Michael E. Bergen
2 Beaverbend Cres.
Winnipeg, MB
Canada, R3J 0T1

Date: ____________

University of Alberta

Constraint-based Assembly Line Sequencing

by

Michael E. Bergen

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Constraint-based Assembly Line Sequencing** submitted by Michael E. Bergen in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Peter van Beek (Supervisor)

Dr. Fraser Forbes

Dr. Russ Greiner

Date:

To My Parents

Abstract

A wide variety of combinatorial optimization problems have been studied in recent years. Of particular interest are a class of optimization problems arising from the manufacturing of vehicles on assembly lines. These problems consist of sequencing the vehicles that are going to be produced such that their production is done in an efficient cost effective manner. In this thesis we introduce a real-world vehicle sequencing problem that was provided by TigrSoft, who solved the problem for one of their clients using a greedy search approach. We began by modeling this problem as a constraint satisfaction problem and from there we devised three different solution techniques for solving it. These solution techniques include a simple hill-climbing algorithm, a backtracking algorithm with parameterized soft constraints, and a branch and bound algorithm that is capable of finding optimal solutions. We were able to improve results, compared to TigrSoft's algorithm, using any of these three solution techniques. For our best method, a branch and bound technique with a decomposition into smaller sub-problems, we obtained improvements ranging between 3% and 13% for six real-world problem instances.

Acknowledgements

This research was funded in part by a post graduate scholarship from the Natural Sciences and Engineering Research Council of Canada (NSERC). Funding was also provided by TigrSoft Incorporated as well as a collaborative research and development grant from NSERC.

To begin with, I would like to thank Peter van Beek for being my supervisor. Without his endless patients, support, and guidance; this thesis would never have been completed.

Thanks to the people at TigrSoft who made this research project possible. In particular, thanks to Tom Carchrae for always seeking out answers to my questions. Thanks to my examining committee members, Fraser Forbes and Russ Greiner, for their comments and suggestions, and for taking the time to be on my committee. Thanks also to the people in the AI lab for sharing ideas and suggestions related to my thesis.

I would like to thank my friends, new and old, for making these last few years very enjoyable. In particular, thanks to Yoko, Oscar, Lourdes, Dima, and Maria for all the great times we spent together. Thanks to the Burrs for sharing their frozen paradise with me. Next time we'll get it lit on time.

Finally, I would like to thank my family. My parents, for their unconditional love and support. My brother, Brian, for helping me move to Edmonton and for always being a good brother.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A wide variety of combinatorial optimization problems have been studied in recent years. These problems consist of searching for the best solution from many possible choices. For small sized problems there are often techniques that are guaranteed to find an optimal solution within a reasonable amount of time. However, for problems that come from real-world situations, the size of the problems often make finding an optimal solution far too complex. Instead, approximate (sub-optimal) solutions that are computed relatively quickly are considered acceptable.

For a particular problem, finding any sub-optimal solution may be quite easy. The challenge is to devise an algorithm that can produce solutions that are as close as possible to an optimal solution and can be found quickly. Given two algorithms that solve the same problem, they can be compared based on the quality of the solution and the complexity of the algorithms. If both algorithms find solutions within a reasonable amount of time (and space), then the algorithm that finds the best solution can be considered better.

A particular class of optimization problems arises in the domain of scheduling (see [23] for an overview of constraint-based scheduling). Within the domain of scheduling problems are problems that arise from the manufacturing of vehicles on assembly lines. These problems consist of sequencing the vehicles that are going to be produced such that their production is done in an efficient cost effective manner. Each of these problems has unique features depending on the manufacturing company that they come from. Some of these unique features include the technology used in the manufacturing process, the types of vehicles that are manufactured, and the goals of the company.

Of these problems, only a few have been examined by the research community. One vehicle sequencing problem that has been studied comes from Chrysler Corporation. The problem was solved using algorithms produced by ILOG. However, there is little information about this research besides a press release [18] and a set of presentation slides [7]. The information presented in the slides indicates a problem with a variety of constraints, and their press release indicates that Chrysler was able to save \$500,000 at a typical assembly plant by just reducing the number of times paint colours are changed during the assembly process.

Peugeot-Citroen, Europe's second largest vehicle manufacturer, also has plans to use ILOG's software to enhance the sequencing of vehicles on its assembly lines [19]. Once again there is no detailed information about this research, and there is unlikely to be any more in the future.

The majority of research has focused on a particular problem referred to as the Car Sequencing problem, introduced by Parrello and Kabat in [14]. It is unclear if this is a real world problem since Parrello and Kabat give no mention of its origin. The problem consists of sequencing different types of vehicles that are to be produced on an assembly line. Each vehicle that is produced requires that certain options be installed (e.g. air conditioning, sun-roof, and radio). The vehicles are classified by the options that they require. Each class of vehicles has a production requirement that indicates how many vehicles that belong to the particular class need to be produced. For each of the options there is an associated capacity constraint. A capacity constraint is defined by a ratio ($r : s$) which indicates that at most r vehicles with a particular option can be placed in any subsequence of s vehicles. For example, the sun-roof option might have a capacity constraint ratio of 2:5. This means that for any subsequence of five vehicles, only two of them can be vehicles that require a sun-roof.

For problem instances containing up to 200 vehicles, it is often the case that solutions do not exist. To deal with this, the problem can be defined as an optimization problem. In [14], Parrello and Kabat redefine the capacity constraints to incur a penalty value whenever a constraint is not satisfied. The weight of this penalty value depends on the option that the constraint is defined for (violating a constraint for some options is more expensive than others), how many vehicles exceed the capacity constraint, and how close these vehicles are sequenced together (see [14] for a detailed description of how these penalty values are calculated). For this representation, the optimization problem is defined as the minimization of penalty values.

The Car Sequencing problem has been solved using a variety of techniques. Van Hentenryck *et al.* [22] used a constraint logic programming (CLP) approach on solvable problem instances (i.e. no optimization was required). The Car Sequencing problem was modeled with finite domains and in an arithmetic manner. The CLP language used takes advantage of this model by applying specialized finite domain, arithmetic, arc consistency propagators to the problem. Furthermore they introduced specialized combinators (constraints) that increase the efficiency of the search.

Régin *et al.* [17] solved the problem using backtracking with a specialized arc-consistency propagator. The constraints in the problem are all converted to global cardinality constraints and the propagator described in [16] is applied to these constraints.

Several local search techniques have been applied to the Car Sequencing problem. Davenport and Tsang [8] defined a new class of problems called the Constraint Satisfaction Sequencing Problem, which is essentially a CSP where the variables all have the same domain values and an all-different constraint is defined over the variables. They solved the problem using hill-climbing with a variation of the min-conflicts heuristic and a value swap neighborhood function (i.e. values are swapped between variables).

Smith *et al.* [20] modeled the optimization version of the problem as a non-linear integer program. The model was solved using a general non-linear program solver, a hill-climbing approach, and a simulated annealing approach. Overall they found that the simulated annealing approach consistently found better solutions than the other two approaches.

The problem presented in this thesis is a vehicle assembly line sequencing problem. It is a real-world optimization problem that was provided by TigrSoft, an Edmonton company that specializes in planning and scheduling software. The company has already solved the problem using a greedy search technique and is interested in improving the quality of the solutions. The problem originates from a client of TigrSoft's that manufactures vehicles. The instances of the problem that we study come from a manufacturing plant that produces approximately 36,000 vehicles in a month, on two assembly lines. This manufacturing plant is currently using the solution technique provided by TigrSoft.

The vehicle assembly line sequencing problem consists of choosing the sequence that the vehicles should be produced on an assembly line. This problem is important because the sequencing of the vehicles affects the cost of production, the quality of the vehicles produced, and even employee satisfaction. Choosing an appropriate sequence can improve on all of these criteria. For example, the cost of production can increase by painting red vehicles immediately before white ones because it requires that the paint machine be cleaned thoroughly (otherwise the white vehicle will turn out pink). Also, producing too many of the same vehicles in a row can cause employees to become bored and thus the quality of production may decrease.

This thesis describes how we modeled a real-world optimization problem as a CSP and solved this model using constraint-based algorithms. The problem instances that were solved consist of one month's worth of vehicles for two assembly lines. The modeling of the problem began with an analysis of TigrSoft's problem specification. This analysis included converting procedurally defined constraints into a declarative form and determining the scope of the problem that we would model and solve. The result of this analysis is a declarative specification of the problem consisting of a description of the problem structure along with the constraints that make up the problem. The constraints defined are either hard (must be satisfied) or soft (can be violated at a cost). Each soft constraint is associated with a penalty value that is incurred every time it is violated. Thus the problem is one of optimization on these penalty values.

We then examined different possible ways to model this specification as a CSP. Of these possible models we selected one to be solved. Three solution techniques were devised to solve this model: two backtracking techniques and a local search technique. The first backtracking technique utilizes a parameterization of the soft constraints, where the parameter value indicates the tightness of the constraint. These parameter values are then adjusted with a restart and relaxation scheme. The second backtracking technique employs a branch and bound approach, which is guaranteed to find optimal solutions. Finally, the local search technique is a simple hill-climbing algorithm. All of these approaches were able to improve on the results of TigrSoft's greedy search algorithm.

The contributions of this thesis can be summarized as follows:

- We model a real-world optimization problem as a CSP.
- We demonstrate the ability to solve these problems using three different constraint-based algorithms. Each of these techniques improves on the results of TigrSoft's greedy search algorithm.

- We demonstrate the importance of decomposing the problem into one-day sub-problems. This decomposition is possible, without significant loss of solution quality, due to an overly tight constraint that makes the sub-problems relatively disjoint from one another.

- For most of these one day sub-problems, we prove optimal solutions using our branch and bound technique.

We begin in Chapter 2 by describing the problem's structure and constraints. Following that, in Chapter 3 we look at possible ways to model the problem as a CSP. After presenting different ways to model the problem, the three solution techniques that were applied to one of the models of the problem are presented in Chapter 4. In Chapter 5, we present the results of applying the three solution techniques to real-world problem instances and compare these results with the solutions of the original greedy search algorithm. Finally, conclusions and future work are presented in Chapter 6.

Chapter 2

Problem Background

The problem addressed in this thesis is the sequencing of vehicles on assembly lines. A typical problem involves sequencing a month's worth of orders, consisting of approximately 36,000 vehicles, on two assembly lines. At first glance, the number of vehicles that need to be sequenced makes the problem seem huge. However, the problem contains structure that significantly reduces its complexity.

In Section 2.1, the structure of the problem is presented. Following that, in Section 2.2, the constraints of the problem are described. Section 2.3 presents the original solution technique, provided by TigrSoft, that was applied to this problem. Throughout the chapter, an example problem is also defined to assist in describing the problem. Section 2.4 provides a solution to this example problem.

2.1 Problem Structure

In this section, we describe the main input of the problem. This can be summarized as a set of vehicles that need to produced, the grouping of these vehicles into lots and batches, and a set of capacity values that restricts how many vehicles can be produced on each day. In section 2.2 the remaining input of the problem, the constraints, are defined.

We begin now by describing how vehicles are grouped together into equal sized units called batches. From there, the problem is redefined as the sequencing of batches, by assigning batches to slots.

2.1.1 Batching

Probably the most important structure of the problem is that individual vehicles are grouped together into equal sized units called batches. All the problem instances examined in this thesis have a *batch size* of sixty (i.e., sixty vehicles are assigned to each batch). Hence, a typical problem with 36,000 vehicles is reduced to 600 batches.

Although the batching process is not part of the problem addressed in this thesis, it defines the main input of the problem. Since the process is somewhat complex, we will give a brief overview here and will explain important details of the process throughout the remainder of this chapter.

The batching process converts a set of vehicle orders into a set of batches. Each order represents a quantity of identical vehicles that needs to be produced. These quantities vary and can be smaller or larger than the batch size. Before batches are created, the orders are split into several smaller quantities of vehicles called *lots*. There are two important rules that determine the number of vehicles in a lot (lot size): it must be less than or equal to the batch size and the lot sizes must be chosen such that all the batches will have sixty vehicles. It is also preferable that the lots be as large as possible. Since many of the orders in a typical problem are larger than the batch size, many of the lot sizes are equal to the batch size. Lots with fewer vehicles than the batch size are grouped together into batches by putting together similar lots with quantities that add up to the batch size. Each batch is assumed to take one hour of time to produce on an assembly line. A typical problem instance has lots with between one and sixty vehicles, and batches with between one and ten lots, with the majority of batches having only one lot.

It is important to note that after batching, the lots are not sequenced in a batch and thus sequencing actually occurs at the lot level. However, the batch structure imposes a great constraint over how the lots can be sequenced (i.e. lots that belong to a particular batch must be sequenced together). Because of this structure, we can define two different problem representations: the *lot representation* and the *batch representation*. We define the lot representation as the sequencing of lots and the batch representation as the sequencing of batches and the sequencing of lots within each batch. Since each lot is assigned to a batch, the lot representation requires that the lots within a batch be sequenced consecutively. On the other hand, the batch representation splits the problem into two separate sub-problems which we refer to as the *batch sequencing problem* and the *internal lot sequencing problem*. For the remainder of this chapter we focus only on the batch representation. In Chapter 3, we present both representations, and how they can be modeled as a CSP.

Table 2.1 represents an example set of lots and the batches they are assigned to. This example problem contains both batches with only one lot (Batch B01) and batches with several lots (Batch B02). Compared to the real-world problem instances examined in this thesis, the example problem is significantly simpler. A typical real-world problem instance contains approximately 600 batches, more than twenty attributes, and each batch contains up to ten lots.

Although the number of lots and batches in this example is relatively small, it is sufficient to describe the structure and constraints that are defined for real-world problem instances. Throughout the remainder of this chapter, we use these lots to describe the different components of the problem.

2.1.2 Internal Lot Sequencing

Within each batch is a set of unordered lots that need to be sequenced. There is only one preferential constraint that always affects the sequencing of lots within a batch: the lot containing the most vehicles is sequenced last. As we will see later, the order of the lots within a batch is also influenced by a few problem instance specific constraints. Satisfying these constraints takes precedence over placing the largest lot last. Since the majority of constraints are not influenced by the order of lots in a batch, the remainder of this chapter will focus on the batch sequencing problem and the internal lot sequencing problem will be referred to when needed.

Lot	Batch	Lot Size	Line On	Line Off	Model	Exterior Colour	Sun Roof
L01	B01	60	1	2	M1	B	Y
L02	B02	20	1	2	M1	R	Y
L03	B02	40	1	1	M1	R	N
L04	B03	10	1	2	M2	G	Y
L05	B03	20	2	2	M2	R	N
L06	B03	30	1	2	M2	B	Y
L07	B04	10	1	2	M3	R	N
L08	B04	10	1	2	M3	G	Y
L09	B04	10	1	2	M3	R	Y
L10	B04	30	1	2	M3	G	N
L11	B05	60	1	2	M1	G	N
L12	B06	60	1	2	M1	B	Y
L13	B07	60	2	2	M1	B	Y
L14	B08	60	1	1	M1	B	N
L15	B09	60	2	2	M1	G	N
L16	B10	60	1	2	M2	R	Y
L17	B11	60	1	1	M2	R	Y
L18	B12	60	1	2	M2	G	N
L19	B13	60	1	2	M3	R	N
L20	B14	60	1	2	M3	G	Y

Table 2.1: Example lots and their batch assignments

2.1.3 Batch Sequencing

As mentioned earlier, batches all have the same production time of one hour. Thus batch sequencing can be redefined using slots, where a slot is defined to be an interval of time over a day with a fixed duration. In other words, a slot has a date, a start time and an end time, where the difference between the start and end times is equal to the duration. If we define a set of disjoint slots each with a duration of one hour, where the number of slots equals the number of batches, then a sequence of batches can be described as an assignment of batches to slots.

The benefit of having batches with a homogeneous production time is that batches can be assigned to slots that are fixed in time. If the production time differed between batches, a slot would not have an interval of time associated with it and its position in time would vary for different sequences of batches.

2.1.4 Capacity

As stated earlier, the problem consists of sequencing vehicles on two assembly lines over a month. As part of the input, a capacity value is assigned to each combination of day and assembly line. Each capacity value represents the number of batches that can be produced for a particular day and assembly line. In other words, the capacity for a day represents the number of slots assigned to that day. If no vehicle production is desired on a particular day, then the capacity for that day is zero. The capacities are assigned such that the sum of all the capacities for each day and assembly line equals the total number of batches that need to be produced for the month. Hence, there is no excess capacity.

A typical problem instance consists of two assembly lines each with 20 days of non-zero capacities. Each of these capacities is approximately fifteen batches, which gives a total capacity of 600 batches.

The example problem presented in Table 2.1 has fourteen batches. These batches will be sequenced on one assembly line over two days, where each day is assigned a capacity of seven batches.

2.1.5 Slot Structure

Using the definition of slots, we can define the problem structure for the batch sequencing problem as follows. A solution to the batch sequencing problem consists of a mapping from batches to slots, where the following is true:

- slots are disjoint from one another and hence are totally ordered,
- each slot belongs to a day and assembly line,
- the number of slots belonging to a particular day and assembly line is equal to the capacity for that day and assembly line, and
- every slot must be assigned one and only one unique batch.

By assigning batches to slots, which are already ordered, we get an ordering of the batches. In essence, the problem consists of finding a bijection from batches to slots and hence a sequence.

2.2 Constraints

Clearly, finding a solution to the problem we have defined so far is trivial. However, each problem contains a set of constraints that restricts which sequences are acceptable. Before describing these constraints, we introduce the concept of an attribute followed by a description of different ways to classify constraints.

2.2.1 Attributes

Constraints rely on information about the problem in order to evaluate a solution. They need to know, for example, what is similar about two batches and what is different, how two slots are oriented to one another in the sequence, and whether they are on the same day and assembly line.

We can represent this information by defining attributes for the three components of the problem structure: lots, batches, and slots. An attribute consists of a finite set of values, where each of a component's elements (a lot, batch or slot respectively) is assigned one of the values. For example, engine type is an attribute of the lots, four cylinder is an attribute value, and every lot is assigned an engine type attribute value. It is also assumed that for each of the components, elements can only be assigned one value from each attribute. For example, the engine type of a lot cannot be both four cylinder and six cylinder. We now describe typical attributes for the three components of the problem structure.

Slot attributes remain the same for all problems. These attributes are: start time (hour), day, and assembly line.

Lots and batches have attributes that remain the same for all problem instances, as well as attributes that are user definable and thus specific to each problem instance. Attributes that are common to all problems are: assembly lines that a lot/batch can be produced on, the date a lot/batch must be produced after (line-on date), the date a lot/batch must be produced by (line-off date), the number of vehicles a lot/batch contains (size), and in the case of batches the set of lots it contains. All other attributes are specific to each problem and are either selected from a set of basic attributes such as vehicle model, exterior colour, type of engine, and type of transmission; or are constructed from these basic attributes using set operations such as union, intersection, and Cartesian-product. For example, two attributes can be combined by taking the Cartesian-product of their attribute values to form a new attribute.

Table 2.1 lists six lot attributes, three unique to this problem: Model, Exterior Colour, and Sun-roof. The Model attribute has three values: M1, M2, and M3. The Exterior Colour attribute has three values: (B)lue, (R)ed, and (G)reen. The Sun-roof attribute has two values that represent if a lot has vehicles that require a sun-roof: (Y)es and (N)o. These attributes are all considered basic attributes.

Since a batch can only be assigned one attribute value for each attribute, and a batch can contain several lots each with different attribute values, batch attribute values are derived from the attribute values of its lots. Each attribute has a different method for deriving batch attribute values.

Table 2.2 shows the five batch attributes of the example problem. The batch attributes Model, Exterior Colour, and Sun-roof are all derived by selecting the attribute value that occurs for the most vehicles in the batch. For instance, in Table

Batch	Line On	Line Off	Model	Exterior Colour	Sun Roof
B01	1	2	M1	B	Y
B02	1	1	M1	R	N
B03	2	2	M2	B	Y
B04	1	2	M3	G	N
B05	1	2	M1	G	N
B06	1	2	M1	B	Y
B07	2	2	M1	B	Y
B08	1	1	M1	B	N
B09	2	2	M1	G	N
B10	1	2	M2	R	Y
B11	1	1	M2	R	Y
B12	1	2	M2	G	Y
B13	1	2	M3	R	N
B14	1	2	M3	G	Y

Table 2.2: Example batches

2.1 batch B03 has 10 Green vehicles, 20 Red vehicles, and 30 Blue vehicles. Thus, batch B03 has the batch attribute value Blue.

2.2.2 Constraint Classification

Constraints can be classified as either a batch constraint, or a lot constraint. Lot constraints rely on lot attributes, and influence the sequencing of lots and hence the sequencing of batches. Similarly, batch constraints rely on batch attributes and influence the sequencing of batches with no concern for the sequencing of lots within a batch. Furthermore, each batch constraint has a method for deriving the batch attribute values from lot attribute values.

Constraints can also be classified as either soft or hard. A hard constraint cannot be violated in a solution, while a soft constraint can be violated but imposes a penalty value for each violation. As part of the input to the problem, each soft constraint is assigned a penalty value; the higher the penalty value, the more undesirable the violation. A problem is evaluated by adding up the penalty values incurred by soft constraint violations. The lower the total value, the better the solution.

The constraints, along with their classifications are listed in Table 2.3. As the table shows, all batch constraints are hard constraints and all lot constraints are soft constraints. We now describe these constraints in detail.

2.2.3 Assembly Line

The problem instances that are examined in this thesis contain two assembly lines. Each assembly line has unique equipment needed to build certain vehicles. Thus, some vehicles can only be assembled on one of the lines, while others can be assembled on either line. If a batch contains a lot that can only be assembled on one of the assembly lines, then the batch must be assembled on that assembly line.

Constraint	Lot/Batch Constraint	Hard/Soft Constraint
Assembly Line constraint	Batch	Hard
Line-on and Line-off constraint	Batch	Hard
Even Distribution constraint	Batch	Hard
Distribution Exception constraint	Batch	Hard
Batting Order constraint	Batch	Hard
Change-over constraint	Lot	Soft
Run-length constraint	Lot	Soft

Table 2.3: Problem constraints and their classification

The batching process rules out any possibility of a batch containing lots that have conflicting line assignments.

2.2.4 Line-On and Line-Off

Each vehicle that is ordered must be produced sometime during the month. Some orders have more stringent scheduling requirements and must be produced during a specific range of days. For example, a particular vehicle's parts may only be available after a certain day of the month or an order might need to be shipped before the end of the month. For this reason, each lot is assigned a line-on and line-off day. A lot can only be produced on or after its line-on day, and on or before its line-off day. A batch's line-on day is selected by picking the maximum line-on day of its lots. For example, in Table 2.1, batch B03 has three lots (L04, L05, and L06) and lot L05 has a line-on day of two. Thus, in Table 2.2, batch B03 has a line-on day of two. Similarly, the line-off attribute value is selected by picking the minimum line-off day of its lots (see batch B02). The batching process rules out any possibility of a batch containing lots with conflicting line-on and line-off days.

2.2.5 Even Distribution

It is considered a desirable trait of a sequence that on each day, an assembly line produces many different types of vehicles. Reasons for this include maintaining workers skills for making all types of vehicles, reducing boredom of workers, and producing certain amounts of each type of vehicle prior to any unexpected assembly line shutdown. To achieve this goal, an even distribution constraint is defined to spread similar batches evenly over a month.

The even distribution constraint relies on a batch attribute (usually constructed from the Cartesian-product of several basic attributes such as model, and exterior colour) to determine if batches are similar or not. Two batches are considered similar, if they are assigned the same batch attribute value. The even distribution constraint spreads the batches by restricting the number of batches, with a particular attribute value, that can be produced on each day. Specifically, for each attribute value a numerical distribution value is designated for each day of the month. These values represent the number of batches with a particular attribute value that must be produced on each day and are provided as part of the input of the problem.[1]

[1]The even distribution constraint is actually defined in the TigrSoft algorithm as a process of

Day	Attribute Value	Distribution Value
1	M1-Y	2
1	M1-N	2
1	M2-Y	2
1	M2-N	0
1	M3-Y	0
1	M3-N	1
2	M1-Y	1
2	M1-N	2
2	M2-Y	2
2	M2-N	0
2	M3-Y	1
2	M3-N	1

Table 2.4: Example even-distribution values

In our example problem, the even distribution attribute is the Cartesian-product of the model and the sun-roof batch attributes. The distribution values are listed in Table 2.4. In Table 2.2, there are three batches with attribute values Model "M1" and Sun-roof "Y" attribute values. The distribution values defined in Table 2.4 say that two of these batches must be sequenced on the first day and one batch must be sequenced on the second day.

2.2.6 Distribution Exception

In some cases, an even distribution is inappropriate. For instance, when a new model year is introduced, production teams need time to learn new procedures. In this case, the distribution of new models needs to be restricted so fewer new models are produced early in the month. To do this, distribution exception constraints are defined to restrict the production of certain vehicles during the month.

Essentially a distribution exception constraint is a more general version of the even distribution constraint. The distribution exception constraint allows constraints to be specified that restrict the production of certain batches during a particular period in the month. Specifically, a distribution exception constraint specifies a minimum and maximum number of batches with a particular attribute value that must be produced on each day during a specified period of days in the month. Provided the even distribution values are known, it is possible to emulate the even distribution constraint by using several distribution exception constraints.

For the example problem, we define a distribution exception constraint on the Exterior Colour attribute value "G" for the first of the two days with a minimum value of one batch and a maximum value of two batches. Thus, batches B04, B05, and B12 cannot all be sequenced on the first day since they all have the attribute value "G".

assigning batches to days. To describe this constraint declaratively we determine these distribution values based on how TigrSoft's algorithm assigns batches to days. For a brief description of TigrSoft's algorithm, see Section 2.3.

2.2.7 Batting Order

Each day, it is desirable that a similar sequencing pattern be followed. One reason for this is to sequence simple vehicles at the beginning of the day and gradually progress to more difficult vehicles. This allows the production teams to warm up before building more complicated vehicles.

The batting order constraint ensures a similar sequencing pattern is followed each day by defining a total ordering of an attribute's values and applying this ordering when sequencing the vehicles on each day. Specifically, on each day a batch must be produced before another batch if its attribute value is ordered before the attribute value of the other batch. For batches with more than one lot, the attribute value assigned to the most number of vehicles is chosen to represent the batch.

For the example problem, we define the batting order constraint using the attribute Model, where the attribute values are ordered as follows: M1 $<$ M2 $<$ M3. This ordering implies that for each day, M1 batches should be produced first, followed by M2 batches, and then M3 batches.

It is also important to note that since the even distribution constraint for the example problem is partially defined on the Model attribute, the batting order constraint can be simplified further. From Table 2.4 we know that four M1 batches must be sequenced on the first day. Since the batting order constraint states that all M1 batches must be sequenced first in a day, the first four slots on the first day must be assigned an M1 batch. This is true for all days and all batting order attribute values. All the real-world problem instances examined in this thesis also have this characteristic. Later we see how this influences the modeling of this constraint within a CSP.

2.2.8 Change-Over

The definition of the change-over constraint is complicated. We will first present an intuitive, simplified version and in the next section discuss the complicating issues.

In a sequence, transitions from one lot attribute value to another lot attribute value may be undesirable. For instance, painting a white vehicle immediately after a red one is undesirable because the paint machine must be thoroughly cleaned before the white vehicle is painted (otherwise the white vehicle will turn out pink). To avoid such transitions, the change-over constraints are defined to incur a penalty value every time an undesirable transition occurs.

A change-over constraint is a soft constraint and thus is assigned a penalty value. It relies on two attributes to evaluate a transition between two sequenced lots: one attribute for the former lot (former attribute) and another attribute for the latter lot (latter attribute). Each change-over constraint has an attribute value assigned for each of the two attributes: former attribute value and latter attribute value. Using two attributes, instead of just one, allows change-over constraints to be defined between different attributes. For instance, a constraint can be defined that says do not sequence red vehicles immediately after two-door vehicles.

For the example problem, we define a change-over constraint using the Exterior Colour attribute with a penalty value of 100. The former attribute value will be "R" and the latter attribute value will be "NOT R", where "NOT R" means all Exterior Colour attribute values except "R". Thus, sequencing lot L17 followed by

L18 would incur a penalty value of 100, since L17 has value “R” and L18 has value “NOT R” (“G”).

2.2.9 Change-Over Special Case

We have just defined the change-over constraint as a constraint between two consecutive lots in a sequence. However, the constraint also includes the following additional rule: if a lot belongs to two violated instances of the same constraint then only one constraint violation is counted. For example, define a change-over constraint with penalty value 100 and with former attribute value “M1” from the Model attribute and latter attribute value “R” from the Exterior Colour attribute. If we sequence L01, L02, and L03 in this order, we would expect a total penalty value of 200 since L01 has value “M1”, L02 has values “R” and “M1” and L03 has value “R”. However, L02 belongs to both constraint instances, so only one violation is counted. Thus the total penalty value for this sequence is only 100.

What if three or more lots are sequenced together that all contain the values “M1” and “R”? In general, if a change-over constraint with penalty value p fails n consecutive times then the total penalty value is $\lceil n/2 \rceil \times p$. For example, 11 or 12 consecutive violations of a constraint with penalty value 100 would have a total penalty value of 600, while 10 consecutive violations would only have 500. Fortunately, this situation does not occur if both the former and latter attribute values come from the same attribute.

For the problem instances examined in this thesis, there are typically around forty different change-over constraints defined and the penalty values for these constraints range between one and a hundred. Most of these have former and latter attribute values that come from the same attribute. Typically a few change-over constraints (usually three) are defined using two different attributes. However violations of these constraints normally are either very rare or only occur within a few slots for each day (due to similarity with the batting order attribute). Because of this, it is highly unlikely that a change-over constraint is violated consecutively more than three or four times.

2.2.10 Run-Length

It is also desirable that certain attribute values are not repeated too much. For example, it may be undesirable to consecutively paint too many vehicles the colour red. Avoiding monotony of an attribute value can improve the effectiveness of production and quality inspection teams, and avoid part supply problems. A run-length constraint is a soft constraint that incurs a penalty value whenever the number of consecutive vehicles with a particular attribute value exceeds a specified limit (run-length value). One penalty value is counted for each lot that exceeds the run-length value.[2]

For the example problem, we define a run-length constraint on the Exterior Colour attribute value “R” with a run-length value of 120 vehicles and a penalty value of 200. Thus sequencing lots L16, L17, and L19 consecutively would incur a

[2]For a sequence of lots that violates a run-length constraint, the lots at the end of the sequence that exceed the run-length value are used to calculate the penalty values.

penalty value of 200 since they all have the attribute value "R" and in total they contain 180 vehicles.

For the problem instances examined in this thesis, there are usually around five different run-length constraints defined and the penalty values for these constraints range between ten and three hundred. Typically, a run-length constraint is defined on the same or similar attribute value as one of the change-over constraints. When a run-length constraint and a change-over constraint are defined on similar attribute values, the constraints are in conflict with each other and the constraint with the smaller penalty value is usually violated. In general, when similar attribute values are used, the run-length constraint is usually defined with a higher penalty value than its related change-over constraint, and hence is less likely to be violated.

2.3 Original Solution Technique

The original solution technique used on this problem is a greedy search algorithm that was created by TigrSoft. The actual process is complex and can be summarized by three stages: assign batches to an assembly line, assign batches in each assembly line to a day, sequence batches and their lots for each day.

2.3.1 Assembly Line Assignment

The first stage is to assign the batches to one of the two assembly lines. To begin with, many batches can only be produced on one assembly line as defined by the assembly line constraint. These batches are thus assigned to their appropriate assembly line. The remaining batches that can be produced on either assembly line are each assigned to the assembly line with batches that have attributes similar to their own. Thus an assembly line will tend to have similar batches.

2.3.2 Day Assignment

After the batches have been assigned to an assembly line, the batches within each assembly line are assigned to a day. This process of assigning batches to days, is essentially what defines the even distribution constraint. We refer to this process in the remainder of this section as the even distribution process.

As described in Section 2.2.5 the purpose of the even distribution constraint is to spread similar batches among the days, where two batches are similar if they have the same even distribution attribute value. In this section we describe the process of distributing batches to days. From this distribution of batches, the distribution values for the even distribution constraint, referred to in Section 2.2.5, are calculated.

The even distribution process begins by determining an ideal distribution of the batches that could be satisfied if no other constraints are defined on the problem. The ideal distribution specifies for each day and even distribution attribute value, the number of batches with the attribute value that can be assigned to the day. These values are calculated based on the number of batches with each distribution attribute value and the capacity of the days.

The even distribution process then attempts to assign each batch to a day based on the ideal distribution. Because of the line-on and line-off constraint and the distribution exception constraints, the ideal distribution is unlikely to be satisfiable.

If a batch cannot be assigned to one of the days specified by the ideal distribution, then it is placed on a different day.

In order to increase the chance of assigning all the batches successfully and achieving a distribution as close to the ideal distribution as possible, several heuristics are used. Although these heuristics are complicated by a significant amount of detail, they can be summarized as an attempt to place priority on the assignment of highly constrained batches to days with the most available capacity and the least contention between unassigned batches.

2.3.3 Batch and Lot Sequencing

Once the batches are assigned to a day, the batches within each day and the lots within these batches are sequenced. This sequencing is influenced by the batting order constraint, the change-over constraints, and the run-length constraints. The sequencing is accomplished in a greedy search manner by sequencing the batch that satisfies the batting order constraint and incurs the least amount of penalty violations.

2.3.4 Performance

The original solution technique produces acceptable schedules, and is fast, taking only a few seconds to solve a problem instance. However, since a non-optimal greedy search algorithm is employed, there is likely room to improve the results using other search techniques such as backtracking or local search.

2.4 Example Problem Solution

A solution to the problem consists of an assignment of batches to slots such that all the hard constraints are satisfied. The quality of a solution is measured by the total penalty values that are incurred by violations of the soft constraints. The lower the total penalty values, the higher the quality of the solution.

In this section we present a solution to the example problem. Looking at just the line-on and line-off constraint, the even distribution constraint, and the batting order constraint; Table 2.5 gives possible batch assignments that can be made. Table 2.6 gives an assignment of batches to slots that satisfies all the hard constraints, and hence it is a solution. This solution satisfies the distribution exception constraint since two batches with the attribute value "G" are assigned to the first day (B12, B04). The change-over constraint is violated three times (L17 $\rightarrow$ L18, L09 $\rightarrow$ L10, and L05 $\rightarrow$ L04), while the run-length constraint is not violated at all. Since the penalty value for each change-over constraint is 100, the total penalty value for this problem is 300. Furthermore, this solution is optimal for the given internal lot sequences.

Day	Slots	Possible Batches
1	1-4	B01, B02, B06, B08
1	5-6	B10, B11, B12
1	7	B04, B13
2	1-3	B05, B07, B09
2	4-5	B03, B10, B12
2	6-7	B04, B13, B14

Table 2.5: Possible batch assignments for example problem

Day	Slot	Batch	Lot	Lot Size	Exterior Colour
1	1	B06	L12	60	B
1	2	B08	L14	60	B
1	3	B01	L01	60	B
1	4	B02	L02	20	R
1	4	B02	L03	40	R
1	5	B11	L17	60	R
1	6	B12	L18	60	G
1	7	B04	L08	10	G
1	7	B04	L07	10	R
1	7	B04	L09	10	R
1	7	B04	L10	30	G
2	1	B05	L11	60	G
2	2	B09	L15	60	G
2	3	B07	L13	60	B
2	4	B10	L16	60	R
2	5	B03	L05	20	R
2	5	B03	L04	10	G
2	5	B03	L06	30	B
2	6	B14	L20	60	G
2	7	B13	L19	60	R

Table 2.6: Possible solution to example problem

Chapter 3

CSP Models

In this chapter we present some possible ways to model the problem as a constraint satisfaction problem (CSP) as well as a constraint satisfaction optimization problem (CSOP). We begin in Section 3.1 by defining the concepts of a CSP and a CSOP. We then present how the vehicle sequencing problem, presented in Chapter 2, can be modeled as a CSP and a CSOP. In Section 2.1.1, we defined two possible representations of the problem: the batch representation and the lot representation. In this chapter, we model the batch representation as the sequencing of batches and require that the sequencing of lots within each batch be dealt with as part of the modeling of the problem (for example fixing the lot sequence for each batch). In Section 3.2, we present two possible CSP models for the batch representation: the slot model and the batch model. In Section 3.3, we present why modeling the lot representation as a CSP seems unsuitable. In Section 3.4, we describe how the CSP can be converted into a CSOP.

3.1 CSP Definition

A CSP is composed of a set of variables, a set of possible values for each variable, and a set of constraints that restrict the possible instantiations of the variables. Many problems can be modeled as a CSP; for example vision, temporal reasoning, and scheduling [24]. CSPs provide a framework for separating the modeling and solving of a problem. Since many techniques have been devised for solving CSPs in general, once a problem has been modeled as a CSP a variety of general solution techniques can be applied to it. Later in this thesis, we examine some of the general solution techniques that can be applied to CSPs. In this section we give a formal definition of a CSP and a few related concepts. For a thorough description of CSPs, see [21].

The CSP can be defined formally as follows:

Definition 1 *An instance of the* constraint satisfaction problem *consists of a set of* n variables, $\{x_1, \ldots, x_n\}$*; their respective* domains, $\{D_1, \ldots, D_n\}$*; and a collection of* m constraints, $\{C_1, \ldots, C_m\}$. *A domain* D_i *consists of a set of values,* $\{a_1, \ldots, a_h\}$. *A variable* x *is instantiated if it is assigned a value* a *from its domain* $(x \leftarrow a)$. *A constraint* C_i *is defined over a set of variables* $x_{i_1}, \ldots, x_{i_k}$ *by a set* R *where* $R \subseteq D_{i_1} \times \ldots \times D_{i_k}$. *The constraint* C_i *is consistent if given an instantiation* $x_{i_1} \leftarrow a_1, \ldots, x_{i_k} \leftarrow a_k$*, the tuple* $(a_1, \ldots, a_k) \in R$.

A *partial solution* consists of an instantiation of some of the variables such that any constraint that is defined over a subset of the instantiated variables is consistent. A *solution* is a partial solution where all the variables are instantiated.

The *scheme* of a constraint is the set of variables that the constraint is defined over. The *arity* of a constraint is the number of variables in its scheme. Constraints can be classified by their arity. Three different classifications are unary, binary, and n-ary. Unary constraints are constraints with an arity of one. Since a unary constraint contains only one variable, all domain values that are not consistent with the constraint can be removed before the search begins. Binary constraints are constraints with an arity of two. N-ary constraint are constraints with arity n, where n is assumed to be greater than two.

Since the problem studied in this thesis is an optimization problem (since it contains soft constraints), a way of modeling optimization problems in terms of constraints is needed. The partial constraint satisfaction problem (PCSP) is a general framework for representing an optimization problem with constraints [9]. A PCSP can be defined as a CSP problem P, a problem space PS where PS contains the problem P along with "relaxations" of P, and a metric function on PS where the metric function defines the quality of each problem in PS relative to P. A solution to a PCSP represents a problem $P' \in PS$ along with a solution to P'. An optimal solution to a PCSP is a solution where P' has the minimum/maximum metric function value for all the problems in PS that are solvable.

The PCSP offers a very general framework for modeling optimization problems, since the set PS can defined in many different ways (e.g. a "relaxation" can include variables being removed from the problem). A more specific form of a PCSP is the constraint satisfaction optimization problem (CSOP). In terms of the definition for PCSPs, a CSOP has a problem space where the variables are not removed from the problem. A CSOP is defined in [21] as a CSP along with a function that maps every solution of a CSP to a numerical value. The function is problem-specific and represents the quality of the solution. The optimal solution of a CSOP is the solution with the minimal (maximal) function value. Essentially within a CSOP the CSP contains only hard constraint while the evaluation function represents all the soft constraints.

For the next two sections, we model the vehicle sequencing problem as a CSP by presenting the soft constraints as though they are hard. In Section 3.4 we describe how the soft constraints can be modeled within a CSOP.

3.2 Batch Representation

It is well known that if a problem can be modeled as a CSP, then there are several possible models (see, for example, [3]). For the batch representation we model the problem as the sequencing of batches, which we call the batch sequencing problem. Since the batch sequencing problem consists of defining a bijection from batches to slots, there are two obvious CSP models of this problem. We either model each slot as a variable and each domain as a set of batches (called the slot model) or model each batch as a variable and each domain as a set of slots (called the batch model). Before we examine these models we first describe how the internal lot sequencing problem is dealt with and then define attribute functions to simplify our discussion.

Function	Function Value
Line(s)	the assembly line that slot s belongs to
Day(s)	the day that slot s belongs to
Position(s)	the position value of slot s in a sequence

Table 3.1: Slot attribute functions

3.2.1 Internal Lot Sequencing

Since the focus of the batch representation is the sequencing of batches, the sequencing of lots within each batch needs to be dealt with in some way. There are several ways to do this.

One way is to pre-order the internal lots before solving the batch sequencing problem. Selecting a sequence to the lots could be done by using the solution provided by the original greedy search algorithm, or solving each internal lot sequencing problem using the lot constraints. Having a fixed ordering of the lots is simple but potentially reduces the quality of a final solution. What may seem like a good ordering of the lots within a batch may turn out to cause unnecessary conflicts when the batches are sequenced.

Another idea is to make explicit all the possible internal lot sequences for each batch containing more than one lot. For batches with only two lots this seems reasonable since there can be at most two sequences. However the problem instances examined have batches with up to ten lots. A batch with ten lots has potentially more than three million different sequences, significantly increasing the complexity of the problem.

A hybrid of the last two ideas would be to select a limited number of possible internal lot sequences. Each possible sequence could be evaluated and the best sequences selected.

In Sections 3.2.3 and 3.2.4 we briefly discuss how two different models of the problem are affected by these different methods of dealing with the lot sequencing problem. Besides this though, we strictly focus in this thesis on a fixed ordering of lots in each batch.

3.2.2 Attribute Functions

To assist in describing each model, we define attribute functions on the lots, batches and slots. These attribute functions represent the attributes that were described in Section 2.2.1 as well as additional functions to assist in our description of the different CSP models.

Table 3.1 presents the attribute functions for the slots. The value returned by the *Position* function (position value) represents an integer, where two slots cannot have the same position value if they are assigned to the same assembly line. A slot is sequenced after another slot if it has a larger position value, and two slots occur consecutively in a sequence, if their position values differ by one.

Table 3.2 presents the attribute functions for the batches. Besides these batch attribute functions, there are lot attribute functions that are defined for each run-length constraint and change-over constraint. These attribute functions are presented in Table 3.3.

Function	Function Value
$LineOn(b)$	line-on day of batch b
$LineOff(b)$	line-off day of batch b
$Lines(b)$	a set of assembly lines that batch b can be assigned to
$EvenDistAttrib(b)$	the even distribution attribute value of batch b
$DistExceptAttrib(b)$	the distribution exception attribute value of batch b
$BatOrdAttrib(b)$	the batting order attribute value of batch b
$FirstLot(b)$	the first lot sequenced in batch b
$LastLot(b)$	the last lot sequenced in batch b
$Lots(b)$	the ordered set of lots in batch b

Table 3.2: Batch attribute functions

Function	Function Value
$RunLenAttrib(l)$	the run-length attribute value of lot l
$ChgOverFormer(l)$	the change-over attribute value of the former lot l
$ChgOverLatter(l)$	the change-over attribute value of the latter lot l
$LotSize(l)$	the number of vehicles in the lot l

Table 3.3: Lot attribute functions

3.2.3 Slot Model

In this section we describe the slot model where the variables represent slots and the domain values represent batches. We begin by defining the variables and domain values, followed by the constraints. To simplify things, we assume that there is only one assembly line when defining all the constraints, except of course for the description of the assembly line constraint.

Variables

For the slot model, the slots are represented by variables and hence the slot attribute functions are applied to the variables. For this model, we represent an arbitrary but fixed variable with the letter s.

Domain Values

The domain values for the slot model are the batches and thus the batch attribute functions are applied to the domain values. We represent an arbitrary but fixed domain value with the letter b.

In Section 3.2.1, we presented different ways of dealing with the internal lot sequencing problem. Since the batches are represented by the domain values, the way we deal with this problem affects the size and structure of the domains. If the sequence of lots within each batch is fixed, then each batch is represented in a domain with a single value. If several internal lot sequences are included for each batch, then each batch is represented in a domain with a value for each lot sequence included. If all possible internal lot sequences are included, then a batch with ten lots would be represented with approximately three million domain values. Clearly, limiting the number of different lot sequences for each batch seems necessary.

Assembly Line

The assembly line constraint is a unary hard constraint. There is an assembly line constraint for each variable that represents a slot. Since each slot belongs to an assembly line, only batches that can be made on that assembly line can be assigned to the slot. An assignment $s \leftarrow b$ is consistent with an assembly line constraint if $\mathit{Line}(s) \in \mathit{Lines}(b)$.

Line-On and Line-Off

The line-on and line-off constraint is a unary hard constraint, where one instance of the constraint is defined for each variable (slot). Since a batch must be produced between its line-on and line-off days, batches can only be assigned to slots that are located between these days. An assignment $s \leftarrow b$ is consistent with a line-on and line-off constraint if $\mathit{Day}(s) \geq \mathit{LineOn}(b)$ and $\mathit{Day}(s) \leq \mathit{LineOff}(b)$.

Even Distribution

One instance of the even distribution constraint is assigned to each day that vehicles are produced on. In other words, the scheme for each constraint instance consists of those variables (slots) that belong to that instance's day and hence its arity is equal to the capacity of that day.

Let d be the constraint's day. Let A be the set of all even distribution attribute values. Let $\mathit{EvenDist}(a, d)$ equal the number of batches with attribute value $a \in A$ that must be produced on day d. Let k be the arity of a constraint. Let $s_1, \ldots, s_k$ represent the variables (slots) on day d. An assignment $s_1 \leftarrow b_1$, $\ldots$, $s_k \leftarrow b_k$ is consistent with an even distribution constraint for day d if for each $a \in A$, $|\{b_i | \mathit{EvenDistAttrib}(b_i) = a\}| = \mathit{EvenDist}(a, d)$.

Distribution Exception

Similar to the even distribution constraint, each instance of a distribution exception constraint is assigned to a production day. In other words, its scheme is the variables (slots) that belong to that day and its arity is equal to the capacity of that day.

Let a be the batch attribute value for the distribution exception constraint. Let d represent a day that the constraint is defined for. Let $\mathit{DistExceptMax}(d)$ ($\mathit{DistExceptMin}(d)$) equal the maximum (minimum) number of batches with attribute value a that can be produced on day d. Let k be the arity of a constraint. Let $s_1, \ldots, s_k$ represent the variables (slots) on day d. An assignment $s_1 \leftarrow b_1$, $\ldots$, $s_k \leftarrow b_k$ is consistent with the distribution exception constraint for day d if $\mathit{DistExceptMin}(d) \leq |\{b_i|\ \mathit{DistExceptAttrib}(b_i) = a\}| \leq \mathit{DistExceptMax}(d)$.

Batting Order

The batting order constraint is a hard constraint and can be represented as a binary constraint between all consecutive pairs of variables (slots) that are on the same day.

Let A be the ordered set of all batting order attribute values, such that for any $a_i, a_j \in A$, $a_i < a_j$ means that a_i comes before a_j in the batting order. Let s_1 and s_2 represent the variables in a batting order constraint, where s_1 precedes s_2 in the ordering of the variables. An assignment $s_1 \leftarrow b_1$ and $s_2 \leftarrow b_2$ is consistent with a batting order constraint if $\mathit{BatOrdAttrib}(b_1) < \mathit{BatOrdAttrib}(b_2)$ in A.

As described earlier, the batting order constraint can be simplified if its attribute is similar to the even distribution constraint's attribute. We redefine the batting order constraint as a unary constraint as follows. Using the even distribution constraint we define the function $\mathit{BatOrdDist}(a, d)$, which represents the number of batches with attribute $a \in A$ that can be assigned to day d. Let $s_1, \ldots, s_k$ represent the variables on a day d, where $\mathit{Position}(s_i) < \mathit{Position}(s_j)$ if $i < j$. Let b be an arbitrary batch and let $a_b = \mathit{BatOrdAttrib}(b)$. Then an assignment $s_i \leftarrow b$ $(1 \leq i \leq k)$ is consistent if $\sum_{a<a_b} \mathit{BatOrdDist}(a, d) \leq \mathit{Position}(s_i) - \mathit{Position}(s_1)$ $+1 \leq \sum_{a \leq a_b} \mathit{BatOrdDist}(a, d)$

Since all the problem instances studied in this thesis have similar attributes for the even distribution constraint and batting order constraint, we use the unary definition of the constraint for the remaining chapters.

Run-Length

The run-length constraint is a soft constraint that is applied to consecutive variables (slots) and its arity depends on the constraints definition.[1] Let *batchsize* represent the number of vehicles assigned to a batch. Let r be the run-length value[2]. The arity of a run-length constraint equals $(r/\mathit{batchsize}) + 1$. For example, if a run-length constraint has a limit of 120 vehicles and the batch size is sixty, then the arity of the constraint is three and an instance of the constraint is defined for every consecutive set of three variables.

Let k be the arity of a constraint. Let a represent the attribute value of the run-length constraint. Let $s_1, \ldots, s_k$ represent the variables in a run-length constraint, where s_i precedes s_{i+1} for $1 \leq i \leq k-1$. For some arbitrary assignment $s_1 \leftarrow b_1$, $\ldots$, $s_k \leftarrow b_k$, let L equal $\bigcup_{1 \leq i \leq k} \mathit{Lots}(b_i)$, where the order of the lots within each batch is maintained and the lots between batches are ordered with respect to the order of the variables that the batches are assigned to. Let L^* be a partition of L, where each element of L^* is a maximal set of consecutive lots that are all assigned the same attribute value from the run-length attribute. An assignment $s_1 \leftarrow b_1$, $\ldots$, $s_k \leftarrow b_k$ is consistent with the run-length constraint if for any $L_p \in L^*$, if for any $l \in L_p$ $\mathit{RunLenAttrib}(l) = a$ then $\sum_{l \in L_p} \mathit{LotSize}(l) \leq r$.

Change-Over

As described earlier, the change-over constraint is complicated by the way its penalty values are counted. We ignore this complication here and deal with it when describing our solution techniques. Thus we can assume that the change-over violation between lots have no influence on the violation of lots around them. Furthermore, we assume that only one domain value represents each batch. In other words, the internal lot sequences are fixed as part of the modeling of the problem. Because of these assumptions, we can ignore the violations between lots within a batch and

[1] In this definition we ignore how many lots within a batch violate a constraint and only check if the constraint fails over a set of batches. We take into account this issue when describing our solution techniques.

[2] To simplify our definition, we assume that r is divisible by *batchsize*.

represent the change-over constraint as binary constraints between all consecutive pairs of variables (slots).

Let s_1 and s_2 represent the variables in a change-over constraint, where s_1 precedes s_2 in the ordering of the variables. Let a_f (a_l) represent the former (latter) lot attribute value of the change-over constraint. An assignment $s_1 \leftarrow b_1$ and $s_2 \leftarrow b_2$ is consistent with a change-over constraint if
$\mathit{ChgOverFormer}(\mathit{LastLot}(b_1)) \neq a_f$ or $\mathit{ChgOverLatter}(\mathit{FirstLot}(b_2)) \neq a_l$.

All-Different Constraint

Since the problem involves defining a bijection between slots and batches, a constraint is needed to insure that the same domain value is not assigned to two different variables. To do this, an all-different constraint is defined over all the variables.

Let $s_1, \ldots, s_k$ represent all the variables. An assignment $s_1 \leftarrow b_1$, ..., $s_k \leftarrow b_k$ is consistent with the all-different constraint if for all s_i, s_j, $b_i \neq b_j$.

3.2.4 Batch Model

In this section we describe the batch model where the variables represent batches and the domain values represent slots. We begin by defining the variables and domain values, followed by the constraints. Like the slot model, we assume that there is only one assembly line when defining all the constraints, except of course for the description of the assembly line constraint.

Variables

For the batch model the batches are represented by variables. Hence the batch attribute functions are applied to the variables. For this model, we represent an arbitrary but fixed variable with the letter b.

As with the domain values of the slot model, the variables of the batch model are influenced by the way the internal lot sequencing problem is dealt with. If the sequence of lots within each batch is fixed, then each batch is represented with a single variable. If several internal lot sequences are included for each batch, then each batch can be represented with several variables, one for each lot sequence. Similar to the slot model, including all possible internal lot sequences for each batch makes the problem significantly larger. Thus, limiting the number of different lot sequences for each batch seems necessary.

Domain Values

The domain values for the batch model are the slots and thus the slot attribute functions are applied to the domain values. We represent an arbitrary but fixed domain value with the letter s.

Assembly Line

The assembly line constraint is a unary hard constraint. There is an assembly line constraint for each variable that represents a batch. Since each batch is assigned a set of assembly lines that it can be produced on, only slots belonging to this set of assembly lines can be assigned to the batch. An assignment $b \leftarrow s$ is consistent with an assembly line constraint if $\mathit{Line}(s) \in \mathit{Lines}(b)$.

Line-On and Line-Off

The line-on and line-off constraint is a unary hard constraint, where an instance of the constraint is defined for each variable (batch). Since a batch must be produced between its line-on and line-off days, it can only be assigned slots that are located between these days. An assignment $b \leftarrow s$ is consistent with a line-on and line-off constraint if $\mathit{Day}(s) \geq \mathit{LineOn}(b)$ and $\mathit{Day}(s) \leq \mathit{LineOff}(b)$.

Even Distribution

For each even distribution attribute value, we define a constraint. The scheme of a constraint is the variables (batches) assigned the constraint's attribute value and its arity is equal to the number of batches with that particular attribute value.

Let a be the attribute value of the constraint. Let D represent the set of production days. Let $\mathit{EvenDist}(a, d)$ equal the maximum number of batches with attribute value a that can be produced on day d. Let k be the arity of a constraint. Let $b_1, \ldots, b_k$ represent the constraint's variables (i.e. all batches with attribute value a). An assignment $b_1 \leftarrow s_1, \ldots, b_k \leftarrow s_k$ is consistent with an even distribution constraint for attribute value a if for each $d \in D$, $|\{s_i | \mathit{Day}(s_i) = d\}| = \mathit{EvenDist}(a, d)$.

Distribution Exception

A distribution exception constraint is assigned an attribute value and its scheme consists of the variables (batches) assigned the attribute value. Hence the arity for a distribution exception constraint is equal to the number of batches with the distribution exception attribute value.

Let a be the batch attribute value for the distribution exception constraint. Let D represent the set of production days. Let $\mathit{DistExceptMax}(d)$ ($\mathit{DistExceptMin}(d)$) equal the maximum (minimum) number of batches with attribute value a that can be produced on day d. Let k be the arity of a constraint. Let $b_1, \ldots, b_k$ represent the constraint's variables (batches). An assignment $b_1 \leftarrow s_1, \ldots, b_k \leftarrow s_k$ is consistent with the distribution exception constraint for attribute value a if for each $d \in D$, $\mathit{DistExceptMin}(d) \leq |\{s_i | \mathit{Day}(s_i) = d\}| \leq \mathit{DistExceptMax}(d)$.

Batting Order

The batting order constraint is a hard constraint and can be represented as a binary constraints between every pair of variables (batches) with different batting order attribute values.

Let A be the ordered set of all batting order attribute values, such that for any $a_i, a_j \in A$, $a_i < a_j$ means that a_i comes before a_j in the batting order. Let b_1 and b_2 represent the variables in a batting order constraint, where $\mathit{BatOrdAttrib}(b_1) < \mathit{BatOrdAttrib}(b_2)$ in A. An assignment $b_1 \leftarrow s_1$ and $b_2 \leftarrow s_2$ is consistent with a batting order constraint if $\mathit{Day}(s_1) = \mathit{Day}(s_2) \rightarrow \mathit{Position}(s_1) < \mathit{Position}(s_2)$.

As with the slot model, when the batting order attribute is similar to the even distribution attribute, the batting order constraint can be simplified as a unary constraint. Let $\mathit{BatOrdDist}(a, d)$ represent the number of batches with attribute value $a \in A$ that can be assigned to day d. Let b be an arbitrary batch and let $a_b = \mathit{BatOrdAttrib}(b)$. Let $s_1, \ldots, s_k$ represent the slots on a day d, where $\mathit{Position}(s_i) < \mathit{Position}(s_j)$ if $i < j$. Then an assignment $b \leftarrow s_i$ $(1 \leq i \leq k)$ is consistent if $\sum_{a<a_b} \mathit{BatOrdDist}(a, d) \leq \mathit{Position}(s_i) - \mathit{Position}(s_1) + 1 \leq \sum_{a \leq a_b} \mathit{BatOrdDist}(a, d)$.

Run-Length

Similar to the slot model's run-length constraint definition, we ignore how many lots within a batch violate a constraint and only check if the constraint fails over a set of batches. For the batch model, a run-length constraint is a soft constraint that is applied to every possible minimal sequenced set of variables (batches) where the number of consecutive vehicles with the run-length's attribute value exceeds the run-length value of the constraint.[3] Let r represent the run-length value of a constraint and let *batchsize* represent the number of vehicles in a batch. If we assume that r is divisible by *batchsize* then the arity of an instance of a constraint is $(r/\mathit{batchsize}) + 1$.

Let k be the arity of a constraint. Let $b_1, \ldots, b_k$ represent the order variables in a run-length constraint, where b_i precedes b_{i+1}, $1 \leq i \leq k-1$. An assignment $b_1 \leftarrow s_1, \ldots, b_k \leftarrow s_k$ is consistent with the run-length constraint if for some i, $1 \leq i \leq k-1$, $\mathit{Position}(s_i) \neq \mathit{Position}(s_{i+1}) - 1$.

Change-Over

Similar to the slot model change-over constraint definition, we ignore how the penalty values of the constraint are counted and assume that the sequence of lots within a batch is fixed. Hence, a change-over constraint can be represented as a binary soft constraint.

Let a_f and a_l represent the change-over constraint's former and latter lot attribute values, respectively. Let b_1 and b_2 represent any two variables in the problem. We define a binary constraint on these variables if $\mathit{ChgOverFormer}(\mathit{LastLot}(b_1)) = a_f$ and $\mathit{ChgOverLatter}(\mathit{FirstLot}(b_2)) = a_l$. An assignment $b_1 \leftarrow s_1$ and $b_2 \leftarrow s_2$ is consistent with a change-over constraint if $\mathit{Position}(s_1) \neq \mathit{Position}(s_2) - 1$.

All-Different Constraint

Since the problem involves defining a bijection between slots and batches, a constraint is needed to insure that the same domain value is not assigned to the two different variables. To do this, an all-different constraint is defined over all the variables.

[3]The method for deciding if a sequence of variables (batches) should have a constraint defined over them is essentially the method described for checking if the run-length constraint is consistent in the slot model. Furthermore, since the constraint is defined for all possible minimal sequences of variables that would violate the run-length constraint if sequenced consecutively, there may be quite a few instances of a constraint that are defined. In practice, many of these constraints can be combined together. However, we do not describe this simplification here because it is relatively complex and the main purpose of this section is to give a declarative statement of the constraint without any serious consideration of how efficiently it could be implemented.

Let $b_1, \ldots, b_k$ represent all the variables. An assignment $b_1 \leftarrow s_1, \ldots, b_k \leftarrow s_k$ is consistent with the all-different constraint if for all b_i, b_j, $s_i \neq s_j$.

3.3 Lot Representation

In the formulation of a problem as a CSP it is crucial to define what the variables and values represent. The problem with formulating the lot representation as a CSP is that within a problem instance, the size of lots can vary from one to sixty vehicles. Because of this, it is not clear if sequencing of lots should be modeled with fixed intervals of time or with slots that have no fixed interval or time reference.

For example, let the variables represent fixed intervals of time and the domains values represent the lots. In this case, the size of the interval would have to be a common divisor of all the lot sizes of the problem. Otherwise, more than one lot could be assigned to an interval and thus using lots for domain values would be inadequate. Hence, the best interval size would be the greatest common divisor of all the lot sizes. Sequencing a lot of size l where the interval size is i would imply that the lot value would be assigned to l/i consecutive variables. To ensure this would require a constraint of arity equal to L/i (where L is the size of the largest lot) to be assigned to all consecutive set of variables that can contain a lot. Now, a typical problem usually contains at least one lot of size one and so the number of variables compared to the batch representation would be 60 times greater.

In contrast, let the variables represent slots with no fixed interval or time reference and the domain value represent the lots. For this model, lots can be assigned to slots in a similar way as the batch representation. However, these slots have no attribute that say what day the slot belongs to. The day that a slot belongs to potentially has to be calculated for every possible sequence of lots. Since many of the constraints rely on information about a slot's position within the month, having to always calculate a slot's position makes it difficult to define these constraints. For example, the line-on and line-off constraint could not be represented as a unary constraint since the day that a slot belongs to is only determined once the preceding (or succeeding) slots are assigned lots.

Both of the above models contain positive and negative aspects. What makes the batch representation attractive is that it takes the positive aspects of both models and leaves behind most of the negative ones. Furthermore, since the majority of the constraints are at the batch level, the constraint definitions are simpler in the batch representation. The only cost of this is the loss of generality of the problem.

3.4 CSOP Model

In the previous sections, we modeled the soft constraints as hard. In this section we redefine these constraints within a CSOP model.

As mentioned earlier, a CSOP model is a CSP along with an evaluation function, where the evaluation function represents the soft constraints. The evaluation function for the vehicle sequencing problem is a function that evaluates a solution to the hard constraints and returns the total penalty value that was incurred by the soft constraints. This total penalty value is calculated by adding up the penalty values incurred by each constraint in the problem. For a change-over constraint the penalty values are added up as described in Section 2.2.9. Similarly the penalty values for a run-length constraint are added up as described in Section 2.2.10.

In the next chapter we describe three solution techniques. Two of these techniques used the CSOP model of the problem. The other technique used a CSP model where the soft constraints are grouped together into parameterized hard constraints. We describe this CSP model as part of our description of the solution technique.

Chapter 4

Solution Techniques

This chapter describes the three solution techniques we devised to solve the vehicle sequencing problem: two backtracking techniques, and a local search technique. Although two models (the slot and batch models) were described in the previous chapter, the solution techniques that were devised for this thesis are specifically for the slot model described in Section 3.2.3. The slot model was selected because it seemed more intuitive (at least from the perspective of a human solving the problem). However, an open question is whether the batch model could outperform the slot model using similar algorithms. It seems reasonable that this question relies significantly on the structure of the particular problem instances that are solved.

We first describe the local search technique in Section 4.1. In Section 4.2 we present the first backtracking technique that uses a restart and relaxation approach and in Section 4.3 we present the other backtracking technique that uses a branch and bound approach. A description of how the problem instances were decomposed into smaller sub-problems is given in Section 4.4.

4.1 Local Search

The first search technique we look at is local search. We begin by giving an overview of local search followed by a description of the algorithm we used.

4.1.1 Background

Local search is a general approach to solving combinatorial optimization problems (see [2] for an overview). An instance of a combinatorial optimization problems is defined by a set of possible solutions L for the problem (referred to as the solution space) and an evaluation function $f : L \rightarrow \Re$. If we assume that the problem is a minimization problem, then an optimal solution to the problem is a solution $l^* \in L$ where $f(l^*) \leq f(l)$ for all $l \in L$. For a CSOP, the solution space is all possible instantiations that satisfy all the hard constraints and f is the evaluation function of the CSOP.

Before describing how local search is applied to optimization problems, we first define the concept of a neighborhood function. A neighborhood function $\mathcal{N}$ is of the form $\mathcal{N} : L \rightarrow \mathcal{P}(L)$, where $\mathcal{P}(L)$ is the power set of the solution space. For any $l \in L$, $\mathcal{N}(l)$ is a set of other solutions in L that are in some way near l. We refer to $\mathcal{N}(l)$ as the neighborhood of l.

There are many possible ways to define a neighborhood function. For instance, for a CSOP the neighborhood of a solution l might be the set of solutions where one variable of l has been instantiated with a different value. In general, the way that the neighborhood function is defined influences the quality of the solutions that a local search algorithm finds and the cost of searching the solution space.

To describe local search, we will now present a simple local search algorithm known as hill-climbing or iterative improvement. Hill climbing begins with an initial solution l and searches the neighborhood $\mathcal{N}(l)$ for a better solution, where $l_i \in \mathcal{N}(l)$ is better if $f(l_i) \leq f(l)$. If better solutions exist then the algorithm selects one of these solutions (usually either the best one found or the first one found) and searches the selected solution's neighborhood for a better solution. This process continues until no better solution exists for the current neighborhood that is being examined. The last solution found is considered a local minimum for the given problem and neighborhood function.

The local minimum that is found by the hill-climbing algorithm is by no means guaranteed to be a global minimum for the problem. Guaranteeing a global minimum for any problem could be achieved by defining a neighborhood function where $\mathcal{N}(l) = L$ for any $l \in L$. However, this is equivalent to generating and testing every possible solution. Thus a good neighborhood function is one that creates reasonable sized neighborhoods but also allows the local search to find reasonably good local minimum solutions.

Even with selecting a reasonable sized neighborhood, there is still room for improving the hill-climbing approach. Several improvements to hill-climbing have been suggested. Two of these are simulated annealing and tabu search.

Simulated annealing improves on hill-climbing by randomizing the selection of a solution from a neighborhood. Instead of always picking a better solution, simulated annealing allows a worse solution to be selected with some small probability. The probability of a worse solution being selected is decreased during the algorithm's execution. How these probabilities are reduced is defined by a *cooling scheduling*. Given an appropriate cooling schedule, it is possible to find an optimal solution. However, such cooling schedules usually take exponential time to find an optimal solution. Thus faster cooling schedules that find sub-optimal solutions are usually adopted. For a more thorough explanation of simulated annealing, see [1].

Similar to simulated annealing, tabu search improves on hill-climbing by allowing worsening solutions to be selected from a neighborhood. At each point in the search, the best solution in the neighborhood is selected even if it is worse than the current solution. This means that when tabu search reaches a local minimum, it will move from the local minimum to a worse solution in its neighborhood. To avoid the likely possibility of moving back to the local minimum in the next iteration (and several iterations after that), tabu search selectively removes certain solutions from a neighborhood. To determine which solutions should be removed from a neighborhood, a list (known as a tabu list) is maintained of solutions that have recently been visited in the search. If a solution in the current neighborhood is found in the tabu list then it is removed from the neighborhood. By removing recently visited solutions, the tabu search can escape local minima and potentially find a new local minimum that is better and possibly optimal. Since the process of maintaining tabu lists and updating neighborhoods can be impractical, a more advanced version of tabu search involves maintaining a list of recent moves instead of solutions. A move is defined as an operation on a solution to obtain a new solution and a set of possible moves is defined for the problem such that the neighborhood can be defined as all solutions that can be obtained by apply one of the moves to the current solution. For a more thorough explanation of move lists and other tabu search enhancements, see [11].

4.1.2 Hill Climbing Approach

The local search algorithm devised for this thesis is a simple hill-climbing algorithm that is applied to a CSOP model of the problem (described in Section 3.4). Algorithm 1 presents an outline of the hill-climbing algorithm that we used, where $\mathcal{N}$ is the neighborhood function and f is the evaluation function. The algorithm begins with an instantiation that satisfies all hard constraints. The default initial solution is the original solution provided by the greedy search algorithm. The neighborhood of a solution consists of any solution where two variables' values have been swapped and no hard constraint is violated. Of the solutions in the neighborhood, the solution that reduces the total penalty value the most is selected. This process is repeated until no solution can be found in the current neighborhood that improves on the quality of the current solution.

Algorithm 1 Hill climbing algorithm

```
input: initial-solution
output: improved-solution

$S \leftarrow$ initial-solution
improvement $\leftarrow$ TRUE
while improvement do
  improvement $\leftarrow$ FALSE
  $N \leftarrow \mathcal{N}(S)$
  for all $S' \in N$ do
    if $f(S') < f(S)$ then
      $S \leftarrow S'$
      improvement $\leftarrow$ TRUE
    end if
  end for
end while
return $S$
```

4.2 Loosening Approach

In this section we describe backtracking on a CSP model with a loosening approach. The algorithm begins with several tight parameterized constraints and loosens them until a solution is found.

In a problem that contains soft constraints, it is common that some soft constraints are not satisfied. Since standard backtracking requires the satisfaction of all constraints, it is possible to treat the soft constraints as hard and remove some of them from the problem. However, the selection of the best constraints to remove is difficult.

To help deal with this, several soft constraint instances can be combined into a single parameterized hard constraint, where the parameter indicates the tightness of the constraint. The tightness of the soft constraints can then be adjusted by changing the constraints parameter value. A problem where the parameterized constraints are very loose will be relatively easy for backtracking to find a solution. However, the quality of this solution relates to the tightness of each parameterized constraint. Thus finding a quality solution requires that the appropriate parameter values are selected such that the problem is solvable and the quality of the solution is reasonable.

Within backtracking, it is possible to do this in many ways. Two possible directions are to start with loose parameter values and tighten until the problem is not solvable (tightening approach) or start with tight parameter values and loosen the problem until it is solvable (loosening approach). In both approaches, the key issue is the selection of which constraints to adjust. Clearly the constraints with high penalty values should be tightened as much as possible or loosened as little as possible. Besides this though, more information is needed on how adjusting a constraint affects the problem. In the case of the tightening approach it is preferable to tighten a constraint that leaves the problem solvable. However, it is unclear how to evaluate a constraint's affect on the problem (besides actually solving the problem). For the loosening approach it is preferable to loosen a constraint that makes the problem solvable. In this case it seems reasonable to select constraints that fail frequently as candidates to loosen. Because of this we chose to implement a loosening approach.

4.2.1 Backtracking Background

Before describing the loosening approach and how we parameterized the soft constraints, we give a brief overview of backtracking. Backtracking search is a technique that extends partial solutions by instantiating variables one at a time. The search only instantiates a variable with one of its domain values if it generates a new partial solution. If at some point in the search instantiating a variable with any of its values does not lead to a new partial solution, the search *backtracks* to the previously instantiated variable. When the search backtracks to a variable, the algorithm instantiates the variable with a different value and attempts to extend the new partial solution to a complete solution. If none of the variable's domain values successfully extends the partial solution then the search backtracks to the variable that was instantiated before the current variable. This process continues until either a solution is found or instantiating all the domain values of the first instantiated variable does not extend to a complete solution.

We will now describe propagation techniques that can be used to enhance the efficiency of backtracking. Following that we look at possible ways of dealing with soft constraints when backtracking.

4.2.2 Soft Constraints

For the loosening approach, each soft constraint's instances that belong to the same day and assembly line are grouped together into a parameterized hard constraint.

Since soft constraint violations can occur between lots that are sequenced on different days, the last slot of the previous day is included in each of these parameterized constraints. Although this does not perfectly model the constraints between days, it is sufficient to recognize most constraint violations that can occur between days.

Let p represent the parameter for an instance of a parameterized constraint. For a change-over constraint the parameter p represents the maximum number of change-over violations that can occur. If more than p violations occur, then the parameterized change-over constraint is not satisfied. Clearly this method has advantages over simply removing selected soft constraints from the problem. It decreases the number of possible selections that need to be made and leaves more decision power to the search algorithm (only the number of constraint failures is chosen, not the removal of specific constraints).

For a run-length constraint the parameter p represents the maximum run-length that can occur on the day. For example, if a constraint is defined on the colour "Red" and its parameter value is 120, then at most 120 vehicles with the colour "Red" can be sequenced consecutively.

4.2.3 Restart Scheme

For the loosening approach, the selection of which constraint to loosen is important. This section describes the circumstances when backtracking restarts and how the parameterized constraints are loosened when a restart occurs.

The backtracking algorithm begins with a problem that is initialized with tight parameter values. For the problem instances in this thesis, each parameterized change-over constraint is initialized with a value of zero and each parameterized run-length constraint, is initialized with the run-length value of the constraint.

The backtracking algorithm attempts to solve a tight problem and counts how many times the parameterized constraint fails. A constraint failure limit is set for each parameterized constraint. The failure limit for a constraint is set relative to its penalty value. Specifically, the constraint's penalty value is multiplied by a fixed constant to calculate its failure limit. Thus, the higher the penalty value of a constraint, the higher its constraint failure limit will be.

The failure limits are used to limit the effort backtracking spends trying to find a solution. Specifically, the backtracking algorithm stops searching if any of the parameterized constraints fail more than their failure limit, otherwise it either searches until it finds a solution or does an exhaustive search and finds no solution.

In the case that a failure limit is exceeded or no solution is found, a parameterized constraint is loosened and the backtracking algorithm is restarted. Which parameterized constraint to loosen is chosen by first selecting the parameterized constraints with the smallest penalty value that failed at least once, and of these constraints, the constraint that failed the most is selected. This method of selecting a constraint to loosen is based on the theory of constraint hierarchies [6] since constraints with the smallest penalty value are always selected. Once a parameterized constraint is selected, it is loosened by adding a value to the parameter. For a change-over constraint, the parameter is incremented by one, allowing one additional violation within the constraint. For a run-length constraint, the parameter is incremented by the batch size, increasing the run-length by sixty vehicles. Once a constraint has been loosened, the backtracking algorithm is restarted, and the relaxation and restart processes is continued until a solution is found. For the remainder of this thesis, we refer to this process of choosing a constraint to loosen as the *relaxation schedule*.

4.2.4 Variable and Value Ordering

The variable ordering is based on two factors: the day the variable belongs to, and the domain size of the variable. The variables are ordered by the day they belong to and the variables with the lowest day are selected. Amongst these variables, a variable with the smallest current domain size is selected.

The value ordering is based on the original greedy search solution. For each variable, the value assigned in the original solution is placed first in the variable's domain. Placing the original solution's batch first provides a reasonable solution for the backtracking algorithm to improve upon.

4.2.5 Consistency Propagators

In this section we begin by describing general propagation techniques that are used in backtracking. Following that we describe the specialized propagators that were devised for each different n-ary constraint.

Background

Propagation is the process of removing domain values from the uninstantiated variables during backtracking search. At any point in the backtracking search, a value can be removed from an uninstantiated variable's domain if, given the current instantiation of variables, no solution exists if the value were instantiated. Given that the removal of the domain value is dependant on the instantiation of some variables, if the instantiation of any of these variables changes then the value is reinstated into its variable's domain. Propagation improves backtracking search by reducing the domain size of uninstantiated variables and thus reducing the size of the search space.

Since deciding if every variable's domain value belongs to a solution can be computationally expensive, different types of propagation have been devised. Before describing two different general propagators, we briefly describe the n-queens problem that we use to describe these propagators. The n-queens problem consists of placing n queens on an $n \times n$ chess board such that no two queens attack each other. For our purposes, we represent this problem as a CSP as n variables that represents the columns of the chess board and n domain values for each variable where each domain value represents where a queen can be placed in a particular column. Figure 4.1 presents a solution to the 6-queens problem where variables $< 1, 2, 3, 4, 5, 6 >$ have been assigned the values $< B, D, F, A, C, E >$, respectively. A constraint is defined for every pair of variables such that only values (queen locations) that do not attack each other are accepted. For instance, if variable 1 is assigned the value B, then the constraint between variable 1 and 2 would allow only values D, E, and F to be assigned to variable 2.

One type of propagation is forward checking [10]. At each point in the search the forward checking propagator examines constraints in which only one variable in the constraint's scheme has not been instantiated. All of this variable's domain values

A				Q		
B	Q					
C					Q	
D		Q				
E						Q
F			Q			
	1	2	3	4	5	6

Figure 4.1: 6-queens solution

A		1			3	2
B	Q	1	1	1	1	1
C		1	Q	2	3	3
D			1	3		
E		Q	2	1	2	2
F			2		1	3
	1	2	3	4	5	6

Forward Checking

A		1		3	3	2
B	Q	1	1	1	1	1
C		1	Q	2	3	3
D			1	3	3	3
E		Q	2	1	2	2
F			2	3	1	3
	1	2	3	4	5	6

Arc Consistency

Figure 4.2: Propagators on 6-queens problem

are checked if they are consistent with the constraint. If instantiating a value does not make the constraint consistent then that value is removed from the domain of the variable. In general, forward checking works well with constraints of small arity (binary for instance) since only a small number of variables need to be instantiated before propagation can take place on a constraint.

Figure 4.2 presents an example of forward checking on the 6-queens problem. For this example, the first three variables were instantiated in the order of their names (1, 2, and 3). The values (1,2,3) on the chess board represent the level in the search when the domain values were removed by the forward checking propagator from the uninstantiated variables. As a result, variable 4 has two remaining domain values (A and F) and variable 5 and 6 only have one domain value remaining (D). Clearly no solution can be extended from this partial solution since variables 5 and 6 only have the same value remaining in their domains. However, at this point in the search forward checking does not recognize this situation since variables 5 and 6 are both uninstantiated.

A more thorough method of propagation is an arc-consistency propagator. The propagator described here (known for binary constraints as AC-1 in [12]) is relatively simple and inefficient compared to more recent arc-consistency algorithms found in [5].

At each point in the search the arc-consistency propagator examines each constraint and examines the domain values of the uninstantiated variables in the constraint's scheme. For each of these domain values, the arc-consistency propagator exhaustively searches in the remaining uninstantiated variables in a constraint's scheme for domain values that if instantiated would make the constraint consistent. If the propagator does not find such an instantiation, the domain value is removed from the variable's domain. This propagation of each constraint is iterated until no domain values are reduced.

Figure 4.2 presents an example of an arc consistency propagator on the 6-queens problem. For this example, the first three variables were instantiated in the order of their names (1, 2, and 3). Like the forward checking example, the values (1,2,3) on the chess board represent the level in the search when the domain values were removed by the arc consistency propagator from the uninstantiated variables. Besides the domain values removed by the forward checking propagator, the arc consistency propagator also removes the remaining values from each of the remaining uninstantiated variables. Thus the domains of all uninstantiated variables are empty and the search can backtrack since a solution cannot be found if any variable's domain is empty.

Although an arc-consistency propagator can reduce the domains of uninstantiated variables more than a forward checking propagator can, it is computationally more expensive. For some problems, backtracking with a forward checking propagator outperforms backtracking with an arc consistency propagator in terms of CPU time.

To achieve a high level of propagation with limited computation, specialized propagators, which take advantage of the constraint's structure, can be devised for particular classes of constraints. Examples of constraints for which specialized propagators have been devised are the all-different constraint [15] and the cardinality constraint [16].

We now describe the specialized propagators that were devised for the loosening approach.

Distribution Propagator

Since the distribution exception constraint is a generalization of the even distribution constraint, the same propagator is used on both. The two distribution constraints are essentially cardinality constraints for which a specialized arc consistency propagator is described in [16]. The propagator described here is simpler and does not achieve arc consistency.

Each constraint is assigned a day, an attribute value, a minimum value and a maximum value. We refer to the batches that are assigned the constraint's attribute value as attribute batches. For the even distribution constraints, the minimum value is assigned the same value as the maximum value (since the number of attribute batches assigned to a day must equal the maximum value).

The scheme of the constraint consists of all variables that belong to the constraint's day. The propagator first calculates the number of variables in the constraint's scheme that are instantiated with an attribute batch. This value is referred to as the instantiated count and represents the number of attribute batches that have already been assigned. The number of uninstantiated variables that have attribute batches remaining in their domain is then added to the instantiated count to get the available count. This value represents the possible number of attribute batches that can be assigned to the variables.

The constraint fails if either the instantiated count is greater than the maximum value or the available count is less than the minimum value.

If the instantiated count equals the maximum value, the propagator removes the attribute batches from all uninstantiated variable domains. If the available count equals the minimum value, the propagator removes all batches besides the attribute batches from all uninstantiated variable domains that contain an attribute batch.

Change-over Propagator

The change-over constraint propagator is presented in Algorithm 2. Attribute functions (Section 3.2.2) are used in this algorithm, as well as two CSP specific functions: *Domain*(s) and *Assignment*(s). The *Domain*(s) function returns the current domain values of the slot s, and the *Assignment*(s) function returns the batch that is instantiated to slot s. The propagator has the following input:

- the maximum change-over value (maxchgovr) is the constraint's parameter value and represents the number of soft constraint violations that can occur in the parameterized constraint,
- the two constraint attribute values: chgformer and chglatter, and
- a set of variables $s_1, \ldots, s_k$ that represents the variables in a parameterized change-over constraint, where $Position(s_i) < Position(s_j)$ if $i < j$.

The propagator removes values from the domains of variables and returns a boolean value that indicates if the constraint has failed.

The propagator calls one function: IntChgOvr (Algorithm 3). This function has the following input:

- the two constraint attribute values: chgformer and chglatter,
- the current batch that is being examined: b,
- the former attribute value (formerattvalue) of the last lot from the previous slot, and
- the recent failure value (recentfail) that indicates if the constraint failed between the previous pair of lots.

The IntChgOvr function returns the number of violations found between the last lot of the previous batch and all the lots within the current batch. Furthermore, the formerattvalue and recentfail variables are passed by reference to the function, so any changes made to those variables in the function are reflected in the procedure that called it.

The change-over constraint propagator presented is specifically for constraints between specific attribute values. In practice, the constraint can also be defined for an entire attribute set. For instance, a constraint can be defined on the attribute Exterior Colour, and a violation occurs if the colour changes between lots. This is equivalent to defining a constraint for each colour where the former attribute value is a specific colour ("Red") and the latter attribute is all colours besides the specified colour ("NOT Red"). However, it is more efficient to combine all the constraints into one. Thus, the implemented propagator handles both constraints between specific attribute values and constraints defined on all the attribute values of a set. It is also important to note that the unusual penalty evaluation never occurs in the case where a constraint is defined on a single attribute. For example, a constraint from "Red" to "NOT Red" cannot occur twice in a row.

Algorithm 2 Change-over constraint propagator

```
input: maxchgovr, chgformer, chglatter, and $s_1$ to $s_k$
output: updated variable domains

formerattvalue $\leftarrow$ NULL
recentfail $\leftarrow$ FALSE
chgovr $\leftarrow$ 0
for $i = 1$ to $k$ do
  if $s_i$ is instantiated then
    $b \leftarrow Assignment(s_i)$
    chgovr $\leftarrow$ chgovr + IntChgOvr(chgformer, chglatter, $b$, formerattvalue, recentfail )
    if chgovr > maxchgovr then
      FAIL
    end if
  else
    min-int-chgovr $\leftarrow$ LARGENUMBER
    for all $b \in Domain(s_i)$ do
      temp-formerattvalue $\leftarrow$ formerattvalue
      temp-recentfail $\leftarrow$ recentfail
      int-chgovr $\leftarrow$ IntChgOvr(chgformer, chglatter, $b$, temp-formerattvalue, temp-recentfail )
      if int-chgovr + chgovr > max-chgovr then
        remove $b$ from $Domain(s_i)$
      end if
      if int-chgovr < min-int-chgovr then
        min-int-chgovr $\leftarrow$ int-chgovr
      end if
    end for
    chgovr $\leftarrow$ chgovr + min-int-chgovr
    formerattvalue $\leftarrow$ NULL
    recentfail $\leftarrow$ FALSE
  end if
end for
```

Algorithm 3 Change-over internal batch violation counter

```
input: chgformer, chglatter, $b$, formerattvalue, and recentfail
output: formerattvalue, recentfail, and int-chgovr

int-chgovr $\leftarrow$ 0
for all l $\in$ *Lots*($b$) {Lots are selected in the order sequenced} do
  if formerattvalue $\neq$ NULL then
    if recentfail = FALSE AND formerattvalue = chgovrformer AND *ChgOverLatter*(l) = chgovrlatter then
      int-chgovr $\leftarrow$ int-chgovr + 1
      recentfail $\leftarrow$ TRUE
    else
      recentfail $\leftarrow$ FALSE
    end if
  end if
  formerattvalue $\leftarrow$ *ChgOverFormer*(l)
end for
RETURN int-chgovr
```

Run-length Propagator

The run-length constraint propagator is presented in Algorithm 4. Like the change-over constraint, attribute functions (Section 3.2.2) are used in this algorithm, as well as two CSP specific functions: *Domain*(s) and *Assignment*(s). These functions are described in the previous section. The propagator has the following input:

- the maximum run-length value (maxrunlength) is the constraint's parameter value and represents the maximum run-length value allowed,
- the run-length attribute value (runlen-attvalue) represents the constraint's attribute value, and
- a set of variables $s_1, \ldots, s_k$ that represents the variables in a parameterized run-length constraint, where $Position(s_i) < Position(s_j)$ if $i < j$.

The propagator removes values from the domains of variables and returns a boolean value that indicates if the constraint has failed.

The run-length constraint propagator presented is specifically for constraints between specific attribute values. Similar to the change-over constraint, the constraint can also be defined for an entire attribute set. For example, a constraint can be defined on the attribute exterior colour where the constraint is violated if any colour is repeated too many times. To improve efficiency, the implemented propagator handles both constraints with a specific attribute value and constraints defined on all the attribute values of a set.

All-Different Propagator

Every time a batch is instantiated to a slot, the batch is removed from any domain that contains it. This propagator could be improved using the all-different constraint

Algorithm 4 Run-length constraint propagator

```
input: maxrunlength, runlen-attvalue, and $s_1$ to $s_k$
output: updated variable domains

runlength $\leftarrow$ 0
for $i = 1$ to $k$ do
  if $s_i$ is instantiated then
    $b \leftarrow$ *Assignment*($s_i$)
    for all l $\in$ *Lots*($b$) {Lots are selected in the order sequenced} do
      if runlen-attvalue = *RunLenAttribute*(l) then
        runlength $\leftarrow$ runlength + *LotSize*(l)
      else
        runlength $\leftarrow$ 0
      end if
      if runlength > maxrunlength then
        FAIL
      end if
    end for
  else
    if runlength > (maxrunlength $-$ batchsize) then
      for all $b \in$ *Domain*($s_i$) do
        intrunlength $\leftarrow$ 0
        for all l $\in$ *Lots*($b$) {Lots are selected in the order sequenced} do
          if runlen-attvalue = *RunLenAttribute*(l) then
            intrunlength $\leftarrow$ intrunlength + *LotSize*(l)
          else
            BREAK
          end if
        end for
        if (intrunlength + runlength) > maxrunlength then
          remove $b$ from *Domain*($s_i$)
        end if
      end for
    end if
    runlength $\leftarrow$ 0
  end if
end for
```

arc consistency propagator described in [15]. Such a propagator would remove more domain values but would also be more computationally expensive.

4.3 Branch and Bound Approach

In this section we describe backtracking on a CSOP model with a branch and bound approach. For this approach, we begin with a loose bound on the evaluation function and tighten the bound until no solution is found.

Applying backtracking to a CSOP model requires backtracking to be extended to incorporate an evaluation function. Two possible approaches are to include either iterative deepening (ID) or branch and bound (BB) as part of the backtracking algorithm. Both of these approaches set a bound on the quality of the solution that backtracking is allowed to find and require a heuristic for estimating the quality of a partial solution to the problem. For both approaches, backtracking occurs if the quality of a partial solution exceeds the bound on the solution. The difference between the two approaches is that ID begins with a tight bound and increases it until a solution can be found, while BB begins with a loose bound (such that a solution can be found) and tightens the bound until no solution can be found.

4.3.1 Evaluation and Heuristic Functions

For the branch and bound approach, all the soft constraints are represented by an evaluation function. The evaluation function takes as its input a solution to the hard constraints and returns the total number of penalty values incurred by the soft constraints.

The heuristic function devised is essentially the evaluation function applied to partial solutions. The heuristic function takes as its input a partial solution to the hard constraints and returns the total number of penalty values incurred by the batches that have been sequenced. It is possible that a heuristic function could be devised that gives an even better estimate of the quality of the partial solution, for instance counting the number of internal violations for the batches that have not been sequenced yet. However, this was not done as part of this thesis.

4.3.2 Branch and Bound Scheme

The backtracking algorithm begins with a problem initialized with a loose bound value. Specifically we set the bound value to the total penalty value incurred by the greedy search solution. By using this value we begin with a problem that is guaranteed to have a solution and is reasonably tight.

After backtracking finds a solution, we take the total penalty value for the solution, reduce it by the largest common divisor of the constraint penalty values in the problem instance (a value of one for the problem instances we examine), and set this as the new bound value. The branch and bound algorithm then continues, and backtracks whenever the heuristic function value of a partial solution exceeds the current bound. If it finds a solution with the current bound value, we reduce the bound value again. This process is continued until no solution can be found. In this case, the last solution found is an optimal solution.

4.3.3 Variable and Value Ordering

The variable ordering for the branch and bound approach is fixed to the ordering of the slots in time. This fixed ordering is used to simplify the way the heuristic function is implemented.

The value ordering is based on the original greedy search solution. For each variable, the value assigned in the original solution is placed first in the variables domain. This value ordering is the same as the one used by the loosening approach.

4.3.4 Consistency Propagators

The distribution propagator and the all-different propagator are the same as the ones described for the loosening approach. Since the change-over constraints and run-length constraints are represented by an evaluation function, there are no propagators for these constraints.

4.4 Problem Splitting

Since the problem is somewhat large, a method for splitting it into smaller sub-problems is used. This process is done for all three of the solution techniques.

The problem is divided into relatively equal size sub-problems by placing, for a particular assembly line, a specified number of consecutive production days in each sub-problem. The specified number of days in a sub-problem is referred to as the split size. If the split size does not divide the number of days on an assembly line, a sub-problem is generated with the remaining days. The sub-problems contain consecutive production days and are solved in order of the days they contain, where the unsolved sub-problem with earliest days is solved next. The domain values are selected for each sub-problem by using the original solution provided by the greedy search algorithm. In other words, for a particular sub-problem we assign batches that are sequenced by the greedy search algorithm to the same days as the days within a sub-problem. Furthermore, since soft constraint violations can occur between sub-problems, after a sub-problem is solved, the batch that was sequenced last is added to the beginning of the next sub-problem. This method of splitting does not completely model the problem, since run-length and change-over constraints may be influenced by batches that are sequenced before the last batch in the previous sub-problem, but it makes visible the majority of violations that can occur between days. It is also important to note that if the split size is one (one day per sub-problem), the soft constraints are the only n-ary constraints in the sub-problems.

Chapter 5

Results

In this chapter we present the results of applying the three solution techniques presented in Chapter 4 to the CSP models of six real-world problem instances. We begin by presenting the six problem instances we used for our experiments in Section 5.1. In Section 5.2, we show the results for the original greedy search algorithm on these problem instances. In Sections 5.3, 5.4 and 5.5 we present the results for the hill-climbing approach, the loosening approach, and the branch and bound approach, respectively. We conclude the chapter with a discussion of the results in Section 5.6.

5.1 Test Problems

There are six real-world problem instances examined in this thesis. Each problem instance represents a month's worth of orders for a manufacturing plant with two assembly lines. Table 5.1 shows for each problem instance the number of lots and batches, the maximum number of lots in a batch, the maximum number of slots in a day, and the number of different types of distribution exception, change-over, and run-length constraints. For the change-over constraints and the run-length constraints, we also present the original number of constraints as defined in TigrSoft's problem specifications as well as the number of combined constraints defined in our problem specification. The constraints were combined by either representing identical constraints that were defined for both assembly lines as one constraint, or by combining certain constraints that are defined on the same attribute. The majority of change-over constraints that were combined were constraints between individual colours and all other colours. In other words, individual constraints restricting change-overs from "X" to "NOT X", where X can be any colour, were combined into one change-over constraint.[1]

[1]For all of the problem instances, two of these colour constraints appeared to be defined wrong in the TigrSoft specification. One was defined from "Colour A" to "ANY LOT" and the other was defined from "Colour B" to "NOT Colour C". These constraints appear to have resulted from data entry errors by the users at the manufacturing company. We included these two constraints separately as part of our specification.

Prb #	# of Lots	# of Batches	Max. Lots in Batch	Max. Slots in Day	# of Dist. Exc.	# of Chg-over		# of Run-len	
						Orig.	Cmb.	Orig.	Cmb.
1	807	577	9	15	1	42	13	7	5
2	771	565	8	19	11	42	13	6	5
3	792	651	2	15	2	42	13	7	5
4	836	659	7	17	2	35	11	6	4
5	856	665	7	17	3	35	11	6	4
6	804	586	10	15	3	35	11	6	4

Table 5.1: Real-world problem instances

Problem	Penalties			Internal Lot Sequences	
	Chg-over	Run-len	Total	Penalties	Percent
1	6618	1400	8018	4191	52%
2	5042	0	5042	2113	42%
3	3362	50	3412	957	28%
4	2098	400	2498	1291	52%
5	2556	400	2956	1644	56%
6	2618	200	2818	1752	62%

Table 5.2: Greedy search

5.2 Greedy Search

The original solution technique used to solve the sequencing problem is a greedy search algorithm, described in Section 2.3. Table 5.2 presents the results of this algorithm when applied to the test problem instances. The table shows the total penalty values incurred by the soft constraints as well as the penalty values incurred between the lots within each batch (internal lot sequences). Since the internal sequence of lots within each batch is fixed, these internal violations can be seen as lower bounds on solutions of the problem instances. On average, about 50% of violations occur between batches and thus only 50% of the violations can potentially be reduced if the internal lot sequences remain fixed.

5.3 Hill Climbing

The first solution technique we examined is a hill-climbing algorithm, described in 4.1. The default settings for this algorithm are as follows.

To begin with, the hill-climbing algorithm used an initial solution provided by the greedy search algorithm. In Section 5.3.2 we examine the affect that a random initial solution had on the hill-climbing algorithm.

The sequencing of lots within a batch was fixed based on the sequence of lots within the original solution. In Section 5.3.3 we look at how optimizing the internal lot sequences affects the hill-climbing solutions.

The problem by default was divided into single day sub-problems. The affect of increasing the sub-problem size to include multiple days is presented in Section

Prb	Penalty Values			CPU	Improvement		
#	Chg-over	Run-len	Total	Time	Days	Diff.	Percent
1	6286	750	7036	114	18/42	982	12%
2	4579	0	4579	158	17/38	463	9%
3	3297	50	3347	136	22/44	65	2%
4	2033	200	2233	134	18/44	265	11%
5	2485	400	2885	133	21/44	71	2%
6	2360	200	2560	139	31/40	258	9%

Table 5.3: Hill climbing results

Prb	Total Penalties			CPU Time	
#	HC	HC-RAND	Percent	HC	HC-RAND
1	7036	7623	-8%	114	220
2	4579	5059	-10%	158	346
3	3347	3543	-6%	136	298
4	2233	2269	-2%	134	367
5	2885	2996	-4%	133	327
6	2560	2605	-2%	139	282

Table 5.4: Hill climbing with random initial solution

5.3.4. In Section 5.3.5 we look at how the removal of the even distribution constraint from the problem affected the solutions of multi-day sub-problems.

5.3.1 Improvement Over Greedy Search

Before examining these different modifications, let us first look at how hill-climbing improved on the greedy search solutions. Table 5.3 presents a comparison of the hill-climbing solutions and the greedy search solutions. The table shows for the hill-climbing algorithm the penalty values incurred by both parameterized constraints, the CPU time (in seconds) needed to solve the problem instances, the number of days improved on, and the difference and percent decrease in penalty values with respect to the greedy search technique. The percent reduction of penalty values for the problem instances ranged between 2% and 12% and approximately only half of the sub-problems are improved on over the greedy search.

5.3.2 Random Initial Solution

The hill-climbing algorithm by default used the greedy search solution as its initial solution. When the initial solution was replaced by a random one, the results significantly deteriorated. Table 5.4 compares the influence of a greedy search initial solution (HC) and a random initial solution(HC-RAND). As the table shows, when a random initial solution was used, the increase in penalty values ranged between 2% and 10% and the CPU time on average doubled. These results clearly indicate the importance of a good initial solution when using a hill-climbing technique on the problem.

Prb	Total Internal Penalties			Total Penalties		
#	Original	Optimized	Percent	Original	Optimized	Percent
1	4191	3868	8%	8018	9468	-18%
2	2113	1857	12%	5042	5091	-1%
3	957	907	5%	3412	3739	-10%
4	1291	1018	21%	2498	2564	-3%
5	1644	1124	32%	2956	3005	-2%
6	1752	1525	13%	2818	2981	-6%

Table 5.5: Greedy search solution with optimized internal lot sequences

5.3.3 Optimized Internal Lot Sequences

In this section we present results where the penalty values were minimized by sequencing the lots within each batch. The optimized lot sequences were produced by a simple generate and test procedure. For each batch, the lot sequences with the smallest total penalty value were selected. In the case of a tie, the sequence with the largest lot sequenced last was selected. Because of complexity issues we only completely optimized batches with seven or fewer lots. For batches with more than seven lots, we optimized the first seven lots in the batch and did not move the remaining lots. The most lots found in a batch for all the problem instances was ten.

Table 5.5 presents the internal and total penalties of the greedy search solutions with their original internal lot sequences and with new optimized internal lot sequences. Although the internal penalties were reduced for all the problem instances from 5% to 32%, the total penalty values increased for all the problem instances from 1% to 18%. This was expected though since reducing penalty values within each batch by rearranging their lots will not necessarily make the penalties disappear. Instead, the penalty values are incurred when the batches are sequenced. Furthermore, the greedy search solution was selected based on the sequence of lots within each batch. Thus it is not surprising that rearranging the lots reduced the quality of the greedy search solution.

Table 5.6 presents hill-climbing on batches with the original internal lot sequences (HC) and with the optimized lot sequences (HC-OPT). The results appear mixed, with changes in penalty values ranging between -2% and 10%. Two possible reasons for these mixed results are as follows.

One possible reason is that hill-climbing with the original lot ordering relies on a reasonable initial solution. As Table 5.5 indicates, when the lots are reordered, the quality of the original greedy search solution was reduced. Since the hill-climbing algorithm finds its solution by improving on its initial solution, it is possible that a worse initial solution will lead to a worse final solution. This is evident in the fact that the problem instances (problems 2, 4, 5) that had only a small increase in penalty values for their "optimized" greedy solutions (Table 5.5) are the same problem instances that improved the most when hill-climbing was applied (Table 5.6).

Another possible reason for mixed results is that the lots within a batch were rearranged such that the lots that are likely to cause a serious violation were moved to the edge of the batch. From within a batch no serious violations appeared, but

Prb	Total Penalties		
#	HC	HC-OPT	Percent
1	7036	7055	0%
2	4579	4470	2%
3	3347	3409	-2%
4	2233	2174	3%
5	2885	2605	10%
6	2560	2577	-1%

Table 5.6: Hill climbing with optimized internal lot sequences

when the batches were sequenced, these violations reappeared between the batches. In fact, it may actually increase the number of violations, since optimizing within each batch may have unforeseen consequences when the batches are sequenced. However, although rearranging the lots may not have eliminated the violations (and possibly increased them), it at least gave the search algorithm a chance to reduce them.

Overall these results demonstrate the potential of reordering the lots within the batches. However, it is not clear what a good internal lot sequence is without consideration of the larger problem.

5.3.4 Multi-day Sub-problems

In this section we examine the affect of multi-day sub-problems on the hill-climbing algorithm. Tables 5.7 and 5.8 present the change in penalty values, the CPU time, and the number of value swaps that occurred between days for problems with two (HC-2) and three (HC-3) days in each sub-problem, respectively. For the most part, the penalty values remained unchanged.

A likely reason for this is that, for the problem instances examined, the even distribution constraint significantly reduced the possibility of improving the solution by solving multiple days at a time. There are two possible reasons for this. First of all, the even distribution constraint for all the problem instances is defined on an attribute that contains more than 200 attribute values. Since there are only approximately 600 batches in each problem, many attribute values only have one or two batches associated with them. Since the even distribution constraint defines for each day and attribute value the number of batches with the attribute value that can be assigned to the day, many days do not share batches. Thus when solving small multi-day problems (two or three days), it is unlikely that the days within a sub-problem will share batches.

Another reason improvements are not likely is that the even distribution constraint attribute is similar to many of the attributes used by the soft constraints. In other words, if two batches are considered similar by the even distribution constraint then it is likely that the two batches are consider similar by the soft constraints. Thus even if two batches are shared by two days, swapping them may not have any affect on the soft constraint violations.

Evidence for both of these arguments can be seen in the fact that for all six problem instances, only once was a value swapped between days. Given the significant increase in CPU time when multi-day sub-problems were solved, solving single

Prb	Total Penalties			CPU Time		Day
#	HC	HC-2	Percent	HC	HC-2	Swaps
1	7036	7034	0%	114	226	0
2	4579	4579	0%	158	317	0
3	3347	3292	2%	136	288	1
4	2233	2233	0%	134	258	0
5	2885	2880	0%	133	239	0
6	2560	2560	0%	139	334	0

Table 5.7: Hill climbing with split size two

Prb	Total Penalties			CPU Time		Day
#	HC	HC-3	Percent	HC	HC-3	Swaps
1	7036	7034	0%	114	441	0
2	4579	4579	0%	158	570	0
3	3347	3336	0%	136	495	1
4	2233	2233	0%	134	416	0
5	2885	2881	0%	133	502	0
6	2560	2561	0%	139	624	0

Table 5.8: Hill climbing with split size three

day sub-problems makes sense.

5.3.5 Removing the Even Distribution Constraint

In the last section we showed results indicating that the even distribution constraint prevents values from being shared between days. In this section we present results for when the even distribution constraint was removed from the problem instances. It is important to note that removing this constraint does not completely undermine the even distribution of batches. Since the problem instances are split into smaller sub-problems, the even distribution constraint is implied by the assignment of batches to each sub-problem. For example, when solving the problem with two day sub-problems, a batch could move at most one day compared to the greedy search solution of the problem. Thus in some sense the batches would remain distributed (as long as the sub-problems are not too big). However, since we have no algorithmic method for determining if a distribution is acceptable (except for the solutions provided by the greedy search algorithm), there is no way to determine if solutions obtained by removing the even distribution constraint are acceptable (besides having each solution instance evaluated manually by an expert).

Tables 5.9 and 5.10 present the penalty values, CPU time, and the number of value swaps that occurred between days for problem instances with no even distribution constraint and two (HC-NE2) and three (HC-NE3) days in each sub-problem, respectively. Unlike the results presented in the previous section, significant improvements were found for each problem instance. Furthermore, the table shows that a significant number of value swaps occurred between days. This is further evidence of the even distribution constraint's influence on the problem. The only down side of removing the constraint is that the CPU time increased significantly

Prb	Total Penalties			CPU Time		Day
#	HC	HC-NE2	Percent	HC	HC-NE2	Swaps
1	7036	6740	4%	114	692	36
2	4579	4421	3%	158	955	27
3	3347	3199	4%	136	816	34
4	2233	2127	5%	134	1037	46
5	2885	2494	14%	133	1269	64
6	2560	2196	14%	139	1002	45

Table 5.9: Hill climbing with no even distribution and a split size two

Prb	Total Penalties			CPU Time		Day
#	HC	HC-NE3	Percent	HC	HC-NE3	Swaps
1	7036	6444	8%	114	1961	39
2	4579	4370	5%	158	3216	43
3	3347	3256	3%	136	2658	44
4	2233	2166	3%	134	3904	74
5	2885	2535	12%	133	3073	58
6	2560	2169	15%	139	2665	57

Table 5.10: Hill climbing with no even distribution and a split size three

for these experiments. This can be attributed to the increase in the neighborhood size that occurred as a result of the even distribution constraint being removed. Overall, these results indicate the potential reduction of penalty values that can be obtained if the even distribution constraint is relaxed.

5.4 Backtracking with a Loosening Approach

The second solution technique that we examine is backtracking with a loosening approach, described in Section 4.2. The default settings for this algorithm were as follows.

First of all, no propagation occurred on the parameterized constraints. We show how propagation on these constraints negatively affected the results in Section 5.4.2.

Each slot's domain values were ordered by placing the batch that was assigned to the slot in the original solution first in the slot's domain. In Section 5.4.3 the affect of removing this value ordering is examined.

After solving a sub-problem, the sub-problem solution was compared with the greedy search solution and the sub-problem solution with lowest penalty value total was selected. We show how not selecting the best solution to sub-problems reduced the quality of the overall solution in Section 5.4.4.

The sequencing of lots within a batch was fixed based on the sequence of lots within the original solution. In Section 5.4.5, we look at how reducing the penalty values that occur within each batch by reordering their lots influenced the solutions.

The failure limit for each parameterized constraint was assigned relative to the penalty value of the constraint. This was done by multiplying the penalty value of the constraint by a fixed value of 200. Thus a constraint with a penalty value of

Prb	Penalties			CPU	Improvement		
#	Chg-over	Run-len	Total	Time	Days	Diff.	Percent
1	6087	1250	7337	916	17/42	681	8%
2	4640	0	4640	662	9/38	402	8%
3	3307	50	3357	63	18/44	55	2%
4	2008	300	2308	320	19/44	190	8%
5	2483	400	2883	336	14/44	73	2%
6	2402	200	2602	345	17/40	216	8%

Table 5.11: Backtracking with loosening approach

50 would have a failure limit of 10000. In Section 5.4.6 we examine the influence of different fixed values, and different ways of counting parameterized constraint failures.

Each problem was also divided into one day sub-problems. In Section 5.4.7 we examine how increasing the number of days within a sub-problem influenced the solutions that were found.

5.4.1 Improvement over Greedy Search

Before examining all of these different modifications, we first look at how the loosening approach, with its default settings, improved on results of the greedy solution technique. Table 5.11 shows the penalty values of the loosening approach solutions and their improvement over the penalty values of the greedy search solutions. The table shows the penalty values for both parameterized constraints, the CPU time (in seconds) of the loosening approach, the number of days that the loosening approach improved results on, the difference in penalty values between the greedy search and the loosening approach, and the percent change in penalty values over the greedy search technique. The percent reduction of penalty values for the problem instances ranged between 2% and 8%. Although the majority of penalty values incurred were from change-over constraints, the loosening approach significantly reduced the run-length penalty values for two of the six problem instances. The algorithm also took between six and fifteen minutes to find a solution.

5.4.2 Soft Constraint Propagation

Although specialized propagators were devised for both the change-over constraints and run-length constraints, for the most part, they proved to be detrimental. Table 5.12 compares the loosening approach with (LN-PROP) and without (LN) the propagation of parameterized constraints. The increase of total penalty values for the problem instances ranged between 0% and 6%. The CPU time (in seconds) used to find a solution decreased significantly when propagation was used (up to eight times faster). This decrease was expected, since propagation reduces the search space. However, the increase in penalty values was not expected. The reason why propagation was detrimental to finding a good solution seems to relate to the influence of propagation on constraint failures. Since propagation reduces the domains of variables that have not been instantiated yet, it reduces the relationship between which constraints "caused" the problem to be unsolvable and which constraints actually

Prb	Total Penalties			CPU Time	
#	LN	LN-PROP	Percent	LN	LN-PROP
1	7337	7628	-4%	916	624
2	4640	4896	-6%	662	78
3	3357	3364	0%	63	38
4	2308	2340	-1%	320	60
5	2883	2917	-1%	336	77
6	2602	2684	-3%	345	68

Table 5.12: Loosening approach with soft propagation

Prb	Total Penalties			CPU Time	
#	LN	LN-NVAL	Percent	LN	LN-NVAL
1	7337	7470	-2%	916	903
2	4640	4737	-2%	662	770
3	3357	3370	0%	63	73
4	2308	2334	-1%	320	319
5	2883	2899	-1%	336	329
6	2602	2614	0%	345	373

Table 5.13: Loosening approach with no value ordering

failed the most. For example, assume we have two arbitrary parameterized constraints A and B, where constraint A must be loosened in order for a problem to be solved and constraint B does not. Then it is possible that constraint A's propagator reduces the domains of the uninstantiated variables, but does not fail, and later in the search constraint B fails because of the domain values that were removed by A. Hence it is possible that constraint B fails more frequently than constraint A and thus B is selected over A to be loosened.

Although it is possible that even without propagation constraint B may be selected over A to be loosened, it seems reasonable that propagation decreases the relationship between which constraints must be loosened to find a solution and how often these constraints fail.

5.4.3 Value Ordering

The loosening approach by default used a value ordering that was based on the greedy search solution. Table 5.13 compares the loosening approach with the value ordering (LN) and without (LN-NVAL). Overall the removal of the value ordering decreased the quality of the solutions. Including the value ordering appears to give the loosening approach a good solution to build on.

5.4.4 Best Solution Selection

When the value ordering was employed, one might expect that the backtracking algorithm would be guaranteed to do the same or better than the greedy search algorithm, since the loosening approach (with the appropriate parameter settings) can find the greedy search solution backtrack free. However, this is not the case

Prb	Total Penalties		
#	LN	LN-NOBS	Percent
1	7337	7491	-2%
2	4640	4692	-1%
3	3357	3418	-2%
4	2308	2315	0%
5	2883	2889	0%
6	2602	2608	0%

Table 5.14: Loosening approach with no best solution selection

Prb	Total Penalties		
#	LN	LN-NVAL-NOBS	Percent
1	7337	8718	-19%
2	4640	5353	-15%
3	3357	3441	-3%
4	2308	2532	-10%
5	2883	3069	-6%
6	2602	3028	-16%

Table 5.15: Loosening approach with no value ordering and no best solution selection

since the loosening approach's relaxation schedule is not perfect. Specifically, the relaxation schedule may loosen the constraints such that the original solution does not satisfy the constraints but a worse solution does.

For this reason, after a sub-problem was solved it was compared with the original solution and the best solution to the sub-problem was selected. Table 5.14 compares the loosening approach with (LN) and without (LN-NOBS) a best solution selection process. The decrease in solution quality can be attributed to sub-problem solutions where the imperfect relaxation schedule missed the original solution and found a worse one.

However, even when the best solution selection process is removed it is possible that the value ordering may provide a backtrack free solution. Thus when both the value ordering and the best solution selection process were removed, the backtracking algorithm had no original solution to fall back on. This situation (LN-NVAL-NOBS) is presented in Table 5.15. As the results clearly show, the relaxation schedule was indeed not perfect.

5.4.5 Optimized Internal Lot Sequences

In Section 5.3.3 we examined the affect of optimizing internal lot sequences on the hill-climbing technique. In this section we present its affect on the loosening approach. Table 5.16 compares the loosening approach with the original internal lot ordering (LN) and with the optimized ordering (LN-OPT). Similar to the hill-climbing technique the results were mixed, with changes in penalty values ranging from -5% and 9%. Reasons for this are similar to those described in Section 5.3.3.

Prb	Total Penalties		
#	LN	LN-OPT	Percent
1	7337	7721	-5%
2	4640	4549	2%
3	3357	3443	-3%
4	2308	2265	2%
5	2883	2618	9%
6	2602	2625	-1%

Table 5.16: Loosening approach with optimized internal lot sequences

5.4.6 Failure Limits

The failure limits for each soft constraint were set by multiplying each constraint's penalty value by a constant value. The results presented so far fixed this value to be 200. In this section we examine the affect modifying this value has on the quality of the solutions. Figure 5.1 presents the influence of different failure limit constants on the total penalty values for each problem instances. The x-axis represents the different failure limit constants that were tried and the y-axis represents the percent improvement over the greedy search results. Each line in the graph, represents either a problem instance or the average of the problem instances.

As the failure limit constants increased, the CPU time required to solve a problem increased as well. For some of the problems (problems 1, 2, 3, and 5) the percent improvement remained relatively the same when the failure limit constant was increased. However, for two problem instances (problems 4 and 6) the percent improvement decreased significantly when the failure limit constant was increased. This shows that a high constraint failure limit does not necessarily imply an increase in the quality of a solution and in fact decreased the quality of the solutions.

We also tried a different method of counting constraint failures. Instead of counting all constraint failures that occurred in the search, we tried only counting the constraint failures that occurred at the deepest level of the search tree. In other words, when the backtrack algorithm was able to reach a new level in the search tree, it only counted constraint failures that occurred at that level of the tree. The intuition behind this idea is that the constraints that fail at the deepest part of the search tree are the constraints that should be loosened, since doing so would allow the search to go deeper. Unfortunately, results showed that this idea does not improve over the original counting technique.

5.4.7 Multi-day Sub-problems

By default, the problem was split into one day sub-problems. When more than one day was included in each sub-problem, the results for the most part decreased slightly in quality. Table 5.17 presents the change in penalty values and CPU time, compared to the loosening approach on one day sub-problems, for problems with two (LN-2) and three (LN-3) days in each sub-problem. Overall the penalty values did not change significantly when the size of each sub-problem was increased.

The reasons for this are explained in Section 5.3.4. However, unlike the hill-climbing technique, backtracking did worse on multi-day sub-problems. This prob-

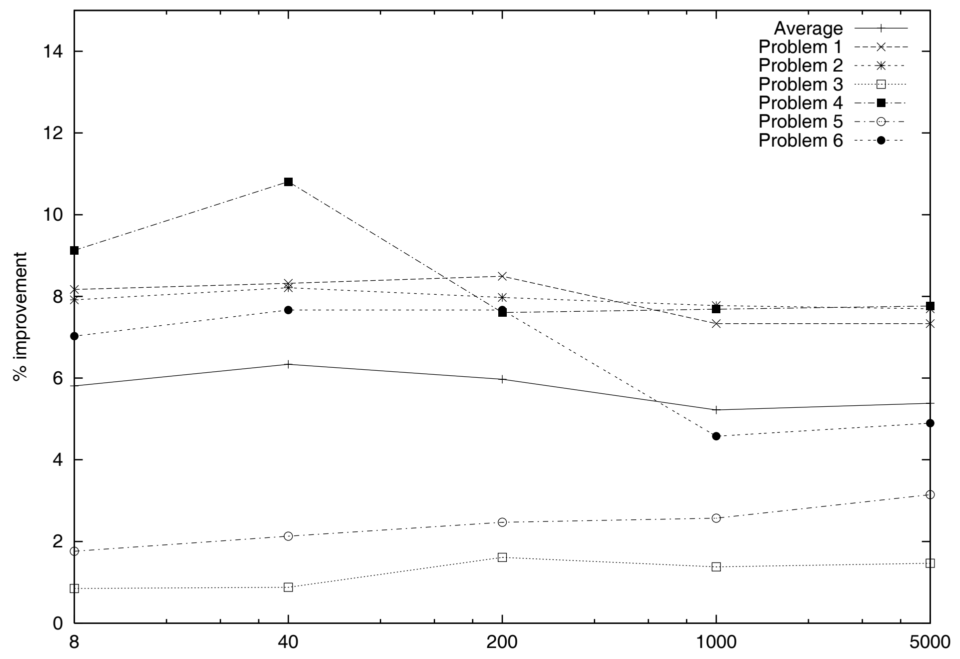

Figure 5.1: Different relative failure limit constants

Prb	Penalties %		CPU Time %	
#	LN-2	LN-3	LN-2	LN-3
1	-2%	-2%	-258%	-775%
2	0%	0%	-319%	-742%
3	1%	-1%	-2132%	-5202%
4	3%	3%	-230%	-481%
5	-1%	-1%	-300%	-684%
6	-1%	-2%	-186%	-555%

Table 5.17: Loosening approach split size two and three results

ably can be attributed to the relaxation schedule. Increasing the size of the sub-problems increases the number of constraints in a problem. This in turn increases the number of constraints that the relaxation schedule must select from. Thus it is more likely that the relaxation schedule will pick an inappropriate constraint to loosen.

5.5 Backtracking with a Branch and Bound Approach

We now present results for backtracking with a branch and bound approach, described in Section 4.3. The default settings for this algorithm were as follows.

Since initial tests of the algorithm proved to be intractable, with one problem instance taking more than five days without returning a solution, a time limit of two hours was set for each sub-problem. In Section 5.5.2 we show results for when the time limit was reduced to one minute.

The sequencing of lots within a batch was fixed based on the sequence of lots

Prb	Penalties			CPU	Improvement		Days
#	Chg-over	Run-len	Total	Time	Days	Percent	Optimized
1	6252	750	7002	1103	28/42	13%	42/42
2	4528	0	4528	48911	24/38	10%	34/38
3	3256	50	3306	442	27/44	3%	44/44
4	1906	300	2206	15385	39/44	12%	43/44
5	2362	400	2762	2677	36/44	7%	44/44
6	2289	200	2489	1305	36/40	12%	40/40

Table 5.18: Backtracking with branch and bound approach

within the original solution. In Section 5.5.3, we look at how reducing the penalty values that occur within each batch by reordering their lots influenced the solutions.

Each problem instance was also divided into one day sub-problems. In Section 5.5.4 we also examine how increasing the number of days within a sub-problem influenced the solutions that were found.

5.5.1 Improvement over Greedy Search

In this section we compare the branch and bound approach results, with its default settings, to the greedy search results. Table 5.18 shows the penalty values of the branch and bound solutions and their improvement over the penalty values of the greedy search solutions. Specifically, the table shows the penalty values for both soft constraints, the CPU time (in seconds), the number of days that the branch and bound approach improved results on, the percent change in penalty values over the greedy search technique, and the number of sub-problem solutions that were proved to be optimal solutions.

The percent reduction of penalty values for the problem instances ranged between 3% and 13%. Of the six problem instances, four of them had all of their sub-problem solutions proven optimal within the two hour per sub-problem time limit. The other two problem instances had in total only five sub-problems with sub-optimal solutions. These five sub-problem solutions may in fact be optimal, but they were not proven so within the time limit.

The CPU time required to solve these problem instances varied significantly. The four problem instances that have optimal sub-problem solutions, took between five and forty-five minutes. The other two problem instances took between four and fourteen hours to solve.

5.5.2 Reduced Time Limit

Since four of the problem instances were proved optimal in a relatively short time, we decided to significantly reduce the sub-problem time limit to one minute. Table 5.19 compares the branch and bound approach with a two hour (BB) and a one minute (BB-FAST) sub-problem time limit. As the table shows, only one problem instance's total penalty values slightly increased when the one minute time limit was imposed. Of the four problem instance solutions that were proven optimal with a two hour time-limit, only one of them was proven optimal with only a one minute time limit. However, the other three solutions all remained optimal but were not

Prb	Total Penalties			BB-FAST	Days
#	BB	BB-FAST	Percent	CPU Time	Optimized
1	7002	7002	0%	422	40/42
2	4528	4528	0%	1081	23/38
3	3306	3306	0%	443	44/44
4	2206	2218	-1%	1489	26/44
5	2762	2762	0%	1012	38/44
6	2489	2489	0%	959	36/40

Table 5.19: Branch and bound approach with reduced time limit

Prb	Total Penalties			CPU Time	Days
#	BB	BB-OPT	Percent	BB-OPT	Optimized
1	7002	6925	1%	1060	42/42
2	4528	4403	3%	50466	33/38
3	3306	3348	-1%	499	44/44
4	2206	2137	3%	15464	43/44
5	2762	2479	10%	2901	44/44
6	2489	2500	0%	1329	40/40

Table 5.20: Branch and bound approach with optimized internal lot sequences

proven so. For the two problem instances that were not proven optimal with a two hour time limit, similar solutions were found with a one minute time limit. However, almost half of the sub-problem solutions were not proven to be optimal. Overall these results seem to indicate that, for this problem, finding a solution to a sub-problem is relatively easy, but proving a solution does not exist is potentially hard.

5.5.3 Optimized Internal Lot Sequences

In Sections 5.3.3 and 5.4.5 we examined the affect of optimizing internal lot sequences on the other two solution techniques. Table 5.20 compares the branch and bound approach with the original internal lot ordering (BB) and with the optimized ordering (BB-OPT). Similar to the other two solution techniques the results were mixed, with changes in penalty values ranging from -1% and 10%. In Section 5.3.3, two possible reasons were given for why these results were mixed. One of the possible reasons was that optimizing the internal lot sequences can have a detrimental affect on the overall quality of possible solutions. The branch and bound approach proves this since two of the problem instances (Problems 3 and 6) have optimal sub-problem solutions that were worse when the internal lot sequences were optimized.

5.5.4 Multi-day Sub-problems

Like the other two solution techniques, the tightening approach was applied to problem instances with multi-day sub-problems. In general the branch and bound approach did significantly worse when applied to multi-day sub-problems. The reason for this is that the complexity of the problem increased significantly. Because

Prb	Penalty Decrease				CPU Time		
#	GR	HC	LN	BB	HC	LN	BB
1	8018	12%	8%	13%	114	916	1103
2	5042	9%	8%	10%	158	662	48911
3	3412	2%	2%	3%	136	63	442
4	2498	11%	8%	12%	134	320	15385
5	2956	2%	2%	7%	133	336	2677
6	2818	9%	8%	12%	139	345	1305

Table 5.21: Summary of results

of this, the two hour time limit was reached for most sub-problems without finding an optimal solution and hence the results were not near optimal. The time required to obtain these solutions took at least one day for every problem instance.

5.6 Discussion

In this chapter we have presented the results of three solution techniques. In this section we review and compare some of the results.

Table 5.21 presents the results of the three solution techniques with their default settings. As the table shows, the branch and bound algorithm (BB) found the best results for all problem instances. This was expected since most of the sub-problem solutions were found to be optimal. The next best algorithm, in terms of penalty value reduction, was the hill-climbing algorithm (HC) which found better (or the same) solutions compared to the loosening approach (LN) for all problem instance.

In terms of CPU time, the hill-climbing algorithm was the best overall, followed by the loosening approach. The branch and bound algorithm took significantly more time to find solutions than the other approaches. However the branch and bound algorithm was also run with a sub-problem time limit of one minute. With this time limit, the branch and bound algorithm found nearly identical results as it did with its default time limit of two hours and only took between 443 and 1489 seconds to obtain these results.

In our discussions with TigrSoft, they explained that solutions with a 5% reduction in penalty values that could be found in less than 30 minutes (1800 seconds) would be considered significant. All three algorithms were capable of finding solutions to the six problem instances within 30 minutes. For four of the problem instances (problems 1, 2, 4, and 6) we were able to obtain more than a 5% reduction in penalty values with any of the three solution techniques. For the remaining two problem instances we obtained at least a 2% penalty value reduction with any of the three solution techniques.

Chapter 6

Conclusions and Future Work

In this chapter we present possible ways to improve and extend the research presented in this thesis. Following that we conclude with a summary of our contributions.

The research we have presented in this thesis is a starting point for possible future work. To begin with, different models of the problem could be examined. An obvious choice would be to apply our solution techniques to the batch model (where the variables are batches). Given the structure and tightness of the constraints, it may be possible that this model will outperform the slot model that we examined. Furthermore, different ways of handling the internal lot sequences could be examined. Instead of fixing the lot sequences within each batch, several possible sequences could be included for each batch. Another way to improve the model would be to redefine the even distribution constraint. This would require an investigation of the manufacturing company's procedures and goals, since there is no clear declarative definition of what a good even distribution is.

There are also many improvements that could be made to the three solution techniques that we devised. To begin with, the hill-climbing algorithm can be improved by including random walking, random restart, and tabu search or simulated annealing. Furthermore, different neighborhood functions could be applied to this problem. For example, instead of swapping values between pairs of variables, values may be rearranged within sets of three variables. However, implementing such a neighborhood function for every possible combination of three variables would increase the size of the neighborhoods.

The backtracking algorithms also have room for improvement. The propagation techniques could be improved for both backtracking algorithms. This would include implementing the all-different and cardinality arc-consistency propagators described in [15] and [16], respectively. Furthermore, the variable and value ordering could possibly be improved. This would involve defining heuristics specific to the problem (see [4] for examples of scheduling heuristics). The loosening approach can clearly be improved by changing how constraints are selected to be loosened. Such a change would require analyzing the search and deducing which constraint needs to be loosened for a solution to be found. Such an improvement might also allow propagation on soft constraints without influencing the constraint selection process. The branch and bound approach can also be improved by introducing a better heuristic. An obvious heuristic would be to count the internal penalty violations of the batches that have not been instantiated yet.

There are also several other solution techniques that can be potentially applied to the vehicle sequencing problem. For instance, Zhao and Goebel [25] present a depth-first search and best-first repair algorithm for solving a dispatcher scheduling problem. Essentially this algorithm extends partial solutions as far as possible. Instead of backtracking, it then applies a repair algorithm (a form of local search) on the partial solution until a new partial solution is found that can be extended. If no extendable partial solution can be found with the repair algorithm, soft constraints are temporarily removed from the problem until the partial solution can be extended.

Oddi and Smith [13] present another possible approach. They proposed an iterative sampling algorithm with stochastic variable and value ordering heuristics and applied this algorithm to an extended version of the job-shop scheduling problem. Iterative sampling alone involves randomly exploring different paths in a search tree. Including stochastic variable and value ordering heuristics implies that the exploration of the search tree is guided by information provided by the heuristics. For the algorithm described by Oddi and Smith, the amount of randomness at any point in the search tree is dependant on the information provided by its heuristics. If a heuristic evaluates a particular choices as being significantly better than any other, then the algorithm is likely to select that choice. However, if several choices are evaluated as being almost as good (or the same) as the best choice, then the algorithm's choice will be more random.

In this thesis, we have introduced a real-world optimization problem that we modeled and solved using a constraint-based approach. We presented several possible ways to model the vehicle assembly line sequencing problem as a CSP. For one of these models, we applied three different techniques. All three of these techniques improved results over TigrSoft's greedy search algorithm for all six problem instances. For four out of the six problem instances (problems 1, 2, 4, and 6) we were able to achieve, for all three solution techniques, improvements considered significant by TigrSoft. Furthermore, all three techniques were capable of finding solutions within TigrSoft's thirty minute time requirement. We also demonstrated the importance of decomposing the problem into one-day sub-problems. We conjectured that because of the tightness of the even distribution constraint and its relationship with the other constraints, such a decomposition has little affect on the potential quality of an overall solution. For nearly all of these one-day sub-problems, we proved optimal solutions within a reasonable amount of time using the branch and bound technique. In even less time, the branch and bound technique was able to find nearly identical results without proving optimality for many sub-problems. The local search technique was also able to find relatively good solutions. Given the simplicity of this algorithm, it is likely that even better results could be found with a local search approach. The loosening approach was the least successful of the three algorithms (most likely due to a poor selection of the constraints to loosen). Improving this approach is likely possible, but the usefulness of such an improvement is questionable due to the quality of the solutions obtained by the other two simpler algorithms. Overall for our best method, the branch and bound technique, we obtained improvements ranging between 3% and 13% for six real-world problem instances.

Given these results, the most promising improvement appears to be in the problem specification. Redefining the even distribution constraint in such a way that it does not tightly constrain batches from being shared between days would allow for the solution quality to be improved even more. The problem with such a modification is that it is not clear what defines a good distribution and redefining it would involve an analysis at the business level. If such a modification was done, it would provide an even more challenging problem with the potential for improved results.

Bibliography

[1] E. Aarts, J. Korst, and P. Laarhoven. Simulated annealing. In E. Aarts and Lenstra J. K., editors, *Local Search in Combinatorial Optimization*, pages 91–120. John Wiley & Sons Ltd., 1997.

[2] E. Aarts and J. K. Lenstra. Introduction. In E. Aarts and Lenstra J. K., editors, *Local Search in Combinatorial Optimization*, pages 1–17. John Wiley & Sons Ltd., 1997.

[3] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 311–318, Madison, Wisconsin, 1998.

[4] J. C. Beck, A. J. Davenport, E. M. Sitarski, and M. S. Fox. Texture-based heuristics for scheduling revisited. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 241–248, Providence, Rhode Island, 1997.

[5] C. Bessière and M.-C. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 108–113, Washington, DC, 1993.

[6] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *LISP and symbolic computation*, 5:223–270, 1992.

[7] T. Chase, C. Klepchak, P. Lavery, M. Subramanian, and D. Vergamini. Centralized vehicle scheduler: An application of constraint technology. http://www.ilog.com/products/optimization/tech/research/cvs.pdf, 1998.

[8] A. Davenport and E. Tsang. Solving constraint satisfaction sequencing problems by iterative repair. In *Proceedings of the First International Conference on the Practical Application of Constraint Technologies and Logic Programming*, pages 345–357, London, 1999.

[9] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[10] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[11] A. Hertz, E. Taillard, and D. de Werra. Tabu search. In E. Aarts and Lenstra J. K., editors, *Local Search in Combinatorial Optimization*, pages 121–136. John Wiley & Sons Ltd., 1997.

[12] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[13] A. Oddi and S. Smith. Stochastic procedures for generating feasible schedules. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 308–314, Providence, RI, 1997.

[14] B. D. Parrello and W. C. Kabat. Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *Journal of Automated Reasoning*, 2:1–42, 1986.

[15] J-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Washington, 1994.

[16] J-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 209–215, Portland, Oregon, 1996.

[17] J-C. Régin and J-F. Puget. A filtering algorithm for global sequencing constraints. In G. Smolka, editor, *Principles and Practice of Constraint Programming - CP97*, pages 32–46. Springer-Verlag, 1997.

[18] ILOG Press Release. ILOG drives productivity improvements at Chrysler. http://www.ilog.com/success/chrysler/index.cfm, 1997.

[19] ILOG Press Release. Peugeot-Citroen standardizes on ILOG optimization components. http://www.theautochannel.com:8080/ news/ press/ date/ 19990803/ press027663.html, 1999.

[20] K. Smith, M. Krishnamoorthy, and M. Palaniswami. Optimal sequencing of car models along an assembly line. In *Proceedings of the 12th National Australian Society for Operations Research (ASOR)*, pages 580–603, Adelaide, Australia, 1993.

[21] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[22] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. Department of Computer Science Technical Report CS-91-62, Brown University, Providence, RI, 1991.

[23] M. Wallace. Applying constraints for scheduling. In B. Mayoh and J. Penjaam, editors, *Constraint Programming.* Springer-Verlag, 1994.

[24] M. Wallace. Practical applications of constraint programming. *Constraints Intelligence*, 1:139–168, 1996.

[25] Q. Zhao and R. Goebel. A method for dealing with potentially relaxable constraints and its application to a dispatcher scheduling system. In *Proceedings of the IJCAI-97 Workshop on Business Applications of AI*, pages 83–88, Nagoya Congress Center, Nagoya, Japan, August 23-29 1997.