears in a paper by Zhan [102]. This work is concerned simply with calculating the earliest possible dates for all activities given minimum and maximum time lags and calendars. The resulting algorithm and proof of correctness are similar to procedure CREATE-WINDOWS and Theorem 3.1.2, respectively. Zhan also considers activities that must be completed within a single span of working units for a calendar (each activity cannot overlap any break times). Handling this would be equivalent to handling the fixed start day and fixed start time issues mentioned above.

The other work that considers the particular calendar issues we discuss here is that of Franck et al. [39]. Their work is similar to what we describe here. They define calendars as zero-one step functions and describe functions for activity durations and time lags that integrate those step functions.[30] This representation fits directly into our GSTP framework.

Frank et al. describe a forward pass and a backward pass algorithm. Together, those produce the windows we get with CREATE-WINDOWS. The algorithms that use their windows include both scheduling and unscheduling steps. However, there is no mention of specific algorithms to update windows; instead, they suggest the

[30]For example, $finish_{A_i}$ is the minimum t for which the sum between $start_{A_i}$ and t is dur_{A_i}.

initialization algorithms be called each time.

The complexity results of Franck et al. are slightly different than ours. They do not assume that the duration and lag functions can be computed in constant time as we do. Also, they describe the overall complexity of CREATE-WINDOWS as $O(ve(b+1))$ (instead of $O(v+ed)$), where b is the number of breaks in the relevant calendar. This is obviously a better bound when time is continuous since d is not included.

Obviously, calendar issues have also been solved to some extent in the implementation of project management software systems. However, we are not aware of documentation that describes these solutions. Also, we suspect the implementations may be somewhat inefficient since they were not designed with optimization algorithms in mind.

CHAPTER 4

An Algorithm for Makespan Minimization with Arbitrary Temporal Constraints

4.1 Introduction

The minimization of makespan is a reasonable objective whenever it is beneficial to complete a project as quickly as possible. This occurs when a shorter makespan allows more projects to be completed (imagine cars on an assembly line) or when financial incentives are associated with a project's completion. Makespan minimization is the most well-studied objective in the project scheduling literature. In most project management systems the only available scheduling algorithm is one that minimizes makespan.

In this chapter,[1] we focus on RCPSP/max problems. These generalize many of the well-known and well-studied classes of scheduling problems (including job-shop problems and their variants) through non-unary resource constraints and, most importantly, arbitrary temporal constraints. The latter permit both minimum and

[1]This chapter is based on material co-authored with John Pyle that is in press [90].

maximum time lags between activities to be represented.[2] This allows many real-world constraints to be modeled, including fixed start times, releases and deadlines, set-up times, minimum and maximum overlaps, activities that must start or finish simultaneously, and shelf life constraints.

RCPSP/max problems have been tackled by both the AI and OR communities [23, 33, 74] with most results achieved by the latter. Successful exact methods incorporate constraint propagation into a variety of branch and bound algorithms. Since these algorithms have limited scalability, a number of non-systematic methods, including Tabu search, simulated annealing and genetic algorithms have also been developed.

In this chapter, we describe how another heuristic approach, squeaky wheel optimization (SWO), can be applied effectively to RCPSP/max problems. The main component of SWO is a priority-based greedy schedule constructor. Search is used to find effective prioritizations for that constructor.

Arbitrary temporal constraints can cause problems for a greedy constructor since it is NP-hard to find any feasible schedule; if most prioritizations do not yield feasible schedules, SWO will be ineffective. Therefore, to increase the power of the greedy constructor, we have added a window-based conflict resolution mechanism, called bulldozing, that allows the greedy constructor to move sets of activities later in a partial schedule to maintain feasibility. Similar mechanisms move sets of activities earlier in a feasible schedule to allow shorter schedules to be discovered.

Our algorithm is compared with the best reported results in the OR and AI literature for benchmark RCPSP/max problems ranging in size from 10 to 500 activities. Competitive for all problem sizes, our algorithm begins to outperform other algorithms as problem size increases.

---

[2] As we saw in Chapter 3, a maximum time lag between A_i and A_j can be represented with a minimum time lag in the other direction (between A_j and A_i). Therefore, allowing maximum time lags is equivalent to allowing cycles in the temporal constraint graph. In this chapter we use the maximum time lag terminology as it is the standard way to discuss RCPSP/max problems.

4.2 The RCPSP/max Problem

We begin with a formal definition of the RCPSP/max problem:

Definition 10. *An RCPSP/max problem contains a set* $\{A_1, ..., A_n\}$ *of activities and a set* $\{R_1, ..., R_m\}$ *of renewable resources. Each activity* A_i *has a start time* $start_{A_i}$ *and a duration* d_i*. Each resource* R_k *has a maximum capacity* c_k *and execution of each activity* A_i *requires an amount* r_{ik} *of resource* R_k *for each time unit of execution. Binary precedence constraints* $[T_{i,j}^{min}, T_{i,j}^{max}]$ *enforce minimum* ($T_{i,j}^{min}$) *and maximum* ($T_{i,j}^{max}$) *time lags between the start times of activities* A_i *and* A_j*.*

The goal of an RCPSP/max problem is to find a feasible schedule such that the project makespan,[3] $MK = max_{1 \leq i \leq n}\{start_{A_i} + d_i\}$*, is minimized.*

The two constraints types of an RCPSP/max problem can be summarized as follows:

- **Precedence Constraints:** $T_{i,j}^{min} \leq start_{A_j} - start_{A_i} \leq T_{i,j}^{max}$
- **Resource Constraints:** $\Sigma_{\{i|start_{A_i} \leq t < start_{A_i}+d_i\}} r_{ik} \leq c_k, \forall t, \forall k$

Unlike scheduling problems without both minimum and maximum time lags, even finding a feasible schedule for the RCPSP/max problem is NP-hard [9].

4.3 Solving RCPSP/max Problems

4.3.1 Time Windows and Constraint Propagation

At the core of our algorithm is a greedy schedule constructor that selects a start time for each activity from a domain of possible values. Therefore, a framework for maintaining time windows forms the foundation for our algorithm.

The GSTP framework described in Chapter 3 can be used to maintain time windows for RCPSP/max problems. However, because we consider benchmark problems without calendar constraints, we use the more common and more efficient framework

[3]The project duration beginning at time 0.

described by Dechter et al. [32]. In Section 4.4.3, we present results using the more generic problem representation based on the GSTP framework that is applicable to real-world problems.

Windows are updated during schedule construction. When a start time t is selected for an activity A_i, the PLACE-ACTIVITY(A_i,t,W) method (as outlined below) does two things:

1. Call SET-START-TIME to update soft windows using temporal constraint propagation.
2. Call INCREASE-WORK-PROFILE which updates the work profiles (W) accordingly. Work profiles represent resource usage as determined by scheduled activities and will be compared with resource capacities to determine where activities can be feasibly scheduled.

PLACE-ACTIVITY(A_i,t,W)
1 SET-START-TIME(A_i,t)
2 INCREASE-WORK-PROFILE(A_i,t,W)

Similarly, when a start time is retracted (as we do in bulldozing), the UNSET-START-TIME-WITH-CYCLES(A_i,W) method decreases the work profiles (W) according to the scheduled start of A_i and expands windows if possible:

UNPLACE-ACTIVITY-WITH-CYCLES(A_i,W)
1 DECREASE-WORK-PROFILE(A_i,$start_{A_i}$,W)
2 UNSET-START-TIME-WITH-CYCLES(A_i)

While the temporal propagation of window maintenance considers only precedence constraints, the propagation of resource constraints can further reduce start time domains [63, 71]. While some resource constraint propagation is likely beneficial, it comes at a computational cost and limits the scalability of algorithms. Also, studies have shown that different problem characteristics seriously affect the effectiveness of any given technique [7].

The algorithms we describe in this chapter explicitly avoid resource constraint propagation as our intent is to show that search can be effective without it and to highlight the differences between our algorithm and others that have been proposed. However, there is no inherent reason for us to avoid resource constraint propagation and it could complement our approach.

4.3.2 Squeaky Wheel Optimization

Squeaky Wheel Optimization (SWO) is an iterative approach to optimization that combines a greedy algorithm with a priority scheme [54]. A number of results suggest that SWO can be applied effectively to a range of real-world problems and scales well [24, 54].

Each iteration of the SWO algorithm can be divided into two stages: construction and prioritization.[4] The construction stage takes a variable ordering and builds a solution using a greedy algorithm. In the prioritization stage, a variable is penalized with 'blame' depending on how well that variable was handled during construction. The updated priorities result in a new variable ordering for the next iteration.

The key to SWO is that elements that are handled poorly by the greedy constructor have their priority increased and are therefore handled sooner (and hopefully better) the next time ("The squeaky wheel gets the grease"). Over time, elements that are difficult to handle drift toward the top of the queue, those that are always easy to handle drift toward the bottom, and the rest settle somewhere in between.

A SWO implementation using a time window framework for the RCPSP/max problem, embedded in a search over possible horizons, is outlined below.

Line 3 and lines 14 through 16 perform something similar to binary search over possible makespans, ranging from the resource-unconstrained lower bound to double the trivial upper bound.[5] Although not included in the pseudocode, our algorithm will quit if the minimum makespan is reached.

Line 4 initializes the priorities. Lines 7 to 11 make up the greedy schedule constructor and priorities are updated on line 17.

---

[4] It has been described as a three step process by Joslin and Clements [54]. However, their analysis and prioritization stages can be combined for our purposes.

[5] $ub = \Sigma_{i=1}^{n} max(d_i, max(T_{i,j}^{min}))$ [33]. We found that doubling this enabled bulldozing to find feasible schedules more easily.

```
SWO(max-iter)
 1  counter ← 1
 2  MK_best ← +∞
 3  SET-HORIZON-LOWER-BOUND
 4  (P_1, ..., P_n) = initial priorities
 5  for counter ← 1 to max-iter
 6        do feasible ← TRUE
 7           for i ← 1 to n
 8                 do
 9                    A_i ← unscheduled activity with highest P_i
10                    if SCHEDULE(A_i) fails
11                       then feasible ← FALSE
12           if feasible
13              then MK_best ← MK_current
14                   DECREASE-HORIZON
15           if 10 iterations without feasible solution
16              then INCREASE-HORIZON
17           (P_1, ..., P_n) = updated priorities
```

The basic SCHEDULE(A_i) method is outlined below. It looks for the earliest time-feasible and resource-feasible place to start A_i. If none is found, resource constraints are ignored and A_i is put at the earliest time-feasible start time ses_{A_i}.

Notice that we explicitly continue even when SCHEDULE(A_i) fails. The reason is that there may be many activities that prove difficult to schedule in any one iteration; we want to be able to blame (and reprioritize) all of them rather than only the first one that fails. While the schedule is no longer resource-feasible once SCHEDULE fails, it is kept time-feasible.

```
SCHEDULE(A_i)
1  t ← earliest resource-feasible time for A_i
          in [ses_{A_i}, sls_{A_i}]
2  if t = DEFAULT-TIME
3     then PLACE-ACTIVITY(A_i, ses_{A_i}, W)
4          return FALSE
5     else PLACE-ACTIVITY(A_i, t, W)
6          return TRUE
```

Example 4.3.1. *Consider the following three activity, one resource problem:*

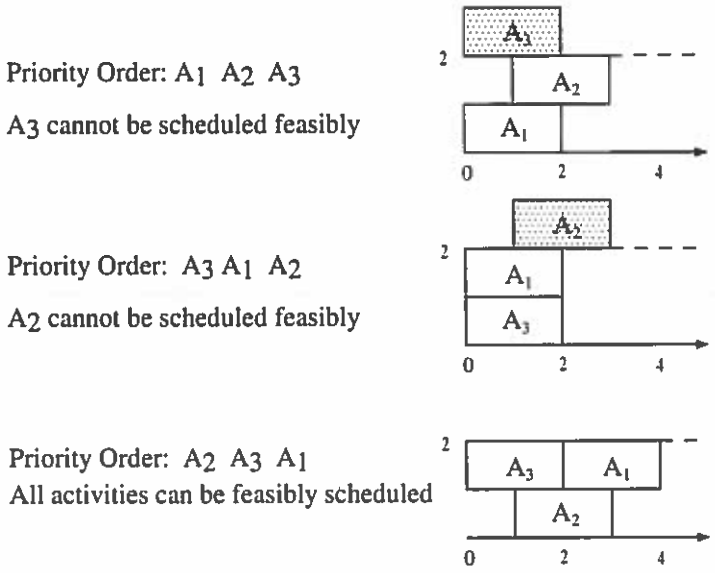


FIGURE 4.1: The third iteration of SWO finds a feasible schedule.

- *Durations* $d_1 = d_2 = d_3 = 2$
- *Resource capacity* $r_1 = 2$
- *Resource requirements* $r_{11} = r_{21} = r_{31} = 1$
- *A constraint that* A_3 *must start exactly 1 unit before* A_2. *That is,* $S_3 - S_2 = -1$ *(or* $T^{min}_{2,3} = T^{max}_{2,3} = -1$*).*

Figure 4.1 demonstrates SWO *beginning with the priority order* (A_1, A_2, A_3). *Using this order, no resource-feasible time is available for* A_3 *after* A_1 *and* A_2 *have been scheduled, so its priority is increased and the priority order* (A_3, A_1, A_2) *is obtained.*[6] *On the second iteration,* A_2 *has the same problem. Finally, in the third iteration, the order* (A_2, A_3, A_1) *leads to a feasible schedule with* $MK = 4$. SWO *will then reduce the horizon to 3, recompute time windows and continue.*[7]

For initial priorities, we have chosen to use the hls_{A_i} values calculated by the

[6]For illustration, an activity is moved to the beginning of the priority queue when it cannot be scheduled feasibly.

[7]Notice that this schedule is optimal. However, with only temporal constraint propagation, a schedule of length 3 cannot be ruled out and SWO will try to find one.

temporal constraint propagation (where the activity with the earliest hls_{A_i} value gets the highest priority).

In our implementation, the priorities of activities that are not scheduled feasibly are increased by a constant amount. To add randomness, priorities of other activities are increased by a smaller amount with a small probability. Since binary search ensures that the horizon is less than the best makespan found, we know that on each iteration either a new best schedule will be found or at least one activity will have its priority changed.

4.3.3 Bulldozing

In Example 4.3.1, we saw how SWO can find a feasible schedule by getting the three activities in the right priority order. For large or highly constrained problems, there can be many sub-problems of this nature and it may be difficult for SWO to get all of them right at the same time.

To avoid this potential problem and strengthen the greedy construction, we add a conflict resolution method called bulldozing to the SCHEDULE method. While we use bulldozing to resolve conflicts in this chapter, we will show in Chapter 6 that it can also be used to enhance window-based local search.

In the RCPSP/max context, bulldozing is inspired by the complication of maximum time lags. Without maximum time lags, a greedy algorithm can easily find a feasible schedule since each A_i can be postponed as long as necessary until resources are available. When there are maximum time lags, however, A_i may not be able to be postponed long enough if activities that constrain it have been placed 'too early' (consider the first iteration of Example 4.3.1 where A_3 cannot be postponed long enough since it is constrained by A_2).

If such an A_i is encountered, bulldozing will attempt to delay the activities that constrain it so that A_i can be scheduled in the postponed position.

Example 4.3.2. *Consider again the simple problem of Example 4.3.1. Figure 4.2 shows what happens if we add bulldozing to the constructor. Activities A_1 and A_2 are placed successfully. When A_3 is considered there is no feasible start time. Times*

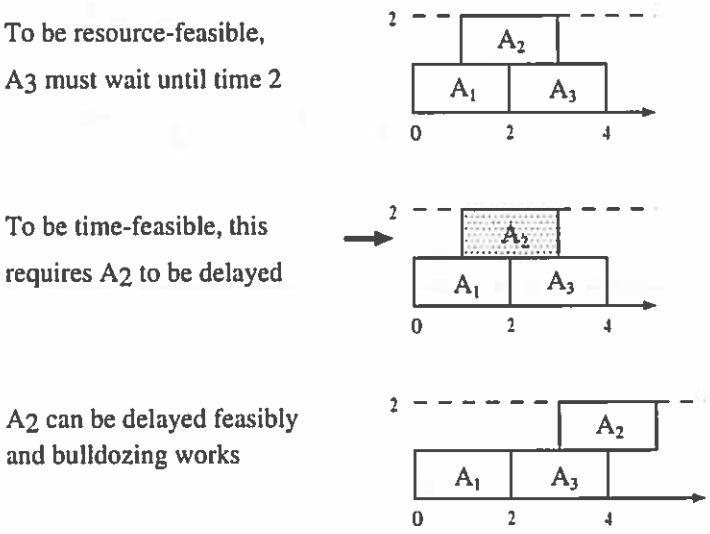


FIGURE 4.2: Bulldozing.

$t \geq 2$ are resource-feasible but only time $t = 0$ is time-feasible since A_3 is constrained by the chosen start time for A_2. Therefore, we place A_3 at 2 and bulldoze A_2. Since A_2 can then be feasibly scheduled at time 3, bulldozing is successful and the resulting schedule is feasible.

We get algorithm SWO(B) by replacing SCHEDULE(A_i) with SCHEDULE-WITH-DOZING(A_i, X), a recursive procedure outlined below (X is the set of activities that must be delayed; in the initial call, $X = \emptyset$). Instead of simply searching the current window $[ses_{A_i}, sls_{A_i}]$ for a feasible time, SCHEDULE-WITH-DOZING considers the larger interval $[ses_{A_i}, hls_{A_i}]$ (recall that hls_{A_i} is the latest start of A_i when no other activities are scheduled). If t is not a feasible time or is before sls_{A_i}, the algorithm proceeds as before.

If the earliest resource-feasible time is outside the current window but within the original window, bulldozing is invoked. Activity A_i is placed at this potential start time and line 9 updates set X with the activities that must be moved.[8] Each A_j in this set is then unplaced (one at a time) and a recursive SCHEDULE-WITH-DOZING(A_j,X) call is made. If all activities can find new feasible start times, bulldozing succeeds.

[8]These will be activities that forced $sls_{A_i} < t$ in the first place.

Otherwise, all delayed activities revert to their previous positions and A_i is placed back at ses_{A_i}.

```
SCHEDULE-WITH-DOZING(A_i, X)
 1  t ← earliest resource-feasible time for A_i
          in [ses_{A_i}, hls_{A_i}]
 2  if t = DEFAULT-TIME
 3     then PLACE-ACTIVITY(A_i, ses_{A_i}, W)
 4          return FALSE
 5     else if t ≤ sls_{A_i}
 6             then PLACE-ACTIVITY(A_i, t, W)
 7                  return TRUE
 8             else PLACE-ACTIVITY(A_i, t, W)
 9                  X ← X ∪ {A_j forced to move by A_i}
10  while X ≠ ∅
11        do A_k ← randomly selected element of X
12           UNPLACE-ACTIVITY-WITH-CYCLES(A_k, W)
13           if SCHEDULE-WITH-DOZING(A_k, X) fails
14             then UNDO-ALL-BULLDOZING
15                  UNPLACE-ACTIVITY-WITH-CYCLES(A_i, W)
16                  PLACE-ACTIVITY(A_i, ses_{A_i}, W)
17                  return FALSE
```

The recursive nature of bulldozing means that more activities can be moved than the original set forced by A_i. In fact, we have observed that A_i itself is often bulldozed further in attempts to settle on start times where all activities are resource-feasible. The reason seems to be that a highly constrained sub-problem may need to be moved out past other activities to fit. This is the motivation for selecting activities to bulldoze randomly on line 11; if a subset of activities proves difficult to schedule, attempts to reschedule them will be done in different orders.

This suggests a side benefit of bulldozing. If a problem has subsets of activities that are highly constrained, the subsets will be pushed out past other activities until they can be scheduled feasibly. It is interesting to note that in other work [74], RCPSP/max problems are explicitly divided into such sub-problems that are solved separately and then combined into an overall solution. Bulldozing similarly isolates subproblems from the rest of the problem but does so only when subproblems prove difficult.

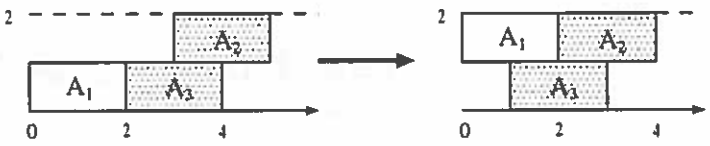


FIGURE 4.3: With refilling, A_2 and A_3 are bulldozed left and the makespan is reduced from 5 units to 4.

If there is a difficult subproblem, SWO without bulldozing may also work because the activities involved are likely to move up in the priority queue and be scheduled before others. However, if there are multiple such subproblems, they will likely be jumbled together in the early part of the priority queue and SWO may have trouble fitting them all together. Bulldozing appears to overcome this problem.

Bulldozing can be considered a form of intelligent backtracking [44] since we unvalue and revalue the variables whose values contribute to an infeasible state. It also has the flavor of iterative repair algorithms that build schedules and then eliminate conflicts through local search. However, a key difference with local repair is that bulldozing is done with partial rather than complete schedules.

4.3.4 Refilling

Consider our simple example once more. In Example 4.3.2, we saw how bulldozing can allow feasible schedules to be constructed with priority queues that result in infeasible schedules without bulldozing. However, notice that the resulting schedule is not optimal. As shown in Figure 4.3, the two activities involved in bulldozing (A_2 and A_3) can be shifted left to fill in the space vacated by the bulldoze.

Algorithm SWO(B,R) results from adding two such 'refilling' pieces to the procedure SCHEDULE-WITH-DOZING:

1. **Left Bulldozing** After a successful bulldoze, an attempt is made to bulldoze the same set of activities back to earlier start times. This often works because of the space vacated by the bulldozed activities. This step is also bulldozing because the subset of activities considered may be highly constrained among themselves (hence the bulldoze in the first place). Therefore, it may not be possible to left-shift any of them individually but they may move as a group.

2. **Left Shifting** After a successful bulldoze, an attempt is made to left-shift activities that can take advantage of resources vacated by bulldozed activities. We simply consider each activity A_i for which $start_{A_i} > ses_{A_i}$.

These refilling mechanisms are able to reduce the makespan of an already feasible schedule or partial schedule. When the horizon is small, they may also indirectly help the construction of more feasible schedules since they leave more room for activities that have yet to be scheduled.

Example 4.3.3. *In the simple problem of Example 4.3.1, there are 6 possible priority orders. For* SWO, *2 of them are both feasible and optimal. For* SWO(B), *all 6 become feasible and 4 are optimal. Finally, for* SWO(B,R), *all 6 are optimal.*

In Figure 4.4 the results of using the three constructors for each priority order are shown. Each row represents a priority order. The first column shows the results of SWO *using that priority order. The second column shows the results of using* SWO(B), *if bulldozing is invoked. Finally, the third column shows the results of using* SWO(B,R) *in the cases where that makes a difference.*[9]

It is straightforward to show that each version of our algorithm is guaranteed to terminate in polynomial time and therefore cannot be guaranteed to find a feasible solution (since the problem is NP-hard).

4.4 Experimental Results

We have tested our algorithms on five sets of benchmark problems (set A is divided into 3 subsets). All benchmarks were generated with ProGen/max [87] using a number of parameters to vary problem characteristics. Table 4.1 lists the benchmarks with

[9]For the two priority orders where refilling changes the schedule, the refilling required is the left bulldozing we have described. It should be noted, however, that the current implementation of SWO(B,R) will not actually succeed in these two cases. The problem is that it tries to bulldoze A_3 back to time 0, discovers that this does not work and then gives up. To fix this would require multiple attempts at left bulldozing (to try different positions for A_3); we suspect this would not be worth the extra computation time. Nonetheless, for the purposes of demonstration, we assume that the refilling is figured out.

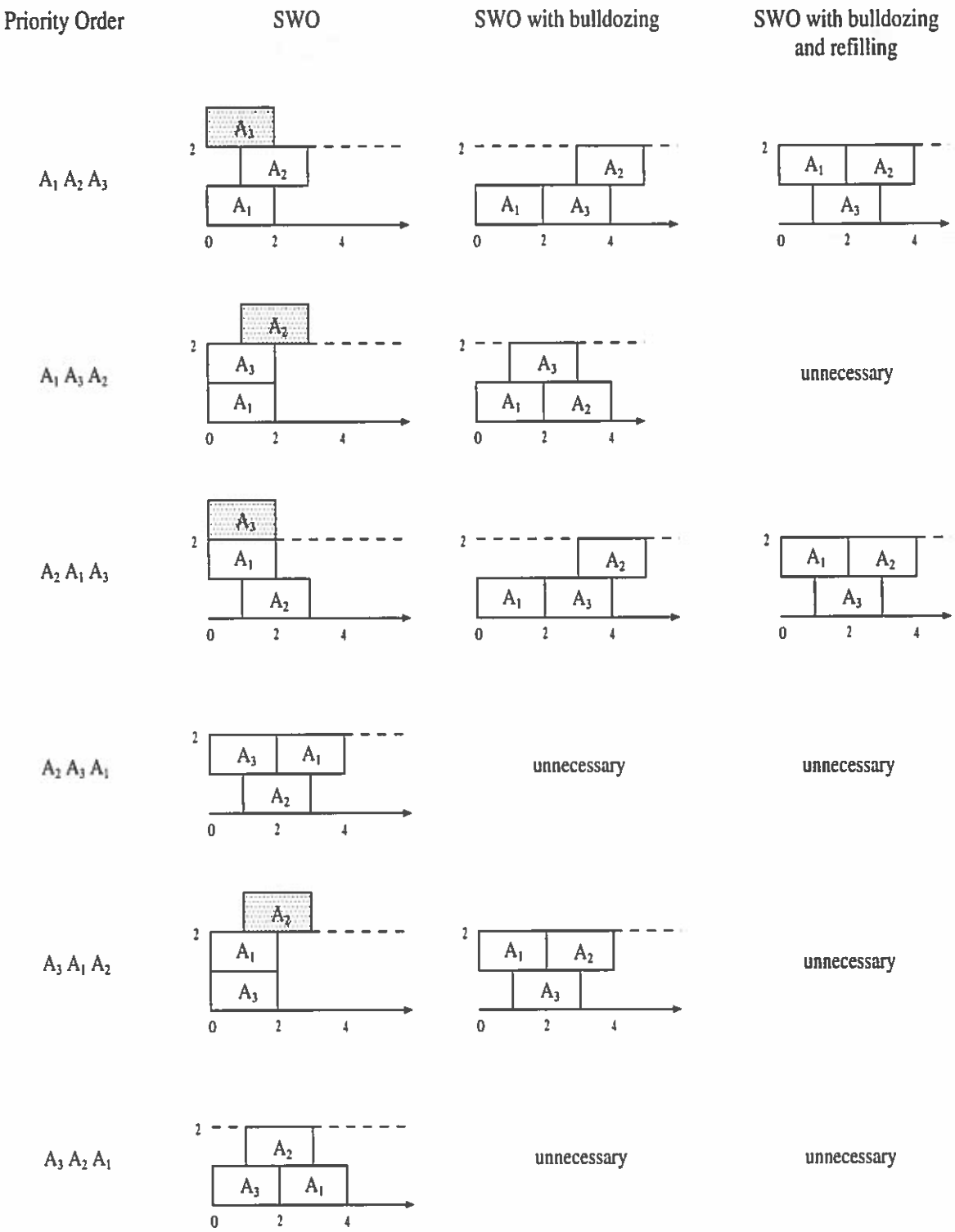


FIGURE 4.4: The results of using the three constructors with each possible priority order as described in Example 4.3.3.

TABLE 4.1: Benchmark names and characteristics.

Set	$N_{instances}$	N_{feas}	N_{act}	N_{res}
J10	270	187	10	5
J20	270	184	20	5
J30	270	185	30	5
B	1080	1059	100	5
C	120	119	500	5

TABLE 4.2: Results of schedule construction with all priority queues for feasible *J*10 problems.

	$\%_{feas}$			$\%_{opt}$		
Constructor	Worst	Avg.	Best	Worst	Avg.	Best
SWO	0	4	54	0	2	53
SWO(B)	18	94	100	0.0002	7	100
SWO(B,R)	15	94	100	0.02	13	100

the number of instances and how many are feasible as well as the number of activities and resources of each instance.

Our algorithm has been implemented in C++ and was run on a 1700 MHz Pentium 4. It uses well under 15 MB of memory on the largest problems.

4.4.1 Quality of the Greedy Constructor

For a SWO algorithm to be effective, it is crucial that the greedy constructor be capable of finding feasible and optimal solutions. We want to be confident that there are at least some (preferably many) priority orders from which optimal schedules will be produced.[10] The *J10* problems are small enough that all 10! = 3628800 possible priority queue permutations can be tried for each instance.

In Table 4.2, we show the percentage of those queues that lead to feasible and optimal schedules using the constructors of our three SWO algorithms: SWO, SWO(B) (bulldozing) and SWO(B,R) (bulldozing and refilling). For both $\%_{opt}$ and $\%_{feas}$, we give the worst and best cases among all 187 feasible problems in *J10* as well as the

[10]It is not obvious that this is the case with maximum time lags.

average.

The benefits of strengthening the constructor are clear; bulldozing yields a jump in the number of feasible schedules produced while refilling yields another jump in the number of optimal schedules produced. While the SWO constructor fails to find any solution to 3 feasible problems, the other two find feasible solutions with at least 15% of the prioritizations for every feasible problem.[11]

Another consideration, of course, is time. On average, the SWO(B) constructor took 3.2 times longer than that of SWO and the SWO(B,R) constructor took twice again as long. Experiments suggest that these computational costs are worth paying.

4.4.2 Benchmark Results

We compare SWO(B,R) with the best methods reported in the literature:

- **B&B$_{S98}$** [88] is a branch and bound algorithm that delays activities by adding precedence constraints.
- **B&B$_{D98}$** [33] is a branch and bound algorithm that reduces the search space with a significant amount of resource constraint propagation.
- **B&B$_{F99}$** [35] is a branch and bound algorithm that dynamically increases the release dates of some activities.
- **ISES** [23] is a heuristic algorithm that begins with all activities scheduled as early as possible and then iteratively finds and levels 'resource contention peaks', by imposing additional precedence constraints and restarting.
- **KDE** [28] improves upon priority rule methods by using models of search performance to guide heuristic-based iterative sampling.

For each of the above, results are available for a subset of the benchmark problems. The results from a number of algorithms that do less well than the above five algorithms are not reported. These include a genetic algorithm, Tabu search and simulated annealing [40], and priority rule-based methods [38].

[11] It is unclear to us why the SWO(B,R) constructor did worse than the SWO(B) constructor in the worst case.

For SWO(B,R), we report the average of 5 runs for each benchmark set. Each run was capped at 10 seconds (or 1000 iterations, whichever came first) in an attempt to ensure that we used no more computation time than other algorithms.

Results are summarized in table 4.3 where we include the following statistics:

- $\Delta_{\mathbf{LB}}\%$ gives the average relative deviation from the known lower bounds.
- $\%_{\mathbf{opt}}$ gives the percentage of feasible problems for which optimal solutions were found (solutions that matched the lower bound or have been proven optimal by some algorithm).
- $\%_{\mathbf{feas}}$ gives the percentage of feasible problems for which feasible solutions were found.
- $\mathbf{Cpu_{sec}}$ gives the average computation time, scaled by processor speed.[12]

Algorithm SWO(B,R) was able to find solutions to all 1733 feasible problems (on all 5 runs). The only other algorithm that does not miss feasible solutions is $B\&B_{F99}$, which does so at the expense of quality, especially on set C.

Not surprisingly, exact methods get the best results for small problems. $B\&B_{S98}$ is able to solve all *J10* problems optimally, does fairly well on the *J20* problems but already fails to find a number of feasible solutions to some of the problems in *J30*. $B\&B_{D98}$ is better able to handle larger problems and is by far the best algorithm on problem set B. However, when the number of activities is increased from 100 to 500 (set C), it fails to find all feasible solutions and solves fewer to optimality than SWO(B,R).

SWO(B,R) is also very competitive with the best non-systematic algorithms. It outperforms *ISES* on all metrics and on all problem sets. While *KDE* is significantly better on measurement $\Delta_{\mathbf{LB}}\%$, SWO(B,R) is slightly better than *KDE* on the other three dimensions.

Finally, in addition to scaling well in terms of relative solution quality, SWO(B,R) appears to scale effectively in terms of running time. For example, while *ISES* takes 33

[12]For example, the times reported for *ISES* have been reduced by a factor of 1700/200 since they were obtained on a Pentium 200 machine. This rough comparison is the most we can hope for given that the algorithms were developed in different languages and run on different platforms. For some systematic algorithms, this time also includes the time to prove optimality.

TABLE 4.3: Results for benchmark problems.

Set	Algorithm	$\Delta_{LB}\%$	$\%_{opt}$	$\%_{feas}$	Scaled Cpu_{sec}
J10	SWO(B,R)	0.2	94.0	100	0.31
	$B\&B_{S98}$	0.0	100	100	-
	ISES	1.3[a]	85.9	99.5	0.08
J20	SWO(B,R)	4.9	66.4	100	0.63
	$B\&B_{S98}$	4.3	85.3	100	-
	ISES	5.4	64	100	0.53
J30	SWO(B,R)	10.3	51.1	100	1.07
	$B\&B_{S98}$	9.6[a]	62.3	98.9	-
	ISES	11.0	49.4	100	2.67
B	SWO(B,R)	6.8	64.2	100	1.85
	KDE	4.6[a]	63.8	99.9	1.97
	ISES	8.0[a]	63.2	99.9	[b]
	$B\&B_{S98}$	9.6	63.7	100	-
	$B\&B_{D98}$	4.6	73.1	100	3.49
	$B\&B_{F99}$	7.0	72.5	100	[b]
C	SWO(B,R)	0.5	79.2	100	3.27
	$B\&B_{D98}$	0.5[a]	71.4	97.5	11.53
	$B\&B_{F99}$	5.2	58.8	100	[c]

(a) Not directly comparable to other numbers since problems not feasibly solved are excluded.
(b) Cutoff of 11.8 seconds used but no average time reported.
(c) Cutoff of 23.6 seconds used but no average time reported.

times longer on *J30* than on *J10*, SWO(B,R) takes less than 4 times longer.[13] Similarly, $B\&B_{D98}$ takes 3.3 times longer on *C* than on *B*; the difference for SWO(B,R) is under 2. It is unclear how *KDE* scales since results are only reported for problems of a single size.

4.4.3 Generalizing SWO(B,R)

As described in Section 4.3.1, our implementation of SWO takes advantage of the simplified nature of benchmark problems. However, a primary goal of this dissertation is to develop algorithms that can handle real-world problems. Therefore, we have also implemented SWO(B,R,G) (where G represents the fact that it is generalized) which differs from SWO(B,R) in three important ways:

1. Time windows are maintained in SWO(B,R,G) using the GSTP framework of Chapter 3.
2. Resource management is also capable of handling calendar issues. This requires a slightly more complicated implementation of INCREASE-WORK-PROFILE and DECREASE-WORK-PROFILE. For example, if an activity is scheduled over the course of a number of weeks but does not work on weekends, work profiles must be increased during the week but not on weekends.
3. The UNPLACE-ACTIVITY-WITH-CYCLES procedure is replaced with the procedure UNPLACE-ACTIVITY, outlined below. This is the version used to unschedule activities in later chapters and does not account for cycles in the constraint graph. As discussed in Section 3.5.3, the resulting windows may not be complete due to maximum time lags. However, window expansion is much faster in practice and we decided this implementation was more appropriate for real-world problems. The end result will be that there will be times that bulldozing may fail even though it would have succeeded if cycles had been handled properly.

UNPLACE-ACTIVITY(A_i,W)
1 DECREASE-WORK-PROFILE(A_i,$start_{A_i}$,W)
2 UNSET-START-TIME(A_i)

In Table 4.4, we show results for the SWO(B,R,G) version of our algorithm compared with the SWO(B,R) results from Table 4.3.

[13] Of course, such comparisons must be taken with a grain of salt; running times depend partly on the cutoff times chosen.

TABLE 4.4: Comparison of SWO(B,R) and SWO(B,R,G).

Set	Algorithm	$\Delta_{LB}\%$	$\%_{opt}$	$\%_{feas}$	Scaled Cpu_{sec}
J10	SWO(B,R)	0.2	94.0	100	0.31
	SWO(B,R,G)	0.1	96.0	100	0.50
J20	SWO(B,R)	4.9	66.4	100	0.63
	SWO(B,R,G)	5.0	65.3	100	1.24
J30	SWO(B,R)	10.3	51.1	100	1.07
	SWO(B,R,G)	10.4	51.1	100	2.39
B	SWO(B,R)	6.8	64.2	100	1.85
	SWO(B,R,G)	7.2	63.1	100	2.68
C	SWO(B,R)	0.5	79.2	100	3.27
	SWO(B,R,G)	0.7	77.1	99.7	3.62

As expected, running times for SWO(B,R,G) are somewhat longer (up to twice as long) than for SWO(B,R). Results are comparable for most problem sets but SWO(B,R,G) has some trouble with set *C* and does not solve all problems feasibly; there is one problem that it solves on only three of the five runs. We suspect this is due both to the slower running time (the 10 second cutoff time is reached most often for set *C*) and the fact that windows aren't kept complete during bulldozing.

4.5 Conclusions

We have described a new heuristic algorithm, SWO(B,R), for RCPSP/max problems. It uses bulldozing and refilling to improve the performance of greedy schedule construction in SWO. On a range of benchmarks SWO(B,R) is competitive with systematic and non-systematic state-of-the-art techniques. Able to consistently solve all feasible problems, it scales well, both in terms of solution quality and running time, relative to the best OR and AI algorithms known.

There are several ways SWO(B,R) could be improved. We have already mentioned that some resource constraint propagation should help guide SWO. Preliminary experiments using one of the consistency tests of Dorndorf et al. [33] give improvements for *J10* and *J20*. However, the benefits seem to be outweighed by the extra

computational costs for the larger problem sizes.

It might be effective to incorporate SWO(B,R) into a meta-heuristic like Tabu search[14] or simulated annealing so that priority space is explored more intelligently; the current search is relatively unstructured.

Finally, SWO(B,R) should be tested on more difficult problems. While it didn't struggle with the above benchmarks, we do not know how it will do on problems that are larger or more difficult.

[14]Tabu search was found to be an effective meta-heuristic to combine with SWO to solve the container ship yard allocation problem [24].

CHAPTER 5

Minimizing Labor Costs

5.1 Introduction

Historically, scheduling research has addressed factory-like settings where the relevant resources are machines. For example, the job-shop scheduling variants that have received so much attention over the years explicitly refer to operations performed by machines. In these settings it makes sense to solve resource constrained scheduling problems where the resource capacities (the number of machines, for example) are fixed and the goal is to optimize a criterion such as makespan without exceeding those capacities.

There are many domains, however, where the relevant resources are people rather than machines. For example, in construction, resources like electricians, plumbers and painters are of more cost and concern than resources like wires, pipes, ladders or paint. In fact, project management systems are designed with the expectation that resources will often be people, allowing the user to specify such details as job titles, skills and overtime costs.

When labor is the important resource, it may not make sense for the resource capacity (staffing level) to be fixed. Instead, the staffing level can likely be increased or decreased as necessary (at some cost) to match demand.

In this chapter, we consider the labor cost optimization problem where the goal is to find a schedule that minimizes total labor costs subject to the problem constraints.

To solve the problem, we must distinguish between the work profile (the amount of work scheduled) and the staffing profile (the number of workers on staff to do the work). These profiles will match when there is just enough work for all workers to be working full time. Often, they will not match: a higher work profile corresponds to overtime (workers doing extra work at a higher cost) while a lower work profile corresponds to undertime (paid workers with nothing to do). Costs will be associated with overtime and undertime as well as with changes in the staffing profile (corresponding to hires or fires).

To simplify discussion throughout this chapter, we assume that a project has a single resource. There is no resulting loss of generality; when there are multiple resources, each one can (and should) be handled separately as discussed here.[1]

We also often refer to time units as days since time units are days in most projects we have seen and days make the most sense intuitively. However, the theory presented is applicable to any time granularity.

In Section 5.2, we show how to calculate labor costs given both a work profile and a staffing profile. The notation introduced there helps us formally define the LCOP in Section 5.3. In Section 5.4, we show how dynamic programming can be used to find an optimal staffing profile given only the work profile. Sections 5.5 and 5.6 show how the properties of the problem can be used to significantly speed up the dynamic programming. In Section 5.7, we address the final goal: optimizing the work profile (schedule) in order to get a minimum labor cost. Other issues are discussed in Section 5.8, and Section 5.9 concludes with a review of related work.

It is important to mention here that most of the algorithms described in this chapter were developed by Andrew Baker. My contributions are a formal description of those algorithms, proof that they have the desired properties, and results concerning their time and space complexity. In addition, the observations of Section 5.6 are my own.

---

[1] An important exception to this is the case of cross-trades mentioned in Section 5.8.2.

5.2 Labor Costs Given Staffing Levels

In this section, we show how we can calculate total labor costs for a project if we are given both a work profile (determined by the schedule) and a staffing profile (determined by staffing decisions).

Labor costs can be calculated given the following five resource rates:[2]

- **Base Rate** (B) is the amount paid to workers to complete scheduled work. This might include wages and any employee benefits but only for time during which work is accomplished.
- **Overtime Rate** (V): is the additional amount paid for overtime work.
- **Undertime Rate** (U) is the amount paid to workers for time during which they are employed but are not accomplishing work. Usually, $U = B$.
- **Hire Rate** (H) is the expense of adding a worker to the staff. This might include recruiting and training costs.
- **Fire Rate** (F) is the cost to decrease the number of workers. This might include severance pay.

Suppose that on some day we have work level w (the amount of work to be done). Let the change in staffing level relative to the previous day be *staff-change*; a negative value means we have fired workers from the previous day. Let the number of extra workers be *staff-excess*; a negative value means there is more work to be done than there are workers.

We can break the labor cost of the day into the following five pieces:

- **Base Cost**: $b(w) = B \cdot w$.
- **Overtime Cost**: $v(\textit{staff-excess}) = V \cdot max(0, -\textit{staff-excess})$.
- **Undertime Cost**: $u(\textit{staff-excess}) = U \cdot max(0, \textit{staff-excess})$.
- **Hire Cost**: $h(\textit{staff-change}) = H \cdot max(0, \textit{staff-change})$.
- **Fire Cost**: $f(\textit{staff-change}) = F \cdot max(0, -\textit{staff-change})$.

[2]In what follows, we assume they are all non-negative.

Finally, we can pair up the two costs associated with *staff-excess* and the two costs associated with *staff-change* as follows:

- **Staff Offset Costs**: $\Delta_{vu}(\textit{staff-excess}) = v(\textit{staff-excess}) + u(\textit{staff-excess})$. The costs incurred for a difference between the work to do and the number of people on staff to do it.
- **Staff Change Costs**: $\Delta_{hf}(\textit{staff-change}) = h(\textit{staff-change}) + f(\textit{staff-change})$. The costs incurred for a change in staffing levels.

Now suppose that we have a schedule spanning m days. Let the work profile (amount to be done by day) be $W = (w_1, w_2, ..., w_m)$ and the staffing profile (number of people by day) be $P = (p_1, p_2, ..., p_m)$. In addition we need a starting staffing level p_0 to represent the staffing level on the day before the project begins; this will be needed to calculate the hire and fire costs for the first day.

The total project cost is then:

$$c_{total} = \sum_{t=1}^{m} b(w_t) + \Delta_{vu}(p_t - w_t) + \Delta_{hf}(p_t - p_{t-1}) \tag{5.1}$$

Example 5.2.1. *Consider a three day project spanning Tuesday through Thursday. Let the work to be done be* $W = (2, 3, 0)$ *and suppose we have staffing levels* $P = (1, 2, 1)$ *and let the staffing level on Monday be* $p_0 = 1$.

Table 5.1 shows how we arrive at a total project cost of 8 using equation (5.1) *with rates* $B = U = 1$ *and* $V = H = F = 0.5$. *The first three rows show the values used to calculate costs. The next 5 rows break down the costs and the last row shows the total cost.*

For example, on Tuesday there are 2 units of work to do which results in a base cost of 2. Since there is only 1 worker, staff-excess $= -1$, *resulting in an overtime cost of 0.5 and no undertime cost. Since there was also 1 worker on Monday, staff-change* $=$ 0 *and there is no hire or fire cost.*

The final column shows the totals for the whole project.

Notice that the base cost depends only on the amount of work and not on the number of people available. Unlike the other costs, the total base cost of a project

TABLE 5.1: Calculation of project cost for Example 5.2.1 using (5.1).

Cost	Tuesday	Wednesday	Thursday	Total
w_t	2	3	0	5
$staff\text{-}excess = p_t - w_t$	-1	-1	1	-
$staff\text{-}change = p_t - p_{t-1}$	0	1	-1	-
$b(w)$	2	3	0	5
$v(staff\text{-}excess)$	0.5	0.5	0	1
$u(staff\text{-}excess)$	0	0	1	1
$h(staff\text{-}change)$	0	0.5	0	0.5
$f(staff\text{-}change)$	0	0	0.5	0.5
c_{total}	2.5	4	1.5	8

corresponds only to the total amount of work and is unaffected by how that work is scheduled or staffed. This can be called the 'value-added' cost.

The other costs, however, depend on the work and staffing profiles and are accrued either when the two are different or when the staffing profile fluctuates. These can therefore be considered unnecessary costs; for an ideal project, with constant work and constant staff profile, they will all be zero and the only cost will be the base cost.

5.3 The LCOP

We are now ready to formally define the LCOP:

Definition 11. *A **labor cost optimization problem** (LCOP) contains a set $A = \{A_1, ..., A_n\}$ of activities and a set $R = \{R_1, ..., R_m\}$ of resources (pools of workers or trades). Each activity A_i has a start time $start_{A_i}$ and a duration d_i and there is an overall deadline D. Execution of A_i requires an amount r_{ikt} of resource k for each time unit t of execution ($start_{A_i} \leq t < start_{A_i} + d_i$).*[3]

Binary precedence constraints $[T_{i,j}^{min}, T_{i,j}^{max}]$ enforce minimum ($T_{i,j}^{min}$) and maximum ($T_{i,j}^{max}$) time lags between the the start times of activities A_i and A_j.

[3]For all of the problems we consider, the resource requirements of each A_i are constant for its duration. However, this need not be the case and all of our algorithms work for this more general definition.

The objective is to find a feasible schedule S with the lowest minimum cost. In other words, the objective is to find S that determines work profile W such that the optimal staffing profile P for W minimizes c_{total} (see (5.1)).

The next three sections describe how to find the minimum cost (and an optimal staffing profile P that gives that cost) given W. Section 5.7 shows how these methods can be incorporated into search for cost effective schedules. In Chapter 6, we describe the ARGOS tool that uses these algorithms to heuristically solve LCOP problems.

5.4 Using Dynamic Programming to Minimize Labor Costs

Above, we showed how to compute labor costs given both a work profile and a staffing profile. We now want to consider the optimization problem of minimizing the total labor cost c_{total} given only the work profile.

Standard dynamic programming can be used. Let p_{max} be the maximum staffing level that might be required during a project.[4] Function $minCost_t(p)$ recursively computes the minimum cost of having p workers for day t based on the values of $minCost_{t-1}(p')$ for all possible staffing levels p' on the previous day. For all p, we define $minCost_0(p) = 0$ to represent the fact that we can have any staffing level on the day before the project begins at no cost.[5] Our recursive function is then:

$$minCost_t(p) = \min_{0 \le p' \le p_{max}} (minCost_{t-1}(p') + \Delta_{hf}(p - p')) + b(w_t) + \Delta_{vu}(p - w_t) \quad (5.2)$$

To compute the overall minimum cost, we simply find staffing level p on the last day m of the project for which $minCost_m(p)$ is lowest:

$$minCost_{total} = \min_{0 \le p \le p_{max}} (minCost_m(p)) \quad (5.3)$$

---

[4]Given a work profile, we can let $p_{max} = \lceil max\{w_t\} \rceil$ since it will never be worth having more staff than that.

[5]In Section 5.8, we show how to handle cases where this is not true.

Consider equation (5.2) again. We can break down the calculation into two pieces. The first piece (in parentheses) calculates the minimum cost to begin day t with p workers by considering the costs of having all of the possible staff levels p' on the previous day and adding the cost to change the staffing level from p' to p.

The second piece of (5.2) adds the new costs for work on day t and has nothing to do with previous days. From now on, we discuss the calculation in terms of these two pieces: one to calculate the minimum starting costs for day t and the other to add the new costs for t.

The results of (5.3) can be achieved by the non-recursive procedure FIND-MIN-COST that follows.

```
FIND-MIN-COST(W)
1  total-min-costs = (0, 0, ...., 0)
2  for t = 1 to m
3        do
4           for p = 0 to p_max
5                 do start-min-costs[p] = FIND-STARTING-MIN(p, total-min-costs)
6                    new-costs[p] = b(w_t) + Δ_vu(p − w_t)
7           total-min-costs = start-min-costs + new-costs
8  return minimum value in array total-min-costs
```

On line 1, FIND-MIN-COST sets the cost of each staffing level to 0 for time 0 (recall that we are allowed to begin with any staffing level at no charge). Then, for each day t, the FIND-STARTING-MIN procedure calculates the minimum starting costs. The new costs for t are calculated on line 6. Finally, the two pieces are summed on line 7. The overall minimum cost is the smallest entry in the array *total-min-costs*$[p]$.

The FIND-STARTING-MIN procedure is outlined below. It is given the minimum total costs at each staffing level for day $t-1$. Using these, it determines the cheapest way to end up with p workers on day t.

```
FIND-STARTING-MIN(p, total-min-costs)
1  answer = ∞
2  for p' = 0 to p_max
3        do this-cost = Δ_hf(p − p') + total-min-costs[p']
4           if this-cost < answer
5              then answer = this-cost
6  return answer
```

Example 5.4.1. *Suppose we want to find the optimal cost to complete the small project of Example 5.2.1. Table 5.2 shows the values computed by* FIND-MIN-COST. *There are three columns for each day:*

1. *The first column shows the minimum cost to begin work on the day with p workers (computed by* FIND-STARTING-MIN*).*
2. *The second column shows the new costs for that day*
3. *The third column is the sum of the first two and shows the minimum cost to complete the project up to that day with p workers.*

Consider the costs on Tuesday. Since we assign no cost to starting the project at any staffing level, the costs in the first column are all 0. The new costs for Tuesday are the costs to do 2 units of work at each staffing level p. If $p = 0$, there are 2 units of work (at cost 1) and two units of overtime (at cost 0.5) which gives the new cost of 3.[6] *The lowest new cost is if there are exactly 2 people. If there are 3 people, the cost is 3 (2 for the work and 1 for the undertime).*

Now consider the minimum starting costs for Wednesday. The minimum cost of 3 to start with 0 workers could actually be achieved three ways:

1. *Using Tuesday's cost for 0 workers (cost 3) without changing the staffing level.*
2. *Using Tuesday's cost for 1 worker (cost 2.5) and reducing the staffing level by 1 (cost 0.5).*
3. *Using Tuesday's cost for 2 workers (cost 2) and reducing the staffing level by 2 (cost 1).*

Notice also that the minimum starting cost for 3 workers on Wednesday is cheaper than the cost to have 3 workers on Tuesday. The cost of 2.5 is achieved by using Tuesday's cost for 2 workers (cost 2) and increasing the staffing level by 1.

The final column gives the minimum cost to complete the project with each staffing level. The optimal cost is 6.5 and can be achieved only by ending with 0 workers on Thursday.

---

[6] We assume for now that the work can be accomplished and ignore the fact that there is nobody to do it.

TABLE 5.2: Calculation of optimal cost for the work of Example 5.2.1.

	Tuesday: $w = 2$			Wednesday: $w = 3$			Thursday: $w = 0$		
Worker level p	Min starting cost	New cost	Total cost	Min starting cost	New cost	Total cost	Min starting cost	New cost	Total cost
0	0	3	3	3	4.5	7.5	6.5	0	6.5
1	0	2.5	2.5	2.5	4	6.5	6	1	7
2	0	2	2	2	3.5	5.5	5.5	2	7.5
3	0	3	3	2.5	3	5.5	5.5	3	8.5

It is worth noting here that the dynamic programming can just as easily proceed from time m to time 1 since the computation is symmetric. We take advantage of this in Section 5.7.

Lemma 5.4.1. *The optimal value, $minCost_{total}$, can be calculated in $O(m(p_{max})^2)$ time using $O(p_{max})$ space.*

Proof: The FIND-STARTING-MIN procedure has p_{max} steps to perform and is called p_{max} times by FIND-MIN-COST for m different times. This is the bottleneck and we get $O(m(p_{max})^2)$ time. It requires $O(p_{max})$ space since only three arrays of size p_{max}, *total-min-costs*, *start-min-costs* and *new-costs*, need to be maintained at any given time.

■

5.4.1 Calculating the Optimal Staffing Profile

The procedure FIND-MIN-COST calculates the minimum cost given a schedule but does not record the staffing profile needed to achieve that cost. If we want to save the staffing profile as well, some extra bookkeeping is required.

During dynamic programming, we must maintain a set of arrays (one for each day) $X = (X_1, X_2, ..., X_m)$ where each X_t is an array $X_t = (x_{t,0}, x_{t,1}, ..., x_{t,p_{max}})$. The value

$x_{t,p}$ gives the staffing level on the previous day $(t-1)$ that results in the minimum cost for staffing level p on day t. An $x_{t,p}$ value will be updated whenever line 5 is reached in the FIND-STARTING-MIN procedure.

Given X, we can use the FIND-OPTIMAL-STAFFING-PROFILE procedure to find the optimal staffing profile, where parameter p is the optimal staffing level for the last time unit m (it is straightforward to modify FIND-MIN-COST to return p as well as the minimum cost).

FIND-OPTIMAL-STAFFING-PROFILE steps backward through X. The optimal level on each day t can be calculated by looking at the entry in X corresponding to the optimal level on day $t+1$. Notice that the loop steps down to time 0; this will give the optimal starting staffing level for the day before the project begins.[7]

FIND-OPTIMAL-STAFFING-PROFILE(X, p)
1 *optimal-profile*$[m] = p$
2 **for** $t = m - 1$ **to** 0
3 **do** *optimal-profile*$[t] = x_{t+1,\textit{optimal-profile}[t+1]}$
4 return *optimal-profile*

Lemma 5.4.2. *Finding the optimal staffing profiles can be done in* $O(m(p_{max})^2)$ *time and* $O(mp_{max})$ *space.*

Proof: Computing X during FIND-MIN-COST does not increase the time complexity given in Lemma 5.4.1 and FIND-OPTIMAL-STAFFING-PROFILE is $O(m)$.

The space complexity is increased to $O(mp_{max})$ because X needs to be stored.

■

Example 5.4.2. *In Table 5.3, we update the table of Example 5.4.1 to include the* $x_{t,p}$ *values for calculating the optimal staffing profile.*

Column X_{Tue} *tells us that the optimal way to have 0, 1, 2, or 3 workers on Tuesday is to have the same number on Monday, as we would expect. In Column* X_{Wed}, *we see similar results for 0, 1 or 2 workers (the cheapest way to have that*

[7]This method will find one of possibly many optimal staffing profiles. The one that is found will depend on how ties are broken when X is created.

TABLE 5.3: Table 5.2 extended to allow an optimal staffing profile to be calculated.

	Tuesday: $w = 2$				Wednesday: $w = 3$				Thursday: $w = 0$			
Staffing level p	Min starting cost	X_{Tue}	New cost	Total cost	Min starting cost	X_{Wed}	New cost	Total cost	Min starting cost	X_{Thu}	New cost	Total cost
0	0	0	3	3	3	0	4.5	7.5	6.5	**2**	0	6.5
1	0	1	2.5	2.5	2.5	1	4	6.5	6	2	1	7
2	0	**2**	2	2	2	**2**	3.5	5.5	5.5	2	2	7.5
3	0	3	3	3	2.5	2	3	5.5	5.5	3	3	8.5

many on Wednesday is to have the same number on Tuesday). However, the final values ($X_{Wed}[3]$) indicates that the cheapest way to have 3 workers on Wednesday is to have only 2 on Tuesday (and to therefore hire another to start on Wednesday).

Given the X_i values, we can see how FIND-OPTIMAL-STAFFING-PROFILE *will work. The optimal staffing level on Thursday is 0 (resulting in cost 6.5). To find the optimal staffing level on Wednesday, we look at $X_{Thu}[0]$ (highlighted) which tells us that having 2 workers on Wednesday is the cheapest way to start Thursday with 0 workers (we will fire them after Wednesday). In the next iteration, we look at $X_{Wed}[2]$ (highlighted) which tells us that having 2 workers on Tuesday is the cheapest way to start Wednesday with 2 workers. Similarly, $X_{Tue}[2]$ tells us that having two workers on Monday is cheapest and we end up with optimal staffing profile $P = (2, 2, 2, 0)$ (starting on Monday).*

5.5 Improvements Over Basic Dynamic Programming

The time and space bounds of Lemmas 5.4.1 and 5.4.2 can be improved by observing that each array of costs ranging over p for day t is a piecewise linear convex (PWLC) function $g(p)$.

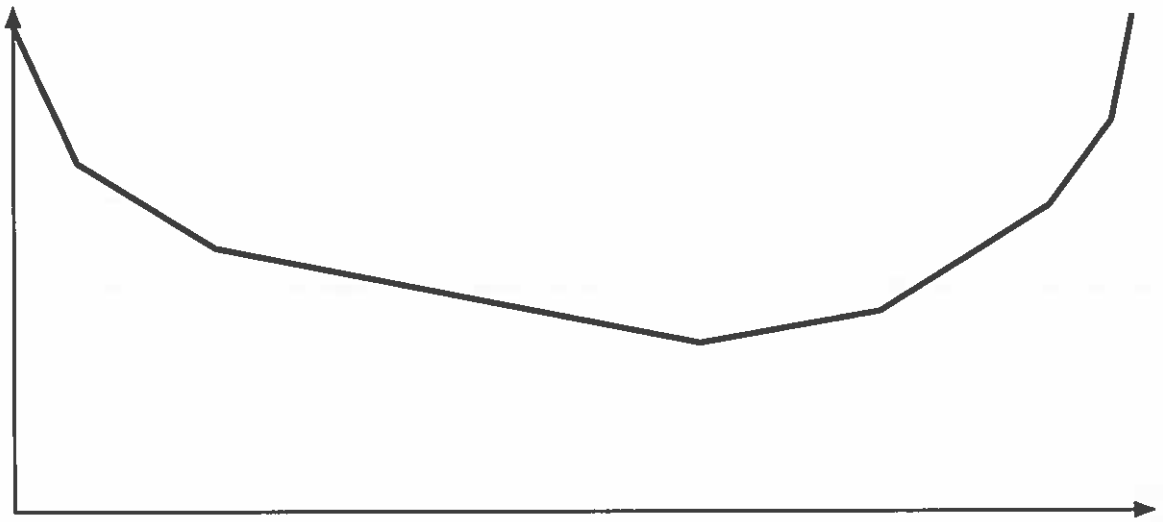


FIGURE 5.1: An example of a PWLC function.

Definition 12. *Given real numbers $b_1 < b_2 < ... < b_k$, called **inflection points**, a continuous function on $\mathbb{R}$ is:*

- ***piecewise linear** if it is linear on each segment $[b_i, b_{i+1}]$ and*
- ***convex** if the slope of segment $[b_i, b_{i+1}]$ is less than that of segment $[b_{i+1}, b_{i+2}]$ for all i. In other words, any line drawn between two points on the function will fall on or above the function.*

An example of a PWLC function (defined over a subset of the real numbers) is shown in Figure 5.1.

Lemma 5.5.1. *The array of daily costs (new-costs) for any time t is a PWLC function.*

Proof: Given a work level w, *new-costs* is a PWLC function $g(p)$ with an inflection point (and minimum value) at $p = w$. For $p < w$, there will be additional overtime costs but no undertime costs and the slope will be $-V$. For $p > w$, there will be additional undertime costs but no overtime costs and the slope will be U. This gives us:

$$g(p) = \begin{cases} p + V(w - p) & p \leq w \\ p + U(p - w) & p > w \end{cases} \tag{5.4}$$

This is clearly a PWLC function.

■

Lemma 5.5.2. *Given a PWLC function $g_1(p)$ representing the total-min-costs for time t, the function $g_2(p)$ representing the start-min-costs array for time $t+1$ is a PWLC function.*

Proof: The set of costs defined by $g_2(p)$ is an update of $g_1(p)$ with the cost of any point p reduced if that staffing level can be reached for lower cost by hiring or firing from another level p' at time t (an example is shown in Figure 5.2).[8] When that happens, there are two possibilities:

1. It is updated by a point $p' < p$. In this case, $f(p'-p) = 0$ and the update is made because
$$g_1(p) > g_1(p') + h(p-p')$$
2. It is updated by a point $p' > p$. In this case, $h(p-p') = 0$ and the update is made because
$$g_1(p) > g_1(p') + f(p'-p) \tag{5.5}$$

Now consider function $g_1(p)$ that is getting updated. Given that it is a PWLC, we divide its segments into 3 sets (see the second graph in Figure 5.2):

1. The first set of segments, A, (possibly empty) contains those for which *slope* $< -F$. We can show that every segment in A will be replaced by a single one of slope $-F$ in $g_2(p)$ (see the shaded area on the left of the third graph in Figure 5.2).

   Let the set of segments in A end at point p_{end}. For every point $p \in A$,
$$\frac{g_1(p_{end}) - g_1(p)}{p_{end} - p} < -F$$
$$g_1(p) > g_1(p_{end}) + F \cdot (p_{end} - p) \tag{5.6}$$
   Since this satisfies (5.5), the cost at p will be updated using the cost at p_{end} unless there is an even better update available. We now show that there isn't. It should be clear that p will not be updated by any $p' < p$ since $g_1(p') > g_1(p)$.

[8] For example, this happened on Wednesday in Example 5.4.2; to start with 3 workers, it is cheaper to have 2 workers on Tuesday and hire another one than it is to have already had 3 workers on Tuesday.

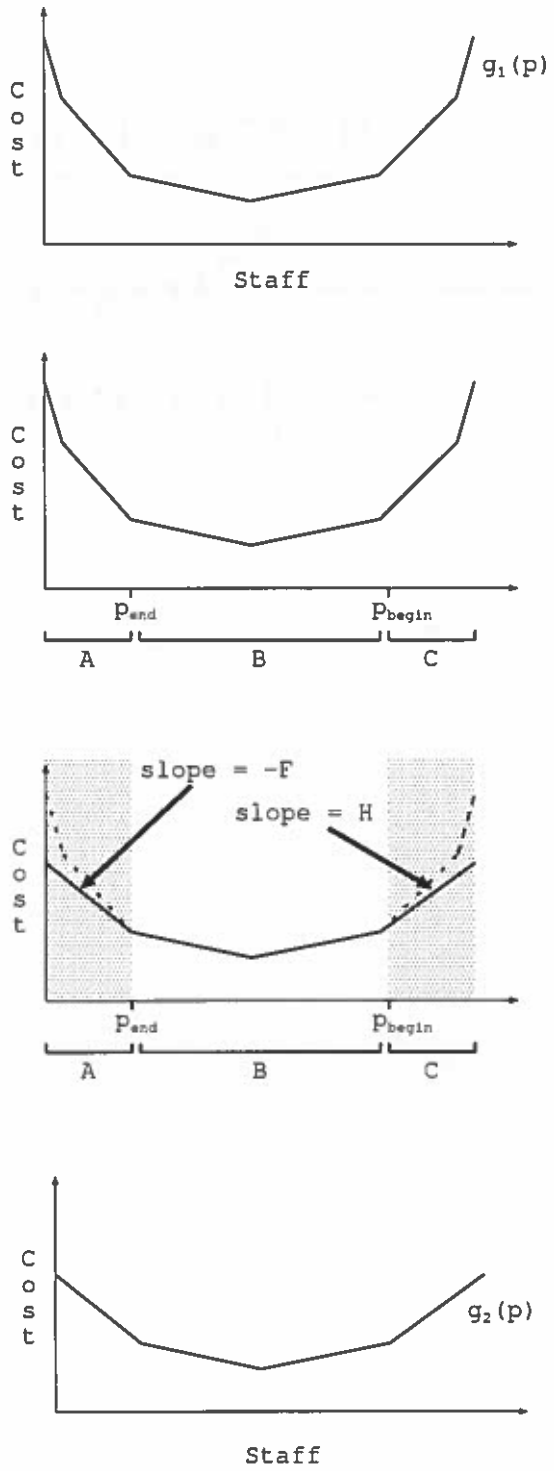


FIGURE 5.2: An example of a PWLC function $g_1(p)$ (*total-min-costs*) for time t updated to produce a PWLC function $g_2(p)$ (*start-min-costs*) for time $t+1$ as described in the proof of Lemma 5.5.2.

Therefore, consider any other point p^- such that $p < p^- < p_{end}$. We have just shown that:

$$g_1(p^-) > g_1(p_{end}) + F \cdot (p_{end} - p^-)$$

Therefore,

$$g_1(p_{end}) < g_1(p^-) - F \cdot (p_{end} - p^-)$$

Adding $F \cdot (p_{end} - p)$ to both sides and rearranging the right side, we get:

$$\begin{aligned} g_1(p_{end}) + F \cdot (p_{end} - p) &< g_1(p^-) - F \cdot (p_{end} - p^-) + F \cdot (p_{end} - p) \\ &< g_1(p^-) + F \cdot (p^- - p) \end{aligned} \tag{5.7}$$

The left side of 5.7 is the new value of $g_1(p)$ if p_{end} is used to updated it and the right side is its value if p^- is used. Therefore, p^- will not be used to update $g_1(p)$. A similar argument shows that for any $p^+ > p_{end}$,

$$g_1(p_{end}) + F \cdot (p_{end} - p) < g_1(p^+) + F \cdot (p^+ - p)$$

Therefore, in the FIND-STARTING-MIN procedure, we set $g_2(p) = g_1(p_{end}) + F \cdot (p_{end} - p)$ for each point p in set A, resulting in a single segment between $p = 0$ and $p = p_{end}$ with slope $-F$.

2. The second set of segments, B, (possibly empty) is those for which $-F \leq slope \leq H$. Similar arguments to the above can show that for each point p in segment B, $g_2(p) = g_1(p)$ since there can be no other p' from which a cheaper starting cost can be obtained.

3. The third set of segments, C, (possibly empty) is those for which $slope > H$. Let C begin at point p_{begin}. Similar arguments to case 1 show that in the FIND-STARTING-MIN procedure, we set $g_2(p) = g_1(p_{begin}) + H(p - p_{begin})$ for each point p in C (see the shaded area on the right of the third graph in Figure 5.2).

Combining the above three cases shows that $g_2(p)$ will consist of a segment with slope $-F$, an already PWLC set of segments, each with slope s_i such that $-F \leq s_i \leq H$, and a final segment with slope H. Therefore, $g_2(p)$ will be a PWLC function as required.

■

Theorem 5.5.3. *The minimum costs obtained by* FIND-MIN-COST *at any time t is a PWLC function $g(p)$.*

Proof: The initialization of *total-min-costs* is a function $g(p) = 0$ which is clearly PWLC. Lemma 5.5.2 shows that the if *total-min-costs* is PWLC, *start-min-costs* will be PWLC as well. Finally, Lemma 5.5.1 shows that array *new-costs* is PWLC. Since the sum of two PWLC functions is also a PWLC function,[9] the *total-min-costs* at the end of each iteration of FIND-MIN-COST will be a PWLC function.

■

We can now replace FIND-MIN-COST with a faster procedure, FIND-MIN-COST-FAST. We represent a PWLC function with the coordinates of the inflection points $((p_0, y_0), ..., (p_k, y_k))$ where $p_0 = 0$ and $p_k = p_{max}$ bound the range of staffing levels we wish to consider.

```
FIND-MIN-COST-FAST(W)
1  min-cost-func = ((0,0),(p_max,0))
2  for t = 1 to m
3        do min-cost-func = ADJUST-FOR-STARTING-COSTS(min-cost-func)
4           min-cost-func = ADD-NEW-COSTS(w_t,min-cost-func)
5  return minimum point in min-cost-func
```

The FIND-MIN-COST-FAST procedure steps through all time units adjusting a single PWLC function *min-cost-func*. Each adjustment has a first step to adjust the function to the minimum starting costs for t and a second step to add on the new costs incurred by t.

The ADJUST-FOR-STARTING-COSTS procedure, outlined below, shows how a cost function can be updated to the minimum starting costs.[10] It proceeds in much the same manner as our proof of Lemma 5.5.2. Lines 3 to 8 replace all segments with slope $s < -F$ with a single segment with slope $-F$. Lines 9 to 11 maintain all of the segments with slope $-F \leq s \leq H$. Lines 12 to 14 replace all segments with slope $s > H$ with a single segment of slope H.

[9]This is well-known and can be shown directly using the definition of a convex function $f(p)$ as one where, for any p_1 and p_2 in the domain of f and any $\lambda \in (0,1)$,

$$f(\lambda p_1 + (1-\lambda)p_2) \leq \lambda f(p_1) + (1-\lambda)f(p_2).$$

[10]For simplicity, we build the new set of inflection points from scratch. In practice, it is better to adjust the current inflection points instead.

The second procedure, ADD-NEW-COSTS, is also outlined below. It adds the costs for this time unit (the PWLC function described in Lemma 5.5.1) to the current *min-cost-func*. This is done by updating the y-value of each inflection point and possibly adding a new inflection point at $p = w$ (if one does not yet exist).

ADJUST-FOR-STARTING-COSTS$((0, y_0), (p_1, y_1), ..., (p_k, y_k))$

```
 1  i = 0
 2  new-values = ()
 3  while i < k and slope of segment [p_i, p_{i+1}] is ≤ −F
 4        do i = i + 1
 5  if i > 0
 6     then y'_0 = y_i + p_i · F
 7          add (0, y'_0) to new-values
 8  add (p_i, y_i) to new-values
 9  while i < k and slope of segment [p_i, p_{i+1}] is ≤ H
10        do i = i + 1
11           add (p_i, y_i) to new-values
12  if i < k
13     then y'_k = y_i + F · (p_k − p_i)
14          add (p_k, y'_k) to new-values
15  return new-values
```

ADD-NEW-COSTS$(w, ((0, y_0), (p_1, y_1), ..., (p_k, y_k)))$

```
 1  i = 0
 2  new-values = ()
 3  while p_i < w
 4        do y'_i = y_i + w + V · (w − p_i)
 5           add (p_i, y'_i) to new-values
 6           i = i + 1
 7  if p_i = w
 8     then add (p_i, w + y_i) to new-values
 9          i = i + 1
10     else  curr-slope = (y_i − y_{i−1})/(p_i − p_{i−1})
11          y'_i = y_i + w − curr-slope ·(p_i − w)
12          add (w, y'_i) to new-values
13  while i ≤ k
14        do y'_i = y_i + w + U · (p_i − w)
15           add (p_i, y'_i) to new-values
16  return new-values
```

To keep the number of inflection points at a minimum (and therefore improve efficiency), we would like to know that the above two procedures will not produce PWLC functions that have two consecutive segments with the same slope.

Consider the PWLC function created by ADJUST-FOR-STARTING-COSTS. It begins with a single segment of slope $-F$. The next set of segments all have slope s where $-F < s < H$[11] and are taken directly from the input PWLC function. Finally, the last segment created has slope H. Therefore, as long as the input PWLC function does not have two consecutive segments with the same slope, neither will the resulting PWLC function.

Now consider the PWLC function g created by ADD-NEW-COSTS; it is a result of adding two PWLC functions, g_1 and g_2 (one for the starting costs for day t and one for the new costs on day t). Suppose g has two consecutive segments, a and b, with the same slope. For there to be an inflection point between them, we can assume that g_1 (without loss of generality) had an inflection point there with the slope of the preceding segment less than the slope of the succeeding segment.

However, the slope of g at any point is the sum of the slopes of g_1 and g_2 at that point. Therefore, for a and b to have the same slope, g_2 must add more slope to the preceding segment of g_1 than to the succeeding segment. This is impossible if g_2 is a PWLC function. Therefore, as long as the input PWLC function to ADD-NEW-COSTS does not have two consecutive segments with the same slope, neither will the resulting PWLC function.

Example 5.5.1. *Figure 5.3 shows how* FIND-MIN-COST-FAST *will maintain PWLC functions to solve the problem of Example 5.4.1. Each segment is labeled with its slope.*

The easiest way to understand these graphs is to refer back to Table 5.2; each graph represents a column in that table and each row of graphs represents a day (divided into minimum starting costs, new costs and total costs). Therefore, the x-axis of each graph is staffing level p and the y-axis is the associated cost.

For example, the first graph is the line $y = 0$ since we can arrive at any staffing

[11]Notice the use of the $\leq$ symbol on lines 3 and 9. Using $<$ instead, as suggested in the proof of Lemma 5.5.2, would make it possible for two segments to end up with the same slope.

level p on Tuesday for no cost. The second graph for Tuesday shows the new costs for that day; since there are 2 units of work to do, the inflection point is at $p = 2$ and the cost increases for any p less than 2 (due to overtime costs) or more than 2 (due to undertime costs). Finally, the third graph on the row is simply the sum of the first two and shows the minimum cost (over the course of the project up to that point) to use p workers on that day.

Notice that every graph is a simple PWLC function. The optimal cost of 6.5 at final staffing level 0 can be seen on the final graph.

Lemma 5.5.4. FIND-MIN-COST-FAST *has time complexity $O(mp_{max})$ or $O(m^2)$. Its space complexity is $O(p_{max})$ or $O(m)$.*

Proof:

The procedures ADJUST-FOR-STARTING-COSTS and ADD-NEW-COSTS are invoked for each time unit and each of them simply steps through the set of inflection points for the PWLC function in question. Therefore, if the maximum number of inflection points is k, we get time complexity $O(mk)$.

There are two ways to bound the number of inflection points. Since we are never interested in values of any PWLC function $f(p)$ for $p > p_{max}$, we know that $k \leq p_{max}$. In addition, ADJUST-FOR-STARTING-COSTS can only decrease the number of inflection points and ADD-NEW-COSTS can only add a single inflection point for each time unit as indicated by the proof of Lemma 5.5.1. Therefore, we also know $k \leq m$. Replacing k with either p_{max} or m in $O(mk)$ gives the desired results.

During the above procedures, we store and modify a single PWLC function. Since that storage requires information for each inflection point, we need $O(k)$ space. As above, we can replace k with either p_{max} or m to get the desired results.

■

5.5.1 Calculating the Optimal Staffing Profile

As before, we may want to acquire the staffing profile that achieves the optimal cost in addition to determining that cost. The arguments of Lemma 5.5.2 show that

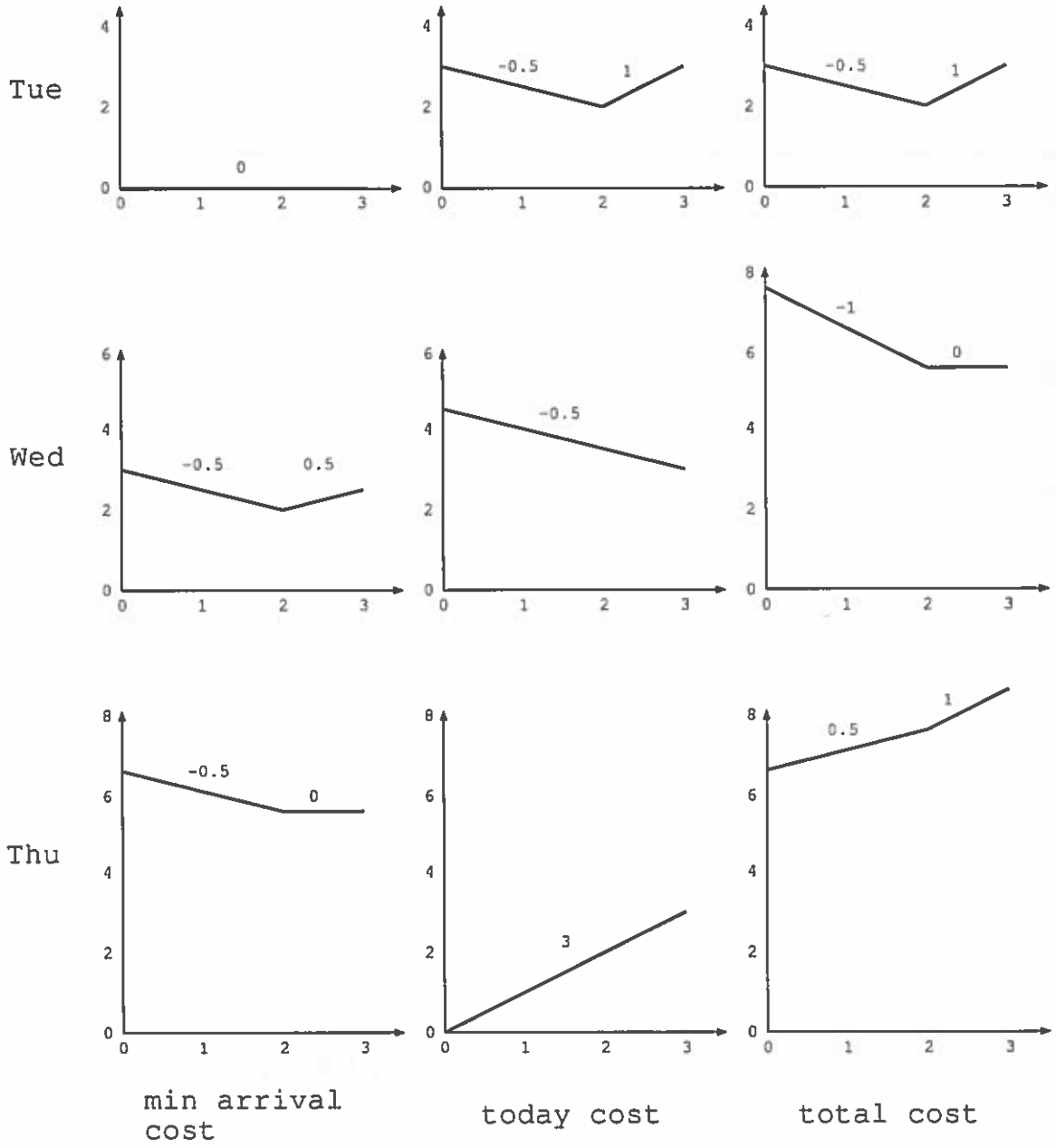


FIGURE 5.3: PWLC functions representing the calculations of FIND-MIN-COST-FAST for Example 5.4.1. The x-axis is the staffing level and the y-axis is cost.

when moving from time unit t to $t+1$, there is an inflection point p_{end} such that for any $p < p_{end}$, the minimum cost to have p workers at $t+1$ is achieved by having p_{end} workers at t and firing to get level p.

Similarly, there is an inflection point p_{begin} such that for any $p > p_{begin}$, the minimum cost to have p workers at $t+1$ is achieved by having p_{begin} workers at t and hiring to get level p. For any points $p_{end} \leq p \leq p_{begin}$, the minimum cost is achieved without a staffing change.

Therefore, to save the optimal staffing profile, we record p_{end} and p_{begin} for each time t. Let *fire-below* be the array of p_{end} values and *hire-above* be the array of p_{begin} values. Then FIND-OPTIMAL-STAFFING-PROFILE will work as follows (where p is the optimal staffing level for time m).

```
FIND-OPTIMAL-STAFFING-PROFILE(fire-below,hire-above,p)
1  optimal-profile[m] = p
2  for t = m − 1 to 1
3        do if optimal-profile[t + 1] < fire-below[t + 1]
4              then optimal-profile[t] = fire-below[t + 1]
5              else if optimal-profile[t + 1] > hire-above[t + 1]
6                       then optimal-profile[t] = hire-above[t + 1]
7              else optimal-profile[t] = optimal-profile[t + 1]
8  return optimal-profile
```

Example 5.5.2. *Consider the graphs of Figure 5.3 that show the PWLC functions calculated by* FIND-MIN-COST-FAST. *For each time unit (row), we get the following values:*

- ***Tuesday**: fire-below[Tue] = 0 and hire-above[Tue] = 3 since there is no need to hire or fire to get to any staffing level.*
- ***Wednesday**: fire-below[Wed] = 0 and hire-above[Wed] = 2. The latter occurs since the cheapest way to arrive at staffing level 3 is to have 2 workers on Tuesday and to hire the third to start Wednesday. This is why the second segment of the last graph on row 1 is replaced by one with lower slope in the first graph on row 2.*
- ***Thursday**: fire-below[Thu] = 2 and hire-above[Thu] = 3. The former occurs since the cheapest way to arrive at any staffing level less than 2 is to have*

2 workers on Wednesday and to fire as necessary to start Thursday with the desired number. This is why the first segment of the last graph on row 2 is replaced by one with slope closer to zero in the first graph on row 3.

FIND-OPTIMAL-STAFFING-PROFILE$((0,0,2),(3,2,3),0)$ *will be the call made using the above six values and will return optimal profile* $(2,2,0)$.

Lemma 5.5.5. *Finding the optimal staffing profile takes* $O(m^2)$ *(or* $O(mp_{max})$*) time and* $O(m)$ *space.*

Proof: Computing *fire-below* and *hire-above* does not increase the time complexity of Lemma 5.5.4 and the FIND-OPTIMAL-STAFFING-PROFILE procedure remains $O(m)$. The additional space complexity of maintaining *fire-below* and *hire-above* is $O(m)$; adding this to the $O(m)$ space complexity of FIND-MIN-COST-FAST does not change the complexity.

■

5.6 A Final Improvement

The algorithm using PWLC functions outlined in the previous section is an improvement over dynamic programming in both theory and practice. From a complexity theory point of view, however, like dynamic programming, the algorithm is only pseudo-polynomial in the problem size. This is because the time complexity ($O(m^2)$) is polynomial with respect to m but m could be exponential with respect to the size of the original problem.[12] Therefore, the algorithm is pseudo-polynomial.

Intuitively, the problem is that we could increase the time granularity without changing the problem; yet changes in time granularity affect the computation time.

In what follows, we show that the we can transform FIND-MIN-COST-FAST into a polynomial procedure. The basic idea is that any sequence of time points for which the work level is constant can be collapsed into a single time point from the point of view of our algorithm.

[12]The other parameter used for the complexity measures described in Lemma 5.5.4, p_{max}, could also be exponential with respect to the problem size for similar reasons.

Lemma 5.6.1. *For any work profile in which two consecutive time points have the same work level, there is an optimal staffing profile in which the two time points have the same staffing level.*

Proof: Assume this is not true. Then there is work profile where $w_t = w_{t+1}$ for which the optimal profile P has $p_t \neq p_{t+1}$ and there is no optimal profile with $p_t = p_{t+1}$. We must show that there does exist an optimal profile P' with $p'_t = p'_{t+1}$ that is no worse, thus contradicting our assumption.

We begin with $P' = P$. We let p'_t and p'_{t+1} be whichever of p_t and p_{t+1} incur less overtime and undertime costs. In other words:

$$p'_t = p'_{t+1} = \begin{cases} p_t & \Delta_{vu}(p_t - w_t) \leq \Delta_{vu}(p_{t+1} - w_{t+1}), \\ p_{t+1} & \textit{otherwise} \end{cases}$$

Now compare the costs of P and P'. Since they each correspond to the same schedule, the base costs are the same. Our above choice for p'_t and p'_{t+1} guarantees that the overtime and undertime costs for P' will be no worse than those for P since we picked the staffing level from P that minimizes those costs.

This leaves the hire and fire costs. To compare them, we will use the following fact that can be derived directly from the definition of Δ_{hf}:

$$\Delta_{hf}(a - c) \leq \Delta_{hf}(a - b) + \Delta_{hf}(b - c) \tag{5.8}$$

In other words, the cost of changing directly from one staff level (a) to another (c) is never more expensive than the cost of changing to any third staff level (b) between the two.

For profiles P and P', let the hire and fire costs for interval $(t - 1, t + 2]$ (the interval where the costs might differ) be *cost* and *cost'* respectively. We have:

$$\begin{aligned} \textit{cost} &= \Delta_{hf}(p_t - p_{t-1}) + \Delta_{hf}(p_{t+1} - p_t) + \Delta_{hf}(p_{t+2} - p_{t+1}) \\ \textit{cost}' &= \Delta_{hf}(p'_t - p_{t-1}) + \Delta_{hf}(p_{t+2} - p'_t) \end{aligned}$$

Suppose that $p'_t = p_t$. This gives us:

$$\textit{cost}' = \Delta_{hf}(p_t - p_{t-1}) + \Delta_{hf}(p_{t+2} - p_t)$$

Now, if we subtract *cost'* from *cost*, the two $\Delta_{hf}(p_t - p_{t-1})$ terms cancel and we get:

$$cost - cost' = \Delta_{hf}(p_{t+1} - p_t) + \Delta_{hf}(p_{t+2} - p_{t+1}) - \Delta_{hf}(p_{t+2} - p_t)$$

We can now use (5.8) to show that $cost - cost' \geq 0$. Similar arguments show the same result in the case where $p'_t = p_{t+1}$. Therefore, the hire and fire costs of P' can be no worse than those of P.

Therefore, P' must also be optimal. This contradicts our assumption and proves the original statement.

■

Corollary 5.6.2. *In the procedure* FIND-MIN-COST-FAST *we can skip the invocation of* ADJUST-FOR-STARTING-COSTS *(line 3) for any time point t for which* $w_t = w_{t-1}$.

Proof: In Lemma 5.6.1, we showed that there is an optimal profile with $p_t = p_{t-1}$. Therefore our algorithm will still return the optimal answer if we make hiring and firing infinitely expensive for time t (set $F = H = \infty$). However, if these two costs are infinite, it is clear that ADJUST-FOR-STARTING-COSTS will make no changes to the function passed to it and therefore can be skipped altogether.

■

Corollary 5.6.3. *For any sequence of consecutive time points with equal work levels, the procedure* FIND-MIN-COST-FAST *can be modified so that the min-cost-func function can be updated a single time for the entire sequence of time points.*

Proof: Consider some sequence of time points where $w_t = w_{t+1} = ... = w_k$. In Corollary 5.6.2 we showed that we need not invoke ADJUST-FOR-STARTING-COSTS for times $t + 1$ through k.

Now consider the calls to ADD-NEW-COSTS. We can show that the updates for $t + 1$ through k can all be done at once. First note that lines 10 through 12 will only be invoked a single time at most; an inflection point can only be added at $p = w_t$ once. The other changes result from lines 4, 8 and 14.[13]

[13]While the pseudocode creates a new PWLC function at each iteration, this is equivalent to updating the points in the current PWLC function.

Consider line 4. For each time point t', it will add $w_{t'} + V(w_{t'} - p_{t'})$. However, since the values of p and w are identical for all time points t through k, we can replace all of these updates with a single update that adds $(1 + k - t)(w_t + V(w_t - p_t))$.

The arguments for the other updates are the same. Therefore, a modified ADD-NEW-COSTS procedure can update the PWLC function for all requisite time points at once.

■

Theorem 5.6.4. *Given a schedule for an LCOP instance, if Δ_w represents the number of changes in the work level, the optimal cost can be computed in $O(\Delta_w^2)$ time using $O(\Delta_w)$ space and therefore using polynomial time and space.*

Proof: We now must update the PWLC function Δ_w times. As before, each such update can add at most 1 inflection point. Since the time required for each update is proportional to the number of inflection points, it is $O(\Delta_w)$ and the overall time complexity is $O(\Delta_w^2)$.

Since we only store a single PWLC function and store it by recording the inflection points, the space required is $O(\Delta_w)$.

■

Corollary 5.6.5. *Given a schedule for an LCOP instance, the optimal cost can be computed in $O(n^2)$ time and $O(n)$ space, where n is the number of activities.*

Proof: It should be clear that $\Delta_w \leq 2n$. Therefore the result follows directly from Theorem 5.6.4.

■

Corollary 5.6.6. *Given a schedule for an LCOP instance, the optimal staffing profiles can be calculated in $O(n^2)$ time and $O(n)$ space, where n is the number of activities.*

Proof: The proof is identical to that of Lemma 5.5.5.

■

5.7 Using Dynamic Programming During Search

We are now able to find a minimum cost (and the associated staffing profile) given a schedule. Our final goal is to take this one step further and search for schedules with minimum cost. That is, we want to solve the LCOP.

In the next chapter, we describe ARGOS, a search algorithm that heuristically solves the LCOP problem. Much like our SWO algorithm in Chapter 4, it will consider activities one at a time and for each will choose a start time from within that activity's time window.

For SWO, we chose for each activity its earliest feasible start time. For LCOP problems, we instead want to choose the least cost start time. This choice depends on the minimum cost for each possible start time. The FIND-BEST-START-TIME procedure outlines a way to compare those costs. Given a set of possible start times $(s_1, s_2, ..., s_k)$, we schedule the activity at each possible s_i and find the cost of that position using the procedure FIND-MIN-COST-FAST.

FIND-BEST-START-TIME is outlined below. It uses INCREASE-WORK-PROFILE to update the work profiles to include activity A_i starting at time t. Similarly, DECREASE-WORK-PROFILE) reduces the work profile to exclude the current position of A_i.

```
FIND-BEST-START-TIME(W, A_i, (s_1, s_2, ..., s_k))
 1  best-start = DEFAULT-TIME
 2  min-cost = ∞
 3  for j = 1 to k
 4        do INCREASE-WORK-PROFILE(A_i,s_j,W)
 5           this-cost = FIND-MIN-COST-FAST(W)
 6           if this-cost < min-cost
 7              then min-cost = this-cost
 8                   best-start = s_j
 9           DECREASE-WORK-PROFILE(A_i,s_j,W)
10  return best-start
```

There is a tremendous amount of work that will be repeated in FIND-BEST-START-TIME since the work profiles for most time units are not affected by the start time of A_i. It is possible to avoid most of that extra computation using procedure

FIND-BEST-START-TIME-IMPROVED described below. The only times for which more than one PWLC function need to be considered are those between s_1 and $s_k + dur_{A_i}$. For each such time point t, the number of PWLC functions considered will be the number of s_i for which $s_i \leq t < s_i + dur_{A_i}$.

The following outlines the procedure FIND-BEST-START-TIME-IMPROVED:

1. Without A_i placed, perform FIND-MIN-COST-FAST from time $t = 0$ to $t = s_k - 1$ and record the PWLC function $f_i^{start}(p)$ right before each start time s_j we wish to consider (at $s_j - 1$).

2. Without A_i placed, perform FIND-MIN-COST-FAST-BACKWARD[14] from time $t = m$ (the end of the schedule) to time $t = s_1 + 1$ and record the PWLC function $f_i^{finish}(p)$ right after each finish time $s_i + dur_{A_i} - 1$ (at $s_i + dur_{A_i}$).

3. For each start time s_j, invoke INCREASE-WORK-PROFILE(A_i, s_j, W) and then perform FIND-MIN-COST-FAST from $t = s_j$ to $s_j + dur_{A_i} - 1$ beginning with PWLC function $f_i^{start}(p)$ (and invoke DECREASE-WORK-PROFILE(A_i, s_j, W) when finished). The resulting PWLC function can easily be combined with the already saved $f_i^{finish}(p)$ to calculate the minimum cost for the staffing profile with A_i scheduled at s_i.

4. Return the s_i that gave the smallest minimum cost.

The original FIND-BEST-START-TIME procedure will compute PWLC functions for mk time points. The first two steps of the improved procedure (FIND-BEST-START-TIME-IMPROVED) will compute PWLC functions for $m + (s_k - 1) - (s_1 + 1)$ time points. The third step will have to consider another $k \cdot dur_{A_i}$ PWLC functions. Altogether, this will take $O(m + k \cdot dur_{A_i})$ time. Compare this with $O(mk)$ time for the original procedures; whenever $dur_{A_i} << m$ and $k \geq 2$ (both of which are usually true), procedure FIND-BEST-START-TIME-IMPROVED will be considerably faster.

Despite these improvements, FIND-BEST-START-TIME-IMPROVED is still a bottleneck for our search algorithms. Therefore, two other speedups have been implemented to further improve performance:

1. As explained in Chapter 3, the set of possible start times for an activity, given a partial schedule, can be represented with a time window $[ses_{A_i}, sls_{A_i}]$. Instead

[14]This is a procedure completely symmetric to FIND-MIN-COST-FAST that begins at the end of time and moves toward the beginning.

of trying all s_i such that $ses_{A_i} \leq s_i \leq sls_{A_i}$ in FIND-BEST-START-TIME, we try a subset. We use the following conventions:

- Always try both ses_{A_i} and sls_{A_i}.
- If the activity had a previous start,[15] s_{prev}, always try $s_{prev} - r \leq s_i \leq s_{prev} + r$ for some given radius r. Our reasoning is that, since we are not trying each start time, a previously selected start time might have just missed the optimal start time. If we came close to the optimal start time, we want to avoid missing it again.
- All other start times are tried with some probability p.

2. Imagine an activity A_i that has a window of length 10 near the end of a project of duration 1000. The costs for times at the beginning of the project are likely to have very little impact on the decision as to where to schedule A_i. Therefore, we only compute FIND-BEST-START-TIME between time $max(0, s_1 - b)$ and $min(m, s_k + dur_{A_i} + b)$ for some buffer distance b. Since hire and fire costs affect the relevance of times farther away from the window considered, b should depend on H and F. It is important to note that cost calculations cannot be exact when the entire makespan is not included. However, both intuition and experience suggest that giving up some accuracy for speed is a trade-off worth making.

5.8 Other Issues

For clarity, the above descriptions avoid many real-world issues that we must also be able to handle. We now mention a number of those that have arisen in discussions with industry and briefly describe how they are handled.

The most important one is a **maximum overtime rate**, ot_{max}, that limits how much overtime work can be done. Without it, a single worker could be scheduled to do the work of 100 if that was cheaper than hiring or firing.[16]

To handle this, we simply define an infinite cost C_{inf} and for maximum overtime rate ot_{max}, the PWLC function will begin with a segment of infinite negative slope at $p = w/(1 + ot_{max})$. The function remains PWLC (although, strictly speaking, it

[15] We sometimes want to consider moving an already scheduled activity to a better start position.

[16] In fact, it would be easy to concoct cases where 0 workers proved to be optimal, even when there was work to do. This doesn't translate well to reality.

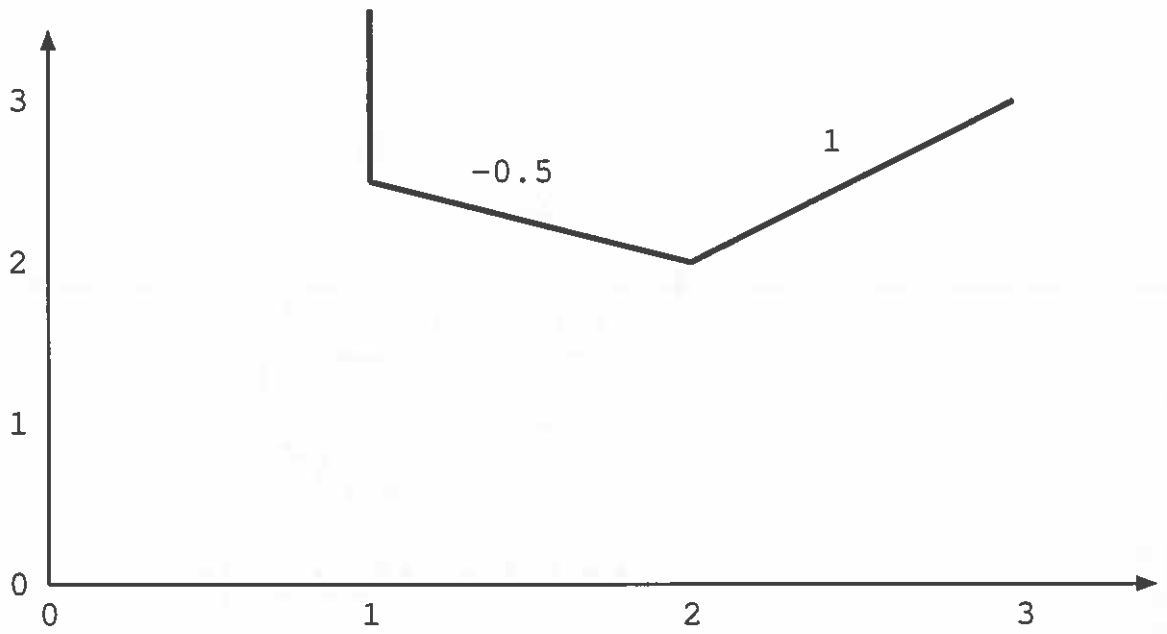


FIGURE 5.4: A daily cost function with a maximum overtime rate.

cannot be called a function anymore) and the only change is some extra bookkeeping since it may no longer be defined for all values of p.

Example 5.8.1. *Figure 5.4 shows the daily cost function for the first time unit of Example 5.4.1 modified for the case where* $ot_{max} = 100\%$. *Since the work level is 2 units, there must be at least one worker or else the cost will be infinite.*

We may want to limit the **maximum staffing level**, p_t^{max}, or **minimum staffing level**, p_t^{min} for any time unit. Handling the latter is identical to the maximum overtime rate; the PWLC function will have infinite negative slope at p_t^{min}. Handling the former is symmetric; the last segment of the PWLC function will start at the maximum staffing level and have infinite slope. We can handle a specified staffing level by setting $p_t^{min} = p_t^{max}$.

We may want to specify a **starting staffing level**, p^{start}, or an **ending staffing level**, p^{end}, to represent the staffing level at the beginning or end of a project. Both of these can be handled as special cases of the maximum and minimum staffing level cases. For the starting staffing level, we already have a cost function for time 0 (one

unit before the project begins) and we simply set $p_0^{min} = p_0^{max} = p^{start}$.[17] For the ending staffing level, we add a dummy time unit $t = m + 1$ to the project and set $p_{m+1}^{min} = p_{m+1}^{max} = p^{end}$.

There also may be a **maximum hire amount**, h_t^{max}, or **maximum fire amount**, f_t^{max}, specified for any time unit. This introduces some complications:

- In ADJUST-FOR-STARTING-COSTS, some segments with *slope* $< -F$ will continue to be replaced with one of slope $-F$. Others, however, may be replaced with segments of smaller slope or may not be affected at all. Some segments may be replaced with two segments with different slope. While the number of inflection points may increase, the function will remain PWLC.

- To find the optimal profile, we must record additional points besides p_{end} and p_{begin} for each time t and the procedure FIND-OPTIMAL-STAFFING-PROFILE is more complicated and time consuming as a result.

- The final improvement suggested in Section 5.6 will no longer always work. That is, there can easily be situations where the work level is constant for a number of days but there is no optimal staffing profile that is constant for those days (imagine having to begin hiring to handle an upcoming spike in the work profile, for example).

If hiring and firing is not allowed during some time unit t (on a weekend for example), we can set $h_t^{max} = f_t^{max} = 0$.

We may want to account for **overtime units**; time units on which all work is done at the overtime rate (weekends and holidays, for example). As long as ADD-NEW-COSTS is made aware of those units that are overtime units, this can be handled easily. It is also straightforward to enforce a different maximum overtime rate for these time units.[18]

Fluctuating staffing levels occur when staffing levels change for reasons other than hiring or firing. Possible reasons include people who quit or retire and vacation

[17] Alternatively, recall that we defined $minCost_0(p) = 0$ to represent the fact that there is no cost to start with any staffing level. Instead, we can define $minCost_0(p^{start}) = 0$ and $minCost_0(p) = \infty$ for all $p \neq p^{start}$. This will force the optimization to use p^{start} as the staffing level at the beginning of the project.

[18] For example, it may be the case that a worker can work a regular day on the weekend but is not allowed to work any additional hours as she might be allowed to do during the week.

or sick leave.[19] In some cases, the staffing levels available for the project in question may also fluctuate.[20] These fluctuating levels can be handled by maintaining a set of fluctuating workers that is used to adjust the work level at each time unit. For example, if we know the fluctuation at some time unit is -2, we can increase the corresponding work level by 2. There are some subtle issues to be considered due to the effects of maximum overtime rates and other factors.

Notice that in equation 5.2, we assume that the staffing level must be an integer. In many cases, this may be appropriate but in others, we may want to allow part-time workers to be hired.

This highlights another advantage of PWLC functions; while dynamic programming no longer works if the staffing level can be any positive real number, PWLC functions can still be used. However, if we really do want the staffing level to be an integer, we must do some work with the PWLC function approach. Specifically, when the amount of work w_t is non-integer (perfectly reasonable), we end up creating an inflection point at a non-integer staffing level. To fix this situation, we simply update ADD-NEW-COSTS to add inflection points at $\lfloor w_t \rfloor$ and $\lceil w_t \rceil$ instead of one at w_t, as demonstrated in Example 5.8.2.

Example 5.8.2. *Using the parameters of Example 5.4.1, suppose that there is a single time unit schedule with 1.5 units of work to do. The first graph of Figure 5.5 shows the resulting PWLC function which obviously gives an optimal staffing level of 1.5. If only an integer staffing level is allowed, this PWLC function must be updated to that shown in the second graph with an optimal staffing level of 1.*

5.8.1 Penalty Costs

In the real world, scheduling problems often have multiple objectives and the labor cost model we have described may not account for all objectives deemed important

[19]The large shipyards apparently see a significant drop in staffing levels on the first day of hunting season.

[20]The large shipyards often share workers with other yards and their staffing levels fluctuate accordingly.

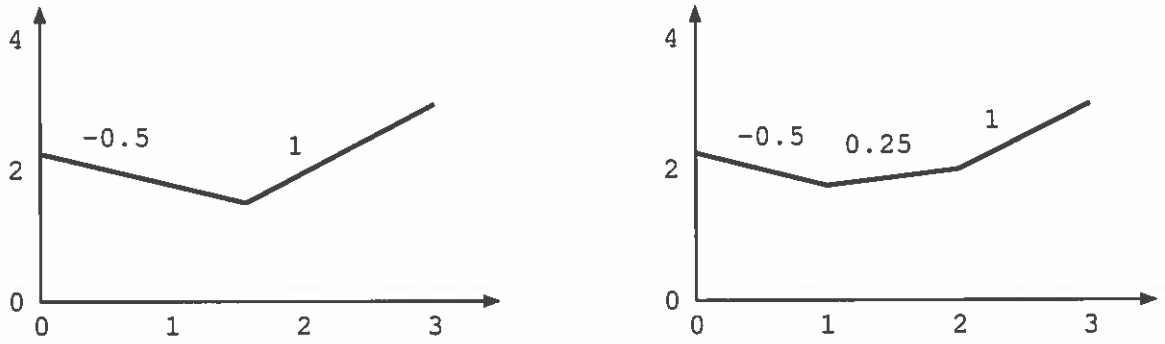


FIGURE 5.5: Updating a PWLC function with a fractional inflection point when only integer staffing levels are allowed.

to the scheduler. To handle this, we add a sixth cost to our calculations, the **penalty cost** ($C_{penalty}$), to incorporate other objectives. To allow the comparison of multiple objectives, we require that all objectives be measured in terms of cost.

A common objective that comes up time and again in industry is that of maximizing **float**. There are two versions of float common in the literature:

Free float is the amount of time an activity can be delayed without delaying any successor activities. Using the notation of Chapter 3, $FreeFloat_{A_i} = sls_{A_i} - start_{A_i}$.

Total float is the amount of time an activity can be delayed without delaying the finish time of the entire project (other activities might need to be delayed). In the notation of Chapter 3, $TotalFloat_{A_i} = hls_{A_i} - start_{A_i}$.

To add float maximization to our cost minimization objective function, we assign a penalty cost to activities whose float is reduced using the function:

$$penalty_{A_i} = \frac{dur_{A_i} \times n \times r \times a}{(float_{A_i}/b) + 1} \tag{5.9}$$

where n is the number of resources used by A_i, r is the total work done by those resources on A_i, $float_{A_i}$ is the total float, and a and b are user defined parameters with the following meaning:

- a is the relative cost to be incurred when there is zero float.
- b is the number of time units of float which is deemed to be half as bad as zero float.

The goal of equation (5.9) is to penalize float in a more rational way than simply looking at free float or total float:

- The penalty increases relative to the duration of A_i. The rationale is that one day of float for a one-day activity is less dangerous than one day of float for a 100 day activity.

- The penalty increases relative to the work done by A_i. The rationale is that one day of float for an activity that involves 1 worker is less dangerous than one day of float for an activity that involves 100 workers.

- The penalty for losing a day of float is more severe as the number of days of float decrease. For example, $float_{A_1} = float_{A_2} = 1$ is better than $float_{A_1} = 0$ and $float_{A_2} = 2$.

With float penalties included, the FIND-BEST-START-TIME procedure will balance the labor cost of the possible start times with the float penalty. Obviously there are two extremes: one without any float penalty (regular labor cost minimization) and one where the float penalty overwhelms all labor costs (in which case every activity will be scheduled at its earliest possible start time, hes_{A_i}). The most appropriate values for a and b between these two extremes depend on both the specific project and the importance of float.

It is easy to make float costs activity dependent. For example, in ship repair, there are a number of 'inspection' activities where part of the ship is inspected to find out if repair work is needed. We could assign float penalties only to inspection tasks since it may be more important for those activities to be completed early.

We can also use $C_{penalty}$ to avoid situations where the cost of a schedule will be infinite. For example, notice that with both a maximum staffing level and a maximum overtime rate, it may be impossible to meet the work level for some time unit t. In these cases, we may be interested in penalizing the situation appropriately but still finding out a relevant cost for the schedule. This would be important if all schedules are infinitely expensive but we still wish to find the best one possible.

5.8.2 Unresolved Issues

In addition to the above issues, there are a few potential issues that appear to make the cost minimization problem significantly harder.

The first is the presence of **cross-trades**. This occurs when one resource (set of workers) is qualified to do the work of another set (nuclear welders doing the work of welders, for example). Given a schedule, it is possible to include cross-trade optimization while minimizing labor costs using linear programming. It is not clear, however, that this can be made fast enough to be used during search.

In a couple of special cases, cross-trades will not make the problem any more difficult. First, if a set of resources have equivalent costs and can substitute for one another, they can all be treated as a single resource. Similarly, if one resource can substitute for another whose costs are all equivalent or higher, we can always use that resource and ignore the one with higher costs. Even in these special cases, additional constraints like maximum hire or fire rates will make the problem more difficult.

In project management systems like Artemis, instead of the form of cross-trades described above, the user is allowed to specify a number of different ways that an activity can be accomplished (for example, perhaps for some strange reason an activity can be performed either with 2 painters and an electrician or with 3 welders). This is equivalent to multi-mode project scheduling [49, 74] and would be better handled by the search side of an algorithm.[21]

Another issue that has not been resolved is **how overtime is accomplished**. The work in this dissertation originated with problems where each time unit is a work day. Therefore, overtime corresponds to workers who stay beyond the regular work hours to finish the required work. If work units instead correspond to other possible time units (hours, minutes, shifts etc.), it may be that all work must be accomplished during the specified time unit.

For example, if there are 3 hours worth of work scheduled from 8:00 to 9:00 and only 2 workers, we can no longer assign the unfinished hour as overtime to one of

[21]The search algorithm might ask the dynamic programming procedures for the cost of various modes; each of those could be calculated using our current approaches.

those 2 workers since the work scheduled at 9:00 likely assumes those 3 hours have been accomplished. This problem could be handled by setting the maximum overtime rate to 0. However, it may be that overtime is still allowed but must be represented differently. For example, if a work day consists of multiple shifts, a worker may be able to work overtime during the time units not on her regular shift.

The concept of **shifts** is the third potential difficulty worth mentioning. Suppose the work day consists of two eight-hour shifts. If all work is scheduled on a daily basis only (activities begin (end) only at the beginning (end) of the day), the day can be combined into a single time unit and the above approach will work. At the other extreme, if all workers must work either the morning or afternoon shift, the problem can be split into two pieces for labor cost minimization since the shifts can be treated separately. The most likely scenario, unfortunately, is one between the above extremes where workers should usually stay on the same shift but may sometimes switch shifts (temporarily or permanently) subject to some costs or restrictions. We do not know the best way to handle this.

5.8.3 Additional Notes

It is interesting to notice that the labor cost framework we have set up can easily be adapted for non-labor resources as well. To represent a machine, for example, we might use the following costs:

- B = cost to run machine
- $V = \infty$ (machines can't work overtime)
- $U = 0$
- H = cost to buy new machine
- F = cost to dispose of a machine

Using such a representation with maximum staffing levels, we can model the RCPSP/max problems discussed in Chapter 4. In the SWO algorithm, for example, the FIND-BEST-START-TIME procedure can be used to find out if there is a resource-feasible time for an activity (it will return an infinite cost otherwise).

Of course, such a representation would be wildly inefficient for RCPSP/max problems. However, this shows that using the LCOP framework is possible for any problems in the range between strict labor cost minimization problems and the resource constrained scheduling problems common in the literature.

5.9 Related Work

The importance of labor scheduling has been recognized for as long as scheduling has existed as a discipline. For example, in a 1967 book, Horowitz [51] discusses 'manpower leveling' and suggests that the scheduler might wish to either avoid fluctuations, keep the staffing level under some threshold, or schedule for a fixed staffing level. His discussion, however, centers around using graphical methods to schedule by hand (!) and there is no mention of quantitative analysis.

The use of dynamic programming to optimize staffing levels given a work profile has been suggested before. For example, Taha [93, page 366] uses the calculation of optimal work force size given hire and undertime costs as an example of a dynamic programming application. However, we are not aware of any previous attempt to incorporate such calculations into a search algorithm.

In the Operations Research community, there is a significant amount of work addressing labor scheduling and cost minimization, much of it for service industries. However, here the goal is generally to come up with optimal daily or weekly schedules for individual workers [6] that incorporate details like days off, specific worker skills [17] and worker learning curves [48]. The time frame, problem size, relevant industries and, as a result, the issues addressed are much different than those we focus on here.

Our labor cost minimization goal represents a nonregular objective function. Recall that a regular objective function is one where it is never advantageous to delay an activity that satisfies all time and resource constraints. While nonregular objective functions are mentioned a number of times in the literature, relevant algorithms and experimental results are scarce.

As mentioned in Section 2.2.2, three types of resource-based nonregular objective

TABLE 5.4: Various resource-based objective functions in the LCOP framework.

Objective	B	V	U	H	F	Notes
RIP	0	∞	0	c_k	0	Must be forced to start with 0 workers
TSUC	0	0	0	0	0	Penalty cost must incorporate objective function
TOC	0	c_k	0	0	0	Set max staffing level to threshold
TAC	0	∞	∞	c_k^+	c_k^-	
RRP	c_k^r	∞	c_k^r	c_k^p	0	

functions have recently been tackled by the OR scheduling community:

- The resource investment problem (RIP) problem where the goal is to minimize the sum (weighted by resource using c_k) of the maximum work levels over all resources.
- The resource leveling problem where the goal is to indirectly minimize labor costs through leveling. There are three variations:
  - Minimize the *total squared utilization cost* (TSUC).
  - Minimize the *total overload cost* (TOC).
  - Minimize the *total adjustment cost* (TAC).
- The resource renting problem (RRP) where the goal is to minimize the sum of resource procurement and renting costs.

In Table 5.4, we show how each of the above objective functions can be represented as an LCOP with appropriate cost constants.[22] Therefore, these objective functions are all subsumed by labor cost minimization. However, the cost of a given schedule is a straightforward calculation for most of them and it is unlikely that we would want to use our cost minimization algorithms for those.

Finally, it is worth noting that the LCOP is an NP-hard problem. This can easily be shown by taking advantage of proofs by Neumann et al. [74] showing the above problems are NP-hard. Since those problem can each be represented as an LCOP, the LCOP is NP-hard.

[22]We show one of many ways each could be represented.

CHAPTER 6

ARGOS: An Algorithm for Labor Cost Optimization

6.1 Introduction

In the previous chapter, we formally defined the LCOP, described efficient methods for calculating minimum costs given schedules and showed how those methods can be incorporated into search.

In this chapter, we describe ARGOS,[1] a suite of algorithms that heuristically solve LCOP problems by using those methods as the inner loop in a number of search procedures. Section 6.2 describes the four techniques (construction, polishing, morphing and annealing) that are used by ARGOS. These are combined in different ways to produce the four hybrid algorithms outlined in Section 6.3. Finally, Section 6.4 discusses some additional details and functionality of ARGOS.

Underlying the ARGOS algorithms is the GSTP framework for maintaining hard and soft windows as described in Chapter 3. Recall that windows are updated whenever activities are scheduled (PLACE-ACTIVITY) or unscheduled (UNPLACE-ACTIVITY).

[1] A Really Good Optimization System. :)

6.2 The Techniques

In this section we describe and outline a number of algorithmic techniques that are used by ARGOS. The first is a schedule construction algorithm. Due to the limited effectiveness of schedule construction by itself (multiple iterations do not help much, for example), the others are local search (or schedule improvement) techniques.

6.2.1 Schedule Construction

The CONSTRUCT-SCHEDULE procedure outlined below is a straightforward single-pass priority dispatch procedure. It steps through activities one at a time, tries a subset of start times for each activity, and greedily places each at the one resulting in minimum cost.[2]

```
CONSTRUCT-SCHEDULE(W)
1  for i = 1 to n
2        do times-to-try=SELECT-TIMES-TO-TRY(A_i,DEFAULT-TIME)
3           t =FIND-BEST-START-TIME-IMPROVED(W,A_i,times-to-try)
4           PLACE-ACTIVITY(A_i, t, W)
```

On line 2, SELECT-TIMES-TO-TRY is used to choose a subset of possible start times for A_i. The rationale behind this procedure, which is outlined below, is described in detail at the end of Section 5.7. Once the subset of start times is chosen, line 3 finds the best of those times and A_i is then placed at that time.

```
SELECT-TIMES-TO-TRY(A_i,currStart)
1  times-to-try = ()
2  add ses_{A_i} to times-to-try
3  add sls_{A_i} to times-to-try
4  for t = ses_{A_i} + 1 to sls_{A_i} − 1
5        do if currStart ≠ DEFAULT-TIME and currStart − r ≤ t ≤ currStart + r
6              then add t to times-to-try
7              else  add t to times-to-try with probability p
```

[2]Note that W is now a set of work profiles (one for each resource). In the previous chapter, we assumed for simplicity that it was a single work profile.

6.2.2 Polishing

The effectiveness of schedule construction is limited because placement decisions for most tasks (all but the last, in fact) are made with incomplete information. A position that has lowest cost when an activity is placed may not end up being a good position for that activity by the time all other activities are placed.

Polishing is a straightforward attempt to address this through schedule improvement; it is hillclimbing, the most basic form of local search. Outlined below, it steps through the set of activities. Each A_i is unplaced, a subset of feasible start times is considered and A_i is scheduled at the best one of these (with ties broken randomly). While A_i may end up right back where it started, there is a good chance that a better position will be found. As can be expected however, the cost improvements of POLISH-SCHEDULE quickly decrease if it is run for more than one iteration; in practice we seem to be close to a local minimum within 5 iterations.

```
POLISH-SCHEDULE(W)
1  for i = 1 to n
2        do
3           times-to-try = SELECT-TIMES-TO-TRY(A_i, start_{A_i})
4           UNPLACE-ACTIVITY(A_i, W)
5           t = FIND-BEST-START-TIME-IMPROVED(W, A_i, times-to-try)
6           PLACE-ACTIVITY(A_i, t, W)
```

6.2.3 Morphing

Consider schedule construction again. When each A_i is scheduled, there will be a subset A' of other activities that have yet to be scheduled. On the one hand, this is an advantage because A_i will be less temporally constrained by activities in A' and may therefore have more possible start times to consider. On the other hand, this is a disadvantage because the costs of those start times may not relate to their effectiveness in the final schedule since the final cost depends on the placement of activities in A'. This advantage and disadvantage are both especially severe near the beginning of schedule construction.

Morphing is an attempt to gain the above advantage while mitigating the effects of the disadvantage. It is a form of schedule construction that uses the work profiles W of a previous schedule to guide construction.

Morphing is outlined in the MORPH-SCHEDULE procedure below. Given a current schedule, line 1 records the start times of that schedule. The soft windows are reset to the hard windows in lines 2 to 5. Activity start times are also reset but we do not use UNPLACE-ACTIVITY to do so because that procedure adjusts work profiles. Unlike what would be done prior to schedule construction, the work profiles W of the current schedule are kept.

Then, when each activity A_i is considered, it is removed from W using its previous start time. A new best start time is found within A_i's soft window and A_i is placed there.

```
MORPH-SCHEDULE(W)
 1  previous-start = [start_A1, ..., start_An]
 2  for i = 1 to n
 3        do start_Ai = DEFAULT-TIME
 4           ses_Ai = hes_Ai
 5           sls_Ai = hls_Ai
 6  for i = 1 to n
 7        do times-to-try = SELECT-TIMES-TO-TRY(A_i,DEFAULT-TIME)
 8           DECREASE-WORK-PROFILE(A_i, previous-start[i],W)
 9           t =FIND-BEST-START-TIME-IMPROVED(W,A_i,times-to-try)
10           PLACE-ACTIVITY(A_i, t, W)
```

Consider the first activity A_1 rescheduled during morphing. We will be able to select any start time within its hard window (all windows have been reset) just as we would for schedule construction. However, we already have the work profiles W corresponding to the previous schedule, giving us some sense of what might be good start times for A_1.

Obviously, morphing is not perfect. For example, we may select a start time for A_1 that gives a terrific cost but discover later that that cost is unattainable. This is because W corresponds to start times for the remaining activities that may not be time-feasible given the new start time of A_1. Therefore, morphing is unlikely to improve an already good schedule. Nonetheless, morphing is the most effective way

we have found to quickly achieve good quality schedules from initial ones. In addition, it provides an interesting way to make large (but non-arbitrary) steps in the search space.

When morphing is included in an algorithm, it is always followed by 5 iterations of polishing to make up for the above difficulties.

6.2.4 Simulated Annealing

The final approach incorporated into ARGOS is a local search algorithm that is a form of simulated annealing. The goal is to avoid the local minima into which polishing quickly falls.

The main piece of our simulated annealing is a version of polishing that includes a temperature t. In this version, the temperature is used to encourage activities to move; for each start time s_j considered for an activity A_i, the actual cost of starting at s_j is decreased by $t \cdot (\mid s_j - start_{A_i} \mid)$. This gives a preference to start times that are far from $start_{A_i}$ as long as they are not too much more expensive.

```
ANNEAL(W, temperature)
 1  num-without-improvement= 0
 2  best-cost = Σ_{r=1}^{|W|} FIND-MIN-COST-FAST(W_r)
 3  while num-without-improvement < 2
 4        do
 5           polish schedule once using temperature
 6           for i = 1 to 5
 7                 do POLISH-SCHEDULE(A,W)
 8           cost = Σ_{r=1}^{|W|} FIND-MIN-COST-FAST(W_r)
 9           if cost < best-cost
10              then best-cost = cost
11                   num-without-improvement = 0
```

The entire simulated annealing algorithm, ANNEAL, is outlined above. Line 5 performs the modified polish where the temperature plays its role. This is followed by 5 regular polishes that approach the local minima for this new schedule. If the resulting schedule is not as good as the best schedule seen, it is thrown out and subsequent iterations start with the best schedule seen.

The whole loop is repeated until 2 consecutive iterations find no improved schedule. This avoids the need to fine tune the temperature setting since the correct settings are quite problem dependent. We can begin with a high temperature and reduce it when no progress is made. This allows self-tuning; the algorithm will naturally settle at effective temperatures and move on when they become ineffective.

6.3 ARGOS

The ARGOS scheduling tool consists of four hybrid algorithms based on the above techniques. The main difference is the number of expected iterations each will perform which in turn affects the expected solution quality. The fast ones can be used to get results quickly but the slower ones should be allowed to run when a low final cost is the main goal.

The first, ARGOS1, simply constructs a schedule and performs two polishes. ARGOS2 performs 2 iterations of morphing (each followed by 5 polishes) followed by 5 polishes. ARGOS3 performs 3 iterations of morphing followed by annealing at 7 different temperature levels between 50 and 0. Finally, ARGOS4 performs 5 iterations of morphing followed by annealing at 19 different temperature levels between 100 and 0. ARGOS4 produces the best results but can take days on large problems.

ARGOS1(A)
1 initialize W (all profiles empty)
2 INITIALIZE-WINDOWS-WITH-CYCLES
3 CONSTRUCT-SCHEDULE(W)
4 POLISH-SCHEDULE(W)
5 POLISH-SCHEDULE(W)

```
ARGOS2(A)
1  initialize W (all profiles empty)
2  INITIALIZE-WINDOWS-WITH-CYCLES
3  CONSTRUCT-SCHEDULE(W)
4  for i = 1 to 2
5        do MORPH-SCHEDULE(W)
6           for j = 1 to 5
7                 do POLISH-SCHEDULE(W)
8  for j = 1 to 5
9        do POLISH-SCHEDULE(W)
```

```
ARGOS3(A)
 1  initialize W (all profiles empty)
 2  INITIALIZE-WINDOWS-WITH-CYCLES
 3  CONSTRUCT-SCHEDULE(W)
 4  for i = 1 to 3
 5        do MORPH-SCHEDULE(W)
 6           for j = 1 to 5
 7                 do POLISH-SCHEDULE(W)
 8  temperature = 50
 9  while temperature ≥ 20
10        do ANNEAL(A, W, temperature)
11           temperature = temperature −10
12  temperature = 15
13  while temperature ≥ 5
14        do ANNEAL(A, W, temperature)
15           temperature=temperature−5
```

6.4 Additional Notes

6.4.1 Bulldozing

All of the ARGOS algorithms can be run with a version of bulldozing. Recall that bulldozing was shown to be a crucial piece of the SWO(B,R) algorithm for makespan minimization (Chapter 3). The basic idea for bulldozing in ARGOS is that instead of trying to schedule an activity A_i at a new time in its soft window, the entire hard window is considered. When start times outside of the soft window appear to

be preferable, other activities are bulldozed out of the way as necessary in order to make the desired times time-feasible. If the cost of the resulting schedule remains preferable to the cost obtained with A_i starting in its soft window (it may not since other activities are forced to move), the modified schedule is kept. Otherwise, the modifications are undone and the best start time within the soft window is chosen after all.

```
ARGOS4(A)
 1  initialize W (all profiles empty)
 2  INITIALIZE-WINDOWS-WITH-CYCLES
 3  CONSTRUCT-SCHEDULE(W)
 4  for i = 1 to 5
 5        do MORPH-SCHEDULE(W)
 6           for j = 1 to 5
 7                 do POLISH-SCHEDULE(W)
 8  temperature = 100
 9  while temperature ≥ 50
10        do ANNEAL(A, W, temperature)
11           temperature=temperature−10
12  temperature = 25
13  while temperature ≥ 20
14        do ANNEAL(A, W, temperature)
15           temperature=temperature−5
16  temperature = 18
17  while temperature ≥ 10
18        do ANNEAL(A, W, temperature)
19           temperature=temperature−2
20  temperature = 6
21  while temperature ≥ 1
22        do ANNEAL(A, W, temperature)
23           temperature=temperature−1
```

The FIND-BEST-START-TIME-IMPROVED procedure is modified to keep track of the bulldozing and do all the necessary bookkeeping. The actual implementation details will not be described here as they are both confusing and unenlightening. Here are a couple of pertinent details:

- Bulldozing could occur in either direction since preferable start times could

be before or after the soft window.[3] However, each direction can be handled independently and there are no additional difficulties.

- If there are multiple times outside a soft window that seem preferable, it is not always clear which to consider. For example, suppose that t_1 is the best start time within the soft window and both t_2 and t_3 are preferable where $t_1 < sls_{A_i} < t_2 < t_3$. If $cost(t_2) \leq cost(t_3)$, we do not consider t_3 since any bulldozing required to make t_3 time-feasible will also make t_2 time-feasible.[4] However, if $cost(t_2) > cost(t_3)$, it is not at all clear which is better. It is quite possible that the extra bulldozing required to make t_3 time-feasible negates the advantage it appeared to have. Therefore, our bulldozing implementation pursues up to five (an arbitrary cutoff) reasonable possibilities in parallel to find out which, if any, turns out to be best.

6.4.2 Local Search

The simulated annealing and polishing steps in ARGOS are each a version of local search in schedule space. The operators to produce neighbor schedules are the following.

- A **simple move** replaces the start time $start_{A_i}$ of some activity A_i with a new start time $start'_{A_i}$ in A_i's soft window. In other words, $ses_{A_i} \leq start'_{A_i} \leq sls_{A_i}$.
- A **bulldoze move** replaces the start time $start_{A_i}$ of some activity A_i with a new start time $start'_{A_i}$ in A_i's hard window. In other words, $hes_{A_i} \leq start'_{A_i} \leq hls_{A_i}$. In addition, a subset of the other activities is shifted so that the resulting schedule is time-feasible.

As implied by the name, bulldoze moves are made only when bulldozing is turned on. Any simple move is also a bulldoze move.

Because ARGOS relies heavily on this local search, we would like to know if it can explore the entire space. That is, we want the search space to be connected with respect to the move operators. The following shows that with an acyclic constraint graph, only simple moves are necessary to ensure the space can be explored.

---

[3] In SWO(B,R), we would only ever try to bulldoze in one direction at a time.

[4] It is possible to concoct examples where t_3 does turn out to be better but these seem unlikely to occur in practice.

Lemma 6.4.1. *For an acyclic problem, given any two time-feasible schedules, S_1 and S_2, there are $O(n)$ simple moves that will change schedule S_1 into S_2.*

Proof: We prove this by producing one such series of simple moves. Because the problem is acyclic there is some topological ordering of the activities. We can produce S_2 with two passes through the topological order.

The first pass moves each activity A_i to hes_{A_i} to produce the early start schedule (ESS). Because we proceed in topological order, it is straightforward to show that when A_i is moved, $hes_{A_i} = ses_{A_i}$ (since all predecessors have already been shifted) which is necessary since only simple moves are allowed.

The second pass is exactly the opposite. We step though the activities in reverse topological order and move each activity later to the start time specified by S_2.

■

The following simple example shows why Lemma 6.4.1 does not hold for cyclic problems.

Example 6.4.1. *Consider a two activity problem with constraints $(A_1, A_2, SS, 0)$ and $(A_2, A_1, SS, 0)$. In other words, the two activities must be scheduled at the same time. Now suppose we have feasible schedule S_1 where $start_{A_1} = start_{A_2} = 0$ and another feasible schedule S_2 where $start_{A_1} = start_{A_2} = 1$.*

There is no way to make simple moves to get S_2 from S_1. The problem is that A_1 and A_2 lock each other in place and neither of them can be moved using a simple move. In fact, the only way to produce a different feasible schedule is to move them both together.

Fortunately, if bulldoze moves are allowed, the whole search space can be reached even for cyclic problems.

Lemma 6.4.2. *Given any two feasible schedules, S_1 and S_2, there are $O(n)$ bulldoze moves[5] that will change schedule S_1 into schedule S_2.*

[5]In the worst case, a single bulldoze move could move n activities so we might move $O(n^2)$ activities.

Proof: We can step through the activities and for each A_i, move it directly to its start time in S_2. The only thing that we must ensure is that activities we have already moved will not be bulldozed out of place by subsequent moves. However, since S_2 is time-feasible, once an activity is in its new start position it will not conflict with the new start position of any other activity.[6]

■

6.4.3 Activity Orderings

The ARGOS algorithms outlined above omit a commonsense modification that has been implemented. Instead of stepping through all n activities in each procedure, we divide the activities into the following three sets and treat them accordingly:

- **Set I** includes each A_i for which $hes_{A_i} = hls_{A_i}$. Since these activities will have $start_{A_i} = hes_{A_i}$ in all schedules, this set is scheduled first during CONSTRUCT-SCHEDULE (since this will presumably help make more informed decisions concerning the placement of subsequent activities) and is ignored during all schedule improvement procedures (since these activities will never be moved).
- **Set II** includes any A_i not in Set I that uses no resources. These activities are common in real-world scheduling problems; they serve as milestone or bookkeeping activities. The start times of these activities do not affect schedule costs.[7] Therefore, these activities are never scheduled by ARGOS since to do so would unnecessarily constrain other activities whose start times do affect the cost.[8]
- **Set III** includes the remaining activities. These are the activities that matter. During polishing, morphing and annealing, it is this subset that is considered.

We did not mention the order the activities in Set III will be considered in the ARGOS procedures. In initial schedule construction and morphing, we have chosen to

[6]We have been purposefully vague about how bulldozing moves other activities. It is easy to define it so that activities are bulldozed only enough to maintain feasibility; this is how our current implementation works. Another reasonable approach would be to look for times that are not only time-feasible but also cost effective for the bulldozed activities. This latter approach is similar to the bulldozing used in SWO(B,R).

[7]Notice that even the float penalty described in the previous chapter will be 0 for an activity that doesn't use resources.

[8]When ARGOS outputs a schedule, it simply schedules each such A_i at ses_{A_i}.

order Set III based on hard window sizes; activities with the smallest hard windows are handled first. This is based on the same intuition that suggests Set I should be scheduled first. The idea is that activities which have very little leeway will be handled first while those with large windows can be more opportunely handled toward the end.

Unlike construction and morphing, Set III is randomly ordered before each iteration of polishing. We experimented with a range of different orderings for the different procedures. In general, ordering seems to have very little impact on the overall effectiveness of the different techniques; the above orderings were slightly better than other choices.

Another option would be to avoid ordering the activities altogether. For example, polishing could randomly choose activities from the entire set, making it likely that some activities would be considered more often than others. Brief experiments showed this tends to decrease performance.

Finally, an approach that might be effective but hasn't been attempted is to dynamically order the activities. If the ordering were based on soft windows, we could immediately schedule every A_i for which $ses_{A_i} = sls_{A_i}$, as we probably should. This might lead to improvements.

6.4.4 Additional Functionality

Often, a scheduling problem from industry includes a schedule that has already been developed. This might be a schedule produced by a project management system, a schedule produced by hand, or (as is often the case), some hybrid of the two. It is straightforward to start ARGOS with such a schedule rather than from scratch - we simply replace the initial schedule construction step with a schedule input step.

In the real world, it is unlikely that a schedule will ever be followed exactly due to random variability and unexpected changes. Therefore, it is often desirable to reschedule once work has begun. ARGOS includes a **freeze** capability; any work that is complete or underway at a given point in time is locked in place before the algorithms modify the remainder of the activities.

Finally, as a problem changes, a previously developed schedule may no longer satisfy all constraints. Therefore, ARGOS has a **legalize** function that attempts to modify a schedule as little as possible so that all constraints are satisfied. This constructs a schedule in two phases:

1. In the first phase, any A_i for which $start_{A_i}$ continues to satisfy all constraints is scheduled at $start_{A_i}$.
2. In the second phase, any A_i not yet scheduled is placed at the point in its soft window closest to $start_{A_i}$.

Obviously, the number of activities moved and the amount they are moved depends on the order they are considered in the two phases. Therefore, the above approach could be replaced with an optimization algorithm that attempts to minimize the total disruption. We have yet to find the need for such an algorithm.

6.4.5 Possible Algorithmic Modifications

A couple of possible modifications to the ARGOS algorithms are worth mentioning. First, we may want to add a cutoff time to ARGOS3 and ARGOS4. This is important if we care at all about running time since the number of iterations performed by ANNEAL can vary wildly from problem to problem (recall that annealing continues at the same temperature until it stops making improvements).

Finally, the many modifications and options discussed in 5.8 have all been incorporated into ARGOS.

CHAPTER 7

ARGOS Experimental Results

7.1 Introduction

In this chapter, we present results of running the ARGOS algorithms on a number of real-world scheduling problems obtained from various sources. Section 7.2 describes the origin and characteristics of those problems. Section 7.3 describes some of the specific ARGOS settings used and general results are presented in Section 7.4.

In Section 7.5, we look at how the running time of ARGOS is distributed among the various pieces. Section 7.6 focuses on a specific problem and looks in more detail at some results to better understand what is going on.

Because most of the ARGOS runtime is spent calculating labor costs, it is reasonable to ask if there is another objective function that is easier to calculate but approximates labor cost well enough to be used instead. We consider the objective of the resource investment problem in Section 7.7 and see that it cannot consistently be used as a stand-in.[1] Finally, Section 7.9 summarizes our experimental results.

[1] As we will see in Chapter 8, the LCOP objective used by ARGOS is itself a stand-in for the even harder to calculate real-world objective of minimizing *expected* costs.

7.2 Problem Sets

The experiments in this chapter were done using the real scheduling problems from industry described below. They span a range of problem sizes and project durations and originate in different industries and from different project management systems.

- The OB (one boat) problem represents construction of a single ship at an anonymous shipyard. Although the shipyard has two eight-hour shifts per day, we treat each day as a single time unit since durations are specified in days. The amount of resources used by activities in this project have been scaled to protect the original data.

- The WY (whole yard) problem represents a snapshot of much of the scheduled work at the above shipyard over a 14 year span. As for OB, the amount of resources used by activities in this project have been scaled.

- The NSC problem represents a specific project at a different anonymous shipyard. We represent each day (again corresponding to two eight-hour shifts) with a time unit.

- The $KHOV$ problem represents the construction of 50 new houses in a subdivision (each house is identical[2]) by K. Hovnanian Enterprises. Each time unit corresponds to a single 8 hour shift.

- The SC problem represents an oil refinery maintenance project of Synge Energy, a Canadian company. Here activities have durations measured in hours and we represent time units as hours as well. How to correctly represent this type of problem is unresolved (see Section 5.8.2); we handle this by pretending each time unit is a 'day' containing a single one-hour shift.

The shipyard that provided the OB and WY datasets uses the Artemis project management software. The schedules they provided (with which ARGOS is competing) were produced through a lengthy process using the Artemis scheduling tool (performing makespan minimization) together with a lot of hand tweaking. The process takes a number of weeks.

[2]In fact, we received a project representing a single house and, at their suggestion, created a larger project by combining 50 copies.

TABLE 7.1: Dataset characteristics.

	OB	*WY*	*NSC*	*KHOV*	*SC*
Activities	6998	139871	1140	11730	12248
Activities in cycles	598	10025	0	0	0
Resources	125	71[c]	17	42	98
Resource assignments	16530	163911	2133	10404	22060
Person-years[b] of resource use	7047[a]	66523[a]	24	52	152
Constraints	12690	198036	1783	24582	16288
Duration	2787 d	6470 d	171 d	635 d	713 h
Calendars	4	11	4	2	4

(a) Resource usages have been scaled at the request of the shipyard.
(b) Assuming fifty-two 40 hour weeks.
(c) We do not know why *WY* has fewer resources than *OB* since it is a much larger amount of work at the same shipyard. We suspect some of the less important resources are either grouped together or ignored in the *WY* data.

The other datasets come from companies that use Primavera. We do not know how the original schedule for set *NSC* was produced. We produced original schedules for *KHOV* and *SC* using the Primavera leveling functionality.[3]

In Table 7.1 we list the important attributes of each problem. The number of activities ranges from around 1000 (*NSC*) to over 100,000 (*WY*) while the number of resources ranges from 17 to 125. Only the datasets *OB* and *WY* have cyclic temporal constraint graphs; for each, between 5% and 10% of activities are involved in cycles. The average number of resources used per activity ranges from just under 1 (*NSC*) to more than 2 (*OB*). There are between 1 and 3 precedence constraints per activity. The number of time units in a project ranges from 171 (*NSC*) to 6470 (*WY*) and the number of relevant calendars ranges from 2 (*KHOV*) to 11 (*WY*).

7.3 ARGOS Settings

- We use the following resource rates as suggested to us by contacts in the shipyard that sent us datasets *OB* and *WY*:

[3]Primavera's only scheduling mechanism is a makespan minimizing algorithm. To get reasonable results using such an algorithm, resource capacities must be used. We chose capacities that looked reasonable according to visual inspection.

- $B = \$24.00/hr$
- $V = \$12.00/hr$ (time-and-a-half, in other words).
- $U = \$24.00/hr$
- $H = \$2460.00$
- $F = \$3864.00$

Although the relative values of the above costs (overtime cost relative to hire cost, for example) is somewhat important, experiments suggest that there is no need to get these values exactly right in order to achieve good schedules.

- When searching for a start time for activity A_i, we do not try every possibility in A_i's window for speed reasons. We choose times as discussed in Section 5.7 and use probability $p = 0.1$ and radius $r = 4$.
- We arbitrarily impose a 2 hour maximum on the runtime of ARGOS for any problem. The larger problems can run for days otherwise and we suspect that schedulers in industry cannot afford to wait longer than 2 hours in many cases.

7.4 Basic Results

In Tables 7.2 through 7.6, we show ARGOS results for each dataset. For each, we show the results for eight configurations; the four ARGOS versions each with and without bulldozing. For each configuration, we report the average of five runs. For each dataset, we list the costs of the original schedule and the costs of the ARGOS schedule paired with the percentage improvements made by ARGOS. In addition, we list the running time and the number of iterations completed for each run. All experiments run on a 1700 MHz Pentium 4 laptop with 384 MB of memory.

See Table 7.7 for a summary of the ARGOS savings. Within the 2 hour cutoff time, ARGOS reduces the overall schedule costs by 3.8% (WY) to 48% ($KHOV$). The low percentage for WY is partly due to the fact that the problem is so big that only a single iteration of polishing can complete within 2 hours. Argos can increase that number to above 5% if given enough time (more than a day).

Recall that the base cost is necessary while the other costs are excess costs due to scheduling inefficiencies. If we ignore base costs and consider only the excess costs,

TABLE 7.2: ARGOS results for problem *OB*.

Schedule	Bulldozing	Iters	Seconds	Costs ($10,000s) / Savings Relative To Original					
				Base	OT	UT	Hire	Fire	Total
Original	-	-	-	35180.1	1860.6	2284.9	2447.0	3959.1	45731.6
ARGOS1	no	2	228.2	35180.1	1080.8	1015.4	1512.4	2430.6	41218.6
				0%	41.9%	55.6%	38.2%	38.6%	9.9%
ARGOS1	yes	2	3440.3	35180.1	1002.6	952.2	1372.1	2181.9	40689.0
				0%	46.1%	58.3%	43.9%	44.9%	11.0%
ARGOS2	no	17	1862.8	35180.1	887.6	793.3	1427.8	2263.6	40552.4
				0%	52.3%	65.3%	41.7%	42.8%	11.3%
ARGOS2	yes	3	7200	35180.1	934.8	858.6	1363.0	2164.8	40501.2
				0%	49.8%	62.4%	44.3%	45.3%	11.4%
ARGOS3	no	65.4	7200	35180.1	802.2	699.0	1364.7	2158.0	40203.9
				0%	56.9%	69.4%	44.2%	45.5%	12.1%
ARGOS3	yes	2.8	7200	35180.1	950.8	833.1	1385.2	2201.6	40550.9
				0%	48.9%	63.5%	43.4%	44.4%	11.3%
ARGOS4	no	65	7200	35180.1	780.7	701.9	1355.8	2146.8	40165.2
				0%	58.0%	69.3%	44.6%	45.8%	12.2%
ARGOS4	yes	3	7200	35180.1	948.2	833.2	1372.0	2178.8	40512.4
				0%	49.0%	63.5%	43.9%	45.0%	11.4%

TABLE 7.3: ARGOS results for problem *WY*.

Schedule	Bulldozing	Iters	Seconds	Costs ($10,000s) / Savings Relative To Original					
				Base	OT	UT	Hire	Fire	Total
Original	-	-	-	332085	11313.2	8761.2	7918.5	12438.2	372516
ARGOS1	no	1	7200	332085	7496.3	5389.1	5247.0	8241.6	358459
				0%	33.7%	38.5%	33.7%	33.7%	3.8%

TABLE 7.4: ARGOS results for problem *NSC*.

Schedule	Bulldozing	Iters	Seconds	Costs ($10,000s) / Savings Relative To Original					
				Base	OT	UT	Hire	Fire	Total
Original	-	-	-	117.5	9.4	21.8	32.5	54.5	235.6
ARGOS1	no	2	2.7	117.5	8.3	11.8	22.9	38.6	199.1
				0%	12.3%	45.6%	29.5%	29.1%	15.5%
ARGOS1	yes	2	9.4	117.5	8.3	11.0	20.6	35.5	192.9
				0%	11.6%	49.2%	36.7%	34.8%	18.1%
ARGOS2	no	17	24.3	117.5	7.8	11.2	20.5	35.0	192.0
				0%	16.9%	48.4%	37.0%	35.7%	18.5 %
ARGOS2	yes	17	60.3	117.5	7.6	10.3	19.7	34.4	189.5
				0%	19.3%	52.5%	39.4%	36.9%	19.6%
ARGOS3	no	304.8	436.1	117.5	7.4	9.3	19.3	33.8	187.4
				0%	21.8%	57.1%	40.5%	37.9%	20.5%
ARGOS3	yes	328.8	1049.2	117.5	8.3	9.2	18.6	32.8	186.4
				0%	12.5%	57.5%	42.6%	39.9%	20.9%
ARGOS4	no	510	734.1	117.5	7.5	8.4	19.7	33.2	186.3
				0%	20.4%	61.3%	39.2%	39.0%	20.9%
ARGOS4	yes	526.8	1835.1	117.5	8.5	8.5	18.5	32.0	185.1
				0%	9.6%	60.7%	42.9%	41.3%	21.4%

TABLE 7.5: ARGOS results for problem *KHOV*.

Schedule	Bulldozing	Iters	Seconds	Costs ($10,000s) / Savings Relative To Original					
				Base	OT	UT	Hire	Fire	Total
Original	-	-	-	256.6	28.5	162.3	75.8	154.2	677.3
ARGOS1	no	2	16.9	256.6	11.9	46.7	55.7	106.7	477.7
				0%	58.3%	71.2%	26.4%	30.8%	29.5%
ARGOS1	yes	2	210.9	256.6	8.6	22.8	41.8	83.8	413.5
				0%	70.0%	86.0%	44.9%	45.6%	38.9%
ARGOS2	no	17	301.6	256.6	3.7	73.3	51.5	90.0	475.0
				0%	87.2%	54.9%	32.1%	41.6%	29.9%
ARGOS2	yes	17	1429.7	256.6	4.3	14.1	28.4	54.4	357.8
				0%	84.9%	91.3%	62.5%	64.7%	47.2%
ARGOS3	no	624.4	7084.9	256.6	2.5	30.7	22.4	41.7	353.9
				0%	91.3%	81.1%	70.5%	72.9%	47.8%
ARGOS3	yes	36	7200	256.6	4.4	18.6	27.4	52.8	359.7
				0%	84.7%	88.6%	63.8%	65.8%	46.9%
ARGOS4	no	612.2	7200	256.6	3.1	40.1	20.8	39.3	359.8
				0%	89.3%	75.3%	72.5%	74.5%	46.9%
ARGOS4	yes	41	7200	256.6	4.9	21.0	23.5	46.4	352.4
				0%	83.0%	87.1%	69.0%	69.9%	48.0%

TABLE 7.6: ARGOS results for problem *SC*.

Schedule	Bulldozing	Iters	Seconds	Costs ($10,000s) / Savings Relative To Original					
				Base	OT	UT	Hire	Fire	Total
Original	-	-	-	758.8	53.1	349.2	130.6	427.0	1718.7
ARGOS1	no	2	60.7	758.8	39.4	291.4	95.5	362.9	1548.0
				0%	25.9%	16.6%	26.9%	15.0%	9.9%
ARGOS1	yes	2	1164.8	758.8	40.2	264.8	79.9	324.3	1468.0
				0%	24.3%	24.2%	38.9%	24.1%	14.6%
ARGOS2	no	17	720.1	758.8	32.7	279.6	86.1	340.7	1497.9
				0%	38.6%	19.9%	34.0%	20.2%	12.8%
ARGOS2	yes	16.6	6989.3	758.8	42.3	249.1	63.5	298.3	1412.0
				0%	20.4%	28.7%	51.4%	30.1%	17.8%
ARGOS3	no	203.8	7200	758.8	37.8	257.4	69.7	302.6	1426.3
				0%	28.9%	26.3%	46.7%	29.1%	17.0%
ARGOS3	yes	13.4	7200	758.8	40.4	250.0	66.2	602.6	1418.0
				0%	23.9%	28.4%	49.3%	29.1%	17.5%
ARGOS4	no	191.2	7200	758.8	36.1	269.2	69.2	306.8	1440.1
				0%	32.0%	22.9%	47.0%	28.1%	16.2%
ARGOS4	yes	13.2	7200	758.8	38.7	250.7	65.3	303.4	1416.9
				0%	27.1%	28.2%	50.0%	28.9%	17.6%

TABLE 7.7: Best average ARGOS results: overall savings and savings without base costs (excess costs only).

Problem	Overall savings	Excess cost savings
OB	12.2%	52.4%
WY	3.8%	34.8%
NCS	21.4%	42.8%
KHOV	48.0%	77.2%
SC	17.6%	31.4%

ARGOS is able to significantly reduce those costs, saving between 31.4% (*SC*) and 77.2% (*KHOV*) of the excess costs of the original schedules.

Here are a number of other observations that can be made:

- Bulldozing gets mixed results. It is slightly better for *KHOV* and *SC*; for *KHOV*, for example, ARGOS4 is able to get a better score in 41 iterations with bulldozing than in over 600 iterations without bulldozing.

  Bulldozing is not as effective for *OB*. While it still produces better results per iteration, this is outweighed by the extra computation time required per iteration. When *WY* is allowed to run beyond the cutoff time, similar behavior can be observed: although it takes roughly 8 iterations without bulldozing to produce a schedule of the quality seen after a single iteration with bulldozing, the 8 iterations take less time than the single iteration of bulldozing.

  The *NSC* results are inconclusive. Each version of ARGOS is able to achieve better results with bulldozing but it takes much longer. However, the results for ARGOS4 suggest that if time is not a factor, bulldozing may be beneficial simply because it is able to explore the search space more thoroughly.

- On problems *OB*, *WY* and *NSC*, ARGOS reduces the undertime cost by significantly more than the overtime cost. This makes sense because undertime is more expensive; with our settings, a unit of undertime wastes $24.00 while a unit of overtime wastes only $12.00 (since the other $24.00 of overtime pay goes toward real work).

  It is interesting that for problems *KHOV* and *SC*, and ARGOS versions 2, 3 and 4, ARGOS decreases undertime more than overtime only when bulldozing is used. Since these tend to be the runs that get the best results, this may suggest that reducing undertime is important.

- On the one hand, ARGOS is able to produce high quality schedules within a very small number of iterations. The most extreme case occurs with *OB* where

two iterations without bulldozing (in 4 minutes) achieves 80% of the savings achieved in over 60 iterations (in the full 2 hours).

This suggests that ARGOS could be effective as a scenario evaluation tool even in industrial settings with large projects. That is, a scheduler could easily perform a number of 'what if' experiments and run a few iterations of ARGOS on each to get a reasonable idea of overall cost.

- On the other hand, ARGOS continues to slowly make progress on problems for quite some time. Only the *NSC* runs with ARGOS4 had enough time to complete; they stopped making improvements after around 500 iterations. For all other problems, ARGOS4 continued to make progress until the cutoff time was reached.

7.5 Where Does The Time Go?

To get some idea of what parts of ARGOS are costly, Table 7.8 summarizes the results of using the Gnu profiling tool. For each problem, ARGOS2 was run both with and without bulldozing.[4] The ARGOS runtime is broken down into the following pieces:

- **Start** shows the percentage of time used to select start times for activities. For bulldozing, this includes a lot of bookkeeping code in addition to calculating the costs of possible start times.
- **Cost** is a sum of two parts: the percentage of time used to calculate the costs of possible activity start times (procedure FIND-BEST-START-TIME-IMPROVED) and the percentage used to calculate the overall score of a given schedule (the procedure FIND-MIN-COST). The former is the main piece of the ARGOS inner loop.
- **Window** shows the percentage of time used to manage time windows and corresponds mostly to procedures PLACE-ACTIVITY and UNPLACE-ACTIVITY (the latter is more expensive).
- **Resource** shows the percentage of time used to maintain and query work profiles during search. These get updated each time an activity is scheduled or unscheduled. When bulldozing, this includes time to create multiple copies of the work profiles (one for each possible start time we are considering).

[4]For *WY*, we used a large cutoff time of 50,000 seconds because profiling executables are much slower than regular ones. Therefore, the time spent in various techniques was slightly different for this dataset.

TABLE 7.8: Profile results for ARGOS2, with and without bulldozing: the percentage of overall run time spent in various pieces.

Problem	Bulldozing	Start	Cost	Window	Resource	Calendar
OB	no	96.0%	79.9%	0.3%	1.0%	0.0%
	yes	99.8%	12.8%	0.8%	85.3%	0.1%
WY	no	99.9%	99.0%	0.0%	0.0%	0.0%
	yes	99.3%	80.0%	2.6%	16.2%	0.3%
NSC	no	93.8%	67.5%	2.8%	3.3%	1.1%
	yes	97.9%	63.7%	7.7%	15.6%	2.0%
KHOV	no	94.6%	55.8%	2.4%	2.7%	0.4%
	yes	99.8%	76.4%	3.9%	8.7%	0.8%
SC	no	95.4%	65.7%	1.9%	1.8%	0.5%
	yes	99.9%	85.2%	2.1%	11.3%	0.2%

- **Calendar** shows the fraction of time used to convert from one calendar to another (when propagating a temporal constraint between two activities with different calendars, for example). This corresponds to solving the edge function inequalities described in Chapter 3.

It should be clear that the above pieces are by no means disjoint. For example, the FIND-BEST-START-TIME-IMPROVED method is always called during start time selection and time spent in it is therefore included in both the Start and Cost columns. Another example is the calendar conversions which are done within time window management; when bulldozing, this occurs within start time selection as well.

Also, although we have separated the calendar conversion piece, the other pieces are also generalized to handle calendars and are all somewhat slower as a result. For example, adjusting a work profile for an activity involves more work when the activity's span includes time points during which it does not work. It is therefore difficult to measure how much slower ARGOS is as a result of calendar issues. A reasonable prediction would be a difference comparable to that observed for $SWO(B, R)$ in Section 4.4.3 where we compared our algorithm with and without calendar generalizations.

In almost all cases, the majority of the computation time is spent calculating optimal staffing profiles and their associated costs. This is especially true when

bulldozing is off and indicates how important it is to use efficient cost calculation algorithms.

When bulldozing is turned on, the associated bookkeeping costs increase; much of this bookkeeping has to do with the maintenance of work profiles. Specifically, for each bulldoze considered, entire copies of the resource profiles must be made to enable the possible bulldozes to be done in parallel (this is why the cutoff of 5 possible bulldozes at a time is important). This is especially costly for problem OB, probably because it has the highest number of resources.

The time spent maintaining windows and making calendar calculations is relatively small. This suggests that the lookahead temporal constraint propagation is well worth the effort and that the GSTP implementation is effective.[5]

There are two obvious areas for improvement suggested by the table. First, it is possible that further improvements can be made to the cost calculation algorithms of Chapter 5. Second, there are a number of possible ways that making copies of work profiles during bulldozing can be improved or avoided.

Because ARGOS is designed to handle real-world problems, scalability is crucial. However, measuring scalability is difficult. At first glance, we might predict the running time of an iteration to be linear in the number of activities since each iteration steps through the activities and attempts to schedule or reschedule each one. However, this is obviously not the whole story since $KHOV$ and SC have twice as many activities as OB yet complete their iterations in a fraction of the time.

Similarly, running time does not seem to be well correlated with most of the other characteristics listed in Table 7.1. Another possibility would be a correlation with the size of activity time windows since the number of start times considered for an activity is roughly proportional to the number of possibilities. This is no better than other measures, however, as OB has smaller windows on average than both $KHOV$ and SC.

The best correlation with running time we have found is the total amount of

[5]Of course, it is possible that these numbers are low only because they are dwarfed by the cost calculations. Therefore, we might want to find ways to improve them when solving problems with simpler objective functions.

resource use in a project (the person-years row in Table 7.1). This is the only measurement for which the datasets are ranked in the same order as they are for running time (the problem with the lowest resource use has the fastest time per iteration and so on). This can be explained by the fact that most of the computation time is used calculating costs and the dynamic programming, despite the efficient implementation, will be affected by the height of the work profiles.

Total resource use could only be used as a rough predictor of running time and we cannot provide an equation that approximates the scalability. To come up with a good correlation would require many more datasets with a variety of characteristics.

7.6 An Example in Depth

We now look more closely at the ARGOS results for problem *NSC* to get a better sense of what is going on. We choose this problem only because it is small: the characteristics we point out can be observed with all problems. We focus on two of the runs done with ARGOS4; one with bulldozing and one without.

First we compare the original schedule with the schedule produced by the bulldozing run. ARGOS reports that the latter is 21.2% less costly.

In Figure 7.1, we compare the cumulative work levels of the two schedules. In Figure 7.2 we compare the work levels of resource 81, the largest resource pool used by this project.[6] Clearly, ARGOS has discovered a schedule that levels this resource, as well as the whole project, more evenly over time.

Resource 81 costs 19.4% less in the ARGOS schedule than it does in the original. To see where the money is saved for this resource, Figures 7.3 and 7.4 show the work levels of each schedule with the corresponding optimal staffing levels (as calculated by the dynamic programming). Recall that extra expenses are incurred when the staffing level increases (hire) or decreases (fire) and when the work profile is above (overtime) or below (undertime) the work level.

Inspection of the graphs shows that the ARGOS schedule has significantly lower

[6]Due to the desire of the shipyard to protect their data, we have no idea what trade this resource is.

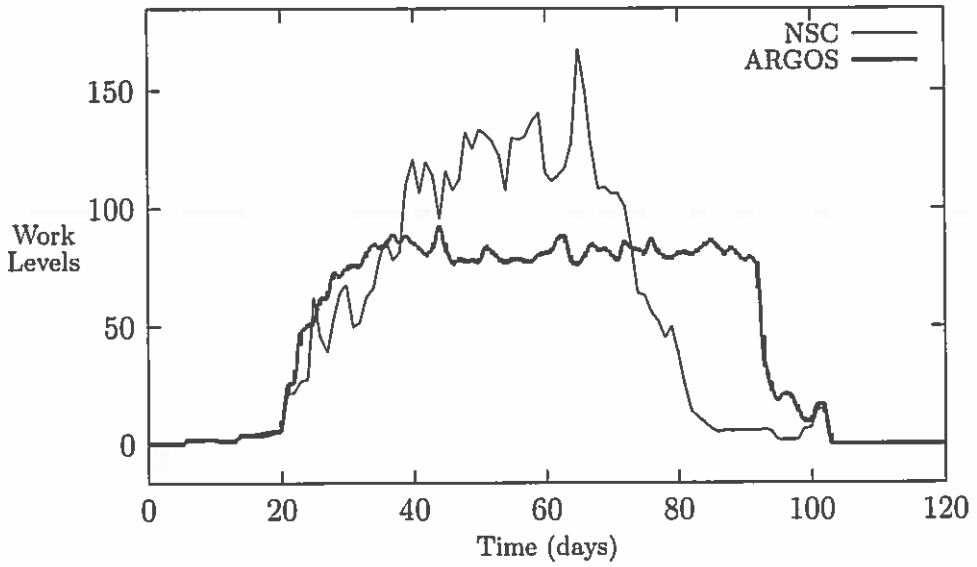


FIGURE 7.1: Cumulative work levels: an ARGOS schedule compared with the original *NSC* schedule.

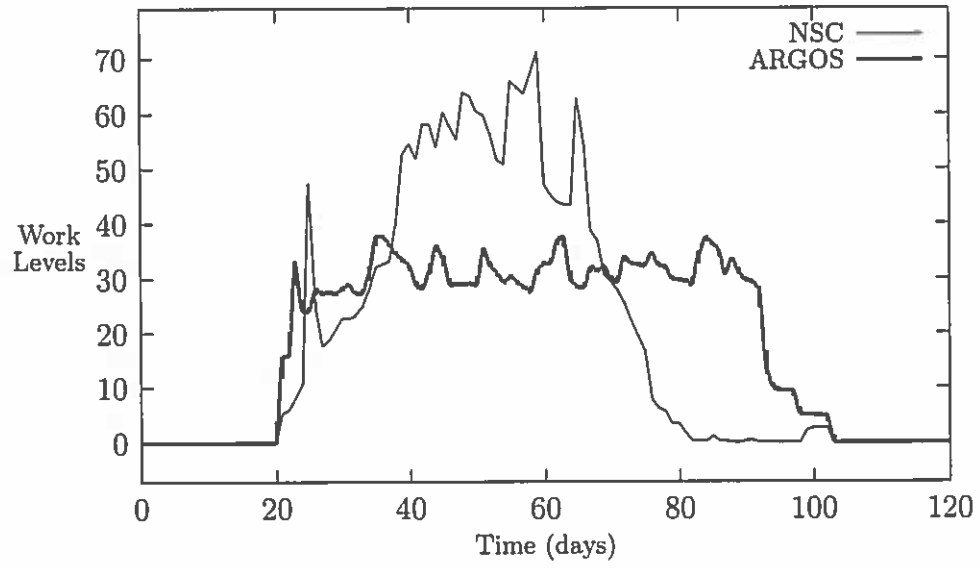


FIGURE 7.2: Work levels of resource 81: an ARGOS schedule compared with the original *NSC* schedule.

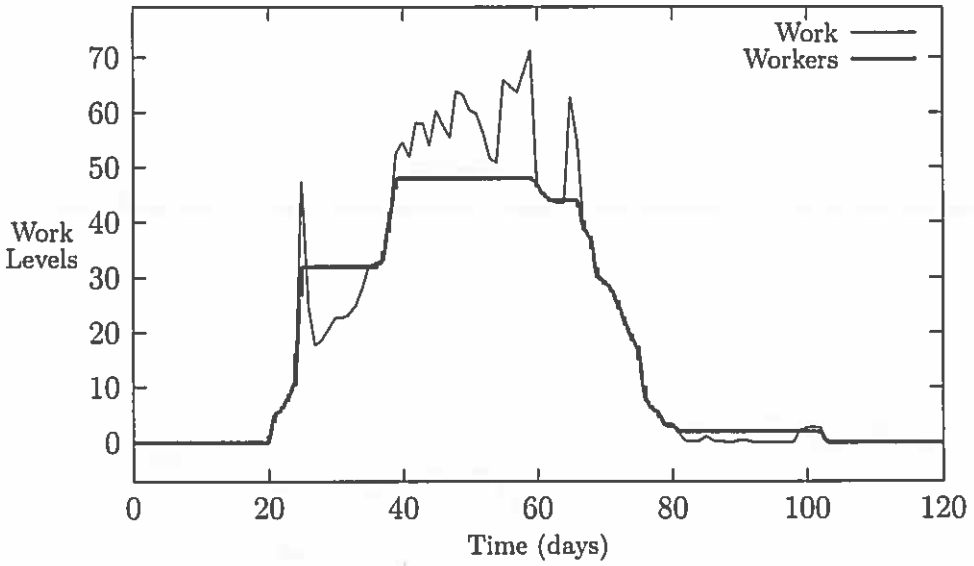


FIGURE 7.3: Original schedule for resource 81 of problem NSC: work levels compared with staffing levels.

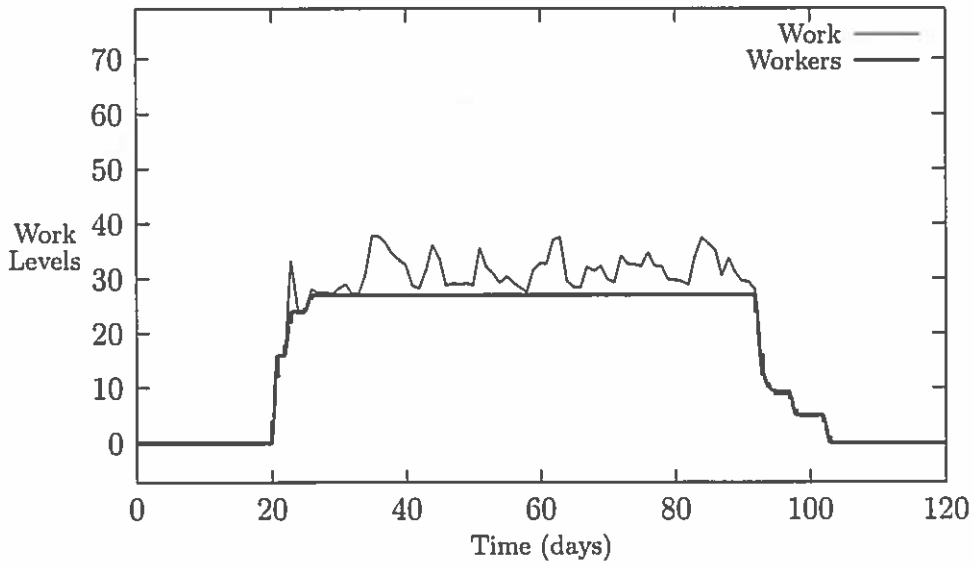


FIGURE 7.4: ARGOS schedule for resource 81 of problem NSC: work levels compared with staffing levels.

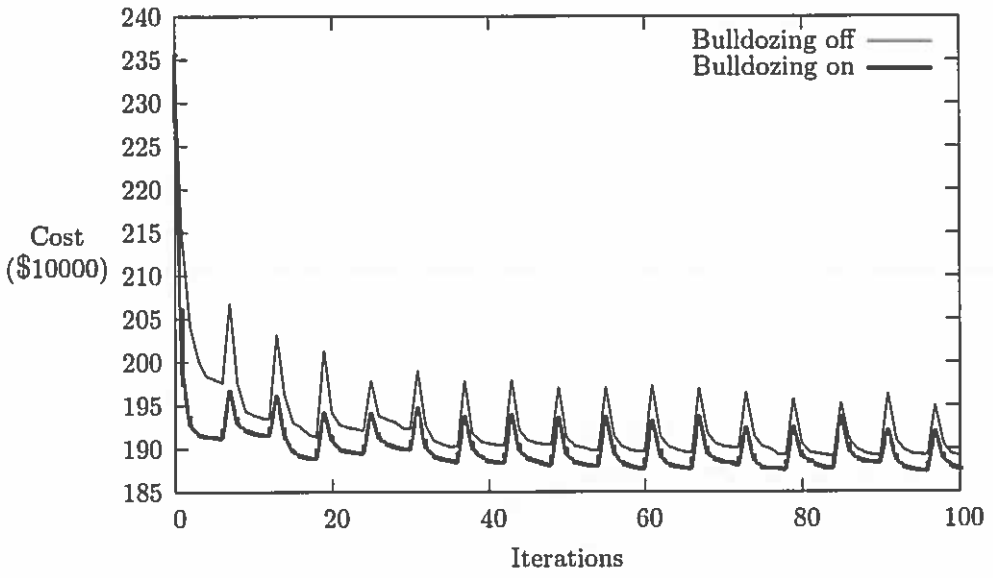


FIGURE 7.5: Schedule quality by iteration for ARGOS4 with and without bulldozing.

costs in each of the above categories except overtime. In fact, the overtime cost has increased by 5.3% while hire and fire costs have decreased by 43.8%, and undertime cost has been virtually eliminated (reduced by 99.5%).

Figure 7.5 shows the quality of schedules produced by ARGOS4 by iteration and compares results with and without bulldozing. Only the first 100 iterations are shown for clarity; the trend continues out to 500 iterations where both runs stop finding better results.

Although the results per iteration are better with bulldozing, bulldozing is slower. Figure 7.6 shows the same runs over time. This makes it unclear which is preferable. As we saw in our overall results, whether the extra effort of bulldozing is worthwhile depends on the specific problem as well as on the importance of overall runtime.

In figures 7.5 and 7.6 we can see the results of the various pieces of ARGOS4. Each peak corresponds to a large move in schedule space. The first five such peaks are due to morphing and the rest to annealing. After each peak, we can see that polishing improves the schedule with diminishing returns over five polishing iterations.

Clearly, ARGOS is able to achieve a huge fraction of its overall savings in the first

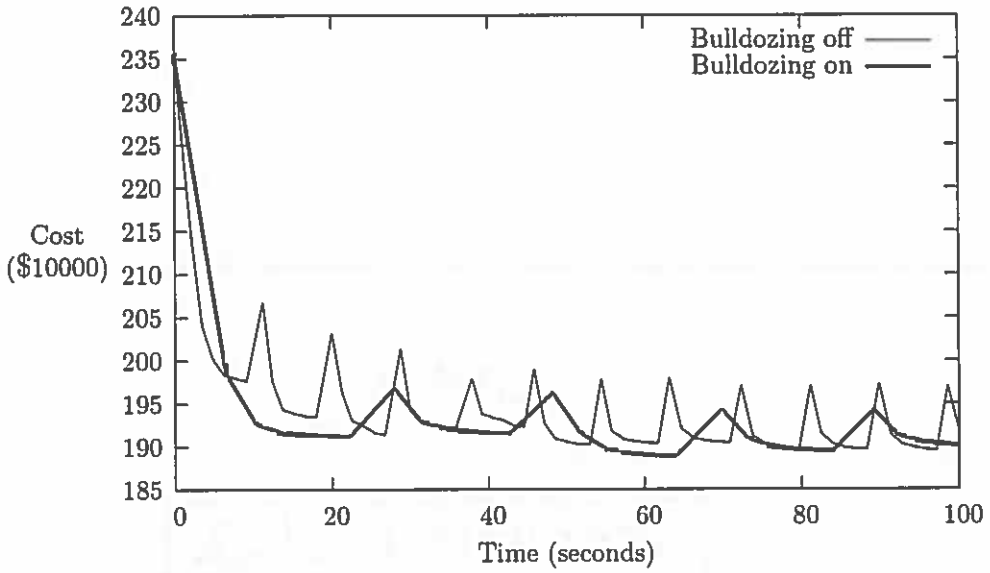


FIGURE 7.6: Schedule quality over time for ARGOS4 with and without bulldozing.

few iterations. However, letting the algorithm run its course is still beneficial if time allows.

7.6.1 Float

The first concern of schedulers in industry when ARGOS is presented is that float should be preserved. Due to the unexpected problems and delays that occur in the real world, they are wary of postponing work for fear that a schedule will become over-constrained in its later stages.

As described in Section 5.8.1, extra costs can be added to the labor costs in ARGOS to give preference to schedules with float.

In Table 7.9, we show results, for the *NSC* problem, of ARGOS runs using different values for the float parameters a (cost of zero float) and b (amount of float that is twice as good as no float) as shown in equation (5.9) in Chapter 5. The three runs are different ways to trade off the objectives of decreasing cost and increasing float. The first run (A) does not include float penalties and is only minimizing cost. In run B, float costs are somewhat important but are only a fraction of the overall cost.

TABLE 7.9: Two runs of float relative to a run without float. Savings on float and other costs relative to the original schedule are presented for each run.

Schedule	Float Parameters		Float Cost		Other Costs	
	a	b	\$10,000s	savings	\$10,000s	savings
ARGOS A	0	0	0	0%	186.6	20.8%
ARGOS B	10	5	78.0	23.7%	193.7	17.8%
ARGOS C	50	5	370.2	27.6%	202.5	14.0%

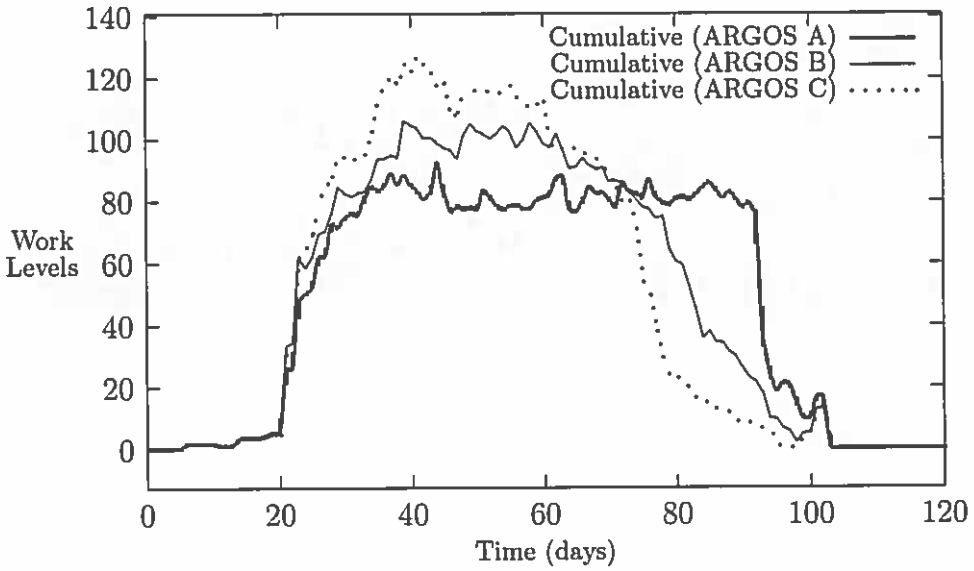


FIGURE 7.7: Cumulative work levels: 3 versions of ARGOS with different emphases on float.

Finally, in run C, float is emphasized and the overall float penalties are higher than the overall labor costs.

In Figure 7.7, we compare the cumulative work levels for each of the schedules in Table 7.9 and Figure 7.8 shows the same thing for resource 81. Notice that the schedules become increasingly left shifted as the importance of float is increased but remain relatively smooth compared with the original NSC schedule depicted in Figures 7.1 and 7.2.

In Figure 7.9, we show the number of days of work being done on activities with different amounts of float in the NSC schedule and ARGOS schedules A,B, and C. Work is divided into buckets of size 10; the y-intercept values correspond to all work

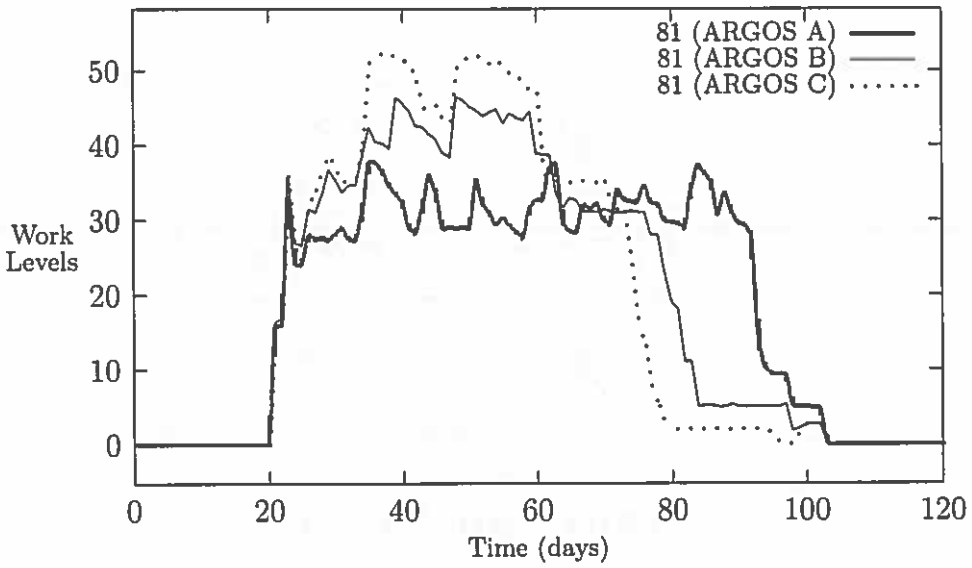


FIGURE 7.8: Work levels of resource 81: 3 versions of ARGOS with different emphases on float.

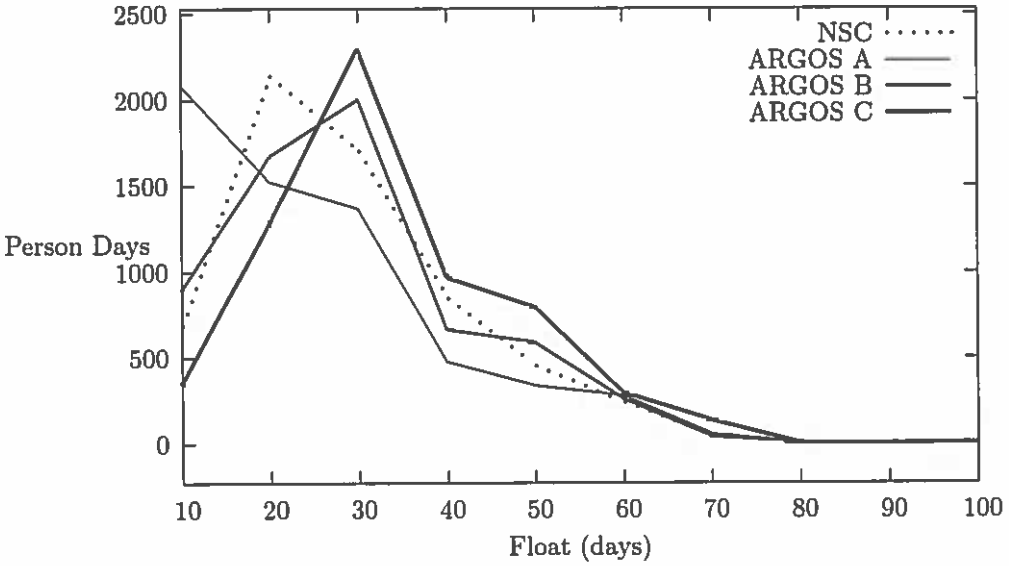


FIGURE 7.9: Total work broken down by the amount of float available for that work: 3 ARGOS schedules and the original *NSC* schedule. Work is divided into buckets of size 10; the y-intercept values correspond to all work done with 10 or fewer days of float.

done with 10 or fewer days of float. Notice that schedule A has more 'critical' work (work with very little float) than the NSC schedule, schedule B is similar to the NSC schedule and schedule C is the best overall (it has the least amount of work on activities with 20 or fewer days of float).

7.7 A Comparison With Indirect Optimization

The huge majority of the ARGOS runtime involves cost calculations. Therefore, it might be possible to optimize with a simpler metric that, while easier to calculate, still roughly measures cost. If improvements in that metric correspond to reductions in cost, the ability to perform more iterations might lead to better overall results.

A candidate metric is that of the resource investment problem (RIP) where the goal is to minimize the sum of the maximum resource levels over all resources. To understand whether this metric could serve as a stand-in for labor costs, we used Procrustes, an existing RIP algorithm designed for large-scale RIP instances,[7] on our datasets.

Procrustes begins with a high constant capacity c_k for each resource k. Each iteration of Procrustes randomly chooses a resource k (with probability proportional to the distance of c_k from its lower bound), lowers c_k and performs three iterations of an algorithm similar to schedule packing [31] to see if a schedule can be found that satisfies all of the resource capacities. If it succeeds, the new c_k is kept and the algorithm continues. Otherwise, c_k is returned to its previous level, resource k is marked as finished, and search continues.

Table 7.10 lists the results of the Procrustes for our datasets according to the RIP objective. Each is the average of 5 runs and shows Σc_k in the original schedule and after optimization as well as the percent reduction. Procrustes was able to reduce Σc_k by between 17.6% (OB) and 82.5% ($KHOV$).

In Table 7.11, we show the results of the Procrustes runs according to our original metric (labor cost). Negative values represent costs for which the Procrustes schedule

[7] In fact, Procrustes was our first attempt to solve the shipyard scheduling problems. Originally, we were asked to level the resources and only later realized that the real objective was to lower cost.

TABLE 7.10: Procrustes results.

Problem	Seconds[a]	Σc_k Original	Σc_k Final	Improvement
OB	10463	6882.8	5672	17.6%
WY	117724	18900.6	13929.6	26.3%
NSC	23.2	192.3	111.2	42.2%
KHOV	21782.2	399	70	82.5%
SC	13083	1569.4	1039.4	33.8%

(a) These runs were done on a number of computers and are not directly comparable to those for ARGOS results. The machines used here are slightly faster on average.

TABLE 7.11: Costs for Procrustes schedules (negative values mean the Procrustes schedule are more expensive than the original).

Problem	Costs ($10,000s) / Savings Relative To Original					
	Base	OT	UT	Hire	Fire	Total
OB	35180.1	1761.4	2390.8	2425.1	3970.1	45731.6
	0%	5.3%	-4.6%	0.9%	-0.3%	0%
WY	332084.7	12824.2	11328.0	9040.3	17491.2	382768.0
	0%	-13.4%	-29.3%	-14.2%	-40.6%	-2.8%
NSC	117.5	12.9	17.3	18.7	32.9	199.3
	0%	-36.7%	20.6%	42.3%	39.6%	15.4%
KHOV	256.5	5.9	139.8	14.6	31.9	448.9
	0%	79.5%	13.8%	80.6%	79.2%	33.7%
SC	758.8	82.2	263.8	74.7	282.5	1462.0
	0%	-54.9%	24.5%	42.8%	33.8%	14.9%

is more expensive (by the given percentage).

The results suggest that the RIP objective is not an effective stand-in for the LCOP objective. For two problem sets (*OB* and *WY*), Procrustes has made no decrease in overall labor costs (for *WY*, it significantly increased most of the costs). For the others, costs have been improved by between 14.9% (*SC*) and 33.7% (*KHOV*). However, even for these problems, ARGOS is able to achieve better results in less time.

7.8 ARGOS on RCPSP/max Problems

In Section 5.8 we mentioned that the labor cost framework can be adapted for non-labor resources and could even be used to handle RCPSP/max problems. Therefore, we can run ARGOS on these problems.

For ARGOS to correctly handle RCPSP/max problems, we adapt it as follows:

- Resource constraints are represented using maximum staffing levels so that ARGOS is not allowed to hire above the given resource constraint.
- Overtime is disallowed by setting the maximum overtime rate to 0.
- We use a finite penalty cost, $C_{penalty} = 20000$, to avoid infinite costs when a schedule has a work profile that exceeds the maximum staffing level.
- We choose to set $B = 1$, $U = 0$, and $H = F = \infty$. The result is that ARGOS will keep the staffing levels constant (probably at the maximum staffing levels). Therefore, the resulting costs of a schedule will only involve the base cost and any penalty costs (incurred if the maximum staffing levels are exceeded).

Using ARGOS with the above settings means that a schedule is feasible[8] if and only if it has no penalty overtime costs.

ARGOS was not designed for RCPSP/max problems and there are a few important reasons to expect it to perform poorly:

- **Unnecessary computation:** ARGOS will be slow because it is constantly computing the value of an objective function that is time-consuming to calculate. To choose a start time for activity A_i, ARGOS will compute labor costs (using PWLC functions) for a subset of possible start time and will choose the one with lowest cost. In contrast, an algorithm designed for RCPSP/max problems will simply compare the work profile with the resource constraint and find the earliest time where A_i will fit.
- **Inappropriate search:** the ARGOS algorithms were designed for problems without resource constraints; there is no notion of resolving resource conflicts. Consider a schedule with a single activity that is not resource-feasible. ARGOS will not pay this activity any more attention than the others because it does not realize that this is the one problem keeping it from producing a feasible schedule.

[8] ARGOS will always create time-feasible schedules but may have trouble finding resource-feasible ones.

TABLE 7.12: ARGOS results for RCPSP/max problems.

Set	Algorithm	Bulldozing	Float	$\Delta_{LB}\%$	$\%_{opt}$	$\%_{feas}$	Scaled Cpu_{sec}
J10	SWO(B,R)	-	-	0.2	94.0	100	0.31
	B&B$_{S98}$	-	-	0.0	100	100	-
	ISES	-	-	1.3[a]	85.9	99.5	0.08
	ARGOS4	no	no	271.4[a]	0	98.9	8.49
	ARGOS4	no	yes	10.7[a]	43.9	98.9	8.68
	ARGOS4	yes	no	271.6	0	100	25.96
	ARGOS4	yes	yes	10.3	40.1	100	30.05
J20	SWO(B,R)	-	-	4.9	66.4	100	0.63
	B&B$_{S98}$	-	-	4.3	85.3	100	-
	ISES	-	-	5.4	64	100	0.53
	ARGOS4	no	no	546.2	0	100	22.02
	ARGOS4	no	yes	16.3[a]	31.0	91.8	26.70
	ARGOS4	yes	no	552.0	0	100	104.74
	ARGOS4	yes	yes	17.9[a]	27.7	98.9	150.36

(a) Not directly comparable to other numbers since problems not feasibly solved are excluded.

- **Different objective:** the objective for RCPSP/max problems is makespan minimization but ARGOS searches for problems with lower costs and does not, by default, pay attention to makespan.

This last problem can be handled by using float costs in ARGOS; by encouraging ARGOS to schedule activities early, we can indirectly encourage it to minimize makespan.

We have run ARGOS4 on the *J10* and *j20* benchmark suite introduced in Chapter 4. We have run it with and without float costs and with and without bulldozing. Results are summarized in Table 7.12 where the ARGOS4 results are compared with state-of-the-art approaches.

Although ARGOS4 does not perform as well as the algorithms designed for these problems, it is able to handle them reasonably well. Here are some observations that can be made:

- Bulldozing enables ARGOS4 to find feasible solutions to all feasible problems (although including float costs kept it from finding feasible solutions for two of the *J20* instances). This was surprising; it is NP-hard to find feasible solutions

and, as mentioned above, the ARGOS algorithms are not designed to search for them.

- As the results without float costs clearly show, ARGOS4 is not paying attention to makespan. However, with float costs included, it does a respectable job at finding short schedules.[9]
- As expected, the runtime of ARGOS4 is much greater than that of other approaches.

7.9 Conclusions

This chapter described results of running the ARGOS algorithms on a number of real-world problems. ARGOS can handle even the largest problems and reduces excess costs by between 31.4% and 77.2% within the 2 hour time limit.[10] Bulldozing increases the ARGOS runtime considerably but is worthwhile for some problems.

On all problems, most of the runtime is devoted to the dynamic programming algorithms that calculate costs; when bulldozing is turned on, it also adds significant bookkeeping costs to the overall runtime.

ARGOS was compared with Procrustes, an algorithm geared toward the RIP objective. Results of this comparison suggest this objective cannot serve as an effective stand-in for total labor cost. We do not know if other objectives could serve that purpose instead.

Finally, ARGOS was run on small RCPSP/max problems. While it does not do as well as state-of-the-art approaches designed for those problems (including the SWO algorithms described in Chapter 4), it is able to find feasible solutions to all feasible problems and float costs are shown to serve as a effective way to encourage ARGOS to minimize makespan.

There are a number of ways that the ARGOS algorithms could be improved. One is to increase the efficiency of the current implementation. Bulldozing is one area

[9]Another way to encourage short schedules, that might be interesting to try, is to use costs $U > 0$, $H = \infty$ and $F = 0$. This would make shorter schedules preferable since they would have lower undertime costs.

[10]On a number of other problems from industry, ARGOS achieves comparable results.

where this is likely to be fruitful. The other possibility is to improve the algorithms themselves. One possibility is the use of Tabu search [46] to explore the search space more effectively. While simulated annealing allows ARGOS to avoid getting stuck in local minima, it may be that the same local minima are visited multiple times. It might be helpful to use the Tabu notion of diversification to ensure that the local search constantly explores new parts of the search space. This could be especially important for ARGOS because the cost calculations are time consuming and therefore the number of points in the search space that can be visited are limited.

CHAPTER 8

SimYard: The Effects of Real-World Complications

In the previous chapter, we saw that ARGOS can reduce excess project costs by between 31.4% and 77.2% for the problems we considered. These projects are large and these savings correspond to millions of dollars in most cases (ARGOS is able to reduce the cost of even the smallest problem, NSC, by more than $500,000).

While the schedulers who provided us with these problems are impressed with ARGOS results and believe that the schedules produced by ARGOS are better than schedules they can produce in any other way, they also doubt the predicted savings. The problem is that the costs calculated by ARGOS assume that a schedule represents exactly how a project will be accomplished.

There are two main reasons that practice diverges from theory:

1. **Inaccurate Models:** the activities, resources and constraints as recorded in a project management system are only approximations of reality. In addition human misunderstanding or errors can result in missing or incorrect information.
2. **Indeterminacy:** even an accurate model cannot account for unpredictable problems and issues that arise after a project has begun.

We spent time discussing the way scheduling works in practice with a number of people at the shipyard that provided data sets OB and WY. In this chapter we describe SimYard, a shipyard simulator that models their shipyard based on the

information they provided. SimYard allows us to compare projected actual savings of ARGOS schedules with theoretical savings.

The results of SimYard experiments suggest that, although the improvements of ARGOS schedules are likely to be muted by real world issues, the ARGOS schedules will continue to be preferable to the original schedules. As mentioned below, the results in this section are preliminary and intended to be indicative rather than definitive; the expected costs reported must be considered in this light.

In Section 8.1, we describe our understanding of shipyard reality that resulted from our discussions. In Section 8.2, we describe SimYard, a simulation tool that attempts to faithfully model that reality. Section 8.3 presents preliminary experimental results of SimYard on the data sets we have been considering. Finally, Section 8.4 suggests ways to expand this preliminary investigation and Section 8.5 mentions some related work.

8.1 How a Schedule Really Works

In a number of conversations with schedulers and supervisors at the shipyard from which we received data sets OB and WY, we discussed the main issues that lead to differences between a theoretical schedule and what gets done in practice. We now outline those problems and then mention the current ways they are handled.

8.1.1 Problems

Especially when the projects are as large as those represented by OB and WY, the scheduling data cannot accurately reflect the problem at the lowest level of granularity.[1] Much of the activity and resource information will be approximate due both to the need to represent it compactly and to uncertainty.[2] The main differences between theory and practice are the following:

[1] At this shipyard, they have a secondary scheduling tool that handles the fine-grained details. However, this information is not available to ARGOS.

[2] Many large projects are one-time jobs. Unlike factory settings, this means there will be less certainty concerning how long things will take and how many resources they will require.

- The resource use of activities is rarely constant as represented in the model. For example, if an activity requires 300 person-days of work by welders and has a duration of 100 days, it will be represented as an activity using 3 welders on each day. In reality, the number of welders needed per day is likely to vary over the course of the activity.
- The total resources required by each activity will not always be right. Often, more or fewer hours than planned will be used by an activity.
- Activity durations often deviate from planned durations.
- Activities may not be able to start when scheduled. This is often due to parts or instructions that are not ready in time.

Another problem in this shipyard is that staffing levels are subject to variability for reasons other than hire and fire decisions. This can be a result of a number of factors including unplanned absences and workers who retire or change jobs. The main cause of variable availability of workers in this shipyard, however, is the existence of an **external pool** of workers for each resource. This is a group of workers who are working on other projects or assigned to other duties.[3] Fluctuations in the external pool spill over into each project and affect the number of workers available for that project on any given day.

8.1.2 Solutions

All of the above problems must be handled by the schedulers and managers responsible for the success of a project. The first way that the shipyard adapts to scheduling changes is by adjusting the available workforce. While this can't be done on a short-term basis (it is difficult to hire a worker who can begin work immediately), a single project can take years and workforce adjustments help adapt to the evolving schedule.

The second, and most important, way that the shipyard handles problems is through one or more **floor managers** who make final scheduling decisions on a daily

[3]At least a quarter of the work done at this shipyard is not incorporated in the project management systems and the corresponding schedules. This includes daily facility maintenance and other overhead work.

basis. Each floor manager considers the available work force, the currently scheduled activities and those scheduled to begin soon. On each day t, she may decide to delay a subset of activities scheduled to start at t and/or start some activities that were scheduled to begin later.

8.2 SimYard

The SimYard tool is a shipyard simulator that models most[4] of the above inaccuracies and uncertainties in an attempt to quantify the actual costs that will be incurred by following any particular schedule. It can be used to compare theoretical costs with expected actual costs.

8.2.1 Parameters

SimYard includes the following parameters that can be adjusted to simulate different conditions in the shipyard. We give the default value for each as specified by the shipyard in question. While we only consider the default settings in experiments reported here, the parameters are mentioned as they are ideal starting points for sensitivity analysis.

- $\mathbf{N_{future-days-considered}} = \mathbf{30}$ represents the window of time the floor manager has available from which to select activities to reschedule. At each time t, she may select activities to begin at t that were not scheduled to start for up to $N_{future-days-considered}$ days.

- $\mathbf{F_{\sigma-for-activity-durations}} = \mathbf{0.2}$ represents the standard deviation between the original and final durations of an activity, as a fraction of the original duration.

- $\mathbf{N_{frozen-manpower-days}} = \mathbf{30}$ represents the number of time units for which the hire and fire decisions cannot be changed, even if the manpower levels are revisited (see the next parameter). For example, if staffing decisions are updated on day t, the actual staff levels can only be changed for days after $t + 30$. This is used to model the time required to hire and train new workers.

[4]It does not currently model activities that may not be able to start when scheduled; see Section 8.4.

- $\mathbf{N_{hire-fire-granularity} = 30}$ represents the number of time units that pass between each time unit on which the staffing levels can be updated.
- $\mathbf{F_{external-pool} = 1/3}$ represents the size of the external pool for each resource as a fraction of the number of workers required for the project.
- $\mathbf{P_{penalty-overtime-ratio} = 20}$ corresponds to the additional cost incurred for time that the work level exceeds the maximum amount of overtime allowed (where a value of 20 means that the cost is 20 times the base rate). Although this penalty is infinite in ARGOS, it must be finite for SimYard since there will invariably be times at which there are not enough staff (due to unexpected variability).

To represent the uneven resource use of an activity, SimYard creates a random resource use profile. To create this profile, strips of resource use of random length and height are placed randomly between the activity's start and finish until the total use matches the amount required.[5] A uniform distribution is used to determine the length and height of each strip.

The same process is used to create the profile of the external pool. The relevant settings and default values for both situations are:

- $\mathbf{N_{min-strip-length}}$ and $\mathbf{N_{max-strip-length}}$ represent the allowed range for the strip length. For activity A_i, $N_{min-strip-length} = 0$ and $N_{max-strip-length} = dur_{A_i}$. For the external pool, $N_{min-strip-length} = 0$ and $N_{max-strip-length} = t_{end} - t_{begin}$ (the project duration).
- $\mathbf{N_{min-strip-height}}$ and $\mathbf{N_{max-strip-height}}$ represent the allowed range for the strip height. For activities, the default settings are $N_{min-strip-height} = 0$ and $N_{max-strip-height} = avg$ where avg is the amount that would be used by the activity each time unit if the resource use was evenly distributed. For the external pool, $N_{min-strip-height} = N_{max-strip-height} = 1$. Therefore, the variability due to the external pool will not be as severe as the variability in the daily resource use of an activity.

8.2.2 Pseudocode

The SimYard algorithm, SIMULATE-SCHEDULE(S), takes a schedule S as input and chronologically steps through it. At each time unit t it simulates the adjustments that occur due to real life problems and events.

[5] When a strip overlaps the end of an activity, it begins again at the activity's start time.

```
SIMULATE-SCHEDULE(S)
1  for t = 0 to t_end
2       do ADJUST-FOR-EXTERNAL-POOL(t)
3          activities-today = SCHEDULE-DAY(t)
4          UPDATE-PLANNED-WORKFORCE-IF-POSSIBLE(t)
5          UPDATE-ACTIVITY-GRANULARITY(activities-today)
```

In the first step, ADJUST-FOR-EXTERNAL-POOL(t), the staff available at t are adjusted due to the external pool. During initialization, SimYard randomly creates a profile representing the external pool of each resource. By centering that profile around 0, we get the number of workers to add or subtract at each time t.

```
SCHEDULE-DAY(t)
 1  activities-today = ()
 2  for i = 1 to n
 3       do A_i = activities-by-decreasing-size[i]
 4          if start_{A_i} = t
 5             then UNPLACE-ACTIVITY(A_i, W)
 6                  new-start = FIND-BEST-START-TIME(W, A_i, (t, t + 1))
 7                  PLACE-ACTIVITY(A_i, new-start, W)
 8                  if new-start = t
 9                     then add A_i to activities-today
10             else if t < start_{A_i} ≤ t + N_{future-days-considered}
11                  UNPLACE-ACTIVITY(A_i, W)
12                  new-start = FIND-BEST-START-TIME(W, A_i, (t, start_{A_i}))
13                  PLACE-ACTIVITY(A_i, new-start, W)
14                  if new-start = t
15                     then add A_i to activities-today
16  return activities-today
```

The second step, SCHEDULE-DAY(t), represents the floor manager. This procedure is outlined above. Each activity A_i is considered, beginning with the biggest. If A_i is scheduled to start at t, the cost of starting it at t and $t+1$ are compared and the better one is chosen. If it was scheduled to start at some $t' < t+N_{future-days-considered}$, the cost of starting at t' is compared with starting at t and the better one is chosen. The end result of SCHEDULE-DAY is a list of activities that will begin at t.

In the third step, UPDATE-PLANNED-WORKFORCE-IF-POSSIBLE(t), the staffing levels are updated (if allowed) for time units beyond $t + N_{frozen-manpower-days}$. This

is only done once in every $N_{hire-fire-granularity}$ time units. This amounts to freezing the staffing profiles from t to $t + N_{frozen-manpower-days}$ and then calling the procedure FIND-OPTIMAL-STAFFING-PROFILE as described in Chapter 5 to get the desired levels.

In the final step, UPDATE-ACTIVITY-GRANULARITY, the activities that start at t (the list produced by SCHEDULE-DAY) are adjusted to use their actual durations and profiles. This simulates the fact that nobody pays attention to the fine-grained details until it has been decided to begin the work.

There are some subtle issues that arise when changing the duration of an activity. The main one is that a new duration could make the scheduled starts of other activities infeasible due to soft temporal constraints. We do not let this occur — if the random duration chosen will make this happen, we adjust it to avoid the problem and also adjust the resource profiles accordingly. In the shipyard modeled, they work very hard to meet important deadlines and instead add workers to activities that risk missing deadlines. Our approach is an attempt to model that.

8.3 Experimental Results

We have run SimYard on all of the datasets using the default settings.[6] For each problem we have simulated each of the 5 schedules produced by ARGOS4 with bulldozing turned on.[7] Each schedule was simulated 100 times (WY schedules were only simulated 25 times due to problem size) and we report the average for all 500 (125 for WY) simulations.

Tables 8.1 and 8.2 outline the results. There are four results to compare:

1. The theoretical costs of the original schedule.

2. The actual costs of the original schedule after simulation.

[6] Although SimYard is designed only to model the company that produced datasets OB and WY, we use it on the other datasets in the absence of a better model for those companies.

[7] Recall that each of the ARGOS results presented in Chapter 7 reports the average of 5 different runs.

3. The theoretical costs of the ARGOS schedule.

4. The actual costs of the ARGOS schedule after simulation.

In Table 8.1, the breakdown of costs for each is given. In Table 8.2, we use the actual costs of the original schedule as a baseline and compare the other three with that.

Notice that there are two numbers given for total cost. This is due to complications of penalty costs. $P_{penalty-overtime-ratio}$ must be finite or else total costs after simulation will almost always be infinite; there are bound to be days when there aren't enough staff available.

The problem is that FIND-OPTIMAL-STAFFING-PROFILE can exploit any finite cost. For example, if the maximum overtime must be exceeded only by a small amount for a single day, it may be better to pay the penalty cost than to hire new workers. Therefore, we use the high value $P_{penalty-overtime-ratio} = 20$ during simulation so that staffing decisions avoid incurring penalty costs as a cheaper solution.

These high penalties are probably higher than the actual costs incurred for not having enough workers (for example, it may be possible to borrow workers from the external pool). Therefore, the second-last column in Tables 8.1 and 8.2 report the SimYard total costs with the penalty ratio at 20 while the last column reports total costs with that ratio decreased to 1.

Unlike tables in previous chapters, the percentages listed in Table 8.2 compare the costs of the schedule in question relative to the actual costs of the original schedule (rather than comparing the *savings* to the actual costs).

We can make a few observations:

- Unlike previous experiments, base cost actually varies slightly from run to run; this is due to the fact that activity durations can increase and decrease.

- While the totals of ARGOS schedules after simulation are always better than the original schedule, there are a few cases where a specific cost is actually higher for an ARGOS schedule. An example is the overtime cost for problem NSC: after simulation, the ARGOS schedule costs 1.7% more than the original schedule.

TABLE 8.1: SimYard results (dollars).

Problem	Schedule	Costs ($10,000s)							
		Base	OT	UT	Hire	Fire	Penalty	Total	Total(reduced)
OB	Original (theory)	35180.1	1881.0	2119.4	2262.2	3667.3	476.6	45586.6	45133.9
	Original (actual)	35226.7	2690.5	3383.6	2582.8	4170.8	22117.6	70172.0	49160.3
	ARGOS (theory)	35180.1	788.2	611.0	1226.3	1943.1	321.2	40069.9	39764.8
	ARGOS (actual)	35130.4	2257.8	2897.0	1448.3	2291.9	22012.9	66038.3	45126.0
WY	Original (theory)	332085.0	11347.7	8609.0	7697.6	12091.2	520.4	372351.0	371856.0
	Original (actual)	332532.0	29068.2	39621.4	7742.7	12162.1	275086.0	696213.0	434881.0
	ARGOS (theory)	332085.0	7517.1	5297.4	5066.2	7957.7	422.3	358345.0	357944.0
	ARGOS (actual)	331550.0	26779.6	39882.7	5342.5	8391.7	250242.0	662189.0	424449.0
NSC	Original (theory)	117.5	10.3	16.4	29.3	48.7	6.0	228.2	222.5
	Original (actual)	117.5	10.9	22.7	29.1	48.4	70.5	299.1	232.2
	ARGOS (theory)	117.5	9.1	7.0	18.9	32.4	1.6	186.4	184.9
	ARGOS (actual)	117.4	11.7	18.0	19.0	32.5	66.1	264.7	201.9
KHOV	Original (theory)	256.6	28.6	162.0	74.0	151.5	3.1	675.8	672.9
	Original (actual)	259.6	29.2	160.4	74.0	151.5	486.7	1161.5	699.1
	ARGOS (theory)	256.6	3.1	40.1	20.8	39.2	0.1	359.7	359.7
	ARGOS (actual)	259.6	11.2	57.7	20.8	39.2	128.6	517.1	395.0
SC	Original (theory)	758.8	65.0	312.9	104.1	376.4	44.4	1661.5	1619.3
	Original (actual)	750.2	68.4	375.4	105.6	378.7	314.4	2001.7	1703.0
	ARGOS (theory)	758.8	48.6	252.0	54.7	277.6	22.1	1413.7	1392.7
	ARGOS (actual)	758.9	56.5	378.0	58.4	283.5	252.6	1788.0	1548.1

TABLE 8.2: SimYard results (percentages).

Problem	Schedule	Percentage of actual costs of original schedule							
		Base	OT	UT	Hire	Fire	Penalty	Total	Total(reduced)
OB	Original(theory)	99.9%	69.9%	62.6%	87.6%	87.9%	2.2%	65.0%	91.8%
	ARGOS (theory)	99.5%	29.0%	18.1%	47.2%	46.3%	1.5%	57.0%	80.6%
	ARGOS (actual)	99.3%	82.9%	86.0%	55.8%	54.6%	99.8%	93.9%	91.4%
WY	Original(theory)	99.9%	39.0%	21.7%	99.4%	99.4%	0.2%	53.5%	85.5%
	ARGOS (theory)	99.5%	25.8%	13.5%	65.4%	65.4%	0.1%	50.9%	82.1%
	ARGOS (actual)	99.3%	91.9%	101.7%	69.0%	69.0%	88.9%	94.1%	97.3%
NSC	Original(theory)	99.9%	94.4%	72.4%	100.7%	100.6%	8.5%	76.3%	95.8%
	ARGOS (theory)	99.6%	79.5%	28.8%	65.1%	67.0%	1.0%	48.5%	77.5%
	ARGOS (actual)	99.5%	101.7%	74.5%	65.4%	67.3%	43.2%	68.9%	84.6%
KHOV	Original(theory)	98.8%	98.1%	101.0%	100.0%	100.0%	0.6%	58.2%	96.2%
	ARGOS (theory)	98.7%	10.4%	25.0%	28.0%	25.9%	0.0%	31.8%	51.5%
	ARGOS (actual)	99.9%	38.2%	36.0%	28.0%	25.9%	28.1%	45.7%	56.6%
SC	Original(theory)	100.0%	95.1%	83.3%	98.6%	99.4%	14.1%	83.0%	95.1%
	ARGOS (theory)	100.5%	71.8%	67.1%	52.2%	73.5%	8.2%	72.5%	82.2%
	ARGOS (actual)	100.5%	83.4%	100.6%	55.8%	75.1%	93.9%	91.7%	91.4%

TABLE 8.3: Cost savings before and after simulation (ARGOS schedules compared with original schedules) as well as the decrease in savings.

	OB	*WY*	*NSC*	*KHOV*	*SC*
Before	11.9%	3.7%	16.9%	46.6%	14.0%
After	8.2%	2.4%	13.0%	43.5%	9.1%
Decrease	31.2%	35.1%	23.1%	6.7%	35.0%

- Hire and fire costs for any schedule before and after simulation are usually quite close. However, overtime and undertime are always significantly higher after simulation. This can be explained by the fact that the disturbances we model mostly have small, local effects. These seem to be absorbed by increases in overtime and undertime and have little affect on long-term staffing decisions.
- For ARGOS schedules for problems OB, WY, and NSC, the difference between the undertime costs before and after simulation are much greater than the same difference for overtime costs. We suspect this is related to the fact that ARGOS reduced the undertime costs by significantly more than the overtime costs for these three problems (see the comments of Section 7.4). Also, in the theoretical ARGOS schedules for these problems, there is twice as much overtime as undertime (since the costs are similar and an hour of undertime costs twice as much as an hour of overtime). Therefore an hour of undertime added during simulation has a greater impact than an added hour of overtime. For comparison, note that in the theoretical ARGOS schedules for $KHOV$ and SC, the undertime cost is significantly higher than overtime cost.

Table 8.3 summarizes the results of Table 8.2 by comparing the theoretical[8] and actual savings of the ARGOS schedule compared to the original for each problem as well as giving the percent decrease in savings between theoretical and actual results.

As should be expected, the theoretical savings of ARGOS are muted somewhat by real-world issues; they are reduced by between 6.7% ($KHOV$) and 35.1% (WY). However, for all problems, the ARGOS schedules remain preferable to the originals.

It should be reiterated that these results are preliminary. While SimYard is an attempt to faithfully model shipyard operations, we do not have the necessary data to validate the model. We have also not included sensitivity analysis to evaluate how the results would change if the parameters were varied.

[8]These numbers are slightly different than those reported in chapter 7 since the numbers here include finite penalty costs and these affect the optimal staffing levels.

TABLE 8.4: Correlation coefficients between theoretical and actual costs.

Problem	Penalty	Correlation coefficient	
		With original schedule	Without original schedule
OB	high	0.763	0.417
	low	0.947	0.924
NSC	high	0.848	0.760
	low	0.970	0.966

Nonetheless, the results suggest that ARGOS schedules are not brittle and susceptible to common real-world changes. This was not necessarily going to be true; a concern of the shipyards was that ARGOS was delaying work and scheduling everything compactly so that small perturbations would cause much more trouble in an ARGOS schedule than in an original one. Therefore, these results do allay these original fears that ARGOS schedules might be difficult to implement in practice.

8.3.1 Are Better ARGOS Schedules Really Better?

Recall that ARGOS can produce high quality schedules within a small number of iterations and subsequently makes progress at a much slower rate over many iterations. If the project cost that really matters is the one predicted by SimYard and not the theoretical savings of ARGOS it is possible that many of the ARGOS iterations do not actually provide much value; the small improvements made after the initial few iterations may become irrelevant due to the noise of simulation.

Given a number of schedules for a particular project, we can compare actual and theoretical costs. Figures 8.1 through 8.4 present results for *NSC* and *OB*. For each problem, we graph the results with both high and low penalty costs.

The top right data point for each graph represents the shipyard's original schedule. The rest represent the schedules produced for the various results reported in Chapter 7.

Correlation coefficients for the four sets of data are presented in Table 8.4. We provide the coefficient for each set with and without the original schedule included.

All scores are positively correlated. With low penalty costs, the coefficients are all

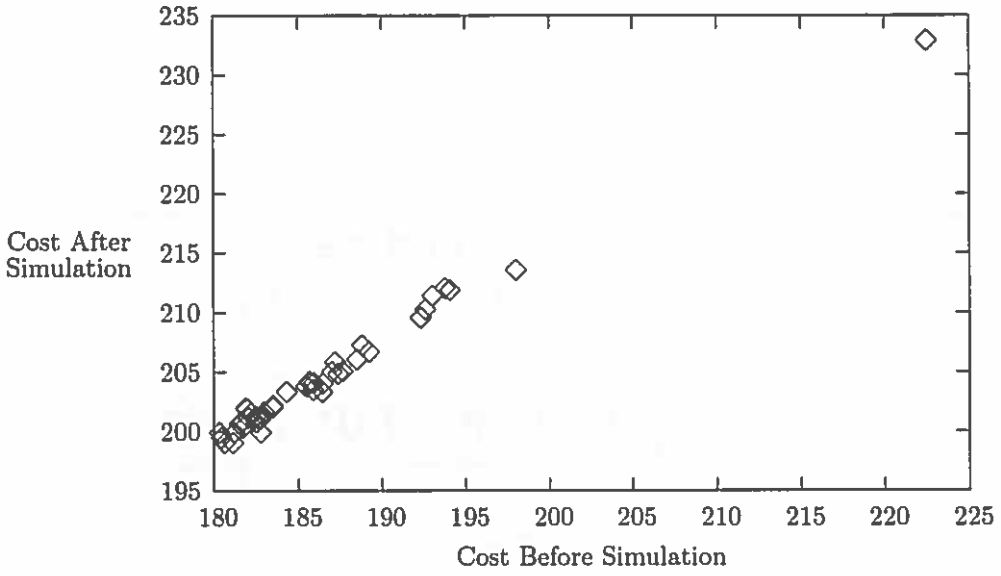


FIGURE 8.1: Theoretical vs. actual costs of various NSC schedules with reduced penalty cost (measured in \$10,000s).

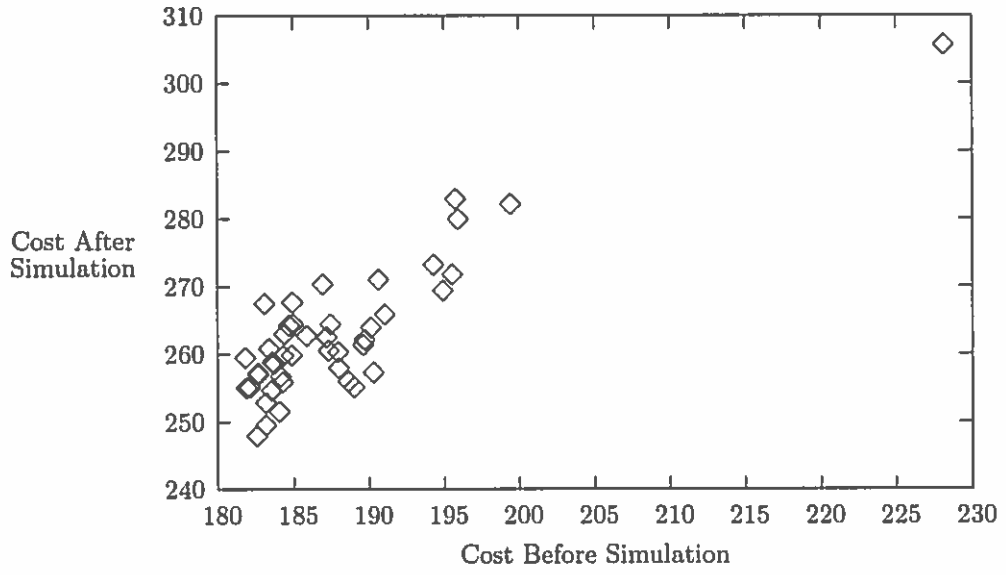


FIGURE 8.2: Theoretical vs. actual costs of various NSC schedules with original (high) penalty cost (measured in \$10,000s).

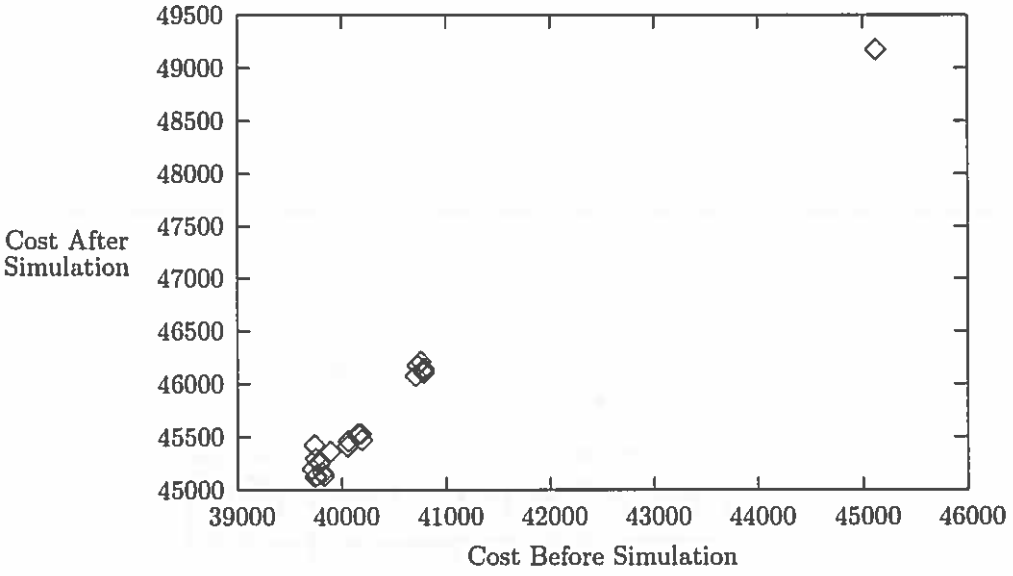


FIGURE 8.3: Theoretical vs. actual costs of various OB schedules with reduced penalty cost (measured in \$10,000s).

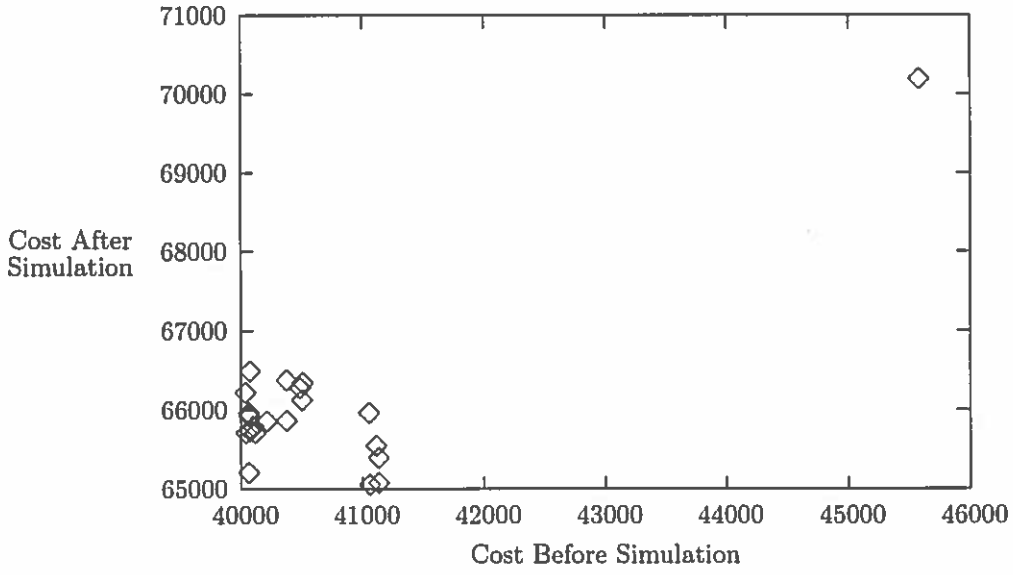


FIGURE 8.4: Theoretical vs. actual costs of various OB schedules with original (high) penalty cost (measured in \$10,000s).

above 0.92. This suggests that if low penalty costs are a close representation of real costs, it is worth letting ARGOS run as long as possible because small theoretical improvements are likely to result in actual improvements.

There is less correlation with high penalty costs. For problem OB, the coefficient has dropped to 0.417 when only ARGOS schedules are considered. This lower number, in addition to visual inspection of Figure 8.4 suggest that, if high penalty costs are more realistic, it is less valuable to run ARGOS for many iterations. In fact, Figure 8.4 suggests there is very little correlation; the positive coefficient in this case is more attributable to the clustered nature of the data points than to the fact that lower theoretical costs yield lower actual costs.

8.4 Future Work

Our simulator is obviously designed to simulate how things work at a particular shipyard. Although we suspect many of the SimYard features apply to a wide range of industries, some of the details may not match reality at other companies. Given an understanding of a different environment, it would be interesting to modify SimYard to model that situation.

There are a number of real-world issues that might be expected to occur but have not been included in our model:

- **Delayed deliveries:** Parts, plans or other requirements could force activities to be delayed. Notice that SimYard only delays activities when the floor manager decides it is locally cost effective; delays are never forced.

- **Emergent work:** In many situations, activities are added to a project after work has begun. This could be a result of rework, added features, or unexpected work.[9]

- **Inaccurate temporal constraints:** The constraints in the original problem may not be accurate. On one hand, there may be some temporal constraints that were imposed out of convenience but can be ignored in practice. On the

[9]Emergent work is common in repair environments where some of the project (inspections, for example) must be accomplished before other parts can be specified.

other hand, there may be others that were missed in the theoretical model but will be apparent to a floor manager when that part of a schedule is reached.

In addition to generalizing the SimYard model for other situations, the results presented here are preliminary and there is considerable work and analysis that could be done with the current model:

- **Sensitivity analysis:** We mentioned a number of parameters in Section 8.2.1 but used only the default setting (the setting suggested by the shipyard in question) in our experiments. It would be interesting to see how results change when these parameters are varied, both individually and together.
- **Allowing ARGOS in the model:** If ARGOS were used by the shipyard, it should probably be used not only to create an initial schedule but also as an aid to optimize rescheduling. Even the results of a couple of iterations of ARGOS are preferable to the decisions of a floor manager who is making local decisions without considering long-term impacts. It would be interesting to know how much this would increase the difference between the expected costs of projects in a yard with ARGOS relative to one without.
- **Optimization with respect to SimYard:** It may be possible for ARGOS to include some notion of robustness in its optimization criteria. The goal would be that, while the theoretical costs might be higher than for current ARGOS schedules, the costs after simulation would be lower. There are a number of possibilities:
  - *Change settings and rates:* The biggest costs in simulation occur when there are not enough people on a given day to get the work done. By increasing the overtime costs, decreasing the maximum overtime rate or adjusting other settings, we could encourage ARGOS to produce schedules whose optimal staffing profiles leave more leeway to handle perturbations in a schedule.
  - *Use float:* Schedulers in industry strongly believe that a schedule must have float in order to be executed successfully. We have seen how ARGOS can increase float at the expense of cost savings. Schedules with float may give better simulation results.
  - *Include SimYard features in ARGOS:* Instead of using theoretical values for activity resource use and duration, ARGOS could perform cost calculations with sample activity profiles that are representations of possible actual profiles. Work profiles and staffing profiles could be handled in the same manner.

- *Analyze SimYard:* By looking closely at simulation results, there may be lessons that can be learned. For example, perhaps it would be clear that ARGOS should never schedule overtime work for a resource that only has one worker because that would leave too little leeway for handling disturbances.

8.5 Related Work

Simulation is a well-established discipline with practitioners in a wide variety of fields. While most project scheduling research has concentrated on deterministic scheduling where real-world complications are ignored, the complications have not gone unnoticed. For example, Stoop and Wiers [91] overview some of the common issues that make scheduling in practice differ, often significantly, from scheduling in theory.

The goal of a typical project scheduling simulation study can be described as one of the following:

1. **Determine effective scheduling approaches:** Use simulation of the real-world environment to determine the effectiveness of various schedules and, as a result, the effectiveness of the algorithms used to produce them. This is the goal of SimYard; to compare ARGOS with current scheduling approaches.

2. **Predict the effects of operational changes:** Use simulation to predict the results of large-scale infrastructure upgrades, technological improvements or procedural changes before implementation to decide whether or not they are appropriate.

A number of researchers have designed simulation studies of semiconductor wafer fabrication facilities [34, 57, 65, 82]. The majority of this work is based on measuring the effectiveness of various dispatch heuristics in real-world conditions. For example, Kim et al. [57] describe methods to reduce the computational needs of simulation so that it can be used in real time to choose among possible dispatch heuristics.

There is some recent work on shipyard simulation. For example, a group of researchers has built a discrete event simulation model of a proposed steel processing

facility at a large shipyard to determine the effectiveness of the proposed facility and investigate possible configurations [68, 101]. Kiran et al. [58] developed a shipyard simulation approach designed to help identify bottlenecks, determine the impact of technological improvement and determine appropriate staffing levels for a given schedule.

The proposed simulation model of McLean and Shao [67] is the most similar to the SimYard model described here. Among their goals are the analysis of different staffing decisions based on the costs of undertime, hiring and firing and the prediction of optimal staffing levels. Their system takes as input both a schedule and proposed staffing levels. It is not clear how the end results of a simulation run are measured; we suspect they measure how well activities are handled (based on lateness issues) rather than the resulting cost of attempting to follow a particular schedule.

CHAPTER 9

Conclusion

We have suggested a window-based approach to project scheduling. A window of feasible start times is maintained for each activity. This allows search algorithms to directly navigate and explore the space of time-feasible schedules.

For problems without resource constraints, such as the LCOP, this is exactly the space we want to explore. For problems with resource constraints, such as the RCPSP/max, bulldozing can be used to resolve resource conflicts by moving directly to other time-feasible schedules in this space.

Defining the search space using windows allows a variety of problems to be solved and a variety of search algorithms to be implemented. An example is our SWO algorithms that use constructive search, augmented with local repair, to address resource-constrained problems with a regular objective function.

A very different example is our ARGOS algorithms that use local search to address problems without resource constraints and a non-regular objective function. Local search can explore the search space directly by using windows to identify neighbor schedules that are achievable via small or large-scale moves in the space.

We showed how to maintain windows efficiently, even when real-world temporal constraints are involved. This allows for scalable algorithms that are effective on large-scale real-world problems.

9.1 Summary

In this dissertation, we addressed a number of real-world issues that are often ignored by the project scheduling community:

- **Scalability:** Many real-world problems have thousands of activities yet experiments with even 1000 activities are rarely mentioned in the literature. Our ARGOS algorithms are scalable and were run on real-world problems ranging in size from 1000 activities to almost 140,000 activities. While the available RCPSP/max benchmark suites only have between 10 and 500 activities, SWO(B,R) was shown to scale well on those benchmarks relative to state-of-the-art approaches.

- **Complex constraints:** The GSTP framework that underlies our algorithm is able to handle two important classes of temporal constraints:

  1. Arbitrary temporal constraints (both minimum and maximum time lags).
  2. Calendar issues common to real-world projects.

  While the former have been considered in the literature on occasion, the latter have almost never been addressed.[1]

- **Varied objective functions:** While most research has focused on regular objective functions, many nonregular objective functions are better representations of actual real-world goals. The LCOP generalizes a number of the nonregular objective functions recently considered in the literature and ARGOS was shown to effectively solve large-scale LCOP instances.

  We also showed that the cost framework of ARGOS can solve multi-objective problems by incorporating other objectives as monetary costs. We considered

[1] It is also worth noting that SWO(B,R) and ARGOS are also capable of handling non-standard resource constraints. For example, both can easily handle resource capacities that vary over time. ARGOS can also handle a number of labor-specific constraints such as maximum hire and fire amounts and fluctuating staff levels.

the example where maximizing float and minimizing cost are competing objectives.

- **Reality:** The experimental results presented for most scheduling algorithms are purely theoretical. The implementation of SimYard as a validation tool enables us to understand how theoretical schedule improvements will translate into real-world results.

9.2 Contributions

9.2.1 The GSTP Framework

We described a generalized simple temporal problem (GSTP) framework and showed how it can be used to efficiently maintain hard and soft time windows (domains of feasible start times) for each activity during schedule construction as well as during schedule deconstruction and local search. The important differences between cyclic and acyclic problems were discussed as well as the complexity of the various procedures in each case.

The use of edge functions extends the GSTP framework beyond the capabilities of similar approaches. An important result is the ability to handle complex temporal constraints, including calendar issues, that arise in real-world scheduling problems.

9.2.2 The LCOP

While makespan minimization has been the primary focus of scheduling research, a more important goal is often to minimize project costs, especially for problems with a fixed deadline or with labor as a significant resource. We defined the labor cost optimization problem (LCOP) and showed that it subsumes a number of other resource-based nonregular objective functions (some of which are designed to indirectly minimize labor costs).

Given a schedule, calculating optimal staffing levels can be done using dynamic programming. We described Andrew Baker's observation that costs can be repre-

sented with piecewise linear convex functions and outlined the efficient algorithms he was able to implement as a result. We showed how these algorithms can be used during search to successfully solve real-world LCOP instances.

9.2.3 Window-Based Search

Most scheduling algorithms fall into one of four categories: chronological, order-based, disjunctive, or constraint-based scheduling. We proposed window-based search as an alternative.

An important benefit of a window-based approach is the ability to move activities around in a schedule while maintaining temporal feasibility. This allows local search to be performed with schedules directly rather than with activity orders (as is most common) and therefore allows local search even if objective functions are nonregular or resource capacities are unlimited.

We introduced the concept of bulldozing, an idea that requires an explicit window-based framework. Bulldozing was shown to be a crucial conflict resolution mechanism for RCPSP/max problems. Although experimental results of bulldozing were mixed for the LCOP, bulldozing is important theoretically because it allows the entire search space to be reached during local search, even with cyclic temporal constraints.

Two very different window-based algorithms were discussed in this dissertation:

1. The procedures SWO(B), SWO(B,R) and SWO(B,R,G) incorporate bulldozing into an order-based scheduling algorithm. On RCPSP/max benchmark suites, SWO(B,R) is competitive with state-of-the-art systematic and non-systematic approaches and scales well. This approach is effective even without any of the resource propagation techniques typically used for these problems.

2. The ARGOS algorithms combine a number of window-based techniques that primarily perform local search in schedule space. In producing significant cost savings relative to currently available algorithms, ARGOS provides a scheduling tool that up until now has not been available. The ability to optimize nonregular (and multiple) objectives, solve large problems quickly and produce schedules that are not simply resource leveled (makespan minimized) has not been available to the industries with which we are familiar.

9.2.4 Shipyard Simulation

SimYard is a shipyard simulator that can be used to compare the theoretical savings of ARGOS for LCOP problems with the savings that can be expected in reality. SimYard could also serve as a useful tool for a shipyard to perform scenario-based evaluations. For example, it could be used to estimate the impact of adding an additional project to the ongoing work in the shipyard.

9.3 Future Work

There are a number of ways the work in this dissertation could be extended:

- **Algorithmic improvements:** As mentioned in the relevant chapters, all of our algorithms have potential room for improvement. For example, the computational efficiency of cost calculations and bulldozing in ARGOS could be improved and Tabu search seems a promising possibility for both SWO(B,R) and ARGOS.
- **Hybrid problems:** The RCPSP/max and LCOP objective functions we considered are two extremes in a range of interesting problems. We showed how the other resource-based objective functions in the literature fall between these two. While ARGOS could solve any problem in the range, we saw that it is not appropriate for RCPSP/max problems. Therefore, it would be interesting to measure the performance of ARGOS in practice and to see at what point other algorithms are more effective. Interesting questions include:
  - How does ARGOS perform on the other resource-based nonregular objective functions?
  - How does ARGOS perform when some or all resources have maximum capacities but the goal is still cost optimization?
- **Optimization using SimYard:** ARGOS currently minimizes the theoretical costs of a schedule. As mentioned in Chapter 8, it would be interesting to see if ARGOS can be modified to minimize expected costs instead.

- **Real-world implementation:** The best measure of the effectiveness of a scheduling algorithm should come from industry. We hope that both ARGOS and SWO(B,R,G) will be incorporated into project management systems for the use of schedulers in the real world. It will be interesting to find out the effectiveness and applicability of these algorithms in real-world settings.

INDEX

BIBLIOGRAPHY

[1] David Applegate and William Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[2] Artemis Management Systems Ltd, Boulder, CO. *Artemis 9000/EX Network Processing User's Manual*, 2000.

[3] Tonius Baar, Peter Brucker, and Sigrid Knust. Tabu-search algorithms for the resource-constrained project scheduling problem. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 1–18. Kluwer Academic Publishers, 1998.

[4] Andrew Baker. *Intelligent Backtracking on Constraint Satisfaction Problems.* PhD thesis, University of Oregon, 1995.

[5] K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974.

[6] K. R. Baker. Workforce allocation in cyclical scheduling problems: A survey. *Operations Research Quarterly*, 27(1):155–167, 1976.

[7] Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP–1997)*, pages 375–389, Schloss Hagenberg, Austria, 1997. Springer-Verlag.

[8] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling.* Kluwer Academic Publishers, Boston, 2001.

[9] M. Bartusch, Rolf H. Möhring, and F. J. Radermacher. Scheduling project networks with resource constraints and time windows. *Annals of Operations Research*, 16:201–240, 1988.

[10] J. C. Beck, Andrew J. Davenport, Eugene D. Davis, and Mark S. Fox. The ODO project: Toward a unified basis for constraint-directed scheduling. *Journal of Scheduling*, 1(2):89–125, 1998.

[11] J. C. Beck, Andrew J. Davenport, and Mark S. Fox. Five pitfalls of empirical scheduling research. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP–1997)*, pages 390–404, Schloss Hagenberg, Austria, 1997. Springer-Verlag.

[12] J. C. Beck, Andrew J. Davenport, Edward M. Sitarski, and Mark S. Fox. Beyond contention: Extending texture-based scheduling heuristics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-1997)*, pages 233–240, Providence, 1997. AAAI Press.

[13] J. Blazewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operations Research*, 93:1–33, 1996.

[14] Peter Brucker. *Scheduling Algorithms.* Springer-Verlag, Germany, second edition, 1998.

[15] Peter Brucker, A. Drexl, R. Möhring, Klaus Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operations Research*, 112:3–41, 1999.

[16] Peter Brucker and Sigrid Knust. Solving large-sized resource-constrained project scheduling problems. In Jan Weglarz, editor, *Project Scheduling: Recent Models, Algorithms and Applications*, pages 27–51, Boston, 1999. Kluwer Academic Publishers.

[17] Michael J. Brusco and Tony R. Johns. Staffing a multiskilled workforce with varying levels of productivity: An analysis of cross-training policies. *Decision Sciences*, 29(2):499–515, 1998.

[18] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, February 1989.

[19] Yves Caseau and Francois Laburthe. Cumulative scheduling with task intervals. In Michael Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 363–377, Bonn, 1996. MIT Press.

[20] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. Scheduling multi-capacitated resources under complex temporal constraints. CMU Robotics Institute Technical Report CMU-RI-TR-98-17, June 1998.

[21] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. Greedy algorithms for the multi-capacitated metric scheduling problem. In *Proceedings of the 1999 European Conference on Planning*, Durham, United Kingdom, September 1999. Springer-Verlag.

[22] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. An iterative sampling procedure for resource constrained project scheduling with time windows. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pages 1022–1033, Stockholm, 1999. Morgan Kaufmann.

[23] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 8(1):109–136, January 2002.

[24] Ping Chen, Zhaohui Fu, and Andrew Lim. The yard allocation problem. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 3–8, Edmonton, 2002. AAAI Press.

[25] Cheng-Chung Cheng and Stephen F. Smith. A constraint-posting framework for scheduling under complex constraints. In *Proceedings of the Joint INRIA/IEEE Symposium on Emerging Technologies and Factory Automation*, Paris, October 1995. IEEE Computer Society Press.

[26] Cheng-Chung Cheng and Stephen F. Smith. A constraint satisfaction approach to makespan scheduling. In *First International Joint Workshop on Artificial Intelligence and Operations Research*, Timberline, OR, 1995.

[27] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms, part II: hybrid genetic search strategies. *Computers and Industrial Engineering*, 36:343–364, 1999.

[28] Vincent A. Cicirello. *Boosting Stochastic Problem Solvers Through Online Self-Analysis of Performance.* PhD thesis, The Robotics Institute, Carnegie Mellon Univ., 2003.

[29] Carlo Combi, Massimo Franceschet, and Adriano Peron. A logical approach to represent and reason about calendars. In *Ninth International Symposium on Temporal Representation and Reasoning (TIME-2002)*, pages 134–140. IEEE Computer Society Press, 2002.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliffort Stein. *Introduction to Algorithms.* MIT Press, Cambridge, 2001.

[31] James M. Crawford. An approach to resource constrained project scheduling. In *Proceedings of the 1996 Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 35–39. AAAI Press, 1996.

[32] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[33] Ulrich Dorndorf, Erwin Pesch, and Toan Phan-Huy. A time-oriented branch-and-bound algorithm for resource-constrained project scheduling with generalised precedence constraints. *Management Science*, 46(10):1365–1384, October 2000.

[34] Zaid Duwayri, Mansooreh Mollaghasemi, and Dima Nazzal. Scheduling setup changes at bottleneck facilities in semiconductor manufacturing. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceeding of 2001 Winter Simulation Conference*, pages 1208–1214, Arlington, 2001. The Society for Computer Simulation International.

[35] A. Fest, R. H. Möhring, F. Stork, and M. Uetz. Resource constrained project scheduling with time windows: A branching scheme based on dynamic release dates. Technical Report 596, TU Berlin, Germany, 1999.

[36] Mark S. Fox. Constraint-guided scheduling, a short history of research at CMU. *Computers in Industry*, 14:79–88, 1990.

[37] Mark S. Fox. ISIS: A retrospective. In *Intelligent Scheduling*, pages 3–28, San Francisco, 1994. Morgan Kaufmann.

[38] B. Franck and Klaus Neumann. Resource constrained scheduling problems with time windows - structural questions and priority-rule methods. Technical Report WIOR-492, Universitat Karlsruhe, Germany, 1998.

[39] B. Franck, Klaus Neumann, and C. Scwhindt. Project scheduling with calendars. *OR Spektrum*, 23:325–334, 2001.

[40] B. Franck and T. Selle. Metaheuristics for the resource-constrained project scheduling problem with schedule-dependent time windows. Technical Report WIOR-546, Universitat Karlsruhe, Germany, 1998.

[41] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W.H. Freeman and Company, 1979.

[42] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, May 1976.

[43] B. Giffler and G. L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8:29–53, July-August 1960.

[44] Matthew L. Ginsberg. Dynamic backtracking. *Journal of AI Research*, 1:25–46, 1993.

[45] Matthew L. Ginsberg, James M. Crawford, and David W. Etherington. Dynamic backtracking. Final Technical Report RL-TR-96-215, ARPA Order No. A009, February 1997.

[46] Fred Glover and Manuel Laguna. *Tabu Search.* Kluwer Academic Publishers, Boston, 1997.

[47] Teofilo Gonzalez and Sartaj Sahni. Flowshop and jobshop schedules: Complexity and approximation. *Operations Research*, 26(1):36–52, 1978.

[48] John C. Goodale and Enar Tunc. Tour scheduling with dynamic service rates. *International Journal of Service Industry Management*, 9(3):226–247, 1998.

[49] Mohamed Haouari and Mohammad A. Al-Fawzan. A bi-objective model for maximizing the quality in project scheduling. DIMACS Technical Report 2002-14, April 2002.

[50] Willy Herroelen, Erik Demeulemeester, and Bert De Reyck. A classification scheme for project scheduling. In Jan Weglarz, editor, *Project Scheduling: Recent Models, Algorithms and Applications*, pages 1–26, Boston, 1999. Kluwer Academic Publishers.

[51] Joseph Horowitz. *Critical Path Scheduling.* The Ronald Press Company, New York, 1967.

[52] *ILOG Optimization Suite: White Paper.* Mountain View, CA, 2001.

[53] A. Jain and S. Meeran. A state-of-the-art review of job-shop scheduling techniques. Technical report, Department of Applied Physics, Electronic and Mechanical Engineering, University of Dundee, Dundee, Scotland, 1998.

[54] David E. Joslin and David P. Clements. Squeaky wheel optimization. *Journal of AI Research*, 10:353–373, 1999.

[55] D. Karger, C. Stein, and Joel Wein. Scheduling algorithms. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computation.* CRC Press, 1997.

[56] Gunnar W. Kau, Neal Lesh, Joe Marks, and Michael Mitzenmacher. Human-guided tabu search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 41–47, Edmonton, 2002. AAAI Press.

[57] Yeong-Dae Kim, Sang-Oh Shim, Bum Choi, and Hark Hwang. Simplification methods for accelerating simulation-based real-time scheduling in a semiconductor wafer fabrication facility. *IEEE Transactions on Semiconductor Manufacturing*, 16(2):290–298, May 2003.

[58] Ali S. Kiran, Tekin Cetinkaya, and Juan Cabrera. Hierarchical modeling of a shipyard integrated with an external scheduling application. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceeding of 2001 Winter Simulation Conference*, pages 877–881, Arlington, 2001. The Society for Computer Simulation International.

[59] Rainer Kolisch and Sonke Hartmann. Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis. In Jan Weglarz, editor, *Project Scheduling: Recent Models, Algorithms and Applications*, pages 147–178, Boston, 1999. Kluwer Academic Publishers.

[60] Rainer Kolisch and Rema Padman. An integrated survey of project scheduling: Models, algorithms, problems and applications. Technical Report, Heinz School of Public Policy and Management, CMU, Pittsburgh, August 1997.

[61] Sarit Kraus, Yehoshua Sagiv, and V.S. Subralimanian. Representing and integrating multiple calendars. University of Maryland Technical Report CS-TR-3751, 1996.

[62] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–34, 1992.

[63] Philippe Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, 2003.

[64] Javier Larrosa and Pedro Meseguer. Generic CSP techniques for the job-shop problem. In Jose Mira, Moonis Ali, and Angel Pasqual Del Pobil, editors, *Proceedings of the Eleventh International Conference on Industrial And Engineering Applications of Artificial Intelligence And Expert Systems (IEA-AIE-98)*, pages 46–55, Castello, Spain, 1998. Springer-Verlag.

[65] Loo Hay Lee, Loon Ching Tang, and Soon Chee Chan. Dispatching heuristic for wafer fabrication. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceeding of 2001 Winter Simulation Conference*, pages 1215–1218, Arlington, 2001. The Society for Computer Simulation International.

[66] Olivier Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-1993)*, pages 232–238, Chambery, France, 1993. Morgan Kaufmann.

[67] Charles McLean and Guodong Shao. Simulation of shipbuilding operations. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceeding of 2001 Winter Simulation Conference*, pages 870–876, Arlington, 2001. The Society for Computer Simulation International.

[68] D. J. Medeiros, Mark Traband, April Tribble, Rebekah Lepro, Kenneth Fast, and Daniel Williams. Simulation based design for a shipyard manufacturing process. In J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, editors, *Proceeding of 2000 Winter Simulation Conference*, pages 1411–1414, Orlando, 2000. The Society for Computer Simulation International.

[69] Itay Meiri. Combining qualitative and quantitative constraints in temporal reasoning. *Artificial Intelligence*, 87:343–385, 1996.

[70] Rolf H. Möhring. Minimizing costs of resource requirements in project networks subject to a fixed completion time. *Operations Research*, 32:89–120, 1984.

[71] Nicola Muscettola. Computing the envelope for stepwise constant resource allocations. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP–2002)*, pages 139–154, Ithaca, 2002. Springer-Verlag.

[72] J.F. Muth and G.L. Thompson, editors. *Industrial Scheduling.* Prentice-Hall, New Jersey, 1963.

[73] Klaus Neumann, Christoph Schwindt, and Jurgen Zimmerman. Recent results on resource-constrained project scheduling with time windows: Models, solution methods, and applications. Technical Report WIOR-617, Universitat Karlsruhe, Germany, April 2002.

[74] Klaus Neumann, Christoph Schwindt, and Jurgen Zimmerman. *Project Scheduling with Time Windows and Scarce Resources.* Springer-Verlag, Germany, 2003.

[75] Klaus Neumann and Jurgen Zimmerman. Methods for resource-constrained project scheduling with regular and nonregular objective functions and schedule-dependent time windows. In Jan Weglarz, editor, *Project Scheduling: Recent Models, Algorithms and Applications*, pages 261–287, Boston, 1999. Kluwer Academic Publishers.

[76] Peng Ning, X. Sean Wang, and Sushil Jajodia. An algebraic representation of calendars. *Annals of Mathematics and Artificial Intelligence*, 36:5–38, 2002.

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

[78] Hartwig Nübel. The resource renting problem subject to temporal constraints. *OR Spektrum*, 23:359–382, 2001.

[79] W. P. M. Nuijten and E. H. L. Aarts. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operations Research*, 90:269–284, 1996.

[80] Masayuki Numao. Development of a cooperative scheduling system for the steel-making process. In *Intelligent Scheduling*, pages 607–628, San Francisco, 1994. Morgan Kaufmann.

[81] Claude Le Pape. Constraint-based programming for scheduling: An historical perspective, working paper. In *Operations Research Society Seminar on Constraint Handling Techniques*, London, 1994.

[82] Oliver Rose. The shortest processing time first (SPTF) dispatch rule and some variants in semiconductor manufacturing. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceeding of 2001 Winter Simulation Conference*, pages 1220–1224, Arlington, 2001. The Society for Computer Simulation International.

[83] Norman M. Sadeh. Micro-opportunistic scheduling: The micro-boss factory scheduler. In *Intelligent Scheduling*, pages 99–136, San Francisco, 1994. Morgan Kaufmann.

[84] Norman M. Sadeh and Mark S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86(1):1–41, 1996.

[85] Norman M. Sadeh, Yoichiro Nakakuki, and Sam R. Thangiah. Learning to recognize (un)promising simulated annealing runs: Efficient search procedures for job shop scheduling and vehicle routing. *Annals of Operations Research*, 75:189–208, 1997.

[86] Craig W. Schmidt. Graph-based schedule builder for tightly constrained scheduling problems. U.S. Patent number 6,490,566, 2002.

[87] Christoph Schwindt. ProGen/max: A new problem generator for different resource constrained project scheduling problems with minimal and maximal time lags. Technical report WIOR-449, Universitat Karlsruhe, Germany, 1995.

[88] Christoph Schwindt. A branch-and-bound algorithm for the resource-constrained project duration problem subject to temporal constraints. Technical Report WIOR-544, Universitat Karlsruhe, Germany, November 1998.

[89] Stephen F. Smith. OPIS: A methodology and architecture for reactive scheduling. In *Intelligent Scheduling*, pages 29–66, San Francisco, 1994. Morgan Kaufmann.

[90] Tristan B. Smith and John M. Pyle. An effective algorithm for project scheduling with arbitrary temporal constraints. In *Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI-2004) (to appear)*, San Jose, July 2004. AAAI Press.

[91] Paul P. M. Stoop and Vincent C. S. Wiers. The complexity of scheduling in practice. *International Journal of Operations and Production Management*, 16(10):37–53, 1996.

[92] G. P. Syswerda. Generation of schedules using a genetic procedure. U.S. Patent number 5,319,781, 1994.

[93] Hamdy A. Taha. *Operations Research.* MacMillan Publishing Company, New York, 1987.

[94] E. P. K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, New York, 1993.

[95] Spyros Tzafestas and Alekos Triantafyllakis. Deterministic scheduling in computing and manufacturing systems: A survey of models and algorithms. *Mathematics and Computers in Simulation*, 35:397–434, 1993.

[96] Raul E. Valdes-Perez. The satisfiability of temporal constraint networks. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-1987)*, pages 256–260, Seattle, 1987. AAAI Press.

[97] Manuel Vazquez and L. Darrell Whitley. A comparison of genetic algorithms for the dynamic job shop scheduling problem. In *Sixth International Conference on Parallel Problem Solving From Nature*, Paris, 2000. Springer-Verlag.

[98] Manuel Vazquez and L. Darrell Whitley. A comparison of genetic algorithms for the static job shop scheduling problem. In Darrell Whitley, David Goldberg, and Erick Cantu-Paz, editors, *Genetic and Evolutionary Computation Conference*, Las Vegas, 2000. Morgan Kaufmann.

[99] Marc Vilain, Henry Kautz, and Peter van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In Daniel S. Weld and Johan de Kleer, editors, *Readings In Qualitative Reasoning About Physical Systems*, San Francisco, 1989. Morgan Kaufmann Publishers.

[100] J. P. Watson, J. C. Beck, A. E. Howe, and L. Darrell Whitley. Problem difficulty for tabu search in job-shop scheduling. *Artificial Intelligence*, 143(2):189–217, February 2003.

[101] Daniel L. Williams, Daniel A. Finke, D. J. Medeiros, and Mark T. Traband. Discrete simulation development for a proposed shipyard steel processing facility. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceeding of 2001 Winter Simulation Conference*, pages 882–887, Arlington, 2001. The Society for Computer Simulation International.

[102] J. Zhan. Calendarization of time planning in MPM networks. *ZOR-Methods and Models of Operations Research*, 36:423–438, 1992.