Consistency Propagation for Stretch Constraints

by

Lars Hellsten

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

Scheduling and rostering problems are among the most common applications of constraint programming. In this thesis, we explore several global constraints for rostering problems. We demonstrate algorithms for efficiently enforcing domain consistency for these constraints, and show empirically that achieving this strongest possible level of consistency is not only of theoretical interest, but also has substantial value in practical applications.

The focus of the thesis is a domain consistency algorithm for the `stretch` constraint based on dynamic programming. We also present an incremental version that sometimes performs better in practice, but requires more memory. We then show how this constraint, along with our algorithms, can be generalized to variants that subsume other rostering constraints from the literature. For certain other extensions of `stretch` that seem intuitively simple and useful, we prove that enforcing domain consistency is NP-hard.

Acknowledgements

I would like to thank my supervisor, Peter van Beek, for his many helpful suggestions, guidance, and collaboration; Gilles Pesant for his assistance in providing benchmark problems and running experiments; and my readers, Gordon Cormack, and Alejandro López-Ortiz for their time. I also would like to express my gratitude to the Natural Sciences and Engineering Research Counsel and the University of Waterloo for supporting my work through scholarships.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Constraint Programming (CP) is a quickly growing interdisciplinary field of Computer Science. As a general programming paradigm for solving hard search problems, it currently has widespread applications, and the increasing power of computer hardware makes it amenable to solving many previously intractable problems. In this thesis, we focus on constraint programming techniques for scheduling and rostering problems.

The history of CP traces back to the study of artificial intelligence (AI) and programming languages. Logic programming languages such as Prolog have traditionally been exploited in AI because their declarative nature is useful for expressing logic and reasoning. Constraint Logic Programming replaces unification in logic programming with constraints to yield a more expressive and efficient model for many problems. From here, CP was born as the general field of solving constraint satisfaction problems without drawing any distinction between imperative, functional, or declarative forms.

One idealistic vision that many would like to see computing evolve towards is the ability to specify a problem in some highly abstracted form involving mathematical notation, or even natural language, and have the computer return an answer. Declarative programming languages bring us closer to this than imperative languages, but in some sense are limited by the fact that there is often a tradeoff between algorithmic expressibility and problem expressibility. By this we mean that Prolog, for example, provides a very clean and elegant

way to express a problem using logic, but more so if one does not care about how the problem is solved. In order to solve a problem efficiently, one often needs to express the problem in a less-than-straightforward manner. Functional languages are the opposite, in that algorithms can be expressed in a very precise and elegant way, but the programs do not really say much about the problem itself. In some sense, constraint programming aims to bridge this gap by decoupling algorithmic implementation details from problem modeling.

1.2 Motivation

One of the major application areas of CP is in scheduling and planning. Problems in this area tend to be NP-hard combinatorial problems, and because of their difficulty, most any approach will involve a search algorithm with heuristics. Thus, while clever one-off algorithms can help dramatically sometimes, they are usually targeted at specific classes of instances of the problem, and there is no guarantee that they will be able to solve general instances efficiently in the worst case.

When one is resigned to solving a problem through brute force application of computing power in the worst case, there may be little reason not to use CP. It is an approach that provides an extraordinarily easy and general way of modeling problems, and most constraint solvers will run efficiently on a wide range of problems, especially if the problem is decomposed into constraints in an effective manner.

In many cases, the propagation techniques and heuristics employed by a constraint solver would take a significant amount of research and development to devise from scratch; and solving the problem in terms of already known constraints that result in good propagation may be the best approach that is found in the end, despite significant effort.

Moreover, scheduling problems often arise naturally in the form of constraints placed on certain variables. It is also quite natural for these constraints to change or be used in different combinations. The differences in constraints may be significant enough that a one-off algorithm or heuristic for solving one problem efficiently is completely ineffective when applied to a slightly different problem. In such situations, the more general framework of CP is particularly useful.

In many business and industrial settings there are certain work tasks that must be performed at specific times of day. Different shifts often require different types (or numbers) of workers, and the organization may operate continuously—around the clock, seven days a week. In such scenarios, one typically wants to create a balanced schedule for the workers. Workers' shifts should be compressed together as much as possible so that the same is true of their free time; they should not work too many shifts in a row; and they should have an appropriate amount of time off between work stretches. The `stretch` constraint was introduced by Gilles Pesant in [14] as a way to model these sorts of problems. Its main features are the ability to enforce lower and upper bounds on the number of shifts worked in a row by an employee, and the ability to restrict the types of shifts that can occur in succession (e.g. forcing work stretches to be followed by rest periods). It is also able to handle rotating schedules, where there is no distinction between the beginning and end of the roster (e.g. 24-hour, seven day a week operations). Gilbert Laporte discusses other techniques for designing rotating schedules in [11].

1.3 Contributions of the Thesis

In this thesis we focus on *sequencing* problems, which we loosely define as scheduling and rostering problems involving variables that are ordered sequentially.

Our main contribution is an algorithm for enforcing domain consistency for the `stretch` constraint, which is defined in Section 2.3.4. This is a stronger level of consistency than any previous algorithm has achieved, and our empirical results in Chapter 4 show that the stronger consistency can make a substantial difference in practice. We also describe a second, incremental algorithm for domain consistency, which has slightly higher space complexity, but manages to save state between invocations to reduce the total amount of work done.

A secondary contribution is the generalization of `stretch` to various forms that subsume the `change`, `smooth`, `group`, `count`, and `among` constraints, all of which appear in the literature [2]. These constraints are discussed in Chapter 5. This work is important because in our case, and indeed most cases where domain consistency can be easily enforced, it is more efficient to use a single constraint than to split the constraint up into multiple

constraints that are enforced independently.

Finally, in Chapter 5 we show that certain other generalizations that seem useful and arise naturally from `stretch` turn out to be intractable problems by proving the NP-completeness of the corresponding decision problems. This implies that it is NP-hard to achieve the strong level of consistency propagation that we were able to for the `stretch` constraint.

Chapter 2

Background

2.1 Preliminaries

In this section we give definitions of the basic terminology of Constraint Programming that we will use throughout this thesis.

Definition (CSP). A *Constraint Satisfaction Problem* (CSP) is a tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where

- $\mathcal{V} = \{x_1, \ldots, x_n\}$ is a finite set of *variables*.
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a finite set of *domains*. There is always one domain D_i corresponding to each variable x_i. We will frequently use the notation $\mathrm{dom}(x)$ to mean the domain associated with x, or $\mathrm{dom}(V) = \bigcup_{x \in V} \mathrm{dom}(x)$ for a set of variables.
- $\mathcal{C} = \{C_1, \ldots, C_k\}$ is a set of *constraints*. For a constraint C_i, $\mathrm{vars}(C_i) \subseteq \mathcal{D}$ is the set of variables to which C_i applies, and C_i is a subset of the Cartesian product of the domains of these variables.
- A *value* is simply an element of some domain. We will often use the term to mean a value of a specific variable's domain when it is clear from context (e.g. "removing a value" never means removing that value from all domains).

Definition (Variable Assignment). A *variable assignment* for a given CSP is a tuple $t \in D_1 \times \cdots \times D_n$. In other words, for each variable we pick one value from its domain. The notation $t[x]$ refers to the value assigned to the variable x in t.

Definition (Solution). A *solution* to a CSP is a variable assignment S that satisfies each of the constraints; for all $C \in \mathcal{C}$, $S \in C$. We say that a CSP is *consistent* if it has a solution, and *inconsistent* otherwise.

Definition (Support).

- A *domain support* for a value v of a variable x with respect to a constraint C is a variable assignment $t \in C$ with $t[x] = v$. If a value does not have a domain support with respect to any of the constraints of a CSP, then it is useless and can be removed. Detecting all such values is often difficult, however.
- When the domains are totally ordered, we denote the smallest and largest values in a domain D as $\min(D)$ and $\max(D)$, respectively. An *interval support* for a value v of a variable x, with respect to a constraint C, is a tuple $t \in C$ such that $t[x] = v$ and for every $y \in \text{vars}(C)$, $\min(\text{dom}(y)) \leq t[y] \leq \max(\text{dom}(y))$.

So, in summary, a CSP is a set of variables that each have a domain of values associated with them, and constraints on what combinations of values are allowed to be assigned to the variables. Our formalization of constraints is not especially intuitive, since we rarely express a constraint as an explicit set of tuples when there is a more compact way, and often the set $\text{vars}(C)$ for a constraint C is made implicit by the description of the constraint. For example, we might specify a problem in terms of arithmetic constraints such as $E : x_5 + x_2^2 < x_9$. Here, $\text{vars}(E) = \{x_2, x_5, x_9\}$. Any variable that is not in $\text{vars}(E)$ is allowed to take on any value in its domain without violating the constraint.

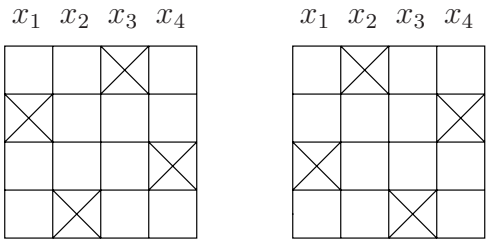
An example of a CSP is the N-queens problem, which is to place N queens on an empty $N \times N$ chess board in such a way that no queen can attack another. In other words, no two queens share the same rank (row), file (column), or diagonal. Table 2.1 shows how we might model the 4-queens problem using a variable for each column, and binary inequality constraints. Figure 2.1 shows the two solutions.

Algorithms for solving CSPs usually attempt to reduce the domains of the variables as much as possible while preserving all of the solutions. The general problem of deciding whether a given CSP is consistent is obviously NP-complete, since it is trivial to model most NP-complete problems in CSP form. In practice, though, the constraints we employ

Table 2.1: CSP formulation for the 4-queens problem. There are four variables and $3\binom{4}{2} = 18$ binary inequality constraints.

`4queens`$(\{x_1, x_2, x_3, x_4\})$
• $\mathrm{dom}(x_i) \subseteq \{1, 2, 3, 4\}$
• $x_i \neq x_j$ (for $i > j$)
• $x_i \neq x_j - (i - j)$ (for $i > j$)
• $x_i \neq x_j + (i - j)$ (for $i > j$)

Figure 2.1: Solutions to the 4-queens problem.



are not just arbitrary tuples, and have a structure that is characteristic of some aspect of the problems which they can model. The science of CP involves looking for ways to exploit this structure to solve problems more effectively.

2.2 Search Techniques

2.2.1 Backtracking

The typical approach to solving CSPs is to use some variant of backtracking, combined with domain reduction. The generic backtracking algorithm is essentially a refinement of brute force search, which systematically generates all possible solutions and tests if they are valid. In backtracking, this process is combined with consistency checks. In the simplest backtracking approach, variables are assigned values one at a time in a recursive manner.

After each variable is bound, we test if any of the constraints are violated by the variable assignments that have been made to this point. If so, we retreat and try a different value for the most recently assigned variable, avoiding the cost of needlessly binding any remaining variables when there is no way to reach a solution. Once all possible values for a variable under consideration have been tried, we backtrack to the variable that preceded it and try assigning the next possible value to it.

There are more sophisticated backtracking algorithms. For example, backjumping records the most recently bound variable that has caused a conflict with the variable currently being considered. The variables bound after this latest conflicting variable and before the variable under consideration have no bearing on the consistency of the problem, so there is no point in binding them to different values until the value of the conflicting variable has changed. Therefore, backjumping reverts directly to the most recent conflict. This approach typically performs better than the simple backtracking described above, and never any worse. Kondrak and van Beek outline different backtracking algorithms in [10], and characterize them based on the nodes they visit in the search tree.

In the rest of this section we look at three of the common heuristics used to speed up the search process in constraint solving.

2.2.2 Variable Ordering

The order in which a backtracking algorithm assigns values to variables can make a very large difference in its running time. There are two major reasons for this: first, some variables may be independent of any solution; second, some variables may give us more information than others and lead to better constraint propagation. We will discuss the latter in more detail below.

There are two main types of variable orderings:

- In a *static variable ordering* heuristic, the ordering is chosen before the algorithm begins, and does not change throughout its execution.
- A *dynamic variable ordering* heuristic chooses the next variable to consider on the fly, once the current variable has been bound to a value.

Ordering variables in increasing order of the sizes of their domains is an example of a strategy that is often useful, and can be implemented either statically or dynamically. Often dynamic orderings work best, but some heuristics are based on information that does not change throughout the search, or that is too expensive to be worth computing more than once.

Variable ordering heuristics are usually based on the so-called "fail first" principle described by Haralick and Elliott [7], which says that it is usually preferable to try branches that are likely to fail first. On the surface this may seem counter-intuitive, since we would like to find a solution as quickly as possible, but this is resolved by realizing that all but a negligible amount of the time taken to find a solution is spent ruling out bad choices. Therefore, the earlier the search detects the bad choices it makes, the faster it will complete.

2.2.3 Value Ordering

When a given variable is considered by a backtracking algorithm, it may have multiple values remaining in its domain. The order in which these values are considered can affect the running time of the search. In contrast to variable ordering, the goal of value ordering heuristics is based on the "succeed first" principle: it is better to prefer values that are most likely to yield a solution.

This difference in principles from variable ordering is explained by the fact value ordering only affects the order in which branches are visited in the search tree, and not the depth or breadth of the tree. If a branch does not contain a solution, then value ordering is useless within it, since the entire tree will be searched regardless of the ordering. It is preferable to avoid these useless branches. This also means that value ordering is most effective when employed as a counterpoint to a good variable ordering heuristic.

As with CSPs where there is no solution, if we want to find all solutions of a CSP, value ordering is not helpful.

2.2.4 Constraint Propagation

Constraint propagation is the process of reducing the domains to obtain a simpler CSP that is equivalent to the original, where equivalence is defined as having the same set of

solutions. Reducing the domains can speed up the execution of a solver significantly, by eliminating branches of the search tree.

Propagation algorithms are essentially inference algorithms that attempt to deduce that certain values cannot be used in any solution that satisfies a certain constraint. We use the term *(local) consistency* in reference to the goal of a propagator for a specific constraint. Note that this differs from the notion of (global) consistency defined in Section 2.1. The latter deals with the overall solubility of the CSP. We will see that local consistency does not imply global consistency, and vice-versa.

There are different kinds of consistency that occur frequently, some of which are stronger than others, and we mention several of them below. It is useful to classify propagators according to the level of consistency they achieve, although it should be noted that the effectiveness of a particular class depends heavily on the constraint considered. For some constraints, anything other than domain consistency may be too weak to be of much interest. Certain types of consistency may be inapplicable (e.g. bounds consistency when the domains are not totally ordered). In other some cases, what initially appears to be a weaker form of consistency may even turn out to be equivalent to domain consistency.

Arc Consistency

Arc consistency (AC) is the most elementary type of consistency, and applies to binary constraints. Recall that a *support* for a value of a domain with respect to some constraint is a variable assignment that is consistent with the constraint. A binary constraint on (x_1, x_2) is AC if $\text{dom}(x_1)$ and $\text{dom}(x_2)$ are both non-empty, each value in $\text{dom}(x_1)$ has a support, and each value in $\text{dom}(x_2)$ has a support.

Unlike any of the subsequent levels of consistency we discuss, AC is simple enough that it is commonly applied on a global basis. A CSP consisting entirely of binary constraints, or a binary constraint network, is said to be AC if each of the binary constraints is AC. As we alluded to above, simply propagating AC for each constraint does not ensure global AC. The removal of values when considering a specific constraint may cause the CSP to become non-AC with respect to a constraint that was previously considered.

Iteratively propagating AC for every constraint solves this problem, but it turns out to be feasible and often less expensive to propagate global AC using a direct algorithm that

considers all of the constraints at once. The problem has been well-studied, largely because any CSP can be transformed into a binary constraint network. The two most common approaches are compared by Bacchus and van Beek in [1]. (However, such transformations often require exponentially greater space and running time than the original non-binary constraint.) Many algorithms for propagating global AC have been proposed. Mohr and Henderson [13] were the first to achieve the optimal worst-case complexity of $O(ed^2)$ with their AC-4 algorithm. Here e is the number of binary constraints, and d is the size of the largest domain. The bound corresponds to the worst-case complexity of the input size; when each constraint consists of $O(d^2)$ tuples. Several other AC algorithms have since been proposed that also achieve theoretical optimality, for example by Bessiére [4].

Bounds Consistency

The remaining types of consistency we cover apply to constraints on arbitrary numbers of variables. Unlike in binary constraint networks, it is impractical to consider such constraints as a set of allowed tuples. As a result, any efficient propagator must take into account the specific structure of the constraint involved. Therefore, it does not make sense to consider constraint-independent algorithms for achieving consistency globally as with AC.

Bounds consistency is often of interest when the domains are totally ordered. A constraint C is bounds consistent if for every $x \in \text{vars}(C)$, both $\min(\text{dom}(x))$ and $\max(\text{dom}(x))$ have an interval support in C. Bounds consistency is often particularly useful for constraints involving intervals, where supports for the minimum and maximum values likely imply support for everything in between. For some constraints, bounds consistency implies range consistency.

Range Consistency

Like bounds consistency, range consistency is specific to totally ordered domains. A constraint C is range consistent if for every $x \in \text{vars}(C)$, every $v \in \text{dom}(x)$ has an interval support in C. For constraints where an interval support implies a domain support, range consistency is equivalent to domain consistency.

Domain Consistency

Domain consistency is sometimes called generalized arc consistency (GAC), and as that name implies, it is a generalization of AC to non-binary constraints. A constraint C is domain consistent if for every $x \in \text{vars}(C)$, every $v \in \text{dom}(x)$ has a domain support. Propagating domain consistency means removing every value that cannot appear in some variable assignment that satisfies C. It is therefore the strongest level of local consistency possible.

In general, deciding if a CSP is globally domain consistent is NP-complete, since the case where all domains are empty is equivalent to deciding whether there is a solution. However, deciding whether an individual constraint is domain consistent is often tractable.

2.3 Global Constraints

We can classify the types of constraints we deal with in CP into two categories: k-ary constraints, and global constraints. The former are simply, as the name implies, constraints that act on k variables at once, for some fixed value of k. The most common type are binary constraints, which act on two variables; for example, arithmetic constraints of the form $x_1 < x_2$, $x_1 \neq x_2$, etc. Global constraints act on an arbitrary number of variables, often depending on the specific problem instance they are being applied to.

All of the constraints we discuss in this thesis are global constraints. There are two particularly compelling reasons why global constraints are important. The first is that they provide a way to formulate complex relationships between variables. Such relationships may not be easy to express otherwise, and for many of the constraints discussed later in this thesis, modeling them as binary constraints would complicate their formulation substantially (this will become obvious once we introduce the constraints).

Secondly, the use of global constraints can lead much better domain reduction than would otherwise be possible. There are many global constraints, such as those we introduce later in this section, which are amenable to efficient domain consistency propagation, whereas other representations of the constraint would not even come close. This is because they have a global structure of which one can only capture bits and pieces with just a fixed number of variables worth of information.

2.3.1 A Motivating Example

To motivate much of our discussion during the consideration of specific global constraints, we present an example of a realistic sequencing problem. Consider a computing support help desk staffed by students from 9am until 5pm, Monday to Friday. The diligent scholars work part-time for several hours a week to help subsidize the lavish lifestyle that graduate school entails. For various reasons of convenience, each day is broken down into hour-long shifts, to which the staff is assigned. Creating the weekly timetable is often challenging, because it must be planned around the classes and other activities of the workers.

The obvious way to model this problem includes a variable for each of the 40 shifts during the work week, $\{s_0, s_1, \ldots, s_{39}\}$, with the domains indicating which staff members are available to work the shift. This allows us to easily account for the shifts where certain staff members are unavailable. But this alone is clearly unacceptable. One possible solution to this CSP would be to simply assign each variable the first value in its domain, possibly resulting in one student working all 40 shifts!

To be more realistic, we need to prescribe a minimum and maximum number of shifts that each person will work. For this, binary constraints would only be sufficient if the maximum number of shifts is one. Otherwise, one must look at more than two variables to determine if the conditions are satisfied. Short of reformulating the problem as a CSP exponential in the maximum number of shifts, global constraints are necessary. We thus devote the rest of this section to some specific global constraints that can help us solve our problem, and to further developing this example with the help of additional constraints.

2.3.2 The `alldifferent` Constraint

The `alldifferent` constraint is one of the simplest and most common global constraints. As the name indicates, it forces all variables to be assigned different values. Régin describes an algorithm based on bipartite matching for achieving domain consistency for `alldifferent` in [17] which has complexity $O(dn\sqrt{n})$, where d is the maximum number of values in any variable's domain, and n is the number of variables.

In order to see how this constraint might be useful, we revisit the help desk example. Suppose that the noon hour shifts tend to be particularly busy, and that in order to fairly

`alldifferent`$(\{x_1, x_2, \ldots, x_n\})$
• $x_i \neq x_j$ for $i \neq j$

distribute the workload, we want to avoid having the same person work two different noon hour shifts during the week. The variables $x_{(8i+3)}$ correspond to these shifts, and one solution is to impose the 10 binary constraints:

$$x_{(8i+3)} \neq x_{(8j+3)} \text{ where } 0 \leq i < j \leq 4$$

But using $n(n-1)/2$ binary constraints is not a very compact representation if one has to list them all. Using `alldifferent`, we only need a single constraint:

$$\texttt{alldifferent}(x_3, x_{11}, x_{19}, x_{27}, x_{35})$$

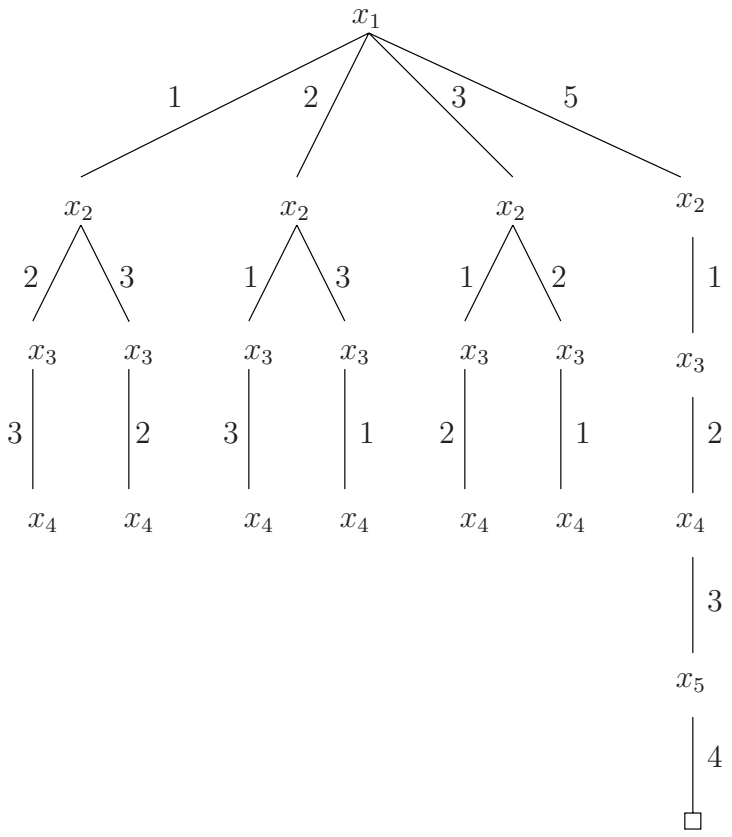
Table 2.2: AC and binary inequalities vs. GAC and `alldiffferent`.

Variable	AC domains	GAC for `alldifferent`
x_1	$\{1, 2, 3, 5\}$	$\{5\}$
x_2	$\{1, 2, 3\}$	$\{1, 2, 3\}$
x_3	$\{1, 2, 3\}$	$\{1, 2, 3\}$
x_4	$\{1, 2, 3\}$	$\{1, 2, 3\}$
x_5	$\{1, 2, 3, 4, 5\}$	$\{4\}$

More importantly than the representation, enforcing AC on the binary constraints is far less powerful than enforcing domain consistency (or even weaker forms of consistency like range or bounds consistency) on the `alldifferent` constraint. This is illustrated in Table 2.2. Now, imagine solving this example using standard (chronological) backtracking with AC propagated on the binary constraints after each variable assignment, with variables and values considered in lexicographic order. Figure 2.2 shows the corresponding search tree, in which all but the path from the root to the rightmost leaf represents wasted work that does not lead to a solution. In contrast, modeling the problem with the `alldifferent`

constraint and propagating domain consistency ensures that no backtracking occurs – the search tree becomes linear.

Figure 2.2: Search tree using backtracking with AC.



We can also use the `alldifferent` constraint to address the problem of restricting the number of shifts each person works during a week, by formulating the problem slightly differently. Let h_i denote the maximum number of hours the i-th staff member should work during the week. Instead of having a single type of value for each employee, we introduce h_i values for employee i, and an `alldifferent` constraint over all variables. Since none of the values corresponding to worker i can be used more than once, he is allocated at most h_i shifts. This formulation has the drawback of significantly increasing the number of

values, which will increase the cost of propagating domain consistency, and may blow up the size of the search tree by an exponential factor. It also does not address the problem entirely, since we still would like to place a lower bound on the number of shifts worked. Otherwise, a few employees who don't mind working many hours may monopolize all of the shifts. Fortunately, the constraint we describe next resolves all of these concerns.

2.3.3 The Global Cardinality Constraint

The Global Cardinality Constraint (`gcc`) is a generalization of `alldifferent`. It allows us to place a minimum and maximum cardinality on the occurrences of each value.

`gcc`$(\{x_1, x_2, \ldots, x_n\}, \{l_1, \ldots, l_k\}, \{u_1, \ldots, u_k\})$
• The number of variables assigned value v_i is at least l_i, and at most u_i

The `alldifferent` constraint, of course, corresponds to the special case where $l_i = 0$ and $u_i = 1$ for all values v_i. Now we can express the limits on the number of shifts each employee works in the help desk example in a natural way, following the original formulation of the problem.

The existence of the `gcc` constraint does not make `alldifferent` obsolete, as propagation algorithms for the latter are faster. The best known algorithm for domain consistency of `gcc` is an $O(n^2 d)$ approach by Régin based on network flow theory [18]. In [16], an efficient bounds consistency algorithm is presented by Quimper et. al.

2.3.4 The `stretch` Constraint

In most staffing problems, we would like to restrict the number of shifts a person works in a row, in addition to their overall weekly workload. Placing a `gcc` constraint over all of the shifts in a single day may not be a reasonable solution, since it would require that we know ahead of time who is going to work on a given day. An equally important concern is that the shifts not be too spread out. For example, it should be possible to ensure that all of shifts assigned to a specific worker on a given day occur in one contiguous sequence of shifts.

The `stretch` constraint solves this problem by allowing us to constrain the length of any maximal contiguous block of shifts all assigned the same type (also called a *stretch*). This constraint also is the basis for most of the original contributions of this thesis, so we will describe it in more detail than the global constraints we have considered up to this point.

We call the variables that the `stretch` constraint acts on *shift variables*, and we will label them $\mathcal{S} = \{s_0, s_1, \ldots, s_{n-1}\}$ to distinguish them from other types of variables in a CSP. Each shift variable, of course, corresponds to a shift in a roster, and they are assumed to be ordered by index. Note that the variables are indexed from 0. This is for convenience when performing modular arithmetic for problems in which we want to consider s_{n-1} and s_0 to be adjacent. We will discuss such circular versions of the problem in detail in Chapter 3. Values represent *shift types*, and are labeled $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_m\}$.

For a given assignment of values to variables, a *stretch* is a maximal sequence of consecutive shift variables that are assigned the same value. Thus, a sequence $s_i, s_{i+1}, \ldots s_{i+k-1}$ is a stretch if $s_i = s_{i+1} = \cdots = s_{i+k-1}$, $i = 0$ or $s_{i-1} \neq s_i$, and $i + k = n$ or $s_{i+k} \neq s_i$. We say that such a stretch *begins* at s_i, has *span* (alternatively, *length*) k, and is of type value(s_i). We write span(s_j) $= k$ once the value of s_j has been bound to denote the span of the stretch through s_j. The formulation of the constraint is summarized as follows.

`stretch`$(\{s_0, s_1, \ldots, s_{n-1}\}, \Pi, \text{shortest}, \text{longest})$
• $\text{dom}(s_i) \subseteq \mathcal{T} = \{\tau_1, \ldots, \tau_m\}$
• $\text{shortest}[s_i] \leq \text{span}(s_i) \leq \text{longest}[s_i]$
• $s_i = s_{i+1}$ or $(s_i, s_{i+1}) \in \Pi$

The elements of the set $\Pi \subseteq \mathcal{T} \times \mathcal{T}$ are ordered pairs called *patterns*. A stretch of type τ_i is allowed to be followed by a stretch of type τ_j, $\tau_j \neq \tau_i$, if and only if $(\tau_i, \tau_j) \in \Pi$. Note that pairs of the form (τ_k, τ_k) are redundant, since by the definition of a stretch two consecutive stretches do not have the same value. The shortest and longest are arrays that map shift types to lengths, indicating the minimum and maximum lengths allowed for a stretch of any given type. We call a stretch of type τ through a variable s_j *valid* if it

satisfies

$$\text{shortest}[\tau] \leq \text{span}(s_j) \leq \text{longest}[\tau].$$

The `stretch` constraint was originally proposed by Gilles Pesant in [14] as a means to solve real-world timetabling problems. Pesant's original propagation algorithm was based on a series of heuristics, and although it often works well in practice and has a fairly good running time, it does not achieve anything as strong as domain consistency, which we will show in Chapter 3. In order to be successful, it needs a favourable variable ordering or the absence of bad luck. The running time of Pesant's algorithm is $O(m^2l^2)$, where l is the maximum length of any stretch. In Chapter 3, we provide a domain consistency algorithm that has worst-case complexity comparable to this, and does better if l is large. More recently, Pesant has done work on a regular expression constraint [15] for which he has given an algorithm to enforce domain consistency. This constraint is able to model `stretch` instances. However, it is not clear how to use it to model cyclic instances, and its theoretical worst-case runtime and space complexity are both $O(nm^2l)$, which we are able to beat.

2.3.5 Applying Global Constraints

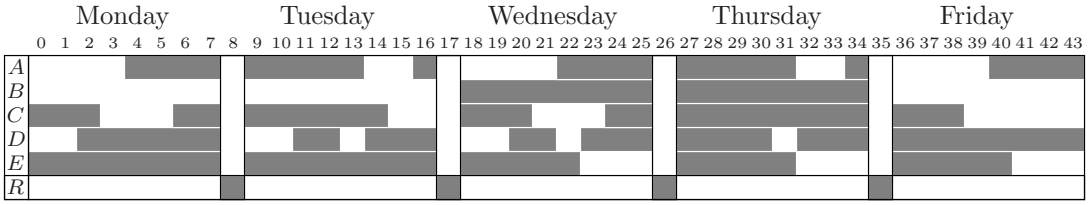
Now that we have a few global constraints in our arsenal, we are ready to revisit the problem described in Section 2.3.1. We will formulate a CSP to create a roster for a single week. Since the shifts do not run for 24 hours, we would like to avoid considerng the last shift of a day and the first shift of the next day to be adjacent. Therefore, we will insert dummy variables (whose domains only contain a dummy rest value, R) between each block of eight variables. We will use the example constraints on worker schedules in table 2.3 to formulate the CSP using both the `stretch` and `gcc` constraints. Figure 2.3 illustrates the corresponding variable domains.

The number of consecutive shift ranges in Table 2.3 map directly to bounds on stretch lengths, and the ranges for the number of shifts each workers can have in total map to the `gcc` lower and upper bounds. Therefore, we can model the example in a straightforward manner using both the `gcc` and `stretch` constraints. The lower and upper bounds for the rest shift type R in the `gcc` constraint is either 4, or the constraint may simply not be applied to the rest shifts, s_8, s_{17}, s_{26}, and s_{35}. The `stretch` instance must include these

Table 2.3: Sample scheduling constraints. Each worker has an associated range for workload (the total number of hours of workload they are willing to work during a week), a range of the number of shifts they are willing to work in a row, and a list of times they are unavailable to work.

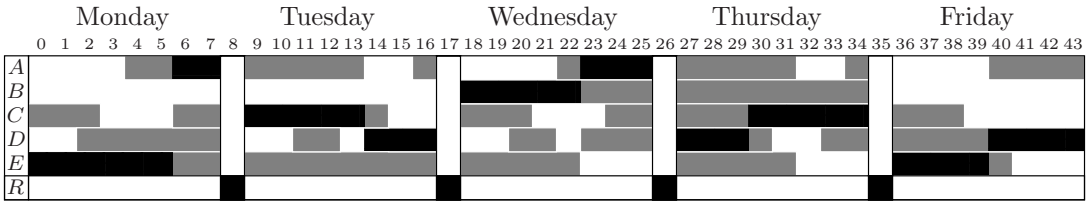
Name	Shift Type	Workload	Length	Times Unavailable
Alice	A	$[5, 5]$	$[2, 3]$	9:00-13:00 Mon, Wed, Fri; 14:00-16:00 Tue, Thu
Bob	B	$[5, 8]$	$[2, 5]$	9:00-17:00 Mon, Tue, Fri
Clara	C	$[5, 10]$	$[5, 5]$	12:00-15:00 Mon, Wed, Fri; 15:00-17:00 Thu, Fri
Dave	D	$[10, 15]$	$[3, 5]$	9:00-11:00 Mon, Tue, Wed; 13:00-14:00 Tue, Wed, Thu
Elise	E	$[10, 10]$	$[4, 6]$	14:00-17:00 Wed, Thu, Fri

Figure 2.3: Domains corresponding to the times workers are available in Table 2.3. Rows and columns correspond to shift type (values) and shifts (variables), respectively. A row and column is shaded if the domain of the corresponding variable contains the corresponding value.



variables, however, since their purpose is to force stretches to be interrupted. We set $\text{shortest}[R] = \text{longest}[R] = 1$. We do not require patterns, so Π includes all ordered pairs. One of the possible solutions to the CSP is shown in Figure 2.4.

Figure 2.4: The highlighted stretches indicate one possible solution to our example.



Chapter 3

Propagation Algorithms for the Stretch Constraint

In this chapter, we will present three propagation algorithms for enforcing domain consistency for `stretch`. The first algorithm, DC (Domain Consistency), is simplest conceptually and sometimes works best in practice because of its low overhead. It also forms the basis for the extended variants of `stretch` that we discuss later. The second algorithm, FC (Fast Consistency), is a slightly more complicated improvement of DC, but has a better worst-case asymptotic complexity. Finally, IC (Incremental Consistency) is an incremental algorithm that trades off space for quick overall running time. We initially restrict our attention to non-cyclic instances of the problem.

All three algorithms rely on the observation that any stretch that appears in a solution is independent of the stretches chosen before and after it, aside from the enforcement of the patterns. Pattern enforcement only depends on the variables adjacent to the stretch, and so when considering whether a stretch appears in a solution, we only need to consider the variables in the stretch, and the two neighbouring ones. So we just need an efficient way to determine whether there are appropriate sequences of stretches as prefixes and suffixes for each possible stretch. We call such sequences *supporting sequences*.

Definition (Supporting Sequences).

- Consider a set of k stretches beginning at each of $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$, where $i_1 < i_2 <$

$\cdots < i_k$, and having types $\tau_1, \ldots, \tau_k$ respectively, where $\tau_i \neq \tau_{i+1}$. Also let $\text{span}(s_{i_p}) = i_{p+1} - i_p$ and $(\tau_p, \tau_{p+1}) \in \Pi$ for all $p < k$. We call this a *sequence of stretches*, which *covers* $s_{i_1}, \ldots, s_{i_k}, s_{i_k+1}, \ldots, s_{i_k+q-1}$ where $q = \text{span}(s_{i_k})$.

In words, we simply mean a contiguous sequence of stretches that would be a solution to the stretch instance restricted to the variables it covers.

- We call a non-empty sequence of stretches *forward compatible* with stretch type τ if $(\tau_p, \tau) \in \Pi$, where τ_p is the type of the last stretch in the sequence, and *backward compatible* if $(\tau, \tau_p) \in \Pi$, where τ_p is the type of the first stretch in the sequence. Additionally, the empty sequence of stretches is forward and backward compatible with all types. We will usually just use the term *compatible* when it is obvious from context which of these is meant.

- A *prefix support* for a stretch beginning at s_i is a sequence of stretches that covers variables $s_0, \ldots, s_{i-1}$. (This sequence may be empty.)

- A *suffix support* for a stretch ending at s_i is a sequence of stretches that covers variables $s_{i+1}, \ldots, s_{n-1}$. (This sequence may also be empty.)

3.1 A Simple Propagation Algorithm

Our first algorithm is based on a simple dynamic programming approach which counts the total number of solutions. The algorithm works by considering prefix supports. If we have initially assigned some shift type τ to variables $s_r, s_{r+1}, \ldots, s_{n-1}$, then the total number of solutions given this assignment at the end of the roster is simply the number of solutions to the `stretch` instance with the same parameters, but restricted to the variables $s_0, s_1, \ldots, s_{r-1}$, and with the added constraint that $(\text{value}(s_{r-1}), \text{value}(s_r)) \in \Pi$. In other words, the number of prefix supports ending at s_{r-1} that are compatible with $\text{value}(s_r)$ with respect to Π. The choices of r and τ are arbitrary, as long as the stretch satisfies the length constraints for τ.

An alternative way to look at the problem is one of computing reachability in a directed graph; the nodes in the graph are (variable, type) pairs, and arcs correspond to stretches.

Our goal is to find all edges that are in some path from some node (s_0, τ_1) to some node (s_{n-1}, τ_2). Here a prefix support is a path from the beginning of the roster to some node, and a suffix support is a path from some node to the end of the roster. If the roster needs to be cyclic, then we may have edges between the ending and beginning positions, the additional constraint that the beginning and ending nodes chosen correspond to valid patterns, and we will want to consider many variables as starting and ending points, not just s_0 and s_{n-1}. The set of all edges found will indicate which values are part of some solution, and which values can be safely removed.

Since the subproblems of counting prefix supports can be solved without any information about what assignments are made to subsequent variables (beyond where the support ends), we can turn what initially appears to be an exponential approach into a polynomial time dynamic programming one. A subproblem can be parameterized by an index r, and a shift type τ. The index indicates that the subproblem is over variables $s_0, \ldots, s_{r-1}$, and τ indicates the shift type assigned to s_{r-1}.

The algorithm `CountSolutions` fills in a two-dimensional table count$[r, \tau]$ that stores the number of solutions to the subproblem beginning at index r with previous shift type τ, making sure to only consider valid stretches (that satisfy the given lower and upper bounds on their length).

Algorithm `CountSolutions`()
1. initialize all entries of count to 0
2. (* consider all the initial stretches *)
3. **foreach** $\tau \in \text{dom}(s_0)$ **do**
4. **for** $l \leftarrow 1$ **to** $\min(\text{longest}[\tau], n)$ **do**
5. **if** $\tau \notin \text{dom}(s_{l-1})$ **then break**
6. **if** $l \geq \text{shortest}[\tau]$ **then** $\text{count}[l, \tau] \leftarrow 1$
7. (* extend all prefix supports *)
8. **for** $r \leftarrow 1$ **to** $n - 1$ **do**
9. **foreach** $(\tau_j, \tau_k) \in \{\text{dom}(s_{r-1}) \times \text{dom}(s_r)\} \cap \Pi$ **do**
10. **for** $l \leftarrow 1$ **to** $\text{longest}[\tau_k]$ **do**
11. **if** $r + l > n$ **or** $\tau_k \notin \text{dom}(s_{r+l-1})$ **then break**
12. **if** $l \geq \text{shortest}[\tau_k]$ **then** $\text{count}[r + l, \tau_k] \leftarrow \text{count}[r + l, \tau_k] + \text{count}[r, \tau_j]$

Each element count$[i, \tau]$ stores the number of ways of forming a prefix support that reaches position i (meaning the final stretch in the sequence ends at position $i-1$), where the value of the last stretch in the support is τ. The total number of solutions is then just the sum of the number of ways of reaching position n over all shift types.

The following small example demonstrates how `CountSolutions` works. We consider a roster with three shift types A, B, and C with the bounds on the stretch lengths and initial domains as shown in Table 3.1 and pattern set $\Pi = \{(A, B), (A, C), (B, A), (C, A)\}$. It is easy to see that the initial configuration is domain consistent. There are five solutions: {*AAABBBAA, AABBBAAA, AAACCCCC, CCCCCAAA, AACCCCAA*}. Each value present in the domains is used in at least one solution. Table 3.2 shows the count table. The rightmost column tells us that there are four solutions ending in a stretch of type A, and one solution ending in a stretch of type C. Column 4 tells us that there is a prefix support ending at s_3 whose last stretch type is C (namely, CCCC). Note that not all a prefix supports appear in a solution.

Table 3.1: (left) Bounds on stretch length; (right) Initial domains.

τ_k	shortest$[\tau_k]$	longest$[\tau_k]$
A	2	4
B	3	3
C	4	5

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A	A	A			A	A	A
		B	B	B	B		
C	C	C	C	C	C	C	C

Table 3.2: Building the count table.

	1	2	3	4	5	6	7	8
A	0	1	1	0	0	0	2	4
B	0	0	0	0	1	1	0	0
C	0	0	0	1	1	1	2	1

To modify the algorithm for propagation, we simply need to determine which values

in each domain are part of some solution. This information can be gathered with a second pass over the count array. We need a second $(n+1) \times m$ array begins_suffix where begins_suffix$[r, \tau_k]$ indicates whether a stretch of type τ_k beginning at r is contained in any solution (i.e. begins a suffix support). If a stretch has a prefix support and a suffix support, then we know that it is contained in a solution. We check whether this is the case for each possible stretch during this second phase, and mark values as being not prunable when it is known that they appear in a solution. At the end of the execution the prunable table indicates which values did not appear, and in lines 15-17 those values are removed.

Algorithm `DC()`

1. initialize all entries of begins_suffix to **false**
2. initialize all entries of prunable to **true**
3. **foreach** $\tau \in \mathcal{T}$ **do** begins_suffix$[n, \tau] \leftarrow$ **true**
4. **for** $r \leftarrow n-1$ **downto** 0
5. **foreach** $\tau_j \in \text{dom}(s_r)$ such that count$[r, \tau_j] > 0$ **do**
6. max_stretch $\leftarrow 0$
7. **foreach** $\tau_k \in \mathcal{T}$ such that $(\tau_j, \tau_k) \in \Pi$ **do**
8. **for** $l \leftarrow 1$ **to** longest$[\tau_j]$ **do**
9. **if** $r + l > n$ **or** $\tau_j \notin \text{dom}(s_{r+l-1})$ **then break**
10. **if** $l \geq$ shortest$[\tau_j]$ **and** begins_suffix$[r+l, \tau_k]$ = **true then**
11. begins_suffix$[r, \tau_j] \leftarrow$ **true**
12. max_stretch $\leftarrow \max(\text{max\_stretch}, l)$
13. **for** $l \leftarrow 1$ **to** max_stretch **do**
14. prunable$[r+l-1, \tau_j] \leftarrow$ **false**
15. **for** $i \leftarrow 0$ **to** $n-1$ **do**
16. **foreach** $\tau_k \in \text{dom}(s_i)$ **do**
17. **if** prunable$[i, \tau_k]$ = **true then** remove τ_k from $\text{dom}(s_i)$

3.1.1 Analysis

It is clear from the loop bounds that the loop from lines 8-12 in algorithm `CountSolutions` dominates the running time. Lines 11-12 are executed $O(nm^2l)$ times, where we define

$l = \max_{1 \leq j \leq m}(\text{longest}[\tau_j])$, so this is the overall time complexity of the algorithm. The space complexity is $O(nm)$, since we require a 2-dimensional table with $(n+1)$ rows for positions $0, \ldots, n$, and m columns, one for each shift type. This is obviously optimal. Similarly, lines 9-12 and line 14 of algorithm `DC` consist of $O(1)$ operations that execute $O(nm^2l)$ times, and the algorithm requires $O(nm)$ space. So the overall runtime complexity is $O(nm^2l)$ and the overall space complexity is $O(nm)$.

Note that there may actually be exponentially (in n) many solutions, so that the integers stored in count would require $O(n)$ space to represent, and $O(n)$ time to add. We ignore this factor, because the algorithm can just as easily be implemented using a table of boolean values to indicate whether a prefix support exists, as with the begins_suffix table. We thought it instructive to present the algorithm in a form that provides more information. One possible application of the counts computed is as a guide to help the CSP solver's variable and value ordering heuristics.

Lemma 3.1. *Algorithm* `CountSolutions` *sets* $\text{count}[i, \tau_j]$ *to a value greater than zero if and only if there is a solution to the subproblem over the variables* $s_0, s_1, \ldots, s_{i-1}$ *with* $s_{i-1} = \tau_j$.

Proof. We can prove this by induction on i. In the base case, $i = 1$, it is easy to see that $\text{count}[1, \tau_j] = 1$ if and only if there is a stretch of length 1 beginning at s_0 with value τ_j. The algorithm begins by trying all such initial stretches on lines 3-6.

Now suppose the result holds for $\text{count}[i', \tau'_j]$ where $i' < i$. Now, inside the loop on lines 8-12, an element $\text{count}[i, \tau_j]$ is added to (thus ensuring $\text{count}[i, \tau_j] > 0$) if and only if there exist i', τ'_j with $i' < i$, $\tau'_j \neq \tau_j$, with $(\tau_j, \tau_{j'}) \in \Pi$ such that $\text{count}[i', \tau'_j] > 0$ and there is a valid stretch of type τ_j of length $i - i'$ starting at $s_{i'}$. Recall that by a valid stretch, we mean that each of the variables $s_{i'}, \ldots, s_{i-1}$ contains τ_j, and $\text{shortest}[\tau_j] \leq i - i' \leq \text{longest}[\tau_j]$.

Since $\text{count}[i', \tau'_j] > 0$, by the induction hypothesis there is a solution to the subproblem consisting of the first i' variables. Since extending this solution with a feasible stretch of τ_j does not violate any of our constraints on stretches, there is a solution to the first i variables that ends with a stretch of type τ_j. □

Lemma 3.2. *Algorithm* `DC` *sets* $\text{begins\_suffix}[i, \tau_j] =$ ***true*** *if and only if there is a solution to the full problem with a stretch of type* τ_j *beginning at* $i - 1$.

Proof. An argument similar to the previous proof shows this, the only difference being we induct on $n - i$ rather than i. □

Lemma 3.3. *Algorithm* DC *sets* prunable$[i, \tau_j]$ = ***false*** *if and only if there is some solution with* $s_i = \tau_j$.

Proof. The algorithm sets prunable$[i, \tau_j]$ = **false** if and only if it can find some values i_1, i_2 with $i_1 \leq i < i_2$ such that a stretch of τ_j from i_1 to $i_2 - 1$ is feasible, count$[i_1, \tau_j] > 0$, and there exists some τ_k such that $(\tau_j, \tau_k) \in \Pi$ and begins_suffix$[i_2, \tau_k]$ = **true**. (Note that since $i < i_2$, begins_suffix$[i_2, \tau_k]$ contains its final value by the time prunable$[i, \tau_j]$ is considered.) □

The next theorem, which follows directly from the previous lemma, states that the algorithm enforces domain consistency.

Theorem 3.4. *Algorithm* DC *prunes a value* τ *from a domain* $\mathcal{D}_{s_i}$ *if and only if there is no satisfying assignment with* $s_i = \tau$.

3.2 A Faster Propagation Algorithm

Our second propagator uses a similar approach, also based on dynamic programming, but takes advantage of the fact that stretches constitute intervals in order to eliminate a factor of $l = \max_{1 \leq j \leq m}(\text{longest}[\tau_j])$ from the running time.

3.2.1 Computing Reachability

The basis of the FC algorithm, like DC, is to use dynamic programming to compute, for each variable s_i and type τ_j, whether there is a prefix support for a stretch beginning at s_i that is compatible with τ_j with respect to the set of patterns. The results of this computation are stored in a matrix of values, forward (see Algorithm ComputeForward). Likewise, we compute whether there is a suffix support for a stretch ending at s_i that is compatible with τ_j, and store the result in backward (see Algorithm ComputeBackward). These are similar to the count and begins_suffix arrays of the DC algorithm.

Once the support information is computed, it is used by a second step that prunes the domains. To make the first step as efficient as possible, we actually store the support information in forward and backward as arrays of prefix sums over the variables. The element forward$[\tau, i]$ indicates the number of variables s_j with $j < i - 1$ such that a prefix support covers $s_0, s_1, \ldots, s_j$, and ends with a stretch type compatible with τ. The prefix sums allow us to, for a given type, query whether a prefix support ends within an arbitrary range in constant time. For example, the difference forward$[\tau, i+1]$ − forward$[\tau, j]$ $(j \leq i)$ is greater than zero if and only if there is some prefix support beginning at s_0 and ending between s_{j-1} and s_{i-1} (inclusive) that is compatible with τ.

Another prefix array, runlength, is precomputed at the beginning of each stage of the algorithm. For each type τ and variable s_i, it stores the size of the maximal contiguous block of variables whose domains contain τ, up to and including s_i (or including and following s_i for `ComputeBackward`). This gives an upper bound on the maximum length of a stretch ending (or beginning) at s_i, which may be less than longest$[\tau]$. Note that to make the algorithm concise, we use 1-based indices when looping over variables, rather than the 0-based indices used for shift variables. Indices 0 and $n + 1$ correspond to initial values.

Algorithm `ComputeForward`()

1. **foreach** $\tau \in \mathcal{T}$ **do** forward$[\tau, 0] \leftarrow 0$
2. **foreach** $\tau \in \mathcal{T}$ **do** forward$[\tau, 1] \leftarrow 1$
3. **foreach** $\tau \in \mathcal{T}$ **do**
4. **for** $i \leftarrow 0$ **to** n **do**
5. runlength$[\tau, i] \leftarrow 0$
6. **for** $i \leftarrow 1$ **to** n **do**
7. **foreach** $\tau \in \mathrm{dom}(s_{i-1})$ **do**
8. runlength$[\tau, i] \leftarrow$ runlength$[\tau, i - 1] + 1$
9. **for** $i \leftarrow 1$ **to** n **do**
10. **foreach** $\tau \in \mathcal{T}$ **do** forward$[\tau, i + 1] \leftarrow$ forward$[\tau, i]$
11. **foreach** $\tau_j \in \mathcal{T}$ **do**
12. hi $\leftarrow i -$ shortest$[\tau_j]$
13. lo $\leftarrow i - \min($longest$[\tau_j],$ runlength$[\tau_j, i])$
14. **if** hi $\geq$ lo **and** forward$[\tau_j,$ hi $+ 1] -$ forward$[\tau_j,$ lo$] > 0$ **then**

```
15.            foreach τ_k ∈ 𝒯 such that (τ_j, τ_k) ∈ Π do
16.                forward[τ_k, i+1] ← forward[τ_k, i] + 1
```

Algorithm `ComputeBackward`()

```
1.  foreach τ ∈ 𝒯 do backward[τ, n+1] ← 0
2.  foreach τ ∈ 𝒯 do backward[τ, n] ← 1
3.  foreach τ ∈ 𝒯 do
4.      for i ← 1 to n+1 do
5.          runlength[τ, i] ← 0
6.  for i ← n downto 1 do
7.      foreach τ ∈ dom(s_{i-1}) do
8.          runlength[τ, i] ← runlength[τ, i+1] + 1
9.  for i ← n downto 1 do
10.     foreach τ ∈ 𝒯 do backward[τ, i-1] ← backward[τ, i]
11.     foreach τ_j ∈ 𝒯 do
12.         lo ← i + shortest[τ_j]
13.         hi ← i + min(longest[τ_j], runlength[τ_j, i])
14.         if hi ≥ lo and backward[τ_j, lo-1] − backward[τ_j, hi] > 0 then
15.             foreach τ_k ∈ 𝒯 such that (τ_j, τ_k) ∈ Π do
16.                 backward[τ_k, i-1] ← backward[τ_k, i] + 1
```

Consider again the example from Section 3.1, which considered the `stretch` instance in Table 3.1. Suppose that suppose we assign $s_2 \leftarrow A$ (see Table 3.6). This has the effect of reducing the set of possible solutions to $\{AAABBBAA,\ AAACCCCC\}$. When we run our algorithm, it should remove the value C from the domains of s_0, s_1, and s_2, and A from s_5. Table 3.3 shows the forward table computed after this assignment has been made. The execution of `ComputeForward` can be traced by looking at successive columns in the table. Table 3.4 shows the backward table.

3.2.2 Pruning Values

Once we have computed the forward and backward support information, we are ready to begin pruning values from domains. This process proceeds by considering, for each type,

Table 3.3: Building the forward table.

	0	1	2	3	4	5	6	7	8	9
A	0	1	1	1	1	1	1	2	3	4
B	0	1	1	2	3	3	3	3	4	5
C	0	1	1	2	3	3	3	3	4	5

Table 3.4: Building the backward table.

	0	1	2	3	4	5	6	7	8	9
A	3	3	3	3	2	1	1	1	1	0
B	5	4	3	3	3	3	2	1	1	0
C	5	4	3	3	3	3	2	1	1	0

every possible stretch of that type. We check if a stretch is in a solution by examining the forward and backward matrices to see if there are supports that can come before and after the one we are considering. If so, we mark the type we are considering, for each of the variables in the stretch. The final pruning step then prunes any value that has not been marked.

In order to make the marking linear in n for each τ_j, we traverse the variables in reverse order, maintaining a queue of possible ending positions. For each position i, we pop any elements from the front of the queue that cannot possibly end a stretch of type τ_j beginning at i. A position $j \geq i$ is not a possible ending position if $j - i + 1 > \text{longest}[\tau_j]$, or if there exists some k, $i \leq k \leq j$ such that the variable s_k does not contain τ_j in its domain, i.e. $j - i + 1 > \text{runlength}[\tau_j, i]$. Notice that if a position is not a valid ending position for i, it is also not valid for any position smaller than i, so it is always safe to remove invalid positions from the queue.

We also need to ensure that recording the marked intervals is efficient. However, this is easy, since the ending positions we consider are non-increasing. Therefore, each interval we add either extends the previous interval, or is disjoint from the previous interval. We end

up with an ordered list of $O(n)$ disjoint intervals which cover a total of $O(n)$ values. We can therefore iterate through the list of intervals and mark all variables within each interval in $O(n)$ time. The merging of intervals is implemented in algorithm `MergeInterval`.

Algorithm `MergeInterval`(s, e)

while true do
  $[s', e'] \leftarrow$ the front element of the interval list (if non-empty)
  if the list of intervals is empty **or** $e < s'$ **then**
    add $[s, e]$ to the front of the list and **return**
  remove $[s', e']$ from the interval list
  $e \leftarrow e'$

Algorithm `MarkValues`()

initialize all entries of prunable to **true**
initialize all entries of runlength to 0
for $j \leftarrow 1$ **to** m
  for $i \leftarrow 1$ **to** n **do**
    if $\tau_j \in \mathrm{dom}(s_{i-1})$ **then** $\mathrm{runlength}[\tau_j, i] \leftarrow \mathrm{runlength}[\tau_j, i-1] + 1$
foreach $\tau \in \mathcal{T}$ **do**
  clear queue and list of intervals
  for $i \leftarrow n$ **downto** 0 **do**
    if $i > 0$ **and** $\mathrm{backward}[\tau, i] - \mathrm{backward}[\tau, i+1] > 0$ **then**
      push $(i-1)$ onto queue
    if $\mathrm{forward}[\tau, i+1] - \mathrm{forward}[\tau, i] = 0$ **then continue**
    repeat
      $e \leftarrow$ front of queue
      remove $\leftarrow (\mathrm{longest}[\tau] < e - i + 1)$ **or** $(\mathrm{runlength}[\tau, e] < e - i + 1)$
      if remove = **true then** pop front of queue
    until remove = **false or** queue is empty
    if queue is not empty **then**
      $e \leftarrow$ front of queue
      if $e - i + 1 \geq \mathrm{shortest}[\tau]$ **then**

```
20.                call MergeInterval(i, e)
21.        foreach (s, e) in the list of intervals do
22.            for i ← s to e do
23.                prunable[i, τ] ← false
```

Algorithm `FC`()

```
1.   call ComputeForward
2.   call ComputeBackward
3.   call MarkValues
4.   for i ← 0 to n − 1 do
5.       foreach τ ∈ dom(s_i) do
6.           if prunable[i, τ] = true then
7.               remove τ from s_i
```

Table 3.5 shows an execution trace of `MarkValues` on the example from the previous section. Finally, Table 3.6 shows the result, in which domain consistency has been re-established.

Table 3.5: Trace of `MarkValues`.

$\tau = A$			$\tau = B$			$\tau = C$		
i	queue	intervals	i	queue	intervals	i	queue	intervals
8	$\{7\}$	$\{\}$	8	$\{\}$	$\{\}$	8	$\{7\}$	$\{\}$
7	$\{7\}$	$\{\}$	7	$\{\}$	$\{\}$	7	$\{7\}$	$\{\}$
6	$\{7\}$	$\{[6,7]\}$	6	$\{5\}$	$\{\}$	6	$\{7,5\}$	$\{\}$
5	$\{\}$	$\{[6,7]\}$	5	$\{5\}$	$\{\}$	5	$\{7,5,4\}$	$\{\}$
4	$\{\}$	$\{[6,7]\}$	4	$\{5\}$	$\{\}$	4	$\{7,5,4\}$	$\{\}$
3	$\{2\}$	$\{[6,7]\}$	3	$\{5\}$	$\{[3,5]\}$	3	$\{7,5,4\}$	$\{[3,7]\}$
2	$\{2\}$	$\{[6,7]\}$	2	$\{\}$	$\{[3,5]\}$	2	$\{\}$	$\{[3,7]\}$
1	$\{2\}$	$\{[6,7]\}$	1	$\{\}$	$\{[3,5]\}$	1	$\{0\}$	$\{[3,7]\}$
0	$\{2\}$	$\{[0,2],[6,7]\}$	0	$\{\}$	$\{[3,5]\}$	0	$\{\}$	$\{[3,7]\}$

Table 3.6: (left) Domains after setting $s_2 \leftarrow A$; (right) Domains after re-establishing domain consistency.

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A	A	A			A	A	A
			B	B	B		
C	C		C	C	C	C	C

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A	A	A				A	A
			B	B	B		
			C	C	C	C	C

3.2.3 Analysis

It is clear that the three stages of the algorithm all terminate, and that each primitive operation in the pseudo-code can be performed in $O(1)$ time. Examining the bounds of the **for** loops, we see that `ComputeForward` and `ComputeBackward` run in $O(nm^2)$ time, where n is the number of shift variables and m is the number of shift types. `MarkValues` runs in $O(nm)$ time, since we iterate through all variables once for each shift type, building a list of disjoint intervals over $[1, n]$. For a given shift type, each variable results in at most one queue insertion, and each element in the queue can form at most one interval. Therefore, there are at most $O(n)$ intervals. A single call to `MergeInterval` can take $O(n)$ time, but since each interval can be removed at most once, the overall running time for a given shift type is $O(n)$.

To achieve domain consistency, we simply need to run these three stages, and remove values which were not marked. The latter step can be performed in $O(nm)$ time, simply by iterating over an $n \times m$ matrix indicating which variable-value pairs (s_i, τ) are contained in some solution. The overall algorithm therefore runs in $O(nm^2)$ time, and requires $O(nm)$ space. In contrast, Pesant's original stretch propagator runs in $O(m^2l^2)$ time, where l is the maximum of the maximum lengths of the shift types. In applications of the stretch constraint that have been seen thus far, l is a small value ($6 \leq l \leq 9$). Thus, our algorithm is more expensive to achieve by a linear factor.

One of the limitations of Pesant's algorithm that he discusses is its inability to consider the entire sequence. It is possible for a value in a variable's domain to be inconsistent because it is incompatible with the domain values of a different, far away variable in the sequence of shifts. Even though the domain filtering acts locally, considering variables near a variable that was assigned to, sometimes the changes will cascade throughout the roster. However, this is not always the case, and particularly for large instances can cause Pesant's filtering method to fail where ours succeeds.

The following is a small example that proves our algorithm achieves stronger propagation (see Table 3.7). We begin with an instance that is initially domain consistent, thus ensuring that any inconsistent values are introduced by incomplete pruning, and were not present to begin with. The problem instance is for a circular roster, with $n = 8$, $m = 3$, and no pattern restrictions (Π contains all ordered pairs of shift types).

Table 3.7: (left) Bounds on stretch lengths; (right) Initial domains.

τ_k	shortest[τ_k]	longest[τ_k]
A	2	4
B	5	5
C	2	4

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A	A	A	A			A	A
	B	B	B	B	B		
C				C	C	C	C

It is easy to see that this configuration is domain consistent. There are three solutions: AAAACCCC, ABBBBBAA, and CBBBBBCC. Each value of each variable's domain is used in some solution. We first assign the value C to variable s_7, and then to variable s_0 (see Table 3.8).

Table 3.8: Domains after setting $s_7 \leftarrow C$.

Algorithm	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
Pesant's	A	A	A	A				
Algorithm		B	B	B	B	B		
	C				C	C	C	C
Domain	A	A	A	A				
Consistency		B	B	B	B	B		
	C				C	C	C	C

After the second assignment, since no stretch of B's can be shorter than 5, clearly choosing the value A for s_1, s_2 or s_3 requires a stretch of 5 C's, which is a violation of the maximum stretch of C's. Therefore, after the second assignment, the solution can be determined. Pesant's algorithm observes that it is possible to choose a stretch of A's or B's beginning at s_1 without violating the length constraints for those values, but does not consider the cascading effect of removing values from those domains.

Having shown that our algorithm enforces a stronger level of consistency than Pesant's algorithm, we now show that our algorithm achieves domain consistency. In order to justify

Table 3.9: Domains after setting $s_0 \leftarrow C$.

Algorithm	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
Pesant's		A	A	A				
Algorithm		B	B	B	B	B		
	C				C	C	C	C
Domain								
Consistency		B	B	B	B	B		
	C						C	C

that our algorithm achieves domain consistency, we must prove that a value is removed from a variable if and only if that value is not contained in some solution to the constraint. We turn our attention to some facts about the intermediate computations.

Lemma 3.5. *The prefix sums computed by* `ComputeForward` *and* `ComputeBackward` *record, for each type τ and position p, the number of previous positions for which some sequence of stretches exists that can be appended with a stretch of type τ. More precisely:*

(a) The value of forward$[\tau, p]$*, $1 \leq p \leq n+1$ as computed by* `ComputeForward` *is equal to the number of variables s_i with $i < p$ such that some prefix support spans variables $s_0, s_1, \ldots, s_{i-1}$, and is either empty, or ends with a stretch of type τ' where $(\tau', \tau) \in \Pi$.*

(b) The value of backward$[\tau, p]$*, $0 \leq p \leq n$ as computed by* `ComputeBackward` *is equal to the number of variables s_i with $i \geq p$ such that some suffix support spans variables $s_i, s_{i+1}, \ldots, s_{n-1}$, and is either empty, or begins with a stretch of type τ' where $(\tau, \tau') \in \Pi$.*

Proof. We prove (a) by induction on p. Case (b) is analogous and omitted. In the base case, $p = 1$, an empty sequence of stretches is compatible with any type. The algorithm handles this on lines 1-2. For $p > 1$, suppose as our induction hypothesis that the lemma holds for $p' < p$. Clearly this hypothesis implies that if we already know forward$[\tau, p-1]$, it is sufficient to take forward$[\tau, p]$ = forward$[\tau, p-1]$ and then increment this count by

one if and only if there is a prefix support compatible with τ spanning $s_0, \ldots, s_{p-2}$. All prefix supports ending at an earlier shift variable have already been counted. We must show that this is precisely what the algorithm does.

Consider iteration $p-1$ of the outer loop, on lines 9-16. This is the only time a given forward$[\tau, p]$ entry will be modified. Line 10 initializes forward$[\tau, p]$ to the count for the previous variables. All types τ' compatible with τ are considered in the j loop, on lines 11-16. The values lo and hi computed give the range of all possible starting points for a stretch of type τ' ending at s_{p-2}. Moreover, by our induction hypothesis, forward$[\tau', \text{hi}+1] -$ forward$[\tau', lo]$ gives the number of variables between lo -1 and hi -1 (inclusive) where a prefix support compatible with τ' ends. If this value is greater than zero, we can append a stretch of type τ' to one such support, and obtain a prefix support spanning $s_0, \ldots, s_{p-2}$, which corresponds to the algorithm setting forward$[\tau, p] =$ forward$[\tau, p-1]+1$. If there is no such support, the count will never be incremented. □

Theorem 3.6 (Correctness of the algorithm). `MarkValues` *marks a value* (p, τ_j) *if and only if there is some solution which assigns shift type* τ_j *to* s_p.

Proof. First, suppose (p, τ_j) is marked. This means that during iteration j of the outer **for** loop, and some iteration $u \leq p$ of the inner **for** loop, the interval $[u, v]$ was recorded, for some $v \geq p$. Thus, forward$[\tau_j, u+1] -$ forward$[\tau_j, u] > 0$. By the lemma, there exists a prefix support spanning variables $s_0, s_1, \ldots, s_{u-1}$, such that the support is either empty, or the last stretch is compatible with τ_j. We also know that v was removed from the front of the queue, and must have been pushed onto the queue during some iteration $k \geq u$. Therefore, backward$[\tau_j, k] -$ backward$[\tau_j, k+1] > 0$, so there exists some suffix support spanning variables $s_{k+1}, s_{k+2}, \ldots, s_{n-1}$ that is either empty, or such that the first stretch is compatible with τ_j. Moreover, line 15 removes any positions v' from the front of the queue which are too distant from u to satisfy longest$[\tau_j]$, or for which a stretch from u to v' is impossible. Meanwhile, line 19 ensures e is far enough from u to satisfy shortest$[\tau_j]$. Hence, we can form a feasible stretch of type τ_j covering variables $s_u, s_{u+1}, \ldots, s_v$, prefix this stretch with a support covering all of the preceding variables, and append to it a support covering all of the remaining variables, giving a solution which assigns τ_j to s_p.

Conversely, consider a solution which assigns τ_j to s_p. Let s_u and s_v be the first and last variables in the stretch containing τ_j. We have backward$[\tau_j, v+1] -$ backward$[\tau_j, v+2] > 0$

by the lemma. Therefore, inside the j th iteration of the outermost loop, we will push v onto the queue when $i = v$. When $i = u$, we have forward$[\tau_j, u + 1]$ − forward$[\tau_j, u] > 0$, so the **repeat** loop will be entered. Following this loop, v must remain on the queue; the condition on line 14 cannot have been satisfied yet, as we know a stretch of type τ_j can exist spanning $s_u, s_{u+1}, \ldots, s_v$. Therefore, in line 20 $[u, v']$ will be added to the list of intervals, for some $v' \geq v$. It follows that all pairs (i, τ_j) such that $u \leq i \leq v$ will be marked, including (p, τ_j). □

3.3 Incremental Propagation

3.3.1 Overview

An incremental propagator is one that can maintain state between calls (besides the content of the domains), which it uses to reduce the overall workload. For example, although it may be somewhat costly to initially enforce domain consistency for an arbitrary set of domains, removing a value may have a limited, localized effect that is cheap to produce. Often when it does not, there are a lot of values being removed, so the work being done is useful work that would need to be done eventually. So it often is possible to amortize the total running time of the propagation over a sequence of domain modifications and achieve better overall run-time than if the algorithm was run from scratch each time.

Propagating `stretch` incrementally is somewhat difficult, because the side-effects of removing a value are not localized. Table 3.10 gives an example where removing the value A from s_0 causes only one other change—removal of the value A from s_9. In general, removing a value can potentially affect any value in the domain of any variable, and leave all others untouched. As a result, we do not see how to guarantee better than $O(knm)$ complexity for a sequence of k invocations of a propagator. Any algorithm that could do so would likely be impractical to implement. Furthermore, any incremental propagator would likely have a worst-case runtime complexity similar to that of FC, because it is possible for a single domain modification to cause $O(nm)$ values to be removed. Nevertheless, in practice the constraint solver may make frequent calls to the propagator where very little work is required, so even if an incremental propagator has poor worst-case theoretical

performance, it may still be useful.

Table 3.10: (left) Bounds on stretch lengths; (right) Initial domains.

τ_k	shortest[τ_k]	longest[τ_k]
A	3	3
B	4	4
C	4	4
D	2	2
E	1	2

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
A	A	A	A			A	A	A	A
		B	B	B	B	B	B		
C	C	C	C			C	C	C	C
D	D							D	D
E				E	E				E

3.3.2 An Incremental Algorithm for `stretch`

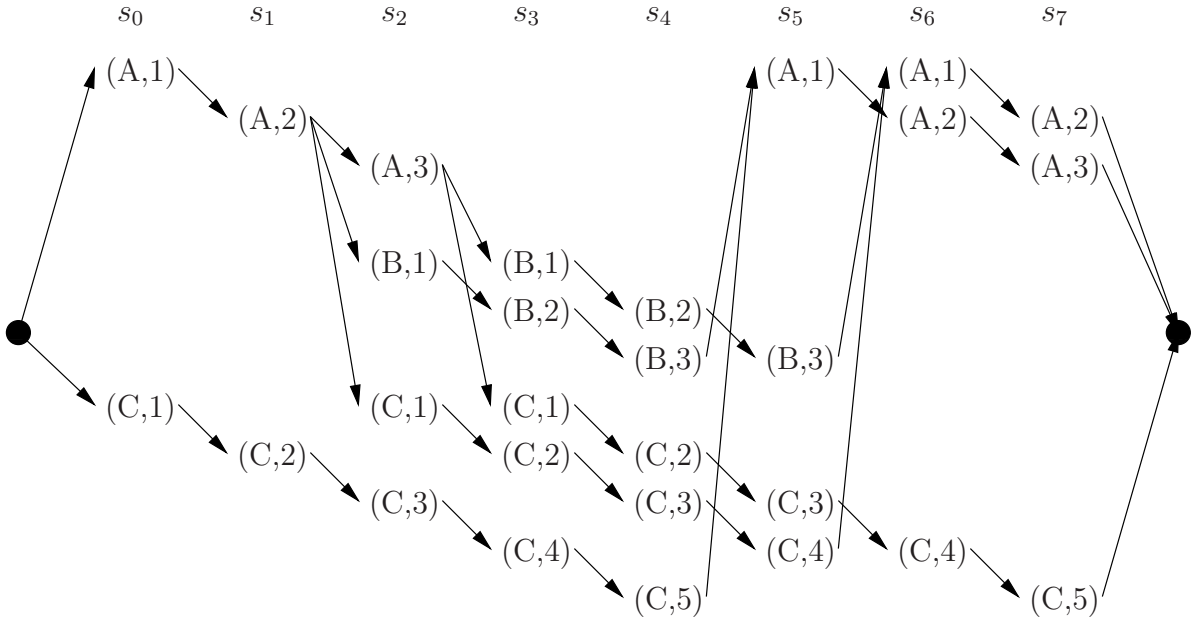
An incremental algorithm for `stretch`, which we will label IC, is possible. The approach draws on ideas from Pesant's regular expression constraint propagator [15], but uses dynamic computation of edges to save time and space. Unfortunately, our algorithm needs to trade a factor of $O(l)$ space in order to reduce overall running time. (Recall that $l = \max_{1 \leq j \leq m}(\text{longest}[\tau_j])$.) However, the total running time over $O(nm)$ value deletions is $O(nm^2l)$, which equals the running time of DC per invocation! The amortized cost per value deletion is $O(ml)$, which is also a significant improvement over the $O(nm^2)$ cost per deletion of FC. In fact, even if FC is run only n times—once when each variable is bound—its total running time of $O(n^2m^2)$ is worse than this incremental approach. When the extra memory overhead is not a big concern, IC is preferable to FC.

The implementation of IC is where our earlier notion of treating the problem as a graph reachability one is particularly useful. We construct an auxiliary directed graph with nodes (s_i, τ_j, p) for each s_i, and $\tau_j \in \text{dom}(s_i)$, and $p = 1, \ldots, \text{longest}[\tau_j]$. This representation denotes the p-th variable in a stretch of type τ_j corresponding to variable s_i. We also add special start and finish nodes, which represent the beginning and end of the roster.

The auxiliary graph also has edges, of course, but as mentioned, they are implicit and determined on the fly to save space. With each node (s, τ, p), we only associate an indegree and an outdegree. We store these in the tables indegree$[s, \tau, p]$ and outdegree$[s, \tau, p]$. The

start node has an outgoing edge to each node of the form $(s_0, \tau_j, 1)$. Each node of the form (s_{n-1}, τ_j, p) where $p \geq \text{shortest}[\tau_j]$ has an outgoing edge to the finish node. Each node of the form (s_i, τ_j, p) has an edge to $(s_{i+1}, \tau_j, p+1)$ if $p < \text{longest}[\tau_j]$, as well as edges to every node of the form $(s_{i+1}, \tau_k, 1)$ if $i + 1 < n$, $(\tau_j, \tau_k) \in \Pi$, and $p \geq \text{shortest}[\tau_j]$. (Naturally, all of the described edges only exist when both of their endpoints exist.)

Figure 3.1: The graph built by IC for the example in Table 3.1 . Deleted nodes are not shown.



Thus, there are $O(nml)$ nodes in the graph, and $O(nm^2l)$ implicit edges. The idea is that each node corresponds to some amount of progress that has been made toward finding a solution. If there is a path from the start node to (s_i, τ_j, p), and a path from (s_i, τ_j, p) to the finish node, it means that there is a solution which makes the assignment $s_i := \tau_j$ where s_i is the p-th variable in its stretch. Using a standard graph reachability test like breadth-first search, we can find all nodes that are reachable from the start node, and all nodes that can reach the finish node. Any node which does not fall into both of these categories can be deleted. We discuss precisely what it means to delete a node below, and how it affects the domain values, but for the moment the reader should take it for granted

that this process will make the variables domain consistent, and produce a graph which can be used later for incremental updates.

It is clear that each node has at most $O(m)$ outgoing edges, which can easily be enumerated, but it is not obvious that we can enumerate incoming edges efficiently. A node corresponding to $p = 1$ may have $O(ml)$ incoming edges, one for each type and stretch length from the preceding variable. Fortunately, there are only $O(nm)$ such nodes, whereas every node corresponding to $p > 1$ has precisely one incoming edge from a node corresponding to $p - 1$. Thus, as long as we make sure our algorithm only traverses or removes each edge a constant number of times, the analysis will remain simple.

The final detail to describe is how the deletion of nodes works. A node can be deleted either during the initialization stage, or during propagation, if its associated shift type is removed from the domain of its associated variable. When a node is deleted, we also delete all incoming and outgoing edges. This involves decrementing the outgoing or incoming edge count of the other endpoint of the edge. If either of these counts ever becomes zero for a node, it too is deleted, since there is either no path to it from the start node, or no path from it to the finish node. Our implementation maintains a queue of nodes scheduled for deletion, and an array which indicates a node is deleted or has been scheduled for deletion, to ensure that no unnecessary work is performed. Node deletions also tell us when to prune values from domains. For a given (s_i, τ_j) pair, we know that the value is used in some solution if and only if there is some p such that (s_i, τ_j, p) is reachable from the start node, and has a path to the finish node. Therefore, we maintain an auxiliary count for each (variable, value) pair indicating how many corresponding active nodes there are. The appropriate counter gets decremented every time a node is deleted, and when one of these counters reaches 0, it is safe to prune the corresponding value from the domain.

Pseudo-code for the initialization phase is given below. In this phase, the edges of the graph are determined using a reachability test and the edge counts for each node are updated accordingly. Then, the graph is scanned for nodes which have an indegree or outdegree of 0. These nodes are scheduled for removal. Finally, the removal algorithm is run.

Algorithm `IC-Initialize`()

1. set all entries of indegree, outdegree, and activecount to 0

```
2.  set all entries of deleted to false
3.  set delete queue to be initially empty
4.  (* traverse forward from start node towards finish node *)
5.  for i ← 0 to n − 1 do
6.      foreach τ_j ∈ dom(s_i) do
7.          for p ← 1 to longest[τ_j] do
8.              activecount[s_i, τ_j] ← activecount[s_i, τ_j] + 1
9.              if indegree[s_i, τ_j, p] > 0 or (i = 0 and p = 1) then
10.                 if p < longest[τ_j] and i < n − 1 then
11.                     outdegree[s_i, τ_j, p] ← outdegree[s_i, τ_j, p] + 1
12.                     indegree[s_{i+1}, τ_j, p + 1] ← indegree[s_{i+1}, τ_j, p + 1] + 1
13.                 if p ≥ shortest[τ_j] then
14.                     if i = n − 1 then
15.                         (* create edge to finish node *)
16.                         outdegree[s_i, τ_j, p] ← outdegree[s_i, τ_j, p] + 1
17.                     else
18.                         foreach τ_k such that (τ_j, τ_k) ∈ Π do
19.                             (* create edge to beginning of next stretch *)
20.                             outdegree[s_i, τ_j, p] ← outdegree[s_i, τ_j, p] + 1
21.                             indegree[s_{i+1}, τ_k, 1] ← indegree[s_{i+1}, τ_k, 1] + 1
22. (* eliminate all dead ends *)
23. foreach (s_i, τ_j, p) do
24.     if (outdegree[s_i, τ_j, p] = 0 or indegree[s_i, τ_j, p] = 0) and not deleted[s_i, τ_j, p] then
25.         deleted[s_i, τ_j, p] ← true
26.         add (s_i, τ_j, p) to delete queue
27. run IC-Delete
```

Algorithm `IC-Delete`()

1. **while** delete queue is not empty **do**
2. $(s_i, \tau_j, p) \leftarrow$ pop front of delete queue
3. indegree$[s_i, \tau_j, p] \leftarrow 0$
4. outdegree$[s_i, \tau_j, p] \leftarrow 0$
5. (* check if the value can be pruned *) activecount$[s_i, \tau_j] \leftarrow$ activecount$[s_i, \tau_j] - 1$
6. **if** activecount$[s_i, \tau_j] = 0$ **then**
7. remove τ_j from dom(s_i)
8. (* delete outgoing edges *)
9. **if** $p \geq$ shortest$[\tau_j]$ **and** $i < n - 1$ **then**
10. **foreach** τ_k such that $(\tau_j, \tau_k) \in \Pi$ **do**
11. **if not** deleted$[s_{i+1}, \tau_k, 1]$ **then**
12. indegree$[s_{i+1}, \tau_k, 1] \leftarrow$ indegree$[s_{i+1}, \tau_k, 1] - 1$
13. **if** indegree$[s_{i+1}, \tau_k, 1] = 0$ **then**
14. deleted$[s_{i+1}, \tau_k, 1] \leftarrow$ **true**
15. add $(s_{i+1}, \tau_k, 1)$ to delete queue
16. **if** $p <$ longest$[\tau_j]$ **and** $i < n - 1$ **and not** deleted$[s_{i+1}, \tau_j, p + 1]$ **then**
17. indegree$[s_{i+1}, \tau_j, p + 1] \leftarrow$ indegree$[s_{i+1}, \tau_j, p + 1] - 1$
18. **if** indegree$[s_{i+1}, \tau_j, p + 1] = 0$ **then**
19. deleted$[s_{i+1}, \tau_j, p + 1] \leftarrow$ **true**
20. add $(s_{i+1}, \tau_j, p + 1)$ to delete queue
21. (* delete incoming edges *)
22. **if** $p > 0$ **then**
23. **if not** deleted$[s_{i-1}, \tau_j, p - 1]$ **then**
24. outdegree$[s_{i-1}, \tau_j, p - 1] \leftarrow$ outdegree$[s_{i-1}, \tau_j, p - 1] - 1$
25. **if** outdegree$[s_{i-1}, \tau_j, p - 1] = 0$ **then**
26. deleted$[s_{i-1}, \tau_j, p - 1] \leftarrow$ **true**
27. add $(s_{i-1}, \tau_j, p - 1)$ to delete queue
28. **else**
29. **foreach** (τ_k, q) such that $(\tau_k, \tau_j) \in \Pi$ **and** shortest$[\tau_k] \leq q \leq$ longest$[\tau_k]$ **do**
30. **if not** deleted$[s_{i-1}, \tau_k, q]$ **then**

31. outdegree$[s_{i-1}, \tau_k, q] \leftarrow$ outdegree$[s_{i-1}, \tau_k, q] - 1$
32. **if** outdegree$[s_{i-1}, \tau_k, q] = 0$ **then**
33. deleted$[s_{i-1}, \tau_k, q] \leftarrow$ **true**
34. add (s_{i-1}, τ_k, q) to delete queue

Notice that each edge is only traversed twice; once in each direction during the initialization phase. During the propagation phase, we only look at edges that are to be deleted. Therefore, each edge gets looked at three times in total. As a result, the algorithm performs $O(nm^2l)$ operations on edges. The actual propagation algorithm is simple with the infrastructure we have built.

Algorithm `IC`(s_i, τ_j)
1. (* τ_j has been removed from dom(s_i) *)
2. **for** $p \leftarrow 0$ **to** longest$[\tau_j]$ **do**
3. **if not** deleted$[s_i, \tau_j, p]$ **then**
4. deleted$[s_i, \tau_j, p] \leftarrow$ **true**
5. add (s_i, τ_j, p) to delete queue
6. run `IC-Delete`

3.3.3 Analysis

As mentioned above, the overall time complexity of the algorithm is $O(nm^2l)$, and the space complexity is $O(nml)$. It is clear from the bounds on the loops that lines 1-26 of algorithm `IC-Initialize` satisfy this, assuming the patterns are stored using an efficient data structure such as an adjacency list. The algorithm adds $O(nml)$ nodes to the delete queue, which is discussed below. All invocations of `IC` require $O(l)$ work in lines 1-5, and since each invocation corresponds to removal of a value, the total work required to remove all values is only $O(nml)$.

Now, when considering `IC-Delete`, notice that during the course of searching for a solution, the main **while** loop is executed precisely once for each node that is deleted. For each node deleted, the loop attempts to remove all incident edges, which takes $O(1)$ time per edge. Lines 6-15 clearly run in $O(m)$ time, and lines 29-34 take $O(ml)$ time, but only

run if $p = 1$. Since there are $O(nml)$ total nodes, and $O(nm)$ nodes with $p = 1$, the overall runtime of `IC-Delete` indeed satisfies $O(nm^2l)$.

To prove the correctness of the algorithm, we establish the relationship between domain consistency, and the consistency the algorithm maintains in the auxiliary graph. Most of the details regarding how the edges are formed were discussed and justified in the previous section, so we omit them here.

Definition (IC-graph consistency). The auxiliary graph built by `IC` is *consistent* if for every node (s_i, τ_j, p) that is not deleted, there is some solution that makes the assignment $s_i := \tau_j$, where s_i is the p-th variable in its stretch, and if every node that is deleted does not correspond to an assignment of this form in any solution.

Lemma 3.7. *The* `IC` *algorithm maintains IC-graph consistency.*

Proof. The algorithm works by initially removing all nodes which do not have both a path from the start node to themselves, and a path from themselves to the finish node. We take this to be already evident, as it is a standard manipulation of a graph. The justification that this is equivalent to the aforementioned form of consistency is that for a given solution, we can construct a path from the start node to the finish node, with every node along the path representing a variable assignment.

When a value is removed from a domain, the algorithm removes all nodes corresponding to that variable and value, thereby eliminating all paths corresponding to solutions that are no longer possible. Consistency is maintained after a deletion if and only if all nodes whose entire set of supporting paths (paths from the start node to the finish node which include the node) pass through the deleted node are removed. To see that this condition is satisfied, consider any path $v_1, v_2, \ldots, v_{k-1}, v_k, v_{k+1}, \ldots, v_{p-1}, v_p$ where the v_i are the nodes on the path; v_1 is the start node, v_p is the finish node, and v_k is the deleted node. When a node is deleted, all of its incident edges are removed, and the corresponding indegree or outdegree counts of adjacent nodes are decremented accordingly. If such a count becomes zero, that node is scheduled to be removed as well. Now, if every supporting path through v_r $(r < k)$ includes v_k, then the same must be true for $v_{r+1}, \ldots, v_{k-1}$. Therefore, when v_k is removed, all outgoing edges from v_{k-1} will be deleted, resulting in its eventual deletion. When v_{k-1} is deleted, v_{k-2} will be scheduled for deletion for the same reason. Repeating

this as many times as needed, eventually v_r will be deleted. The same idea can be used for $r > k$. Eventually all nodes which depend on the originally deleted node will be deleted in this fashion, and therefore consistency is maintained. □

Theorem 3.8. *The* `IC` *algorithm enforces domain consistency.*

Proof. It follows from definition 3.3.3 that the assignment $s_i := \tau_j$ is made in some solution if and only if in the corresponding consistent auxiliary graph, there is some p such that (s_i, τ_j, p) is active (not deleted). Since by lemma 3.7, our algorithm maintains consistency of the auxiliary graph, and since it removes values from domains precisely when there is no such p, the algorithm maintains domain consistency. □

3.4 Cyclic Rosters

For the cyclic version of the problem we are not assured that a stretch starts at position 0, and the first and last stretch must differ. These requirements are easy to account for in the DC and FC algorithms by simply trying all possible starting positions and starting values, and adding a check to ensure that a stretch that ends at position $n-1$ does not have the same value as the initial stretch. The modified algorithm wraps around from the end of the roster to consider the shift variables prior to the starting position as being at the end. So, for a starting position of k, the stretch problem considered would consist of variables $s_k, s_{k+1}, \ldots, s_{n-1}, s_0, s_1, \ldots, s_{k-1}$ in that order. A value is only removed from a domain if it does not become marked as valid during any of the invocations, which means there is no starting position and starting value for which it appears in a solution.

Naively, a modified algorithm using this technique for the circular problem must be invoked $O(nm)$ times. We can improve this by choosing some fixed variable s_i and only considering the possible stretches through s_i. Then the slowdown is at worst $O(m \times \max\{\text{longest}[\tau] : \tau \in \mathcal{T}\})$. By always choosing the variable s_i that minimizes the product $|\text{dom}(s_i)| \times \max\{\text{longest}[\tau] : \tau \in \text{dom}(s_i)\}$ we can greatly reduce this slowdown in most typical problems. If s_i is simply the most recently bound variable, we will have $|\text{dom}(s_i)| = 1$, giving an upper bound of $O(\max\{\text{longest}[\tau] : \tau \in \mathcal{T}\})$ on the slowdown for cyclic instances. At worst, this gives a slowdown of $O(ml)$.

Modifying the incremental algorithm is more complicated, since we do not want to run the algorithm from scratch for each starting point. One solution is to maintain $O(ml)$ different auxiliary graphs, and only remove a value when all of the auxiliary graphs indicate the value does not appear in a solution. This begins to make the memory cost unwieldy, however. Therefore, for cyclic instances, the non-incremental algorithms are preferable.

Chapter 4

Empirical Results

We implemented our domain consistency algorithm for the stretch constraint using the ILOG Solver C++ library, Version 4.2 [9] and compared it to an existing implementation of Pesant's [14] algorithm which enforces a weaker form of consistency (denoted hereafter as WC; also implemented using ILOG Solver).

4.1 Benchmark Problems

Our first class of experiments is benchmark problems. These results are by Hellsten, Pesant, and van Beek, and originally appeared in [8]. The experiments were run on a 3.0 GHz Pentium 4 with 1 gigabyte of main memory.

Our benchmark instances were gathered by Laporte and Pesant [12] and are known to correspond to real-life situations. The problems range from rostering an aluminum smelter to scheduling a large metropolitan police department (see [12] for a description of the individual problems). The constraint models of the benchmark instances combine one or more cyclic stretch constraints with many other constraints including global cardinality constraints [18] and sequencing constraints [19]. The problems are large: the largest stretch constraint has just over 500 variables in the constraint.

On the benchmark problems, our DC propagator offers mixed results over the previously proposed propagator when considering just CPU time (see Table 4.1). When the number of fails is roughly equal, our stronger consistency can be slower because it is more

expensive to enforce. However, our DC propagator also sometimes leads to substantial reductions in number of fails and CPU time. Overall, we conclude that our propagator leads to more robust and predictable performance on these problems. All of the benchmark problems are solved in under one second by our propagator, whereas the propagator based on the weaker form of consistency cannot solve one of the problems in less than twenty seconds.

Table 4.1: Number of failed branches and CPU time (sec.) for cyclic rostering problems from the literature. The variable ordering heuristic fills in weekends first, followed by week days chronologically, and within days either (left) chooses the next shift using minimum domain size, or (right) lexicographically. The value ordering is random.

	WC		DC	
	fails	time	fails	time
atc-1	6	0.01	4	0.00
atc-2	11	0.01	0	0.08
atc-3	48335	13.02	9	0.06
atc-4	108	0.03	25	0.18
alcan-1	0	0.00	0	0.01
alcan-2	970	0.17	967	0.39
burns	1	0.00	80	0.08
butler	2	0.01	1	0.10
heller	3671	1.34	88	0.27
horot	0	0.01	0	0.01
hung	22	0.05	0	0.93
laporte	200	0.04	37	0.05
lau	3	0.01	0	0.09
mot-1	0	0.00	0	0.05
mot-2	9	0.01	9	0.05
mot-3	3331	0.76	17	0.06
slany1	0	0.00	0	0.00

	WC		DC	
	fails	time	fails	time
atc-1	2	0.00	2	0.01
atc-2	266	0.04	1	0.06
atc-3	100982	21.86	2	0.04
atc-4	2356	0.30	1	0.09
alcan-1	0	0.00	0	0.01
alcan-2	955	0.17	995	0.39
burns	6	0.01	6	0.04
butler	2	0.00	2	0.10
heller	3259	1.24	88	0.27
horot	1	0.01	0	0.01
hung	3	0.04	0	0.93
laporte	223	0.04	28	0.04
lau	6	0.01	0	0.09
mot-1	3	0.00	3	0.05
mot-2	9	0.01	9	0.05
mot-3	2799	0.59	39	0.08
slany1	0	0.01	0	0.00

4.2 Random Problems

Table 4.2: Number of failed branches, CPU time (sec.), and number of problems solved within 10 minutes when finding first solution for random cyclic stretch problems. Each fail and time value is the average of only the tests that completed within the time bound of 10 minutes. A total of 50 tests were performed for each combination of n and m.

		WC			DC		
n	m	fails	time	solved	fails	time	solved
50	4	7628.1	0.09	50	0	0.01	50
	6	100089.6	1.21	48	0	0.04	50
	8	138855.7	2.14	50	0	0.08	50
100	4	666002.1	10.17	42	0	0.06	50
	6	281044.4	3.15	38	0	0.15	50
	8	757859.3	11.32	40	0	0.30	50
200	4	246781.2	4.40	19	0	0.26	50
	6	3.5	2.64	24	0	0.59	50
	8	2.9	6.60	22	0	1.09	50
400	4	50653.1	1.19	15	0	1.02	50
	6	90051.8	1.75	17	0	2.40	50
	8	10.2	0.01	14	0	4.25	50

To systematically study the scaling behavior of the algorithm, we also considered random problems. Like the benchmark problems, these results also originally appeared in [8]. The experiments on the random instances were run on a 2.40 GHz Pentium 4 with 1 gigabyte of main memory.

In our first random model, problems were generated that consisted of a single *cyclic* stretch over n shift variables and each variable had its initial domain set to include all m shift types. The minimum shortest$[\tau]$ and the maximum longest$[\tau]$ of the lengths of any stretch of type τ where set equal to a and $a + b$ respectively, where a was chosen uniformly at random from $[1, 4]$ and b was chosen uniformly at random from $[0, 2]$. These

particular small constants were chosen to make the generated problems more realistic, but the experimental results appear to be robust for other choices of small values. No pattern restrictions were enforced (all ordered pairs of shift types were allowed). A random variable ordering was used since, as Pesant [14, p.193] notes, "This not only makes the problem harder to solve but also approximates a more realistic context in which fragments of the sequence may be preassiged or fixed through the intervention of other constraints." In these pure problems nearly all of the run-time is due to the stretch propagators. These problems are trivial for domain consistency, but not so for the weaker form of consistency. We recorded the number of problems that were not solved by WC within a fixed time bound (see Table 4.2). As n increases, the difference between DC and WC becomes dramatic.

In our second random model, problems consisted of a single *non-cyclic* stretch. The domain of each variable was set in two steps. First, the initial domain of the variable was set to include all m shift types. Second, each of the shift types was removed from the domain with some given probability p, $0.0 \leq p < 0.2$. The minimum and the maximum of the lengths of any stretch of type where set equal to a and $a + b$ respectively, where a was chosen uniformly at random from $[1, 25]$ and b was chosen uniformly at random from $[0, 2]$. No patterns were enforced and a random variable ordering was again used. The WC propagator finds these non-cyclic problems much easier than the previous cyclic problems. Nevertheless, on these problems whenever WC is faster than DC the improvement is negligible, whereas our DC propagator can be dramatically faster than WC (see Table 4.3).

Finally, the same random model was used to compare each of the three algorithms we presented in Chapter 3. These results show that IC achieves slightly better performance than FC, but for large instances, the difference may not be substantial enough to warrant the extra memory usage.

Table 4.3: WC versus DC when finding first solution for random non-cyclic stretch problems: ten best improvements in time (sec.) of WC over DC and ten best improvements in time (sec.) of DC over WC. A total of 1500 tests were performed for each value of n. A blank entry means the problem was not solved within a 10 minute time bound.

	10 best for WC		10 best for DC	
n	WC	DC	WC	DC
100	0.05	0.13		0.00
	0.00	0.06		0.05
	0.00	0.06		0.05
	0.00	0.06	164.69	0.05
	0.00	0.06	145.58	0.05
	0.00	0.06	126.70	0.00
	0.00	0.06	51.64	0.05
	0.00	0.06	38.11	0.05
	0.00	0.06	0.80	0.00
	0.00	0.06	0.69	0.00
200	0.06	0.88		0.06
	0.05	0.77		0.06
	0.05	0.77		0.06
	0.05	0.55		0.06
	0.06	0.41		0.06
	0.13	0.48		0.06
	0.13	0.42		0.08
	0.11	0.42		0.11
	0.17	0.44		0.17
	0.17	0.42		0.17

Table 4.4: Average CPU time (sec.) to find the first solution for random non-cyclic stretch problems. A total of 50 tests were performed for each combination of n and m.

		DC	FC	IC
100	8	0.09	0.03	0.02
	16	0.38	0.07	0.05
	32	1.55	0.16	0.16
200	8	0.44	0.15	0.08
	16	1.80	0.30	0.23
	32	7.44	0.68	0.55
400	8	1.90	0.62	0.45
	16	7.85	1.28	0.96
	32	31.79	2.88	2.01
800	8	7.36	2.48	1.65
	16	31.30	5.28	3.34
	32	128.66	12.25	6.86

Chapter 5

Generalizing Stretch

Because so many scheduling problems turn out to be NP-hard, the fact that `stretch` is easily solvable in polynomial time is noteworthy. It begs the question of whether we can generalize the constraint to make it more useful, and whether we can apply the same techniques used in designing our original algorithm to solve such a wider class of problems. It would also be useful to know what kind of generalizations definitely cannot be solved efficiently (unless $P = NP$).

In this chapter, we examine several natural modifications of `stretch`. Some of them lend themselves to efficient domain consistency algorithms, and others are proven NP-hard. To avoid generalizing the constraint in a frivolous manner, we focus on extensions that subsume existing constraints from the literature, or that are intended to be useful for specific kinds of realistic problems.

As we discussed in Chapter 2, independently enforcing domain consistency with respect to two separate constraints does not imply that domain consistency is enforced on the combination of the constraints. Therefore, when we are able to generalize `stretch` to subsume some additional constraint, the new constraint and accompanying propagator provide the potential for more powerful propagation than what would be achieved by using separate independent propagators. Moreover, for several of these constraints our propagators are the first published algorithms for achieving domain consistency.

5.1 Counting Stretches

Counting problems arise frequently in rostering, such as when one wants to restrict the number of shifts an employee might work during a week, or the number of days off. The only form of counting that we have considered within the `stretch` constraint is the restriction of stretch lengths. These restrictions are only local though, in the sense that the validity of a given stretch depends only on whether it has prefix and suffix supports, and not on the structure of any of the supports in particular. It would be useful to also be able to constrain properties of the roster on a global basis using a single constraint, rather than a `gcc` constraint in combination with a `stretch` constraint, for example. Consider the `change` constraint, proposed by Beldiceanu in [3] and defined as follows. Notice that the notion of counting *changes*—pairs of consecutive variables that differ in value—is equivalent to counting stretches.

`change`$(C, \{s_0, s_1, \ldots, s_{n-1}\})$
• $\text{dom}(C) \subseteq \{0, 1, \ldots, n-1\}$
• C equals the number of variables s_i such that $s_i \neq s_{i+1}$

This constraint has a useful and powerful property that we have not discussed yet: the possible change counts in a solution are indicated by a variable. In order to constrain the number of stretches allowed, we can initially configure the domain of C accordingly, or even introduce additional constraints into the CSP that act on C and one or more other variables. This is much more expressive than a version of the constraint in which the possibilities are related to fixed parameters; for example, the specification of an exact count, or a range of possible counts. Beldiceanu argues that we should always try to compose constraints in this fashion [2].

It is quite straightforward to propagate domain consistency for `change` using dynamic programming, although Beldiceanu describes a more generic propagation algorithm for a whole family of constraints which does not necessarily achieve domain consistency. As with our DC algorithm for `stretch`, we can use the fact that if we choose an arbitrary sequence of stretches covering the first k variables of the roster, then the problem reduces to an equivalent subproblem over the remaining $n - k$ variables. Each subproblem can

be specified as a tuple (p, r, τ), where p is the number of variables that have already been assigned values; r is the number of changes we require amongst the remaining variables $s_r, \ldots, s_{n-1}$; and τ is the type used for the previous stretch, and therefore disallowed as a type of the beginning of the stretch beginning at r. By computing similar information with the order of the variables reversed as in DC, we obtain all the information we need to prune the domains. Some extra steps are necessary to ensure that the C variable also has its domain pruned.

Overall, this simple approach yields an algorithm that runs in $O(n^3m)$ time and requires $O(n^2m)$ space. It is likely possible to do better, but it is the similarity to our DC algorithm that is of interest here. Indeed, we omit the details of an algorithm for the `change` constraint alone, and instead focus on a modifying DC so that it enforces both the `stretch` and `change` constraints, giving the new `counted_stretch` constraint.

`counted_stretch`$(C, \{s_0, s_1, \ldots, s_{n-1}\}, \Pi, \text{shortest}, \text{longest})$
• $\text{shortest}[s_i] \leq \text{span}(s_i) \leq \text{longest}[s_i]$
• $s_i = s_{i+1}$ or $(s_i, s_{i+1}) \in \Pi$
• $\text{dom}(C) \subseteq \{0, 1, \ldots, n-1\}$
• C equals the number of variables s_i such that $s_i \neq s_{i+1}$

It is possible to modify both the `DC` and `IC` algorithms to enforce domain consistency for `counted_stretch`. The cost is an extra factor of $O(n)$ in both space and runtime. We present the algorithm `counted_DC` below, which is a modification of the `DC` algorithm.

The idea is to combine the approach described above for the `change` constraint alone with our existing dynamic programming approach, by adding a dimension to the tables which counts the number of stretches. That is, $\text{count}[i, \tau, c]$ now stores the number of ways of forming a prefix support that reaches position i, where the value of the last stretch in the support is τ, and the support consists of c stretches. Similarly, $\text{begins\_stretch}[i, \tau, c]$ indicates whether a stretch of type τ beginning at s_i is contained in any solution, and begins a suffix support that consists of c stretches.

Algorithm `BuildCounts`()

1. initialize all entries of count to 0

```
2.  (* consider all the initial stretches *)
3.  foreach τ ∈ dom(s_0) do
4.      for l ← 1 to min(longest[τ], n) do
5.          if τ ∉ dom(s_{l−1}) then break
6.          if l ≥ shortest[τ] then count[l, τ, 1] ← 1
7.  (* extend to all prefix supports *)
8.  for r ← 1 to n − 1 do
9.      foreach (τ_j, τ_k) ∈ {dom(s_{r−1}) × dom(s_r)} ∩ Π do
10.         for l ← 1 to longest[τ_k] do
11.             if r + l > n or τ_k ∉ dom(s_{r+l−1}) then break
12.             if l ≥ shortest[τ_k] then
13.                 for c ← 1 to n − 1 do
14.                     count[r + l, τ_k, c + 1] ← count[r + l, τ_k, c + 1] + count[r, τ_j, c]
```

Algorithm `counted_DC`()

```
1.  initialize all entries of begins_stretch to false
2.  initialize all entries of prunable to true
3.  foreach τ_j ∈ 𝒯 do
4.      foreach c ∈ C do
5.          begins_stretch[n, τ_j, c] ← true
6.  for r ← n − 1 downto 0
7.      foreach τ_j ∈ dom(s_r), c ∈ {0, ..., n} such that count[r, τ_j, c] > 0 do
8.          max_stretch ← 0
9.          foreach τ_k ∈ 𝒯 such that (τ_j, τ_k) ∈ Π do
10.             for l ← 1 to longest[τ_j] do
11.                 if r + l > n or τ_j ∉ dom(s_{r+l−1}) then break
12.                 if l ≥ shortest[τ_j] and begins_stretch[r + l, τ_k, c + 1] > 0 then
13.                     begins_stretch[r, τ_j, c] ← 1
14.                     max_stretch ← max(max_stretch, l)
15.         for l ← 1 to max_stretch do prunable[r + l − 1, τ_j] ← false
16. for i ← 0 to n − 1 do
17.     foreach τ_k ∈ dom(s_i) do
```

18. **if** prunable$[i, \tau_k]$ = **true then** remove τ_k from dom(s_i)
19. **for** $c \leftarrow 0$ **to** n **do**
20. seen $\leftarrow$ **false**
21. **foreach** $\tau \in \mathcal{T}$ **do**
22. **if** count$[n, \tau, c] > 0$ **then** seen $\leftarrow$ **true**
23. **if not** seen **then** remove c from C

Using analysis similar to that used in Chapter 2, the runtime complexity turns out to be $O(n^2m^2l)$, and the space complexity $O(n^2m)$. The proof of correctness is also analogous.

The `IC` algorithm can be modified in a similar fashion, with the set of nodes being augmented with an extra dimension that indicates the stretch count. A node of the form (s_i, τ_j, p, c) corresponds to s_i being assigned value τ_j, where s_i is the p-th variable in its stretch, and this stretch is the c-th stretch in the solution.

Two related constraints described by Beldiceanu in [2] are the `among` and `count` constraints. The former includes a set of values Σ as a parameter and constrains the number of variables that are assigned one of the types from this set. The latter is simply for the special case where the set of values is a singleton.

`among`$(C, \{s_0, s_1, \ldots, s_{n-1}\}, \Sigma)$
• dom$(C) \subseteq \{0, 1, \ldots, n-1\}$
• $\Sigma \subseteq \mathcal{T}$
• C equals the number of variables $s_i \in \Sigma$

Both of these constraints appear in real world solvers, and both can be incorporated into stretch in much the same way as the `change` constraint. For example, `counted_DC` can be modified so that the count parameter c is incremented by the length of the stretch being considered if that stretch is of a type in Σ (and not incremented otherwise).

5.2 Smooth Stretches

The `smooth` constraint is similar to `change`, except that rather than counting all changes, it only counts *large* changes. A change between s_i and s_{i+1} is large if it satisfies $|s_i - s_{i+1}| > T$,

so we are assuming the values in the domain of each s_i are integers now. As with the `change` constraint, we try to avoid having fixed parameters, so T is a variable. `smooth` turns out to be a generalization of `change`, which corresponds to the case where $\mathrm{dom}(T) = \{0\}$. Thus, we can create an even more general stretch constraint.

`smoothed_stretch`$(C, T, \{s_0, s_1, \ldots, s_{n-1}\}, \Pi, \text{shortest}, \text{longest})$
• $\text{shortest}[s_i] \leq \text{span}(s_i) \leq \text{longest}[s_i]$
• $s_i = s_{i+1}$ or $(s_i, s_{i+1}) \in \Pi$
• $\mathrm{dom}(C) \subseteq \{0, 1, \ldots, n-1\}$
• $\mathrm{dom}(T) \subseteq \{0, 1, \ldots, d\}$, where $d = \max(\tau \in \mathrm{dom}(\{s_0, \ldots, s_{n-1}\}))$
• C equals the number of variables s_i such that $\lvert s_i - s_{i+1}\rvert > T$

It is fairly straightforward to see how to modify the `counted_DC` algorithm in a naive way to enforce domain consistency for this constraint. For a fixed value of t, anywhere the original algorithm would use the table entry $\text{count}[r, \tau_j, c+1]$ for a pattern (τ_i, τ_j), we modify it to instead refer to the entry $\text{count}[r, \tau_j, c]$ if $|\tau_i - \tau_j| \leq t$. Repeating this for each $t \in \mathrm{dom}(T)$, we can find out which values of t have solutions and which can be pruned. The intersection of the prunable tables generated by all invocations tells us which values can be pruned from the shift variables, and the intersection of the change count values tells us how to prune C. The running time is therefore $O(n^2 m^2 l |\mathrm{dom}(T)|)$.

Unlike the `counted_stretch` constraint, having T as a variable costs us in running time. If we instead set t to be a fixed parameter, the algorithm would have the same complexity as `counted_stretch`. This shows that in practice it may sometimes be beneficial to provide specialized versions of a constraint.

5.3 Grouping Types

Another type of constraint proposed by Beldiceanu in [2] is the `group` constraint. Its notion of a *group* is very similar to the notion of a *stretch*. A group is a maximal sequence of variables whose types are all chosen from a set $\mathcal{G}$, which is given as a parameter. The constraint allows one to specify the size of the smallest and largest groups, the minimum

and maximum distances between groups (or between a group and the beginning or end of the roster), the total number of groups in a solution, and the number of variables who take their value from $\mathcal{G}$. If we treat all of the values in $\mathcal{G}$ as one type, and the values not in $\mathcal{G}$ as a second type, this almost becomes a combination of `counted_stretch` and `among`. The only difference is that `group` says that there *must* be groups whose lengths are exactly equal to the lower and upper bounds.

`group`$(C_g, C_v, \{s_0, s_1, \ldots, s_{n-1}\}, \mathcal{G}, \text{MinGroup}, \text{MaxGroup}, \text{MinDist}, \text{MaxDist})$
• $C_g \subseteq \{0, \ldots, n\}$ the number of groups
• $C_v \subseteq \{0, \ldots, n\}$ is the number of variables s_i such that value$(s_i) \in \mathcal{G}$
• The smallest group is of size MinGroup
• The largest group is of size MaxGroup
• The distance between groups or a border and a group is at least MinDist
• The distance between groups or a border and a group is at most MaxDist

One can use an approach similar to our algorithm for `counted_stretch` to enforce consistency with respect to C_g and C_v. To handle the fact that groups are like stretches whose variables can be assigned different values, it is easiest to consider groups one variable at a time. This can be accomplished by augmenting the dynamic programming subproblems with an extra parameter p, indicating how many variables in the group currently being considered have been assigned, similar to how we constructed the auxiliary graph for the `IC` algorithm. To handle the MinGroup and MaxGroup parameters, we can add two extra boolean parameters to the dynamic programming subproblems, indicating whether a group of size MinGroup or a group of size MaxGroup has been included in the support the subproblem corresponds to. The MinDist and MaxDist parameters are handled exactly like shortest and longest.

The dynamic programming subproblems end up being specified by tuples of the form $(r, p, \text{ingroup}, \text{minused}, \text{maxused})$, where the last three parameters are boolean parameters. The boolean valued ingroup parameter takes the place of the τ parameter for the most recent stretch type for previously discussed constraints, since we only need to know whether the previous value was in $\mathcal{G}$ or not. Thus, the space required is $O(2^3 nl) = O(nl)$, where

l = MaxGroup. When considering each subproblem, there are at most m possible values to consider for the next variable, and so the running time is $O(nml)$.

It is quite feasible to combine the `group` constraint with the concepts of stretches, and even smoothed and counted stretches, into one super stretch constraint that combines all of these concepts, using the same principles we have discussed throughout this chapter. We have elected to discuss the concepts separately for simplicity, and because it is unlikely that any realistic problem would require all of these features simultaneously. Moreover, a domain consistency algorithm for such a constraint would likely be impractical to use for anything but very small problems, because all of the variables and parameters to keep track of would result in inordinately high time and space complexity.

5.4 Intractable Variations

In this section, we present some variations of the constraint that seem simple and useful, but turn out to be NP-complete to fully propagate.

5.4.1 Forcing Shift Appearances

It is often useful to force a shift of a certain type to occur at least once. For example, there may be a mandatory cleaning shift that we want to include in a daily roster, but we don't care when it occurs. One approach would be to simply schedule the cleaning shift at a fixed time by binding variables ahead of time to create a pre-defined stretch. This has the drawback that it may limit the possible arrangements for the remaining shifts though. It would be better if we were to incorporate this capability directly into the constraint:

`forced_shift_stretch`$(\{B_1, \ldots, B_m\}, \{s_0, s_1, \ldots, s_{n-1}\}, \Pi, \text{shortest}, \text{longest})$
• $\text{shortest}[s_i] \leq \text{span}(s_i) \leq \text{longest}[s_i]$
• $s_i = s_{i+1}$ or $(s_i, s_{i+1}) \in \Pi$
• $B_j = 1$ if there is a stretch of type τ_j, and 0 otherwise

The idea is that if we want to force a stretch of a certain type to appear, we can set the domain of the corresponding B_j variable to $\{1\}$. (It is easy to force no stretch of a

given type to appear by simply removing the type from all domains.) Unfortunately, this constraint turns out to be much harder than `stretch`.

Theorem 5.1. *Deciding whether an instance of* `forced_shift_stretch` *has a solution is NP-complete.*

Proof. A witness for the problem is a set of supports, one for each value in each variable's domain. This is polynomial in n and m, which shows that the problem is in NP. To show completeness, we proceed with a reduction from HamiltonianPath.

An instance of HamiltonianPath consists of an undirected graph $G = (V, E)$, and the answer is "yes" if and only if there is a path in G that visits each vertex precisely once. We introduce a shift type τ_v for each vertex $v \in V$, and $|V|$ shift variables. Each shift variable's domain contains all types. For each edge $(u, v) \in E$, we let (τ_u, τ_v) be a pattern in Π. Finally, set $B_v = \{1\}$ for each $v \in V$. It is easy to see that by construction, G contains a Hamiltonian path if and only if the corresponding `forced_shift_stretch` instance has a solution, and the construction has size $O(|V|^2)$. □

Corollary 5.2. *Enforcing domain consistency for* `forced_shift_stretch` *is NP-hard.*

A consequence of this is that any constraint which individually restricts the number of occurrences of each shift type is intractable. Consider the stretch constraint where the minimum length shortest$[\tau_i]$ and the maximum length longest$[\tau_i]$ of every stretch of type τ_i is replaced with a set variable l_i which contains the lengths of the possible stretches of that type. Unfortunately, it is intractable to enforce domain consistency on such a generalized stretch constraint.

As another example, consider the stretch constraint which is extended to include domain variables c_i, where c_i denotes the number of possible stretches of type τ_i. Such a constraint would prove useful in modeling real-life rostering problems (see [12] for examples). Unfortunately, it is intractable to enforce domain consistency on this extended stretch constraint. One can model restrictions on the number of stretches of a certain type using a combination of a (regular) stretch constraint and a generalized cardinality constraint over auxiliary variables. However, the amount of pruning achieved by enforcing domain consistency on such a decomposition will necessarily be less than on the extended

stretch constraint since individually the problems are tractable but the combination is intractable (see [5]).

5.4.2 Creating Multiple Rosters

All of the constraints we have presented have dealt with the problem of creating a single roster. It may, however, be useful to try to create a second roster using only values that were not assigned to the first roster. For example, each roster might represent a job that is composed of multiple tasks, each of which can only be completed by certain workers. Thus, workers do not spend all their time working on a single job, and could be scheduled to work on another job during any remaining shifts. In other words, we would like to find disjoint solutions to a `stretch` instance: a set of solutions such that no two of them assign the same value to any variable.

One way to model this problem is by iteratively solving several CSPs. Each CSP includes a `stretch` constraint (and possibly other constraints). When a solution is found to the first CSP, we remove the corresponding values from the initial domains of the variables, and solve the new CSP. This is repeated until we have as many solutions as we need or no more solutions are possible. It turns out that this will not necessarily generate the maximum number of solutions. Consider the example `stretch` instance in Table 5.1. The maximum number of disjoint solutions is two. One such pair of solutions is AAABCC and CCBAAA. However, what if the first solution that the constraint solver finds is CCBBCC? Removing these values from the domains gives the `stretch` instance in Table 5.2, which has no solutions.

Table 5.1: (left) Bounds on stretch length; (right) Initial domains.

τ_k	shortest[τ_k]	longest[τ_k]
A	3	3
B	1	2
C	2	2

s_0	s_1	s_2	s_3	s_4	s_5
A	A	A	A	A	A
		B	B		
C	C	C	C	C	C

Table 5.2: (left) Bounds on stretch length; (right) Initial domains.

τ_k	shortest[τ_k]	longest[τ_k]
A	3	3
B	1	2
C	2	2

s_0	s_1	s_2	s_3	s_4	s_5
A	A	A	A	A	A
		C	C		

Not only does the simple greedy approach above fail to produce a maximum set of solutions, but it may produce a much smaller set. One might hypothesize that a more clever greedy approach might work better. The problem also looks suspiciously like something that might be solved using network flow theory on one of the graph representations previously used to model `stretch`. Any maximum set of disjoint solutions will form disjoint paths in such a graph. However, because all stretches are not the same length, it turns out that this does not work either. The problem turns out to be NP-complete.

As we have stated it, this is an optimization problem. It can be turned into a CSP by introducing a variable whose values are the numbers of disjoint sets possible, and replacing each shift variable with m indicator variables which are 0 if the corresponding value is not used in the solution, and k if it is used in the k-th disjoint solution. We will spare the details, since proving that the decision version of the problem is NP-complete suffices to imply that it is NP-hard to enforce domain consistency for any polynomial-sized CSP formulation.

`parallel_stretch`
Given an instance of the `stretch` constraint, and a value K, determine if there is a set of K or more disjoint solutions. That is, a set of at least K solutions, no two of which assign the same value to any variable.

Theorem 5.3. *Deciding* `parallel_stretch` *is NP-complete.*

Proof. We will reduce from the three-dimensional matching problem, 3DM. An instance of 3DM consists of a set $M \subseteq W \times X \times Y$, where W, X, and Y are disjoint sets having the same number of elements, q. An instance is a "yes"-instance iff there is a subset $M' \subseteq M$ such that $|M'| = q$ and no element of W, X, or Y appears in more than one element of M'. See for example [6] for more details on 3DM and a proof of NP-completeness.

The idea is to construct a `stretch` instance where each element of M corresponds to a distinct solution by defining the stretch lengths so that only stretches corresponding to elements of M interlock in a way that leads to a solution. Each solution will consist of four stretches. The first, second, and third of these correspond to the elements of W, X, and Y, and the minimum and maximum stretch lengths are constrained so that for each element in one of these sets, any corresponding stretch must include a certain variable. This ensures that two disjoint solutions cannot include the same element. The fourth stretch is used to ensure that only triples contained in M can be considered in solutions. Any sequence for the first three stretches which does not correspond to an element of M will not have a corresponding fourth stretch that can be appended to fill out the roster.

To arrange our stretches so that any two stretches of a type τ corresponding to an element of W, X, or Y necessarily overlap, consider the mapping $\varphi : W \cup X \cup Y \to \mathbb{N}$ in which the i-th element in W (the order is set arbitrarily), the j-th element in X, and the k-th element in Y map to i, $j(q+1)$, and $k(q+1)^2$ respectively. Define $\phi : W \times X \times Y \to \mathbb{N}$ as $\phi(v) = \varphi(v_W) + \varphi(v_X) + \varphi(v_Y)$, where v_W, v_X, and v_Y indicate the components of v corresponding to sets W, X, and Y. This maps each element in M to a number whose digits in base-$(q+1)$ indicate the components of the triple. Now, for each value $u \in W \cup X \cup Y$, we will create a corresponding value in our `stretch` instance, with $\text{shortest}[u] = \text{longest}[u] = \varphi(u)$. We also create tail stretch values for each $v \in M$, with $\text{shortest}[v] = \text{longest}[v] = \phi(v)$. The domains are defined as follows:

- For each $w \in W$, variables $s_0, s_1, \ldots, s_{\varphi(w)-1}$ contain w. The total number of variables containing w is exactly the length of any stretch of type w.
- For each $x \in X$, variables $s_1, s_2, \ldots, s_{q+\varphi(x)-1}$ contain x. The total number of these variables is $q + \varphi(x) - 1$, and since $\varphi(x) > q - 1$, any two valid stretches of type x must overlap.

- For each $y \in Y$, variables $s_{q+2}, s_{q+3}, \ldots, s_{q+q(q+1)+\varphi(y)-1}$ contain y. The total number of these variables is $q(q+1)+\varphi(y)-2$. Again, since $\varphi(y) > q(q+1)-2$, any two valid stretches of type y overlap.

- For each $v \in M$, variables $s_{\phi(v)}, s_{\phi(v)+1}, \ldots, s_{n-1}$ contain v, where $n = (q+1)^3$. The purpose of these variables is to ensure that only triples corresponding to elements of M can be chosen in a solution to the `stretch` problem.

We have set up our stretch lengths and domains so that for any $v \in W \times X \times Y$, a corresponding prefix support containing three stretches can be found. Moreover, this prefix support covers precisely the first $\phi(v)$ variables. There will be a stretch of type v from $s_{\phi(v)}$ to the end of the roster if and only if $v \in M$. Table 5.3 gives an example of a roster produced by this transformation. Notice that triples that are not in M such as $(w_1, x_1, y_2$ do not correspond to solutions. Choosing stretches of types w_1, x_1, and y_2 covers variables s_0 up to $s_2 1$, but there is no tail stretch to append.

The correctness of our transformation follows easily from the construction. For any solution $M' \subseteq M$ to a "yes"-instance of 3DM, it is clear by construction that there is a set of q disjoint solutions to the `parallel_stretch` instance, since no two elements of M' share the same component, and they each have a corresponding tail stretch. Conversely, a solution to `parallel_stretch` of size q corresponds to a "yes"-instance of 3DM by virtue of the previously noted properties that no two solutions can use a stretch of a given type; and each choice for the first three stretches of a solution covers a distinct number of variables, which can only be extended to the end of the roster if the first three stretches correspond to an element of M.

It is obvious that we can verify a solution to `parallel_stretch` in polynomial time, by checking the stretches in the solution to ensure they are valid and do not overlap. The transformation we have given from 3DM can also be computed in polynomial time, since the resulting stretch problem has $O(q^3)$ variables and $O(q^3)$ values. □

Resource optimization problems are often NP-complete, so it is to be expected that most stretch variations of an optimization nature are intractable. As another example, consider a variation in which only solutions in which the number of shift types (e.g. employees) does not exceed some threshold are considered. Whereas `parallel_stretch` would be useful to

maximize the productivity of a fixed number of workers, this is something of a counterpart and would be useful to minimize the number of workers needed to perform a fixed task. However, this problem is also NP-complete, by a simple reduction from Min-Cover.

5.5 Other Extensions

The extensions we described in the first part of this chapter all correspond to existing constraints that have been proposed in the literature and are useful in practice. For example, the `counted_stretch` constraint would be useful if we wanted to restrict the number of employee changeovers that occur during a work schedule. With the original `stretch` constraint, we could attempt to accomplish this by narrowing the bounds on stretch lengths, but this would not work effectively if we want to still allow very short or very long work stretches. A application of the `smoothed_stretch` is to force some property to change gradually. If the values represent production quantity, one may want to set high values near Christmas time, low values earlier in the year, and force the quantities to be ramped up over time.

There are many more tractable variations of `stretch` that may be useful which we have not discussed, since they do not generalize previously studied constraints. Most of these can be solved using small modifications to one of the algorithms we have presented. Some examples are:

- Restrict the length of the k-th stretch for some k, or the possible types of the k-th stretch to some subset of the shift types. It is simple to augment the propagator for `counted_stretch` to handle arbitrarily many constraints of this form.
- Allow different pattern sets to be used for specific pairs of variables, or specific stretch counts.
- Define a cost function for each variable or stretch that depends on the type, and only permit solutions whose cost does not exceed a certain value. This can be done by modifying the `DC` algorithm to store the minimum cost in its tables rather than the number of solutions.

Table 5.3: (top) Bounds on stretch length; (bottom) Domains; The `parallel_stretch` instance corresponding to the 3DM instance with $M = \{(w_1, x_1, y_1), (w_1, x_2, y_2), (w_2, x_2, y_2)\}$. Here $q = 2$, and t_1, t_2, and t_3 correspond to the values of M in the order they are listed.

τ_k	shortest$[\tau_k]$	longest$[\tau_k]$
w_1	1	1
w_2	2	2
x_1	3	3
x_2	6	6
y_1	9	9
y_2	18	18
t_1	14	14
t_2	2	2
t_3	1	1

s_0	s_1	...	s_4	...	s_7	...	s_{13}	...	s_{16}	...	s_{25}	s_{26}
w_1												
w_2	w_2											
	x_1	...	x_1									
	x_2	...	x_2	...	x_2							
			y_1	...	y_1	...	y_1	...	y_1			
			y_2	...	y_2	...	y_2	...	y_2	...	y_2	
							v_1	...	v_1	...	v_1	v_1
											v_2	v_2
												v_3

Chapter 6

Conclusions

Much of the past research on CSPs has focused on binary constraint networks, in which all constraints apply only to pairs of variables. This class of CSPs is justifiably interesting, because any non-binary constraint network can be transformed into a binary one. However, in practice CSPs are rarely solved this way because the domains of the transformed problem can become exponentially large, and any inherent structure in the problem can be muted. Recent trends involve placing more emphasis on techniques for solving CSPs expressed using N-ary constraints.

The study of non-binary constraints presents some of its own unique issues. Most of the work on binary CSPs of course does not generalize easily, if at all, to higher-arity constraints. This means that most work must focus on specific constraints. However, there are infinitely many constraints that one might imagine! Far fewer are useful or interesting. We feel that one of the most important goals in the area should be to produce constraints that are as general as possible while maintaining efficient consistency propagation, since it is often better for propagation to model a problem using a single constraint rather than multiple independent ones.

One approach to generalization is to extend existing constraints where possible, so that they can handle a wider class of problems. This is one of the contributions of this thesis, and an earlier example is the `gcc` constraint, which generalizes `alldifferent`. It is also useful to identify the limits of these generalizations–what kind of changes make it intractable to achieve certain levels of consistency? This way we have a clear demarcation

of what problems can be solved within a constraint family.

6.1 Stretch Problems

This thesis presented efficient algorithms for enforcing domain consistency propagation of the stretch constraint, both incrementally, and non-incrementally. We showed how to extend the `stretch` constraint and our domain consistency algorithms to generalize multiple constraints from the literature. We also discussed other types of generalizations for which enforcing domain consistency is intractable.

The experimental results comparing our algorithm to Pesant's original algorithm underscore the importance of being able to enforce a strong level of consistency. While Pesant's algorithm is actually quite effective most of the time, there are occasional instances that it cannot solve in any reasonable period of time. Being able to enforce domain consistency guarantees that any bottlenecks are outside of the `stretch` propagator, and are related to the constraint solver and interference by any other constraints which are also being propagated.

6.2 Future Work

The treatment of the `stretch` constraint in this thesis solves most of the algorithmic side of the problem of propagating domain consistency. It may be possible to squeeze small performance improvements out of our FC algorithm, but in practice these would likely not make a huge difference in performance. Of greater interest is the $O(nml)$ memory consumption of the IC algorithm. An incremental approach that lowers this to the optimal complexity of $O(nm)$ would be useful, and eliminate the main reason one might prefer to use FC.

One other promising avenue for future work is Pesant's recent regular expression constraint [15], and his corresponding algorithm for enforcing domain consistency. It allows a sequence of variables to be constrained to match a given regular expression or DFA. Indeed, this constraint subsumes the original `stretch` constraint, and although it has higher worst-case complexity than our approach, its expressive nature allows it to handle a wide variety

of problems. If it could be combined with counting features to handle our extensions of `stretch`, it would be an extremely useful and general constraint.

Bibliography

[1] 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*, pp. 311–318, Madison, Wisconsin, 1998.

[2] N. Beldiceanu. Global constraints as graph properties on structured networks of elementary constraints of the same type. (T2000/01) (2000).

[3] N. Beldiceanu. Pruning for the cardinality-path constraint family. Technical Report T2001/11A, SICS, 2001.

[4] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence* **65** (1994), 179–190.

[5] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. The complexity of global constraints. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, San Jose, California, 2004.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

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

[8] L. Hellsten, G. Pesant, and P. van Beek. A domain consistency algorithm for the stretch constraint. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, Toronto, 2004.

[9] ILOG S. A. ILOG Solver 4.2 user's manual, 1998.

[10] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 541–547, Montréal, 1995.

[11] G. Laporte. The art and science of designing rotating schedules. *J. of the Operational Research Society* **50** (1999), 1011–1017.

[12] G. Laporte and G. Pesant. A general multi-shift scheduling system. *J. of the Operational Research Society* (2004). Accepted for publication.

[13] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence* **28** (1986), 225–233.

[14] G. Pesant. A filtering algorithm for the stretch constraint. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pp. 183–195, Paphos, Cyprus, 2001.

[15] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, Toronto, 2004.

[16] C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pp. 600–614, Kinsale, Ireland, 2003.

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

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

[19] J.-C. Régin and J.-F. Puget. A filtering algorithm for global sequencing constraints. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pp. 32–46, Linz, Austria, 1997.