Combinatorial Reasoning for Sets, Graphs and Document Composition

Graeme Keith Gange

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

Department of Computing and Information Systems
The University of Melbourne

December 2012

Abstract

Combinatorial optimization problems require selecting the best solution from a discrete (albeit often extremely large) set of possible candidates. These problems arise in a diverse range of fields, and tend to be quite challenging. Rather than developing a specialised algorithm for each problem, however, modern approaches to solving combinatorial problems often involve transforming the problem to allow the use of existing general optimization techniques.

Recent developments in constraint programming combine the expressiveness of general constraint solvers with the search reduction of conflict-directed SAT solvers, allowing real-world problems to be solved in reasonable time-frames. Unfortunately, integrating new constraints into a lazy clause generation solver is a non-trivial exercise. Rather than building a propagator for every special-purpose global constraint, it is common to express the global constraint in terms of smaller primitives.

Multi-valued decision diagrams (MDDs) can compactly represent a variety of common global constraints, such as REGULAR and SEQUENCE. We present improved methods for propagating MDD-based constraints, together with explanation algorithms to allow integration into lazy clause generation solvers.

While MDDs can be used to express arbitrary constraints, some constraints will produce an exponential representation. `s-DNNF` is an alternative representation which permits polynomial representation of a larger class of functions, while still allowing linear-time satisfiability checking. We present algorithms for integrating constraints represented as `s-DNNF` circuits into a lazy clause generation solver, and evaluate the algorithms on several global constraints.

Automated document composition gives rise to many combinatorial problems. Historically these problems have been addressed using heuristics to give *good enough* solutions. However, given the modest size of many document composition tasks and recent improvements in combinatorial optimization techniques, it is possible to solve many practical instances in reasonable time.

We explore the application of combinatorial optimization techniques to a variety of problems which arise in document composition and layout. First, we consider the problem of constructing optimal layouts for k-layered directed graphs. We present several models models for constructing constructing layouts with minimal crossings, and with maximum planar subgraphs; motivated by aesthetic considerations, we then consider weighted combinations of these objectives – specifically, lexicographically ordered objectives (first minimizing one, then the other).

Next, we consider the problem of minimum-height table layout. We consider existing integer-programming based approaches, and present $A^\star$ and lazy clause generation methods for constructing minimal height layouts. We empirically demonstrate that these methods are capable of quickly computing minimal layouts for real-world tables.

We also consider the guillotine layout problem, commonly used for newspaper layout, where each region either contains a single article or is subdivided into two smaller regions by a vertical or horizontal cut. We describe algorithms for finding optimal layouts both for fixed trees of cuts and for the free guillotine layout problem, and demonstrate that these can quickly compute optimal layouts for instances with a moderate number of articles.

The problems considered thus far have all been concerned with finding optimal solutions to discrete configuration problems. When constructing diagrams, it is often desirable to enforce specified constraints while permitting the user to directly manipulate the diagram. We present a modelling technique that may be used to enforce such constraints, including non-overlap of complex shapes, text containment and arbitrary separation. We demonstrate that these constraints can be solved quickly enough to allow direct manipulation.

Declaration

This is to certify that

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Graeme Keith Gange

Preface

- Chapter 3 is substantially previously published as:

  G. Gange, P. Stuckey, R. Szymanek, "MDD Propagators with Explanation" *Constraints* 2011, **16**:407–429

- Chapter 4 is substantially previously published as:

  G. Gange, P. Stuckey, "Explaining Propagators for s-DNNF Circuits" *Proceedings of the 9th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming* 2012, to appear.

- Chapter 5 is substantially previously published as:

  G. Gange, P. J. Stuckey, K. Marriott, "Optimal k-level Planarization and Crossing Minimization" *Proceedings of the 18th International Symbosium on Graph Drawing* 2010, 238–249

- Chapter 6 is substantially previously published as:

  G. Gange, K. Marriott, P. Moulder, P. Stuckey, "Optimal Automatic Table Layout" *Proceedings of the 11th ACM Symposium on Document Engineering* 2011, 23–32.

- Chapter 7 is substantially previously published as:

  G. Gange, K. Marriott, P. Stuckey, "Optimal Guillotine Layout" *Proceedings of the 12th ACM Symposium on Document Engineering* 2012, to appear.

Acknowledgements

I always assumed that this section would be my opportunity to settle scores and air any grudges I accumulated over the years. Alas, I am instead forced to confess my gratitude towards a great many people without whom this work would never have been completed. To those who I have neglected to include in the following list, I offer humble apologies; the omission is due solely to my faulty and incomplete memory, rather than any lack of appreciation.

Thanks, then, are due first and foremost to my supervisor, Peter Stuckey, for tolerating me for all these years. Kim Marriott, for the reliable supply of interesting problems to solve. My thesis committee, who attended several meetings on extremely short notice as deadlines loomed. My office-mates Ben, Alex, Kian, Abdulla, Matt, Shanika and Jorge, for many a diverting conversation.

A debt of gratitude is also owed to those who helped me stay fed and supplied with caffeine; the University of Melbourne, for providing the Henry James Williams scholarship, and those who employed me over the years – Michael Kirley, Alistair Moffat and Harald Søndergaard.

All those with whom I've shared a house during my candidature; notably Kent, Nhi, Mike and Anh, who are all far more wonderful and tolerant than anyone could ask. The #7/Grand Fed ex-pats and associated diaspora, for helping me to keep hold of what little sanity remains. Michelle, for demonstrating that research was an option. Kate, for putting me back together again. Jackie, because I still remember.

And finally, to *you*. Because if you've read this far, either you're someone who should have been on this list, or there's a reasonable chance you're planning to read further. And there's nothing more meaningless than a piece of writing that is never read.

Contents

List of Figures

List of Tables

1

Introduction

COMBINATIONAL optimization addresses the task of finding the best solution for a problem from a (potentially extremely large) discrete set of possible choices. These problems arise in an exceptionally diverse range of fields, from the logistics of transporting goods [Crainic and Rousseau, 1986] and packing crates [Martello et al., 2000], to designing staff rosters [Demassey et al., 2006, Brand et al., 2007], cutting glass panes [Dyson and Gregory, 1974] and scheduling sporting tournaments [Rasmussen and Trick, 2008]. Accordingly, a great deal of effort has been expended in developing techniques for solving combinatorial optimization problems; indeed, early work on the *assignment problem* dates to at least the 18^{th} century.

It was not until the late 1940s, with the development of linear programming, that the field of combinatorial optimization developed dramatically. Linear programming gives the first example of a separation between the problem model and the optimization procedure. Where previously it was necessary to painstakingly develop an optimization procedure for each individual problem, now any problem that could be expressed as a linear program over continuous domains could be solved using the simplex algorithm [Dantzig, 1963]; even better, any improvements to solution techniques could immediately benefit the entire class of problems. The development of integer programming techniques [Gomory, 1960] soon after finally provided the ability to solve arbitrary problems over discrete domains.

Even though it is possible to model a problem as an integer program, that doesn't necessarily mean that the resulting model will be concise or perform well.

While linear and integer programs were excellent for modelling problems involving sets of linear constraints, it was often inconvenient to model problems with a complex structure – such as problems involving disjunctions, inequalities or permutations. However, the principle of separation between model and solver was instrumental to the eventual development of constraint programming [Jaffar and Lassez, 1987]. Where integer programs are limited to inequalities over linear sums, CP solvers can support arbitrary relations over sets of variables. Unfortunately, since each constraint is enforced independently and communication occurs only through variable domains, CP solvers cannot take advantage of the same degree of global reasoning available to integer programming solvers.

Parallel to the development of integer programming techniques and constraint programming solvers, algorithms were developed for Boolean satisfiability (SAT) problems. SAT problems are a very restricted class of combinatorial problem; instances contain only Boolean variables, and all the constraints are disjunctions of either positive or negated variables. Although heavily restricted, SAT is nevertheless NP-complete [Cook, 1971]. While the core algorithm for solving SAT problems was developed in 1962 [Davis et al., 1962], the solvers were not particularly effective for hard instances. It was only with the development of conflict-based learning algorithms [Zhang et al., 2001], allowing the elimination of large parts of the problem space, that it became feasible to solve large industrial instances, and worthwhile to transform other combinatorial problems into SAT. Being restricted to Boolean variables, however, it is difficult to encode numerical problems with large domains.

As a wide range of combinatorial optimization problems require both complex constraints and non-Boolean variables, we would like to take advantage of the learning properties of SAT solvers, while maintaining the expressiveness of finite-domain CP solvers. Lazy-clause generation solvers [Ohrimenko et al., 2009] achieve this by maintaining a dual representation of the problem – they use a finite-domain CP engine to propagate inferences, but also construct a partial SAT model for the active parts of the problem. When a conflict is detected, conflict-based clause learning is then performed on this partial SAT model. Recently, this combination of techniques has dramatically improved solver performance on a wide range of optimization and satisfiability problems. This improved performance comes at a price, however – implementing a complex global propagator in a conventional

finite-domain constraint solver requires only the definition of a filtering algorithm; a propagator in a lazy clause generation solver must also be able to (often retrospectively) generate an explanation for any inferences it makes.

Document composition and layout are fields that give rise to an extraordinary variety of combinatorial problems. Even seemingly simple problems, such as table layout (described in Chapter 6), often turn out to be NP-hard. Document composition has historically been dominated by manual composition; however, with the rise of automatically generated and customized media, it is no longer possible for every document to be manually designed.

Most *automated* document composition methods [González et al., 1999, Strecker and Hennig, 2009, Di Battista et al., 1999], then, tend to use a variety of heuristics to generate an acceptable solution. In many cases, this may be sufficient. However, evaluation of heuristics tends to be problematic. We can compare the performance of heuristics relative to one another, but without some method for determining the optimal solution we cannot determine how good the heuristic is. Now with modern solver technology, we can often find solutions to practical instances quickly enough that there is no reason to use a heuristic rather than computing the optimal solution. A further advantage of using standard solver technology is (as we shall see in Chapter 5) that it permits easier exploration of related problems – it is possible to quickly experiment with different objective values and side-constraints, rather than having to construct a completely new heuristic for each modified objective.

Sometimes, however, completely automated composition is not desirable. Constraint-assisted layout allows elements of the document to be directly manipulated, and adjusts the other elements such that specified constraints are maintained. These systems are generally limited to constraints that can be conveniently represented as linear constraints, such as alignment, distribution and linear separation. In Chapter 8 we present a modelling technique to integrate complex geometric constraints (such as non-overlap of non-convex polygons) into a simplex-based constraint-assisted layout system.

This thesis makes a number of contributions. First, we improve lazy clause generation constraint solvers by developing algorithms for quickly constructing efficient propagators for problem-specific global constraints. Second, we develop

models for finding optimal solutions to a variety of combinatorial layout problems. And finally, we develop techniques that allow the integration of complex, disjunctive constraints into a constraint-assisted layout system.

1.1 Overview

In Chapter 2, we will introduce the general classes of optimization (and satisfiability) problems we consider – integer and linear programming, constraint satisfaction problems, Boolean satisfiability and dynamic programming – and the common approaches used for finding optimal solutions to each.

Developing global propagators for problem-specific constraints in a lazy clause generation solver tends to be time-consuming and error-prone. In Chapter 3, we present algorithms for integrating constraints expressed as Multi-valued Decision Diagrams (MDDs) into a lazy clause generation solver, permitting the convenient construction of problem-specific global constraints. We demonstrate that these MDD propagators can outperform state-of-the-art constraint solvers.

A disadvantage of MDDs is that they are limited in expressiveness – some classes of constraints produce an MDD with an exponential number of nodes. In Chapter 4, we generalise the algorithms described in Chapter 3 to `s-DNNF`, a representation that admits polynomial representations of a wider class of constraints, and compare these propagators to comparable SAT-based decompositions for the constraints.

We then move from developing new constraint-solving techniques to applying combinatorial optimization techniques to a variety of problems that occur in document and diagram composition. In Chapter 5, we consider the problem of constructing optimal layouts for k-layered directed graphs. We present SAT and MIP-based models for layouts with minimal crossings, and maximal planar subgraph. Motivated by aesthetic considerations, we also consider variants with combined objectives – first minimising the number of crossings then finding the maximum planar subgraph, and first planarization then crossing minimization. We then evaluate these models on a set of both collected and randomly generated graphs.

In Chapter 6 we present several models for constructing minimum-height layouts for HTML-style tables. We compare integer-programming, $A^\star$and lazy clause

generation approaches on a combination of real-world and artificially generated tables, both with and without column- and row-spans.

Related to table layout is the problem of constructing guillotine layouts for a collection of documents, such as when laying-out pages for a newspaper. In Chapter 7, we present dynamic programming methods for constructing optimal guillotine layouts for a moderate number of articles, and for efficiently updating a fixed layout with a new display width.

As observed in Chapter 7, it is sometimes convenient to update a layout by moving to a nearby solution according to user input. In Chapter 8, we develop modelling techniques for handling complex disjunctive constraints in a constraint-based diagram layout system. We then demonstrate the effectiveness of these techniques by demonstrating non-convex polygon non-overlap constraints and flexible text-containment.

2

Background

THIS chapter introduces the various classes of (mostly-)combinatorial problems we shall be considering throughout this thesis, together with common approaches used to solve them.

2.1 Linear Programming

A *linear program* is a constrained continuous optimization problem over a set of variables $X = \{x_1, x_2, \ldots, x_n\}$, of the form

$$\min \sum_i c_i \cdot x_i$$
$$\mathbf{s.t} \bigwedge_j \sum_i a_{ij} \cdot x_i \leq k_j$$

the feasible region of which forms a convex polytope. The most common method for solving such problems is the simplex method [Dantzig, 1963], however interior-point methods (such as the ellipsoid [Khachiyan, 1979] method) also exist.

The simplex method relies on the observation that the optimal solution must lie on an *extreme point* (i.e. vertex) of the feasible region. The algorithm first finds a feasible extreme point, then greedily walks between adjacent extreme points along the boundary of the feasible region until no move improves the objective. In the worst case, this may require an exponential number of steps; however the average case complexity over a family of randomly perturbed linear programs is polynomially bounded [Spielman and Teng, 2004], and the algorithm performs well in practice.

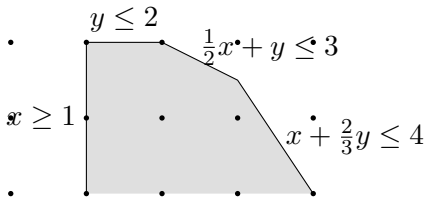


Figure 2.1: Feasible region for the linear program in Example 2.1. Integer coordinates are marked with a dot.

When solving a series of closely related linear problems, the simplex algorithm can be given a *warm-start*, by using the optimum for a previous problem to construct the initial basis for the next (see, e.g., Maros and Mitra [1996]).

Interior point methods, conversely, traverse the interior of the feasible region, iteratively refining a conservative approximation of the optimal solution. The ellipsoid method, an early interior point method, is primarily of use in complexity proofs – it is of polynomial complexity, but too slow to be useful in practice. More recent methods, such as Karmarkar's method [Karmarkar, 1984], outperform simplex-based methods on large linear programs [Karmarkar and Ramakrishnan, 1991]. However, for both integer programming (Section 2.1.1) and interactive constraint-based layout (Section 8.2), closely related problems must be repeatedly solved, so simplex-based methods are usually used.

The first step in the simplex algorithm is transforming the problem into *standard form*. In standard form, the problem is reformulated as a minimization problem, and all inequalities are transformed into equalities by introducing an additional *slack variable* for each constraint, indicating the amount of slack between the current solution and the constraint.

Example 2.1. *Consider the linear program*

$$\begin{aligned} \textbf{\textit{max}} \quad & x + y \\ \textbf{\textit{s.t.}} \quad & \frac{1}{2}x + y \leq 3 \\ & x + \frac{2}{3}y \leq 4 \\ & x \geq 1 \\ & y \leq 2 \\ & x, y \geq 0 \end{aligned}$$

The feasible region for this problem is shown in Figure 2.1. The problem is then transformed into standard form by introducing slack variables:

$$\begin{aligned} \textbf{\textit{min}} \;\; & f(x,y) = -x - y \\ \textbf{\textit{s.t}} \;\; & \frac{1}{2}x + y + s_1 = 3 \\ & x + \frac{2}{3}y + s_2 = 4 \\ & x - s_3 = 1 \\ & y + s_4 = 2 \\ & x, y, s_1, s_2, s_3, s_4 \geq 0 \end{aligned}$$

□

Once the problem is reformulated, the simplex algorithm proceeds in two phases. In Phase I, we construct an initial basic feasible solution. A *basic* solution for a problem with n variables and c constraints is a solution with at most c non-zero variables – referred to as *basic* variables. We can construct a basic solution by making all initial problem variables 0 and maximising the introduced slack variables; however, if there are any $\geq$ constraints, this will not be feasible.

Instead, we introduce an additional *artificial* variable for each such constraint. This allows us to construct a feasible solution to the modified problem. We then proceed to minimize the artificial variables – once we find a basic solution with no artificial variables in the basis, we have a basic feasible solution to the original problem.

Example 2.2. *We cannot immediately compute a basic feasible solution for the problem in Example 2.1, as this would result in s_3 having a negative value. Instead, we introduce an artificial variable a, giving us the following equations:*

$$\begin{aligned} s_1 &= -\frac{1}{2}x - y + 3 \\ s_2 &= -x - \frac{2}{3}y + 4 \\ a &= -x + s_3 + 1 \\ s_4 &= -y + 2 \\ \hline f &= -x - y \\ g &= -x + s_3 + 1 \end{aligned}$$

Using this notation, variables on the left-hand side of the equations form the basis*; all other variables take the value 0. The new objective g represents our goal of removing the artificial variable a from the solution. We remove a from the basis by replacing all occurrences of x with $s_3 - a + 1$:*

$$\begin{aligned} s_1 &= -y - \frac{1}{2}s_3 + \frac{1}{2}a + \frac{5}{2} \\ s_2 &= -\frac{2}{3}y - s_3 + a + 3 \\ x &= s_3 - a + 1 \\ s_4 &= -y + 2 \\ \hline f &= -s_3 + a - y - 1 \\ g &= a \end{aligned}$$

This gives us a basic feasible solution $(x, y, s_1, s_2, s_3, s_4) = (1, 0, \frac{5}{2}, 3, 0, 2)$ to the original problem. □

Once we have a basic feasible solution, we move to Phase II of the simplex algorithm. In Phase II, we progressively move between adjacent basic feasible solutions by *pivoting* – replacing variables in the basis with others that can give an improved solution.

Example 2.3. *Continuing the previous example, the simplex tableau generated for the current solution (after removing all occurrences of a) is as follows:*

$$\begin{aligned} s_1 &= -y - \frac{1}{2}s_3 + \frac{5}{2} \\ s_2 &= -\frac{2}{3}y - s_3 + 3 \\ x &= s_3 + 1 \\ s_4 &= -y + 2 \\ \hline f &= -s_3 - y - 1 \end{aligned}$$

We need to select a variable to move into the basis. Reducing either s_3 or y will improve the objective value, as they have negative coefficients in the equation for f. Once we decide to swap s_3 into the basis, we must determine which variable to swap out; we must pick the equation

$$v = -cs_3 + \ldots + k$$

with the minimum value of $\frac{k}{c}$ – otherwise the resulting tableau will be infeasible. In this case, we add s_3 to the basis by removing s_2, then eliminate s_3 from all the other equations.

$$\begin{aligned} s_1 &= -\frac{2}{3}y + \frac{1}{2}s_2 + 1 \\ s_3 &= -\frac{2}{3}y - s_2 + 3 \\ x &= -\frac{2}{3}y - s_2 + 4 \\ s_4 &= -y + 2 \\ \hline f &= -\frac{1}{3}y + s_2 - 4 \end{aligned}$$

The only variable with a negative coefficient in the objective is now y, which can replace s_1 in the basis.

$$\begin{aligned} y &= -\frac{3}{2}s_1 + \frac{3}{4}s_2 + \frac{3}{2} \\ s_3 &= s_1 - s_2 - \frac{1}{2}s_2 + 2 \\ x &= s_1 - s_2 - \frac{1}{2}s_2 + 3 \\ s_4 &= \frac{3}{2}s_1 - \frac{3}{4}s_2 + \frac{1}{2} \\ \hline f &= \frac{1}{2}s_1 + \frac{3}{4}s_2 - \frac{9}{2} \end{aligned}$$

This solution cannot be improved, as no variables have a negative coefficient in the objective. This gives an optimal solution $(x, y) = (3, \frac{3}{2})$, with $f(x, y) = \frac{9}{2}$. □

Often it is useful to know how much impact a given constraint has on the optimal value. The *Lagrange multiplier* λ_j for a given constraint indicates how much the objective value would improve if the j^{th} constraint in the problem were to be relaxed. At an optimal solution, the Lagrange multiplier λ_j is given by the coefficient of the slack variable s_j in the objective row.

Example 2.4. *Consider the final tableau given in Example 2.3. The coefficient of s_1 in the objective row (and hence the value of λ_1) is $\frac{1}{2}$. If the constraint $\frac{1}{2}x + y \leq 3$ is relaxed by ϵ, the objective value will be improved by $\frac{1}{2}\epsilon$. Conversely, the coefficient of s_3 is 0 – relaxing $x \geq 1$ will not result in any improvement of the objective function.* □

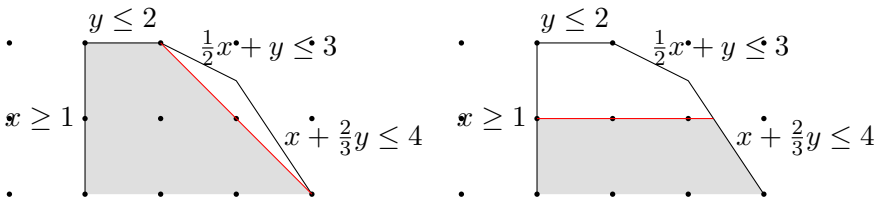


Figure 2.2: Resulting problem if the solver adds a cutting plane $x + y \leq 2$ (left) or branches on $y \leq 1$ (right). In the latter case, the solver will later need to search $y \geq 2$.

2.1.1 Integer Programming

Mixed integer programming (MIP) is a superset of linear programs, where some or all of the variables x_i may be required to take integral values. In general, solving integer programs is NP-hard.

Most MIP solvers, such as CPLEX[1] and GUROBI[2] are based on branch-and-cut techniques (described in detail in Aardal et al. [2005]). These solvers operate by solving the problem without integrality constraints (the *linear relaxation* of the problem), which provides a lower bound for the objective function. Solvers may also add *cutting-planes* [Gomory, 1958], additional constraints which remove regions not containing any integral solutions. If the feasible region contains no integer solutions, or the lower bound is worse than the best solution found so far, the solver can terminate the current branch. If the optimal solution is integral, the solver returns the current solution. Otherwise, the solver must pick a *branch*, adding a new constraint to split the feasible region, and recursively perform the same procedure until an optimal solution is found.

Example 2.5. *Consider the example used in Example 2.1, but with the added restriction that $x, y \in \mathbb{Z}$. The optimal solution for the linear relaxation is $(x, y) = (3, \frac{3}{2})$.*

A potential cutting plane is at $x + y \leq 2$, as it only removes fractional solutions – the resulting feasible region is shown on the left of Figure 2.2. If this constraint is added, then the solution to the new relaxation is integral, and therefore must be the optimum. If a cutting plane is not added, the solver must select a branch to reduce the search space. The feasible region resulting from branching on $y \leq 1$ is shown on the right of Figure 2.2.

□

[1] http://www.ibm.com/software/integration/optimization/cplex-optimizer/
[2] http://www.gurobi.com

2.2 Binary and Multi-valued Decision Diagrams

Reduced Ordered Binary Decision Diagrams [Bryant, 1986a] are a well-known method for representing Boolean functions on Boolean variables using directed acyclic graphs with a single root. Every internal node $n(v, f, t)$ in a BDD r is labelled with a Boolean variable $v \in \mathcal{B}$, and has two outgoing arcs — the 'false' arc (to BDD f) and the 'true' arc (to BDD t). Leaf nodes are either $\mathcal{F}$ (false) or $\mathcal{T}$ (true). Each node represents a single test of the labelled variable; when traversing the tree the appropriate arc is followed depending on the value of the variable. A node $n(v, f, t)$ can be interpreted as representing the constraint $\ll n \gg$, where

$$\ll n \gg \quad \equiv \quad (v \wedge \ll t \gg) \vee (\neg v \wedge \ll f \gg)$$

Reduced Ordered Binary Decision Diagrams (BDDs) [Bryant, 1986b] require that the BDD is: *reduced*, that is it contains no identical nodes (nodes with the same variable label and identical then and else arcs) and has no redundant tests (no node has both then and else arcs leading to the same node); and *ordered* (with respect to a linear ordering $\prec$), if there is an arc from a node labelled v_1 to a node labelled v_2 then $v_1 \prec v_2$. A *long edge* is an arc from a node labelled v_1 to a node labelled v_2 such that, for some v', $v_1 \prec v' \prec v_2$.

The key property of BDDs, for the purposes of this thesis, is that a BDD can be efficiently constructed from a sequence of Boolean operations. This allows us to give a declarative specification of a Boolean function, and have an equivalent BDD automatically constructed. Pseudo-code for applying a Boolean operator to two BDDs is given in Figure 2.3. As will be shown in Sections 2.3.3 through 2.3.6, this allows us to easily express a variety of complex global constraints. While the constructed BDDs are not guaranteed to be polynomial in the number of variables, many of the constraints we shall consider have concise BDD representations.

As with BDDs, Multi-valued Decision Diagrams (MDDs) [Srinivasan et al., 1990] are directed acyclic graphs representing functions. However, MDDs instead represent Boolean functions over variables with arbitrary finite domains. Each internal node in an MDD G, $n_0 = node(x, [(v_1, n_1), (v_2, n_2), \cdots, (v_k, n_k)])$ is labeled with a variable x and outgoing arcs consisting of a value v_i and a destination node n_i. Define $node.var$ as the variable label appearing in the node, i.e. $n_o.var = x$. Each value

```
bdd_apply(op, x = n(v_x, f_x, t_x), y = n(v_y, f_y, t_y))
    if(terminal(x) ∧ terminal(y))
        return op(x, y)
    if(v_x < v_y)
        v := v_x
        f := bdd_apply(op, f_x, y)
        t := bdd_apply(op, t_x, y)
    else if(v_x > v_y)
        v := v_y
        f := bdd_apply(op, x, f_y)
        t := bdd_apply(op, x, t_y)
    else
        v := v_x
        f := bdd_apply(op, f_x, f_y)
        t := bdd_apply(op, t_x, t_y)
    if(f == t)
        return f
    else
        return n(v, f, t)
```

Figure 2.3: Simple pseudo-code for applying a Boolean operator to a pair of BDDs. An efficient implementation will also cache recent calls to bdd_apply to avoid repeatedly constructing the same subgraph.

v_i is in the initial domain of x. There is a final node $\mathcal{T}$ which represents $true$ (the $false$ terminal is customarily omitted for MDDs). Let $G.root$ be the root node of an MDD G. We can understand an MDD node G where $n_0 = G.root$ as representing the constraint $\ll n_0 \gg$ where

$$\ll n_0 \gg \equiv \bigvee_{i=1}^{k} ((x = v_i) \wedge \ll n_i \gg)$$

and $\ll \mathcal{T} \gg \equiv true$. We denote by $|G|$ the number of edges in MDD G. Algorithms for constructing MDDs can be defined analogously to those for BDDs.

We assume that MDDs are *ordered* and without *long edges*, that is there is mapping σ from variables in the MDD to distinct integers such that for each internal node n_0 of the form above $\sigma(n_i.var) = \sigma(n_0.var) + 1, \forall 1 \leq i \leq k$ where $n_i \neq \mathcal{T}$. The condition can be loosened to $\sigma(n_i.var) > \sigma(n_0.var)$ (which allows long edges) but this complicates the algorithms considerably, as a single edge no longer corresponds to a single (var, val) pair – processing an edge then requires checking the destination node, and updating information for all skipped variables. In practice this complication usually overcomes any benefits of treating long edges directly

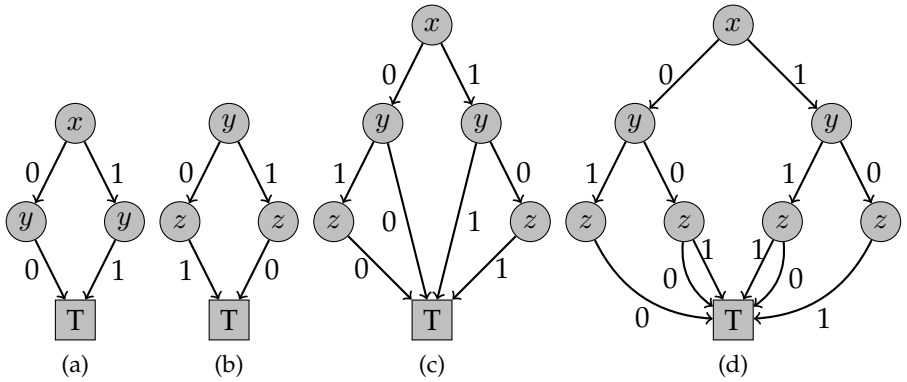


Figure 2.4: BDDs for (a) $x \Leftrightarrow y$, (b) $y \oplus z$, and for $(x \Leftrightarrow y) \vee (y \oplus z)$ both (c) with, and (d) without long edges.

(unlike the case for BDDs). The i^{th} level of an MDD is the set of nodes corresponding to the i^{th} variable (and the outgoing edges from those nodes).

For convenience, we will refer to an edge e as a tuple (x, v_i, s, d) of a variable x, value v_i, source node $s = n_0$ and destination node $d = n_i$. We will refer to the components as $(e.var, e.val, e.begin, e.end)$. An edge $e = (x, v_i, s, d)$ is said to be *alive* if it occurs on some path from the root of the graph to the terminal $\mathcal{T}$. Otherwise, it is said to be *killed*. An edge e becomes killed if v_i is removed from the domain of x, all paths from the root r to s cross killed edges (*killed from above*), or all paths from d to $\mathcal{T}$ cross killed edges (*killed from below*).

We use $s.out\_edges$ to refer to all the edges of the form $(\_, \_, s, \_)$, those leaving node s, and $d.in\_edges$ to refer to edges of the form $(\_, \_, \_, d)$ those entering node d. Similar to above, a node is said to be *killed* if it does not occur on any reachable path from the root node r to $\mathcal{T}$. A node becomes killed if either all incoming or all outgoing edges become killed. As a result, we can determine if a given node n is killed by examining its incoming or outgoing edges. We use $G.edges(x, v_i)$ to record the set of edges of the form $(x, v_i, \_, \_)$ in MDD G.

Example 2.6. *Consider the construction of a BDD for $(x \Leftrightarrow y) \vee (y \oplus z)$. First, we construct BDDs for $(x \Leftrightarrow y)$ and $(y \oplus z)$, shown in Figure 2.4 (a) and (b) respectively. We then use* bdd_apply *to compute the disjunction of the two BDDs. This result is part (c) of Figure 2.4. The presence of long edges (edges that skip variables) adds substantial complexity to a variety of algorithms – Figure 2.4 (c) shows the same function with additional nodes introduced to eliminate long edges.* □

2.3 Finite-domain constraint satisfaction problems

Consider a set of variables $X = \{x_1, \ldots, x_n\}$. Let $\mathbb{D}(x_i)$ denote the *domain* of x_i, the set of values that x_i may take. An assignment θ is a mapping from each variable $x_i \in X$ to a value $v \in \mathbb{D}(x_i)$. Let $\mathbb{D}_X = \mathbb{D}(x_1) \times \ldots \times \mathbb{D}(x_n)$ be the set of possible assignments for the variables in X. A constraint c is a function $\mathbb{D}_{X'} \rightarrow \{true, false\}$ which restricts the allowed values for some subset X' of variables in X.

The objective of a *constraint satisfaction problem* (CSP) is, given a set of variables X and a set of constraints C, to find an assignment $\theta \in \mathbb{D}_X$ such that each constraint $c \in C$ is satisfied – that is, $c(\theta) = true$. It is not normally feasible to directly construct a satisfying assignment for an arbitrary set of constraints – CSPs in general are NP-hard. Similarly, it is usually not practical to directly represent the set of possible assignments to X permitted by the constraints even when the variable domains in X are finite, since there are $2^{|\mathbb{D}_X|}$ possible sets. Indeed, sometimes it is too expensive to maintain an exact representation of the domain of a variable. Instead, constraint solvers keep track of (an approximation of) the possible values for each variable independently. Let $\mathcal{D}_X$ denote these stored variable domains. For convenience, we introduce a partial ordering $\sqsubseteq$ over domains such that $\mathcal{D}'_X \sqsubseteq \mathcal{D}_X \Leftrightarrow \forall x \in X \ \mathcal{D}'_X(x) \subseteq \mathcal{D}_X(x)$. Often it is useful to reason about lower and upper bounds for a variable x. We use $\mathsf{lb}(x)$ to refer to the smallest value in the domain of x, and $\mathsf{ub}(x)$ for the largest value.

In this thesis we will consider primarily finite-domain CSPs, where each variable must take a value from a discrete set of possibilities. The simplest approach to solving a finite-domain CSP is to enumerate the set of possible assignments, and test each one to determine if it satisfies the constraints. This process is typically performed using *backtracking search*. During backtracking search, the solver maintains a current partial assignment, and progressively extends this assignment by selecting a variable x and removing a set of values V from its domain. If the current partial solution cannot be extended to a complete solution, the solver *backtracks* to the previous partial solution, and tries again with the domain of x set to V (the values that where previously removed). However, since $|\mathbb{D}_X|$ is exponential in the number of variables, this is only viable for very small or extremely under-constrained problems.

In order to avoid exploring all possible solutions, constraint solvers apply *propagation* to eliminate values that cannot be extended to a complete solution given the current domains of other variables. A propagator f is a function over variable domains ($\mathcal{D}_X \rightarrow \mathcal{D}_X$) which is *monotonically decreasing* $f(D) \sqsubseteq f(D')$ whenever $D \sqsubseteq D'$, and *contracting* $f(D) \sqsubseteq D$. Propagators remove values from variable domains that are inconsistent with respect to a specific constraint. f is said to be *idempotent* if $f(\mathcal{D}) = f(f(\mathcal{D}))$ for any domain $\mathcal{D}$ – that is, f immediately reaches a fixed point with respect to itself. During propagation, which occurs immediately after the solver has restricted the current partial solution, the solver repeatedly applies propagators until the variable domains reach a fixed point – that is, no propagators can remove additional values from the domains. Once propagation has reached a fixed point, backtracking search continues as before.

A domain $\mathcal{D}_X$ is said to be *domain consistent* with respect to a constraint c if for every variable x_i and value v_i, $v_i \in \mathcal{D}_X(x_i)$ if and only if there is some allowed assignment, with x_i assigned to v_i, that satisfies c. Formally, this requirement is:

$$\forall_{x_i \in X, v_i \in \mathcal{D}(x_i)} \exists_{\theta \in \mathcal{D}_X} \theta(x_i) = v_i \wedge c(\theta) = true$$

A domain $\mathcal{D}_X$ is said to be domain consistent if it is consistent with respect to all constraints $c \in C$. For problems with large domains, it is sometimes impractical to enforce domain consistency. In these cases weaker forms of consistency, such as bounds consistency [Lhomme, 1993] may be used. A domain $\mathcal{D}_X$ is said to be *bounds consistent* with respect for a constraint c if and only if:

$$\forall_{x_i \in X, [\check{v}_i, \hat{v}_i] = \mathcal{D}(x_i)} (\exists_{\theta \in \mathcal{D}_X} \theta(x_i) = \check{v}_i \wedge c(\theta) = true) \wedge (\exists_{\theta \in \mathcal{D}_X} \theta(x_i) = \hat{v}_i \wedge c(\theta) = true)$$

Example 2.7. *Consider the variables x, y with domains $\mathcal{D}(x) = \mathcal{D}(y) = \{1, 2, 3\}$, and constraint $x < y$. As there is no value $v_y \in \mathcal{D}(y)$ such that $3 < v_y$, we can remove 3 from $\mathcal{D}(x)$. Similarly, 1 can be removed from $\mathcal{D}(y)$, giving $\mathcal{D}(x) = \{1, 2\}, \mathcal{D}(y) = \{2, 3\}$.* □

Example 2.8. *Consider variables x, y, z with domains $\mathcal{D}(x) = \{1, 2, 3\}, \mathcal{D}(y) = \mathcal{D}(z) = \{2, 3\}$, and constraints $x \neq y$, $y \neq z$ and $x \neq z$. When propagating $x \neq y$, we find that for $x = 2$, $y = 3$ satisfies the constraint (and similarly for $x = 3$). We find the same when propagating $x \neq z$. As we propagate each constraint independently, we cannot prune any values from the domain of x.*

With a global propagator enforcing domain consistency over `all_different`(x, y, z)*, however, we discover that if* $y = 2$*, then* $z = 3$ *and* x *must be* 1*; similarly, if* $y = 3$*, then* $z = 2$ *and* x *again must be* 1*. Therefore, we can determine* $\mathcal{D}(x) = \{1\}$*.* □

2.3.1 Propagation engine

The propagation engine is responsible for applying propagators until the variable domains reach a fixed point. The propagation engine maintains a work-list of propagators which need to be executed, and a list of propagators in which each variable occurs.

Upon each iteration, the solver selects a propagator from the work-list to be executed, removes it from the work-list, and applies it to the current variable domains – if the propagator is idempotent or makes no change, it is then removed from the work-list. If the propagator detects a conflict – that is, a variable has an empty domain, the propagation terminates.

If all the variable domains are unchanged, the solver iterates again, picking another propagator. If the domain for a variable v is changed, the engine scans the list of propagators for v, and adds each propagator (excluding the current propagator) to the work-list. This process then repeats until there are no more propagators in the work-list, at which point all the propagators have reached a fixed point.

At this point the solver cannot directly make any further inferences; it must then create a restricted subproblem by selecting a variable and removing one or more values from its domain. This creates a new *decision level* – if this restricted problem turns out to be infeasible, either because propagation detected a conflict or because further search proved there were no solutions, the solver will backtrack to this point and continue with only the reduced domains. This interleaved sequence of search and propagation is repeated until either all variables become fixed (in which case we have a satisfying assignment), or a contradiction is found at the top level (and the problem is unsatisfiable).

There are typically two approaches used to facilitate this restoration of solver state. The simplest method, copying, duplicates the variable domains for each decision level. This means that backtracking simply requires changing a pointer to the relevant copy; however, using copying for problem instances with many variables and large domains can consume prohibitive amounts of memory. The

$\mathcal{D}$	WORK-LIST
$\mathcal{D}(X) = \{1, 2, 3\}$ $\mathcal{D}(Y) = \{2, 3\}$ $\mathcal{D}(Z) = \{2, 3\}$	
DECISION: $X = 2$	
$\mathcal{D}(X) = \{2\}$ $\mathcal{D}(Y) = \{2, 3\}$ $\mathcal{D}(Z) = \{2, 3\}$	$X \neq Y$ $X \neq Z$
PROPAGATE: $X \neq Y$	
$\mathcal{D}(X) = \{2\}$ $\mathcal{D}(Y) = \{3\}$ $\mathcal{D}(Z) = \{2, 3\}$	$X \neq Z$ $Y \neq Z$
PROPAGATE: $X \neq Z$	
$\mathcal{D}(X) = \{2\}$ $\mathcal{D}(Y) = \{3\}$ $\mathcal{D}(Z) = \{3\}$	$Y \neq Z$
PROPAGATE: $Y \neq Z$	
CONFLICT!	

Figure 2.5: A possible sequence of propagator executions for the problem in Example 2.9, if the solver branches on $X = 2$. Updated domains, and the corresponding queued propagators, are shown in purple.

other approach is known as *trailing*; trailing solvers record the sequence of changes made during propagation; upon backtracking, the solver walks backwards along the sequence of changes, inverting each until the previous state is restored. If there are relatively few changes at each decision level, this approach can require substantially less memory and computation; however, it introduces a slight overhead on every propagation, as the solver must record enough information to revert the change.

Example 2.9. *Consider again the problem described in Example 2.8, with primitive inequalities. Assume the solver decides to branch on $X = 2$. First the domain of X is updated, then the propagators involving X are added to the work-list. We then select a propagator from the work-list to execute – in this case, $X \neq Y$. Executing this propagator removes 2 from the domain of Y; we must then add propagators involving Y to the work-list. However, we do not need to add $X \neq Y$ back onto the work-list, as the propagator is idempotent. The resulting sequence of actions and updates is shown in Figure 2.5. Once the conflict is reached, the solver will backtrack to the decision, and instead remove 2 from the domain of X.* □

We now describe several example propagators.

2.3.2 Propagation of constraints represented as BDDs and MDDs

As shall be illustrated in the remainder of this section (and in Chapter 3), a variety of constraints can be conveniently represented as MDDs or BDDs. However, to be useful in a finite-domain constraint solver, we must also provide an algorithm for propagation over the constraint. Domain consistent propagation of a constraint represented as a BDD [Gange et al., 2008] or MDD [Pesant, 2004] is reasonably straightforward. The graph is traversed from the root node, marking each reached node (so that it is not revisited) with whether or not the node still has a path to $\mathcal{T}$ given the current domains of variables. Any edge (x, v_i, s, d) on such a path gives support for the value v_i for x. Any values in the current domain of x that are not supported after the traversal is finished are removed. This algorithm is given in Figure 2.6.

Cheng and Yap [2008] made this process more incremental by recording nodes in the graph that were previously determined not to reach $\mathcal{T}$, and sped up the search by recording for which variables all values in the current domain are still supported.

An alternative is to decompose the MDD, introducing state variables for each level and implementing the transition relation with primitive constraints [Beldiceanu et al., 2004]. In this case, the overall constraint maintains domain consistency if and only if the transition constraints are domain consistent.

Example 2.10. *Propagation of the MDD shown in Figure 2.7(b), after $x_2 \neq 1$ and $x_3 \neq 1$, traverses the MDD from the root visiting all nodes except $\{5, 6, 14, 16, 17\}$. The shown arcs of Figure 2.7(c) are traversed by the propagation algorithm; the doubled arcs are determined to be on paths from the root to $\mathcal{T}$ and hence support values. There is no support found for $x_0 = 0$, $x_1 = 0$, or $x_5 = 0$ so their negations are new inferences made by the propagation algorithm.* □

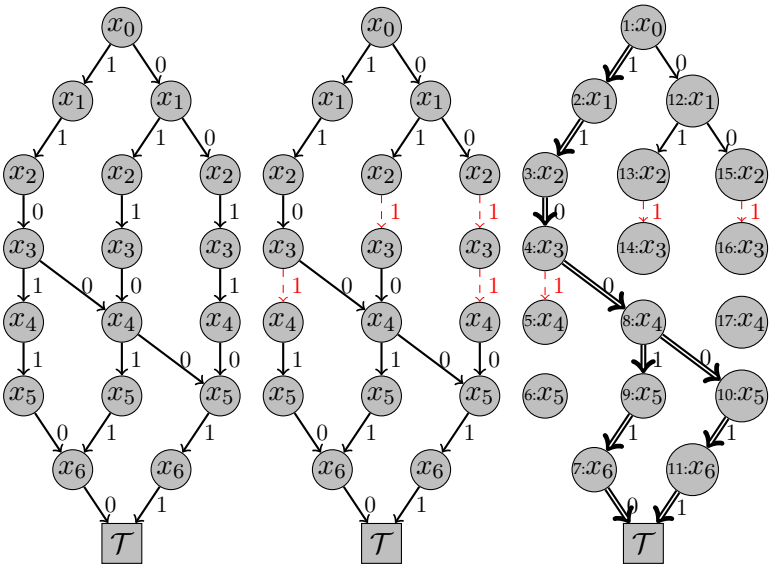
2.3.3 `regular`

A *deterministic finite-state automaton* (DFA) is a model of computation for determining if a given input sequence matches the desired pattern. Formally, a DFA consists of a 5-tuple $D = (Q, q_0, \Sigma, \delta, F)$. The set Q defines the possible states of the automaton, and q_0 gives the initial state. Σ is the alphabet, defining the set of possible

```
% G is the constraint MDD
% D is the current domains
propagate_mdd(G,D)
    unsupp := {}
    clear_cache()
    % Mark all available values as unsupported
    for(var ∈ D)
        for(val ∈ D(var))
            unsupp ∪:= {(var, val)}
    if(¬propagate_rec(D, unsupp, G.root))
        return FAIL
    return unsupp

propagate_rec(D, unsupp, node)
    % If the node has already been processed,
    % use the cached value.
    c := lookup(node)
    if(c ≠ NOTFOUND) return c
    if(node == 𝒯) return true
    c := false
    for((val, n') ∈ node.out_edges)
        if(val ∈ D(node.var))
            if(propagate_rec(D, unsupp, n') ≠ false)
                % Found a support for the current value.
                unsupp \:= (node.var, val)
                c := true
    cache(node, c)
    return c
```

Figure 2.6: Basic algorithm for propagating MDD constraints. unsupp holds the set of (var, val) pairs that haven't yet occurred on a path to $\mathcal{T}$. The algorithm traverses the constraint depth-first, and (var, val) pairs as supported once they occur on a path to $\mathcal{T}$. The operation cache$(key, value)$ is used to store $(key, value)$ pairs in a global table. lookup(key) returns $value$ if there is a corresponding entry in the table, and NOTFOUND otherwise. clear_cache removes all entries from the table.



(a) Constraint (b) $x_2 \neq 1, x_3 \neq 1$ (c) Propagation

Figure 2.7: An example MDD for a `regular` constraint $0^*1100^*110^*$ over the variables $[x_0, x_1, x_2, x_3, x_4, x_5, x_6]$, and the effect of propagating $x_2 \neq 1$ and $x_3 \neq 1$. Edges traversed by the propagation algorithm are shown in (c) – doubled edges are on a path to $\mathcal{T}$.

input values. δ is a function $(Q \times \Sigma) \to Q$ which defines how the automaton state is updated given a new input. If the final state of the automaton is an element of the accept states F, then the input was in the language defined by D.

A *non-deterministic finite-state automaton* (NFA) has the same structure as a DFA; however, the transition function δ is replaced with a relation on $(Q \times \Sigma \cup \{\epsilon\} \times Q)$. This permits a state to have either zero or multiple transitions on an input, as well as ϵ-transitions which consume no input. An NFA will accept on a sequence of inputs if there is *any* corresponding path through the NFA which ends in an accept state.

A `regular` constraint takes a DFA D and a sequence of variables $[x_1, \ldots, x_k]$, and requires that the values of $[x_1, \ldots, x_k]$ must be in the language defined by D. This can be defined recursively as follows:

$$\texttt{regular}(D = (Q, q_0, \Sigma, \delta, F), [x_1, \ldots, x_k]) = \texttt{regular}_D(q_0, [x_1, \ldots, x_k])$$

$$\texttt{regular}_D(q, [\,]) = \begin{cases} \mathit{true} & \textbf{if } q \in F_D \\ \mathit{false} & \textbf{otherwise} \end{cases}$$

$$\texttt{regular}_D(q, [x_i, \ldots, x_k]) = \bigvee_{(q,v) \in \delta_D} [\![x_i = v]\!] \wedge \texttt{regular}(\delta(q, v), [x_{i+1}, \ldots, x_k])$$

In the worst case, this constructs an MDD with $O(k|G|)$ nodes.

This construction, the propagator of Pesant [2004] and the decomposition described by Beldiceanu et al. [2004] all work without modification for NFAs without ϵ-transitions. An arbitrary NFA can be transformed to remove ϵ-transitions by computing the ϵ-closure of each state (see, e.g., Sipser [2006] for details of this transformation).

2.3.4 `table`

Sometimes, particularly in configuration and sequencing problems, it is useful to specify a constraint *extensionally* – that is, specify the constraint by giving an exhaustive list of permitted combinations (or, in the case of negative extensional constraints, a list of disallowed combinations).

A positive table can be formulated as follows:

$$\texttt{table}([\,], [x_1, \ldots, x_k]) = \mathit{false}$$

$$\texttt{table}([V_i = (v_{i1}, \ldots, v_{ik}), \ldots, V_n], [x_1, \ldots, x_k]) = \begin{array}{ll} & (\llbracket x_1 = v_{i1} \rrbracket \wedge \ldots \wedge \llbracket x_k = v_{ik} \rrbracket) \\ \vee & \texttt{table}([V_{i+1}, \ldots, V_n], [x_1, \ldots, x_k]) \end{array}$$

where each $V_i = (v_{i1}, \ldots, v_{ik})$ is a row in the table. This constructs an MDD with at most $O(nk)$ nodes. A negative table can be obtained simply by negating this MDD; this also has $O(nk)$ nodes, but in the worst case may introduce an additional $O(|\,\mathbb{D}(x_i)|)$ edges per node (as previously omitted paths to false must be introduced).

A number of propagation algorithms have been developed for extensional tables, either using the list of rows directly [Lecoutre and Szymanek, 2006], or augmenting this with some form of acceleration structure [Lhomme and Régin, 2005, Lecoutre, 2008].

2.3.5 Global cardinality (gcc)

A *cardinality* constraint $\texttt{card}([x_1, \ldots, x_k], v, l, h)$ requires that between l and h variables in $\{x_1, \ldots, x_k\}$ take the value v. This can be formulated as follows:

$$\texttt{card}([x_1, \ldots, x_k], v, l, h) = \texttt{card}([x_1, \ldots, x_k], v, l, h, 0)$$

$$\texttt{card}([\,], v, l, h, c) = \begin{cases} \mathit{true} & \textbf{if } l \leq c \wedge c \leq h \\ \mathit{false} & \textbf{otherwise} \end{cases}$$

$$\begin{aligned} \texttt{card}([x_i, \ldots, x_k], v, l, h, c) = \quad & ([\![x_i \neq v]\!] \wedge \texttt{card}([x_{i+1}, \ldots, x_k], v, l, h, c) \\ \vee \quad & ([\![x_i = v]\!] \wedge \texttt{card}([x_{i+1}, \ldots, x_k], v, l, h, c+1) \end{aligned}$$

The *global cardinality* constraint $\texttt{gcc}([x_1, \ldots, x_k], [(v_1, l_1, h_1), \ldots, (v_m, l_m h_m)])$ restricts the number of occurrences of a set of values $\{v_1, \ldots, v_m\}$ to be within the given bounds. This can be encoded as a conjunction of `card` constraints:

$$\texttt{gcc}(X, V) = \bigwedge_{(v_i, l_i, h_i) \in V} \texttt{card}(X, v_i, l_i, h_i)$$

Unfortunately, this direct implementation can propagate quite weakly, and building the `gcc` constraint into an MDD produces an exponential number of nodes – for small numbers of variables, the MDD approach can be feasible. A propagation algorithm based on flow networks [Régin, 1996] enforces domain consistency in $\mathrm{O}(|X|^2|V|)$ time.

2.3.6 Context-free grammar (grammar)

Context-free grammars (CFGs), like DFAs, allow us to specify a desired set of permitted sequences. A CFG consists of a tuple $C = (V, \Sigma, R, S)$, for a set of non-terminal symbols V, alphabet Σ, production rules R and start symbol S. Each rule in R describes a transformation wherein a non-terminal T is replaced with either a series of non-terminal or terminal (alphabet) symbols, or the empty string ϵ. A string in the language is constructed by starting with S and sequentially applying rewrite rules until only terminal symbols remain.

Example 2.11. *The language* $\{0^i 1^j \mid i \geq j\}$ *(where* a^k *denotes the symbol* a *repeated* k *times) can be recognized by the CFG:*

$$S \rightarrow 0S1 \mid T$$
$$T \rightarrow 0T \mid \epsilon$$

with non-terminals $\{S, T\}$ *and start symbol* S*.*

We can then generate any string in the language by starting with S *and applying some sequence of production rules:*

RULE	STRING
—	S
$S \rightarrow 0S1$	$0S1$
$S \rightarrow 0S1$	$00S11$
$S \rightarrow T$	$00T11$
$T \rightarrow 0T$	$000T11$
$T \rightarrow 0T$	$0000T11$
$T \rightarrow \epsilon$	$\mathbf{000011}$

□

As for `regular`, the constraint `grammar`(G,$[x_1, \ldots, x_k]$) requires that the values assigned to $[x_1, \ldots, x_k]$ form a string in the language recognized by the CFG G. A dedicated propagator [Sellmann, 2006, Quimper and Walsh, 2006] and a decomposition [Quimper and Walsh, 2007] have been presented for enforcing `grammar` constraints, both based on the structure of the CYK parsing algorithm [Younger, 1967].

2.4 Boolean Satisfiability (SAT)

Boolean Satisfiability (SAT) is a well-studied restricted class of CSP. The problem variables must be Boolean. A *literal* is either a variable v_i or its negation $\neg v_i$. A SAT problem consists of a set $\mathcal{B}$ of Boolean variables together with a set of *clauses* of the form $\bigvee_i l_i$, where each l_i is a literal from $\mathcal{B}$. A solution is an assignment to all variables in $\mathcal{B}$ such that at least one literal in each clause is true.

Example 2.12. *Consider a SAT problem with variables $\mathcal{B} = \{x, y, z\}$, and clauses*

$$(x \vee y) \wedge (\neg y \vee \neg z) \wedge (z \vee \neg x)$$

A satisfying assignment to this problem is $\theta(x, y, z) = \{true, false, true\}$. □

Example 2.13. *Consider again the problem given in Example 2.12, but with the additional constraints*

$$(x \vee \neg y) \wedge \neg z$$

This problem is unsatisfiable. Since $z = false$ is asserted, we can only satisfy $(z \vee \neg x)$ by fixing $x = true$. However, $(z \vee \neg x)$ forces $x = false$. Since we cannot have both x and $\neg x$, the set of clauses cannot be satisfied. □

Notice that since every clause must be true, whenever all but one literal in a clause becomes false, the remaining literal must become true. The process of detecting such clauses (referred to as *unit clauses*) and asserting the remaining literal is known as *unit propagation*, and is the foundation of the DPLL (Davis-Putnam-Loveland-Logemann) procedure [Davis et al., 1962], on which all modern complete SAT solvers are based.[3] As with a conventional finite-domain constraint solver, these solvers interleave search (by picking a literal to assert) with unit propagation to find a satisfying assignment. A *decision literal* is a literal that is chosen by the search strategy after unit propagation reaches a fixed point.

Unit propagation can be implemented efficiently by using a two-literal watching scheme [Moskewicz et al., 2001]. Observe that we perform unit propagation exactly when the second last literal in a clause becomes false. Specifically, so long as we know that at least 2 literals in the clause have not become false, changes to the other literals cannot cause unit propagation.

With each literal l in the problem, we associate a list of clauses that may become unit clauses if l becomes true. For each clause c, we pick two literals w_0 and w_1 to be *watched literals* (or watches), and add c to the lists for $\neg w_0$ and $\neg w_1$. Consider the case when $w_0 = false$ is asserted (the case for w_1 is analogous). We first check if w_1 is true; if this is the case, the c is already satisfied, and we don't need to look for a replacement for w_0. Otherwise, we scan c to find any literal w' (other than w_1) that is not yet false. If a literal w' is found, the clause does not propagate, but (w_0, w_1) is

[3] Local search methods are used for stochastic SAT solvers, however tend not to be effective for industrial or structured problems.

no longer a valid pair of watched literals. So we remove c from the list of watched clauses for w_0, and add it to the list for $\neg w'$. If no replacement literal is found, we check if w_1 is false. If so, the current partial solution is inconsistent, so we backtrack to a previous decision level. If w_1 is unfixed, we know w_1 must be true under the current assignment.

Example 2.14. *Consider variables $\mathcal{B} = \{w, x, y, z\}$ and clauses:*

$$c_0 = w \vee x$$

$$c_1 = x \vee \neg y \vee \neg z$$

$$c_2 = \neg w \vee \neg x \vee \neg y$$

If we select the first two literals in each clause as watches, we get the initial watch lists:

w	$\{c_2\}$	$\neg w$	$\{c_0\}$
x	$\{c_2\}$	$\neg x$	$\{c_0, c_1\}$
y	$\{c_1\}$	$\neg y$	
z		$\neg z$	

Assume search first asserts $\neg x$. $\neg x$ is watched by c_0 and c_1. As c_0 is a binary clause, we cannot find a replacement watch, so we propagate w. We then scan c_1 for a replacement watch. z has not yet been given a value, so we pick $\neg z$ as our watch, and move c_1 from the watch list for $\neg x$ to the list for z.

We then examine the watch list for w, which contains only c_2. However, $\neg x$, the second watch for c_2 is already true, so the clause is satisfied – we then don't need to find a new watch for c_2.

This gives the updated watch lists as follows:

w	$\{c_2\}$	$\neg w$	$\{c_0\}$
x	$\{c_2\}$	$\neg x$	$\{c_0\}$
y	$\{c_1\}$	$\neg y$	
z	$\{c_1\}$	$\neg z$	

□

Over the past 15 years, several improvements have been developed which substantially improve the performance of basic DPLL algorithm on a wide range of problems. Conflict analysis [Marques-Silva and Sakallah, 1999] allows the solver

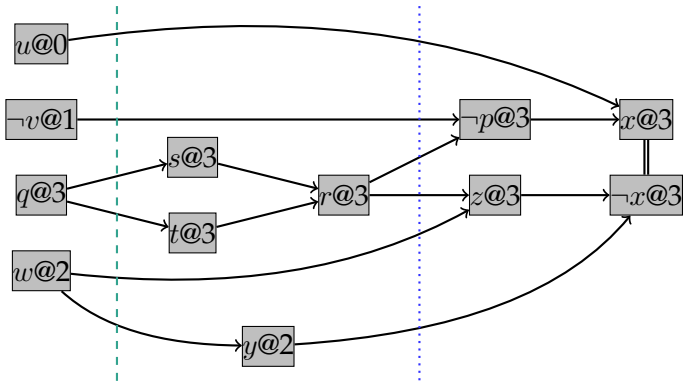


Figure 2.8: Implication graph for Example 2.15. The dashed line indicates the decision cut. The dotted line indicates the 1-UIP cut.

to substantially reduce search space by avoiding similar regions of the search tree, and activity-directed search drives the solver towards a solution by concentrating search on variables that have recently been involved in conflicts.

Conflict analysis in SAT solvers is generally used to construct a *nogood*, a clause that is (a) implied by the clause database, (b) unsatisfiable under the current assignment and (c) contains only one literal at the current decision level. Requirement (a) ensures that the clause won't eliminate a satisfying assignment. Requirements (b) and (c) ensure that, when we backtrack to the previous decision level and add the nogood to the clause database, the current branch will be eliminated.

The simplest conflict clause is the negation of all decision literals. While this is a correct nogood (and sufficient to ensure eventual termination), nogoods constructed in this way tend to be large, and cannot prune additional branches of the search space. These can be improved by including only those decisions that participated in the conflict [Bayardo and Schrag, 1997], however these still propagate relatively infrequently; it is desirable to construct stronger nogoods which will eliminate more of the search space. Most SAT solvers construct nogoods according to the 1-UIP (first unique implication point) scheme [Zhang et al., 2001]. 1-UIP nogoods can be constructed by sequentially resolving the most recent clauses in the implication graph together until there is exactly one literal at the current level remaining. 1-UIP nogoods are not guaranteed to be smaller (and therefore stronger) than the decision nogoods, but appear to perform well in practice.

Example 2.15. *Consider the following sequence of inferences made during SAT solving:*

Level	*Inference*	*Reason*
0	u	—
1	$\neg v$	—
2	w	—
	y	$\neg w \vee y$
3	q	—
	s	$\neg q \vee s$
	t	$\neg q \vee t$
	r	$\neg s \vee \neg t \vee r$
	$\neg p$	$v \vee \neg r \vee \neg p$
	$\mathbf{x}$	$p \vee \neg u \vee x$
	z	$\neg w \vee \neg r \vee z$
	$\neg \mathbf{x}$	$\neg z \vee \neg y \vee \neg x$
	CONFLICT	

This forms the inference graph shown in Figure 2.8. Each node represents a clause that has become unit and propagated; for example, the node marked r@3 shows that the clause $(\neg s \vee \neg t \vee r)$ propagated asserted r at decision level 3 – the inbound edges indicate the set of literals that caused propagation. A conflict was found, as both x and $\neg x$ were inferred at the current decision level, so we must construct a nogood for the conflict.

The decision (or last-UIP) nogood is $(\neg u \vee v \vee \neg q \vee \neg w)$ – this is indicated by the dashed line in Figure 2.8. To construct the 1-UIP nogood, we start with the conflicting clause $C = (\neg z \vee \neg y \vee \neg x)$. We then walk back along the sequence of inferences and progressively resolve the conflict clause with each reason. The most recent inference was z, so we resolve C with $(\neg w \vee \neg r \vee z)$, giving an updated clause $(\neg y \vee \neg x \vee \neg w \vee \neg r)$. We then continue this process until exactly one literal at the current decision level remains:

Inference	*Reason*	*Learnt*
—	—	$\neg z \vee \neg y \vee \neg x$
z	$\neg w \vee \neg r \vee z$	$\neg y \vee \neg x \vee \neg w \vee \neg r$
x	$p \vee \neg u \vee x$	$\neg y \vee \neg w \vee \neg r \vee p \vee u$
$\neg p$	$v \vee \neg r \vee \neg p$	$\neg y \vee \neg w \vee \neg r \vee u \vee v$

At this point, $\neg r$ is the only literal at the current decision level; the current learnt clause $(\neg y \vee \neg w \vee \neg r \vee u \vee v)$ is the 1-UIP nogood. After backtracking to the previous level, adding this clause will then cause $\neg r$ to be propagated.

Notice that both $\neg w$ and $\neg y$ are in the 1-UIP nogood. By testing for subsumed literals, we can determine that y is a consequence of w, and as such can be eliminated from the clause [Sörensson and Biere, 2009]. This gives a resulting nogood of $(\neg u \vee v \vee \neg r \vee \neg w)$. □

The solver generates a nogood for each conflict, which is then added to the clause database. However, maintining a continually increasing set of clauses causes the solver to gradually degrade. In practice, solvers periodically scan the database of learnt clauses to remove any clauses that haven't recently participated in conflict (and aren't currently asserting a literal).

2.4.1 Tseitin Transformation

One of the major challenges in using a SAT solver to solve combinatorial problems is constructing a clausal representation of the problem. A direct conversion of an arbitrary Boolean expression into an equivalent CNF may produce an exponential number of clauses – indeed, some formulae require an exponential number of clauses for the minimal CNF representation.

Example 2.16. *Consider the formula $(x_1 \wedge y_1) \vee \ldots \vee (x_n \wedge y_n)$. A direct conversion of this formula into CNF produces the formula:*

$$\bigwedge_{l_1 \in \{x_1, y_1\}} \bigwedge_{l_2 \in \{x_2, y_2\}} \cdots \bigwedge_{l_n \in \{x_n, y_n\}} (l_1 \vee l_2 \vee \ldots \vee l_n)$$

which has $O(2^n)$ clauses. □

However, we don't need an equivalent CNF formula, merely an *equisatisfiable* formula. The *Tseitin transformation* [Tseitin, 1968] produces an polynomial sized CNF encoding of a formula by introducing intermediate variables to represent the value of each subformula.

Example 2.17. *Consider the formula given in Example 2.16. By introducing an additional variable c_i for each conjunction, we can construct the following formula:*

$$(c_1 \vee c_2 \vee \ldots \vee c_n) \wedge \bigwedge_{i=1}^{n} (c_i \Leftrightarrow x_i \wedge y_i)$$

By directly expanding this formula, we produce the following set of clauses:

$$(c_1 \vee c_2 \vee \ldots \vee c_n) \wedge \bigwedge_{i=1}^{n} (x_i \vee \neg c_i) \wedge (y_i \vee \neg c_i) \wedge (\neg x_i \vee \neg y_i \vee c_i)$$

This introduces an additional n variables, but requires only $O(n)$ clauses, rather than the $O(2^n)$ of the direct encoding. □

There are some disadvantages to applying these transformations. The transformed formula often doesn't maintain the same level of consistency as the original formula – it has weaker propagation. Also, the introduced variables can interfere with the conflict-directed search strategies which most SAT solvers use. Finally, there can be considerable overhead in maintaining the additional variables and clauses in the SAT solver, particularly as most will propagate very rarely. A significant amount of recent research has been into developing CNF transformations of various classes of formulae which are concise, but still maintain domain consistency over the external variables [Jung et al., 2008, Quimper and Walsh, 2007, Abío et al., 2011].

2.4.2 Pseudo-Boolean CSPs

Linear pseudo-Boolean (PB) constraint problems (also known as binary integer programs) can be considered to be either an extension of SAT, or a restricted case of integer programs. Pseudo-Boolean constraint problems have the same basic structure as conventional integer programs; however, all variables are restricted to 0–1 domains. Solvers for this class of problems are generally either SAT-based or integer programming based. In the absence of an objective function, SAT-based solvers may convert these constraints directly into clauses, as in the case of MINISAT+ [Eén and Sörensson, 2006], or extend the propagation and learning algorithms of the solver, such as GALENA [Chai and Kuehlmann, 2003]. If there is an objective function, the solver will generally solve a sequence of SAT problems constructed with increasingly restricted objective values. Integer programming based solvers, such as SCIP [Achterberg et al., 2008], simply treat the problem as an integer program, and use similar techniques to those described in Section 2.1.1 to solve the problem.

Example 2.18. *Consider the cardinality constraint:*

$$\sum_{i=1}^{n} x_i \leq 1 \ , \ x_i \in [0,1]$$

The direct encoding into SAT produces $O(n^2)$ clauses:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg x_i \vee \neg x_j$$

However, we can introduce partial-sum variables $\{p_1, \ldots, p_{n-1}\}$ such that p_i is true if any of $\{x_1, \ldots, x_i\}$ is true. These new intermediate variables allow us to construct a more concise encoding [Silva and Lynce, 2007] requiring only $O(n)$ clauses:

$$(\neg x_1 \vee \neg p_1) \wedge (\neg x_n \vee \neg p_{n-1}) \wedge \bigwedge_{i=2}^{n-1} (\neg x_i \vee p_i) \wedge (\neg p_{i-1} \vee p_i) \wedge (\neg x_i \vee \neg p_{i-1})$$

It is worth noting that this corresponds to the Tseitin encoding of the corresponding BDD. □

For more general pseudo-Boolean constraints, the constraint is commonly transformed into a Boolean circuit, then decomposed into CNF using transformations similar to those described in Section 2.4.1. Common approaches involve representing the constraint using BDDs [Eén and Sörensson, 2006, Abío et al., 2011] or variants of sorting networks [Eén and Sörensson, 2006, Codish and Zazon-Ivry, 2010, Asín et al., 2011].

Example 2.19. *Consider the BDD used in Example 2.20. When directly using the Tseitin transformation as described in Eén and Sörensson [2006], the following clauses are generated for the node marked n_3 in Figure 2.9(a):*

$$x_1 \wedge n_5 \rightarrow n_3$$

$$\neg x_1 \wedge n_4 \rightarrow n_3$$

$$x_1 \wedge \neg n_5 \rightarrow \neg n_3$$

$$\neg x_1 \wedge \neg n_4 \rightarrow \neg n_3$$

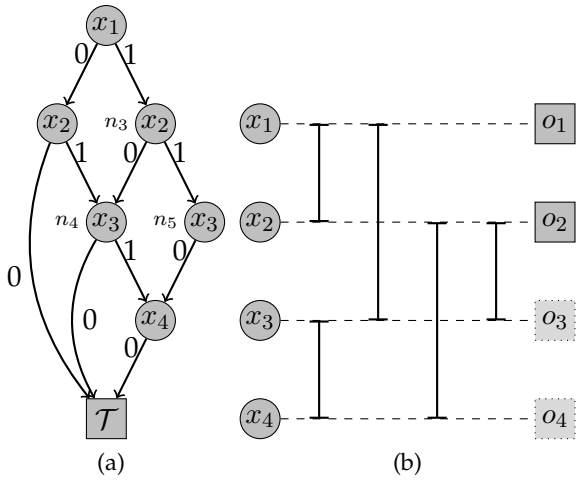


Figure 2.9: The constraint $\sum_{i=1}^{4} x_i \leq 2$ as (a) a BDD, and (b) a sorting network.

However, as observed in Abío et al. [2011], since cardinality constraints are monotone Boolean functions, this set of clauses can be reduced to:

$$\neg n_4 \rightarrow \neg n_3$$

$$x_1 \wedge \neg n_5 \rightarrow \neg n_3$$

□

Sorting networks are Boolean circuits which take an unsorted sequence of inputs, and produce the same values in sorted order as outputs. Sorting networks are generally constructed using *comparators*. A comparator ${}^{x}_{y}\mathrm{I}^{x'}_{y'}$ is a Boolean circuit that takes two inputs x and y, and produces the sorted values x' and y' as its outputs. The comparator can be implemented as

$$\mathsf{comparator}(\ll x, y \gg, \ll x', y' \gg) \equiv (x \vee y \rightarrow x') \wedge (x \wedge y \rightarrow y')$$

Conceptually, sorting networks operate fairly similarly to traditional sorting algorithms. Indeed, most sorting networks used for pseudo-Boolean constraints are based on variants of merge sort; the inputs are divided into two sets (maintaining some precondition), then recursively sorted, and finally merged (using the aforementioned precondition to reduce the number of necessary comparators).

Example 2.20. *Consider the constraint* $\sum_{i=1}^{4} x_i \leq 2$. *Figure 2.9(a) shows this constraint as a BDD (as described in Section 2.2). This BDD can be decomposed by introducing*

additional variables for each internal node, and performing the Tseitin transformation (or related decomposition) as usual.

Figure 2.9(b) gives a 4-input sorting network. Each vertical bar denotes a comparator which takes two Boolean inputs x and y and sorts them. The sorting network uses these comparators to perform a merge sort on the input, producing a unary representation of the number of true inputs. The two leftmost comparators sort their respective pairs of inputs; the remaining three merge the two sorted sequences. The constraint $\sum x_i \leq 2$ can then be enforced by asserting $\neg o_3$ (shown dotted in Fig. 2.9). □

Sorting networks have the convenient property that they can be incrementally restricted – if we have the constraint $\sum x_i \leq k$, we can strengthen the constraint to $\sum x_i \leq k'$ by forcing the $(k'+1)^{th}$ output to be false.

2.4.3 Lazy Clause Generation

As mentioned in Section 2.4, the conflict-learning techniques used by modern SAT solvers often produce substantial reductions in search space. Unfortunately, it is problematic to represent problems with numerical constraints and large integer domains directly in CNF.

One approach, often used in bit-vector solvers such as BOOLECTOR [Brummayer and Biere, 2009] and YICES [Dutertre and De Moura, 2006], is to construct a logarithmic encoding by introducing a Boolean variable for each bit. Integer constraints can then be constructed by building logic circuits similar to those used in normal computation. While this representation is polynomial in the number of bits, this representation propagates very weakly, as inferences only occur once a bit becomes fixed.

Example 2.21. *Let $x = \langle x_2 x_1 x_0 \rangle$, $y = \langle y_2 y_1 y_0 \rangle$ and $z = \langle z_2 z_1 z_0 \rangle$. We can enforce the constraint $z = x + y$ by introducing carry variables $\langle c_2 c_1 c_0 \rangle$ and building an adder:*

$$(c_0 \Leftrightarrow x_0 \wedge y_0) \wedge (z_0 \Leftrightarrow x_0 \oplus y_0) \wedge \neg c_n$$

$$\bigwedge_{i=1}^{n} z_i \Leftrightarrow x_i \oplus y_i \oplus c_{i-1}$$

$$\bigwedge_{i=1}^{n} c_i \Leftrightarrow (x_i \wedge y_i) \vee (x_i \wedge c_{i-1}) \vee (y_i \wedge c_{i-1})$$

□

Example 2.22. *Let $x = <x_3 x_2 x_1 x_0>$, with the constraint $x \geq 3$. We cannot determine the value of any bits, since $x = 3$ has b_1 and b_0 true, but $x = 4$ has b_1 and b_0 false. In fact, bounds cannot produce any inferences until $\mathsf{lb}(x) \geq 8$ or $\mathsf{ub}(x) \leq 7$.* □

The other alternative is to use a unary encoding for the integer variables. In this case, we introduce Boolean variables for each value the variable can take, and add clauses to ensure the ordering is maintained. However, in problems with large domains, many of these literals will likely not be used in solving the problem – they simply serve to introduce overhead into the solver. Also, integer constraints (such as addition and multiplication) tend to be quite expensive to directly encode with this representation.

By integrating SAT-style nogood handling into an FD solver, we can avoid these representation problems while maintaining the advantages of having a SAT encoding. Rather than defining a SAT model initially, the solver uses a standard FD-style engine for propagation, modified to record the *reason* for each inference. It then constructs a clausal representation lazily, only when needed for conflict analysis. Early work integrating conflict analysis and clause learning with FD solvers [Katsirelos and Bacchus, 2003, 2005] used an exhaustive encoding of variable domains, introducing a literal for each value, and enforcing assignments through the FD engine. *Lazy clause generation* [Ohrimenko et al., 2009] solvers extend this by maintaining only a partial representation of variable domains, and introducing literals for representing variable bounds (rather than just values). As such, the solver can benefit from conflict-directed learning without having to maintain a complete SAT model of the problem.

For convenience, we use $[\![x \leq v]\!]$ to indicate the literal representing $x \leq v$. When $\mathsf{lb}(x)$ is set to v, we can assert $\neg\,[\![x \leq v-1]\!]$; when $\mathsf{ub}(x)$ is set to v, we can assert $[\![x \leq v]\!]$. We will often use $[\![x \geq v]\!]$ rather than $\neg\,[\![x \leq v-1]\!]$ when manipulating lower bounds; both notations refer to the same literal. However, bounds literals cannot express an interval with gaps, which is necessary when we want to enforce domain consistency over constraints. In this case, we introduce *equality literals* $[\![x = v]\!]$, which are tied to bounds literals by the addition of the constraint $[\![x = v]\!] \leftrightarrow \neg\,[\![x \leq v-1]\!] \wedge [\![x \leq v]\!]$.

There are a number of variations amongst lazy clause generation solvers in how explanations are generated. When the solver finds a conflict, it begins computing a

conflict clause following the same procedure described in Section 2.4. However, in order to do so, the solver must be able to determine which literals were responsible for each inference. If explanations are generated *eagerly*, as the solver of Ohrimenko et al. [2009], any time an inference is made by a propagator, a corresponding explanation is generated and recorded. In this case, the conflict analysis can proceed precisely as if it were a pure SAT problem. However, often a conflict will only involve a small subset of inferences; if constructing explanations is expensive, it is desirable to avoid generating explanations that will be unused. It is possible to instead use *deferred* explanations [Gent et al., 2010, Gange et al., 2010]. In this case, the solver records a time-stamp with each inference, instead of an explanation – this must give enough information to restore the propagator state to the time the inference was made. During conflict analysis, when the solver attempts to retrieve the reason for an inference, it must first check if a time-stamp is recorded instead. If so, the inferring propagator must construct an explanation which was correct at the specified time; this then replaces the time-stamp, and the conflict analysis continues as usual. An important decision to be made is how to handle the generated explanations. If the explanations are to be used only for conflict analysis, they can simply be discarded once the solver has backtracked beyond the current level – this can be done either eagerly, or by periodic garbage collection (similar to the handling of learnt clauses). However, adding the generated explanations to the clause database allows these explanations to be re-used when the same inference occurs in a different subtree. This can be directly beneficial in cases when explanations are expensive compute; however, since generated explanations for an inference can be wildly different despite occurring in similar subtrees, keeping explanations can also direct the inference graph (and thus, conflict analysis) to focus repeatedly on the same subproblem, hopefully resulting in stronger nogoods. In this case, the explanations (once generated) can be treated exactly as conventional learnt clauses.

Example 2.23. *Consider a problem with variables x, y and z, with $\mathbb{D}(x) = \mathbb{D}(y) = \mathbb{D}(z) = [0, 300]$ and the constraint $z = x + y$. Assume that the following propagations have occurred:*

	$\mathbb{D}(x)$	$\mathbb{D}(y)$	$\mathbb{D}(z)$
	$[0, 300]$	$[0, 300]$	$[0, 300]$
$x \geq 50$	$[50, 300]$	$[0, 250]$	$[50, 300]$
$y \geq 80$	$[50, 220]$	$[80, 250]$	$[130, 300]$

At this point, the following literals have been created:

x	$\{[\![x \leq 49]\!], [\![x \leq 220]\!]\}$
y	$\{[\![y \leq 79]\!], [\![y \leq 250]\!]\}$
z	$\{[\![z \leq 49]\!], [\![z \leq 129]\!]\}$

During conflict analysis, it may be necessary to generate clauses explaining any of the inferences made during propagation.

Generating an explanation for $\neg [\![z \leq 129]\!]$ *will produce*

$$\neg [\![x \leq 49]\!] \wedge \neg [\![y \leq 79]\!] \rightarrow \neg [\![z \leq 129]\!]$$

An explanation for $[\![x \leq 220]\!]$ *would be*

$$\neg [\![y \leq 79]\!] \rightarrow [\![x \leq 220]\!]$$

□

Lazy-clause generation is closely related to *SAT Modulo Theory* (SMT) solvers. SMT solvers combine SAT reasoning with a *theory* solver for reasoning about the non-Boolean parts of the problem. The theory solvers communicate with the Boolean parts of the model through *theory literals*, similarly to the relationship between variable bounds and bound literals in a lazy clause generation solver. A variety of theory solvers have been developed, such as for fixed-width bit-vectors [Brummayer and Biere, 2009], difference logic [Nieuwenhuis and Oliveras, 2005] and linear arithmetic [Dutertre and de Moura, 2006]. As observed by Ohrimenko et al. [2009], lazy clause generation solvers can be seen as a special form of SMT solver where each propagator is a theory solver.

Example 2.24. *Consider again the problem of Example 2.23, but with the added constraint* $z \leq x$*. When we set* $x \geq 50$*, the propagation progresses exactly as in the previous example,*

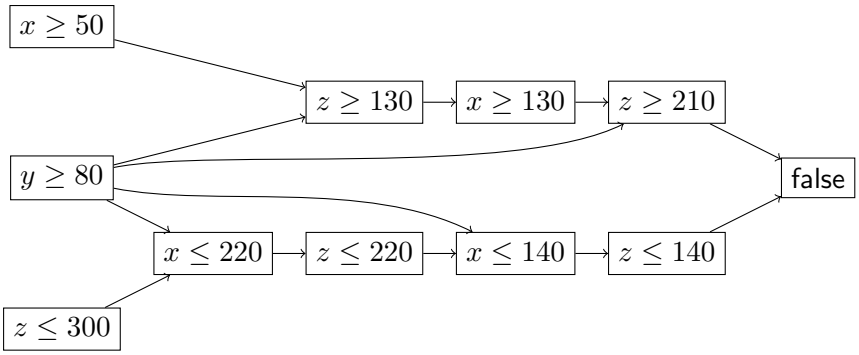


Figure 2.10: Generated inference graph for Example 2.24.

as the propagator for $z \leq x$ fires only when the upper bound of x decreases, or the lower bound of z increases.

As before, asserting $y \geq 80$ forces $z \geq 130$ and $x \leq 220$. The constraint $z \leq x$ then fires, ensuring $x \geq 130$ and $z \leq 220$. Since x and z have changed, we again propagate $z = x + y$, which propagates $x \leq 140$, $y \leq 90$ and $z \geq 210$. When $z \leq x$ propagates again, the propagator attempts to set $z \leq 140$ but fails, as $z \geq 210$.

As before, the solver starts with the conflict clause $[\![z \geq 210]\!] \wedge [\![z \leq 140]\!] \rightarrow \mathit{false}$, and replaces $[\![z \leq 140]\!]$ with its reason, $[\![x \leq 140]\!]$. It then repeats this process, next replacing $[\![x \leq 140]\!]$ with its reason $[\![z \leq 220]\!] \wedge [\![y \geq 80]\!]$, constructing the conflict clause in the same manner as a SAT solver. The inference graph constructed during this process is shown in Figure 2.10. □

The use of nogood learning can provide dramatic improvements to the performance of BDD-based constraint solvers. Several such solvers have been developed, mostly in the context of solving set constraints. The solver of Hawkins and Stuckey [2006] used BDD conjunction and existential quantification to implement propagation and explanation. Subbarayan [2008] introduced a more efficient algorithm for computing explanations, and Gange et al. [2010] incorporated a more efficient propagation algorithm, and took advantage of the conflict-directed search heuristics of modern SAT solvers. The solver of Damiano and Kukula [2003] used very similar techniques, but constructed BDDs from an existing SAT model (with the goal of eliminating variables) rather than preserving the structure of a high-level constraint.

2.5 Dynamic Programming

Dynamic programming [Bellman, 1952] is a technique used to solve optimization problems that can be recursively defined in terms of smaller subproblems. In many cases, the solution to a problem p may require solving subproblems p_α and p_β, both of which depend on subproblem p_γ. In a naive implementation, we will have to solve p_γ (at least) twice – first when solving p_α and again to solve p_β. The key idea of dynamic programming is to store the optimal solution to p_γ so it can be used without being recalculated.

Bottom-up dynamic programming is the simplest form, where the optimal solution is computed for *all* subproblems, ordered such that a problem p is not computed until after all its subproblems. This is convenient, as it requires no special data structures (beyond an array to store the subproblem solutions), but it may compute the solution to subproblems that aren't used in computing the final optimum.

Top-down dynamic programming is slightly more complex than bottom-up, as it requires a hash table or other data structure to record the set of subproblems that have been computed. When solving a problem p, we first check if p is in the table – if so, we return the computed value. Otherwise, we recursively solve the subproblems required by p, compute the optimum, then enter it in the table. If there are many subproblems that don't occur in the dependency tree of p, then top-down can provide a substantial performance improvement over bottom-up. When describing top-down dynamic programming algorithms (such as the MDD propagation algorithm given in Section 2.3.2), we will use the procedures $\mathsf{cache}(key, value)$ and $\mathsf{lookup}(key)$ to insert and find entries in this global table; $\mathsf{clear\_cache}()$ is used to remove all entries from the table (when we are about to solve a new problem).

Dynamic programming problems often involve taking the min or max of a set of possible subproblems. *Bounded* top-down dynamic programming avoids exploring subproblems that cannot result in an improved solution. Assume that we are maximising a dynamic program f. We define a bounding function ub_f such that $ub_f(c) \geq f(c)$ (preferably one that can be computed quickly). ub_f gives a limit on how far the objective value can be improved (if minimizing, we define lb_f similarly). *Local bounding* records the best solution $\hat{f}$ found for the current node, and

avoids expanding any children c where $ub_f(c) \leq \hat{f}$. Local bounding is guaranteed to expand at most as many nodes as the basic dynamic programming algorithm. *Argument bounding* (also called *global bounding*) extends this by keeping track of the best solution found so far along any branch of the search tree, rather than just children of the current node. While it generally reduces the search space, argument bounding can potentially expand *more* nodes than basic top-down dynamic programming, as a node that is cut off initially may be checked repeatedly along different paths with weaker bounds each time.

Example 2.25. *A 0–1 knapsack problem is defined by a set of items $I = \{(w_1, v_1), \ldots, (w_k, v_k)\}$ with weight w_i and value v_i, and a capacity C. The objective is to find the subset of items with maximal value, but can still fit within the knapsack. This can be formulated as follows:*

$$\begin{aligned} \mathsf{knapsack}(I, C) \;=\; & \mathbf{max} \sum_{(w_i,v_i)\in I'} v_i \\ & \mathbf{s.t} \sum_{(w_i,v_i)\in I'} w_i \leq C \\ & I' \subseteq I \end{aligned}$$

When formulated as a dynamic program, this becomes:

$$\begin{aligned} \mathsf{knapsack}(C) \;&=\; \mathsf{knapsack}(|I|, C) \\ \mathsf{knapsack}(k, c) \;&=\; \begin{cases} 0 \textbf{ if } k = 0 \\ \mathsf{knapsack}(i-1, c) \textbf{ if } w_i > c \\ \mathbf{max}(v_i + \mathsf{knapsack}(i-1, c-w_i), \mathsf{knapsack}(i-1, c)) \textbf{ otherwise} \end{cases} \end{aligned}$$

Assuming for convenience that the items are sorted in order of decreasing efficiency (that is, $\frac{v_i}{w_i}$), we can calculate an upper bound for the subproblem (i, c) as follows [Dantzig, 1957]:

$$\mathsf{knapsack\_ub}(i, c) \;=\; \begin{cases} v_i \frac{c}{w_i} \textbf{ if } w_i > c \\ v_i + \mathsf{knapsack\_ub}(i-1, c-w_i) \textbf{ otherwise} \end{cases}$$

Consider the knapsack problem shown in Figure 2.11. As it has 5 items and capacity 18, bottom-up dynamic programming would require calculating all 90 entries in the table. However, the majority of these entries are never used in the construction of the optimal

item	w_i	v_i	$\frac{v_i}{w_i}$
i_5	1	2	2
i_4	6	7	1.16
i_3	7	8	1.14
i_2	8	9	1.12
i_1	9	10	1.11
cap	18		

Figure 2.11: Example knapsack problem.

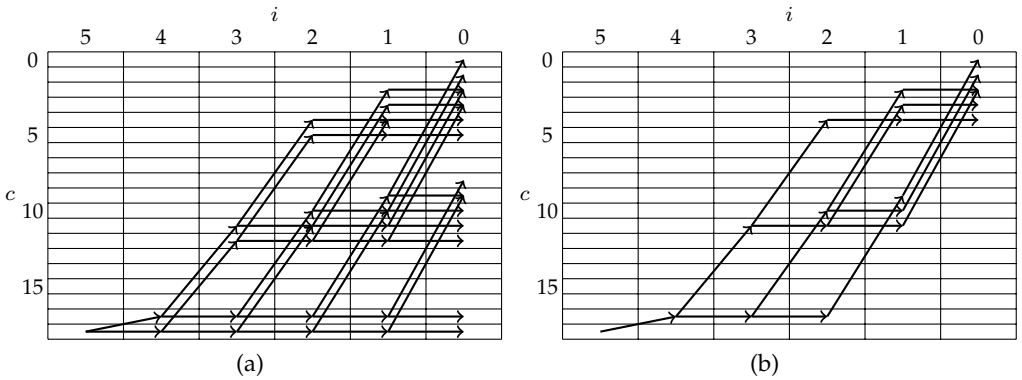


Figure 2.12: Sequence of cells explored when solving the knapsack problem of Example 2.25. (a) Using top-down dynamic programming with no bounding. (b) With local bounding.

solution. Figure 2.12 (a) shows the sequence of calls made during a top-down computation, and Figure 2.13 (a) shows the set of computed values. The optimal value is $v = 21$ *– by tracing back along the call graph, we can determine that the corresponding solution is* $\{i_5, i_2, i_1\}$.

As can be seen from part (b) of Figures 2.12 and 2.13, by using local bounding we can construct the optimal solution while computing fewer subproblems. In this instance, argument bounding provides no improvement over local bounding, as the subproblems end up being traversed in order of increasing value.

□

c	i_5	i_4	i_3	i_2	i_1
0					
1					
2					0
3					0
4				0	0
5				0	0
6					
7					
8					
9					10
10				10	10
11			10	10	10
12			10	10	10
13					
14					
15					
16					
17		19	19	19	10
18	21	19	19	19	10

(a)

c	i_5	i_4	i_3	i_2	i_1
0					
1					
2					0
3					0
4				0	0
5					
6					
7					
8					
9					10
10				10	10
11			10	10	10
12					
13					
14					
15					
16					
17		19	19	19	
18	21				

(b)

Figure 2.13: Table of results computed for the knapsack problem in Example 2.25 with (a) no bounding, and (b) local bounding.

Part I

Generic Propagation Techniques

One of the key properties that makes solving a wide range of combinatorial optimization problems possible is the existence of general techniques that can be applied to multiple problems; either by transforming the problem (as in the case of SAT and MIP), or by describing a solution procedure (as with dynamic programming).

Constraint satisfaction and optimization problems often involve domain-specific constraints that aren't natively supported by the constraint solver. In the case of finite-domain CP and lazy clause generation solvers, it is then necessary to either reformulate the domain-specific constraint in terms of primitive constraints, or design and implement propagation and explanation algorithms for the constraint.

A preferable option is to provide a declarative specification of the constraint, and have an efficient learning propagator automatically constructed to be used by the solver. In Part I, we describe several techniques for constructing learning propagators for arbitrary global constraints, using MDDs or `s-DNNF` as an underlying representation, that can be integrated into a lazy clause generation solver.

3

Multi-valued Decision Diagrams

One of the major challenges in using constraint propagation solvers in general, and particularly lazy clause generation solvers, is the problem of handling problem-specific global constraints. If the solver does not already have a specific propagator for the constraint, the options are generally either to use a decomposition, or implement a propagator. Decompositions are (comparatively) simple, but often introduce large numbers of intermediate variables, and can propagate poorly. Implementing propagators requires a thorough understanding of the constraint, lots of time and is error prone (particularly in the case of lazy clause generation solvers).

An alternative approach is to have a declarative specification of the constraint, and some method for automatically deriving a propagator from this specification. Multi-valued decision diagrams (MDDs), described in Chapter 2, are well suited to this task, as they can be automatically constructed from a series of logical operations and the satisfiability of an MDD G can be tested in $O(|G|)$ time.

In order to use MDD-based constraints in a lazy clause generation based solver, we need two components. First, a propagation algorithm for pruning values from domains; and second, an explanation algorithm to generate explanation clauses for inferences generated. In this chapter, we introduce an incremental propagation algorithm for MDDs that avoids touching parts of the MDD that haven't changed, and describe two explanation algorithms for MDDs: an extension of the algorithm of Subbarayan [Subbarayan, 2008], and an incremental algorithm that attempts to avoid traversing the entire graph.

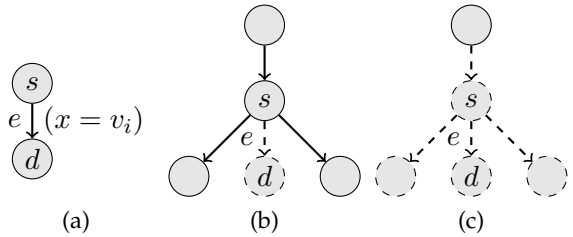


Figure 3.1: Consider an edge e from s to d, shown in (a). Assume e is killed from below (due to the death of d). If s has living children other than e, as in (b), the death of e does not cause further propagation. If e was the last living child of s, such as in (c), s is killed, and the death propagates upwards to any incoming nodes of s. Propagation occurs similarly when e is killed from above – we continue propagating downwards if and only if d has no other living parents. If v_i is removed from the domain of x (and e was alive), we must both check for children of s and for parents of d.

This chapter is organized as follows. In the next section, we describe incremental propagation algorithms for enforcing constraints represented as MDDs. In Section 3.2, we describe several explanation algorithms to integrate MDD constraints with a conflict-learning solver, together with some refinements to reduce learnt clause size. In Section 3.3, we compare the performance of the described methods on a variety of problems, and finally we conclude in Section 3.4.

3.1 Incremental Propagation

When a value v_i is removed from the domain of a variable x, the edges corresponding to that value are killed. An edge (x, v_i, s, d) being killed in this way can only cause changes if it is the last remaining outgoing edge of s, in which case it will kill s and all incoming edges to s, or the last incoming edge of d, in which case it may kill all outgoing edges from d – this is illustrated in Figure 3.1. Thus, if s (and d) have other incoming (and outgoing) edges remaining, we need not explore more distant parts of the graph. If this is not the case, however, we must repeat this process for the new edges that have been killed.

Similarly, removing an edge (x, v_i, s, d) can cause v_i to be removed from the domain of x if and only if all other edges supporting that value are killed. Thus, we want to efficiently determine whether or not a given edge is the last remaining edge for the given value. However, keeping edge counts for nodes and values is not desirable, as we would then have to restore these counts upon backtracking. Accord-

ingly, we adopt a similar method to the two-literal watching scheme [Moskewicz et al., 2001] used in SAT solvers.

We associate with each edge flags indicating whether (a) the edge is alive, and (b) whether it provides support for a value, the node above, or the node below. We initially mark one edge for each value as *watched*, along with one incoming and outgoing edge for each node. When an edge is removed, it is marked as killed, then the watch flags are examined. If none of the watch flags are set, the edge cannot cause any further changes to the graph or domains. If it is watched by a node (not in the direction from which it was killed), we must then search the corresponding node for a new watched edge; if none can be found, the node is killed, and further propagation occurs. Likewise, if it is watched by a value, we must then search for a new supporting edge; if none exists, the corresponding value is removed from the domain. Otherwise, the new supporting edge is marked as watched, and the mark is removed from the old edge. While the liveness flags must still be restored upon backtracking, this is less expensive than updating separate counts for incoming, outgoing and value supports each time an edge is killed or restored.

Pseudo-code for the algorithm is given in Figures 3.2 and 3.3. Algorithm mdd_inc-propagate takes an MDD G and a set of pairs (var, val) where $var \neq val$ is the change in domains by new propagation. The MDD graph G maintains a status $G.status[e]$ for each edge e as either: alive, dom killed by domain change, below killed from below (no path to $\mathcal{T}$ from $e.end$), or above killed from above (no path from the root to $e.start$). It also maintains a watched edge for each node n's input ($n.watch\_in$), output ($n.watch\_out$), and each (var, val) pair ($G.support[var, val]$). For simplicity of presentation, the information about how each edge e is being watched is also recorded as $G.watched[e] \subseteq \{\text{begin}, \text{end}, \text{val}\}$. If begin $\in G.watched[e]$, the edge e is watched by the node $e.begin$; likewise for end and val. The pseudo-code for upward_pass is omitted since it is completely analogous to downward_pass. The graph also maintains a trail of killed edges $G.trail$ (which is initially empty) and, for each (var, val) pair, a pointer to the level of the trail when it was removed ($G.limit[var, val]$). The list *kfa* holds the set of nodes that may have been killed due to removal of incoming edges (killed from above); *kfb* is used similarly with regard to outgoing edges. Note that restoring the state of the propagator, mdd_restore

```
mdd_incpropagate(G, changes)
    kfa := {} % The set of nodes that may have been killed from above.
    kfb := {} % Nodes which may have been killed from below.
    pinf := {} % (var,val) pairs that may be removed from the domain.
    count := length(G.trail) % Record how far to unroll the trail to get back to this state.

    for((var, val) in changes)
        G.limit[var, val] := count % Mark the restoration point.
        % Kill all remaining edges for the value.
        for(edge in G.edges(var, val))
            if(G.status[edge] ≠ alive) continue
            G.status[edge] := dom % Mark the edge as killed due to external inference.
            insert(G.trail, edge) % Add the edge to the trail
            if(begin ∈ G.watched[edge])
                % If this edge supports the above node e.begin,
                % add the node to the queue for processing.
                kfb ∪:={edge.begin}
            if(end ∈ G.watched[edge])
                % Likewise, add the end node if it is supported by the edge.
                kfa ∪:={edge.end}
    pinf := downward_pass(G, kfa)
    if(G.status[T.watch_in] ≠ alive)
        % If T is unreachable, the partial assignment is inconsistent.
        % Otherwise, propagating upwards is safe.
        return FAIL
    pinf ∪:= upward_pass(G, kfb)
    return collect(G, pinf)
```

Figure 3.2: Top level of the incremental propagation algorithm.

shown in Figure 3.3, requires only restoring the status of killed edges to alive. The maximum trail size is the number of edges in G.

The mdd_incpropagate algorithm enforces domain consistency on the MDD. The complexity of mdd_incpropagate is $O(|G|)$ down a branch of the search tree (with a little care in implementation). In each forward computation each edge is only killed once. As each edge corresponds to exactly one (var, val) pair, and each (var, val) pair is killed at most once, each edge is only considered at most once in the inner **for** loop of mdd_incpropagate. Each node n can appear in kfa at most $|n.in\_edges|$ times, hence the **for** loop in downward_pass runs $O(|G|)$ times. By traversing $n.in\_edges$ (in downward_pass) from the previously watched edge, we can guarantee that we only traverse each edge twice down the branch of the search tree. Similarly when traversing $G.edges[var, val]$ (in collect) looking for new support, if we look from the previously watched edge we can guarantee we only traverse each edge at most twice down the branch of the search tree.

```
downward_pass(G, kfa)
  pinf := {};
  for(node in kfa)
    % Search for a new support
    for(edge in node.in_edges)
      if(G.status[edge] := alive)
        % Support found. Update the watches.
        G.watched[node.watch_in] \:={end}
        G.watched[edge] ∪:={end}
        node.watch_in := edge
        break
    if(is_dead(node.watch_in))
      % The node is still dead
      % kill the outgoing edges.
      for(edge in node.out_edges)
        if(G.status[edge] ≠ alive { continue }
        G.status[edge] := above
        insert(G.trail, edge)
        if(end ∈ G.watched[edge])
          % If the edge supports a node,
          % queue it for processing.
          kfa ∪:={edge.end}
        if(val ∈ G.watched[edge])
          % If the edge supports a value,
          % add it to the queue.
          pinf ∪:={(edge.var, edge.val)}
    return pinf
```

```
collect(G, pinf)
  inf = {}
  for((var, val) in pinf)
    % Search for a new support.
    edge = G.support[var, val]
    for(e in G.edges(var, val))
      if(G.status[e] = alive)
        % Support found.
        G.watched[edge] \:={val}
        G.watched[e] ∪:={val}
        G.support[var, val] = e
        break
    edge = G.support[var, val]
    if(G.status[edge] ≠ alive)
      % Still dead.
      inf ∪:={(var, val)}
      G.limit[var, val] = count
  return inf
```

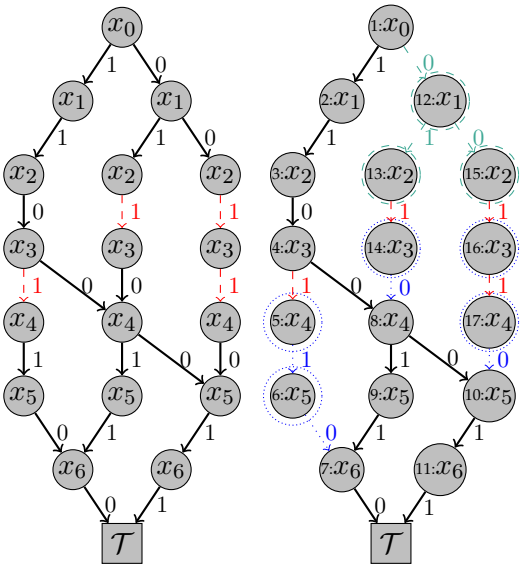
```
mdd_restore(G, var, val)
  % Determine how far to unroll
  lim = G.limit[var, val]
  while(length(G.trail) > lim)
    % Restore the propagator.
    edge = pop_last(G.trail)
    G.status[edge] = alive
```

Figure 3.3: Pseudo-code for determining killed edges and possibly removed values in the downward pass, collecting inferred removals, and backtracking.

Example 3.1. *Consider the MDD shown in Fig 3.4(a). If the values $x_2 = 1$ and $x_3 = 1$ are removed from the domain, we must mark the corresponding edges as removed. These edges are shown dashed.*

Incremental propagation works as follows assuming the leftmost edge leaving and entering a node is watched, and the leftmost edge for each $x = d$ valuation is watched. The removal of the edge $(x_2, 1, 13, 14)$ removes the support for node 13 which is added to kfb, as denoted by operation $\cup:=$, and node 14 which is added to kfa. Similarly 15 is added to kfb and 16 to kfa by the removal of $(x_2, 1, 15, 16)$. The removal of the edges $(x_3, 1, 4, 5)$ and $(x_3, 1, 16, 17)$ leave $kfa = \{5, 14, 16, 17\}$ and $kfb = \{4, 13, 15, 16\}$ before downward_pass *execution.*

*We then perform the downward pass. We find no new supports from above for 5 which means we mark $(x_4, 1, 5, 6)$ as killed from above (*above*) and add 6 to kfa and add $(x_4, 1)$ to the queue of values to check $pinf$. Similarly we kill $(x_5, 0, 6, 7)$ and add 7 to kfa and $(x_5, 0)$ to $pinf$. We do find a new support from above for node 7. We similarly kill edges*



(a) $x_2 \neq 1, x_3 \neq 1$ (b) Propagation

Figure 3.4: An example MDD for a `regular` constraint 0*1100*110* over the variables $[x_0, x_1, x_2, x_3, x_4, x_5, x_6]$, and the effect of propagating $x_2 \neq 1$ and $x_3 \neq 1$ using the incremental propagation algorithm.

$(x_3, 0, 14, 8)$ *but note since this is neither watched by its destination nor its value, nothing is added to kfa or* $pinf$*. We similarly kill the edge* $(x_4, 0, 17, 10)$ *but again this is not watched.*

We then perform the upward pass. We find a new support for node 4 *from below. We find no new supports for nodes 13 hence we kill edge* $(x_1, 1, 12, 13)$ *and add* 12 *to kfb, We similarly kill node 15 and edge* $(x_1, 0, 12, 15)$ *which adds* $(x_1, 0)$ *to* $pinf$*. Examining node 12 we find no support from below and kill* $(x_0, 0, 1, 12)$ *adding* $(x_0, 0)$ *to* $pinf$ *(but not* 1 *to kfb). The killed from below nodes and edges are shown dashed in Figure 3.4(b), while the killed from above nodes and edges are shown dotted.*

We finally consider $pinf = \{(x_4, 1), (x_5, 0), (x_1, 0), (x_0, 0)\}$*. We find a new support* $(x_4, 1, 8, 9)$ *for* $x_4 = 1$*, therefore we remove* val *from* $G.watches$ *of edge* $(x_4, 1, 5, 6)$ *as denoted by the* $\mathrel{\backslash}:=$ *operation. We are not able to find new supports for the other variable value pairs. Propagation determines that* $x_5 \neq 0$*,* $x_1 \neq 0$ *and* $x_0 \neq 0$*.*

□

3.2 Explaining MDD Propagation

In this section, we describe several explanation generation algorithms for MDD propagators.

3.2.1 Non-incremental Explanation

The best previous approach to explaining BDD propagation is due to Subbarayan [2008]. Here we extend this approach to MDDs. It works in two passes. It first traverses the MDD backwards from the true node $\mathcal{T}$ marking which nodes can reach $\mathcal{T}$ in the current state assuming the negation of the inference to be explained holds. It then performs a breadth-first traversal from the root progressively adding back domain values as long as this does not create a path to $\mathcal{T}$. The algorithm creates a minimal explanation (removing any part of it does not create a correct explanation), but it requires traversing the entire MDD once for each new inference.[1] Note that it does not create a *minimum size explanation*; doing so is NP-hard [Subbarayan, 2008]. Pseudo-code explaining the inference $var \neq val$ is given in Figure 3.5.

Example 3.2. *Consider explaining the inference $x_0 \neq 0$ discovered in Example 3.1. The* mark_reachT *call walks the MDD from $\mathcal{T}$ adding which nodes can reach $\mathcal{T}$ into $reachT$ in the state where the inference was performed (Figure 3.4(a)) with the additional assumption that the converse of the inference holds ($x_0 = 0$). It discovers that all nodes reach $\mathcal{T}$ except $\{1, 12, 13, 15, 16\}$. It then does a breadth-first traversal from the root looking for currently killed edges that if not excluded would create a path from the root to $\mathcal{T}$. From the root we only reach node 12 (under the assumption that $x_0 = 0$), from 12 we reach 13 and 15. Restoring the killed edge $(x_2, 1, 13, 14)$ would create a path to $\mathcal{T}$, hence we require $x_2 \neq 1$ in the reason. Once we have this requirement, from 15 we cannot reach 16 and the algorithm stops with the explanation $\neg[\![x_2 = 1]\!] \rightarrow \neg[\![x_0 = 0]\!]$.* □

3.2.2 Incremental Explanation

The non-incremental explanation approach above requires examining the entire MDD for each new inference made. This is a significant overhead, although one

[1] It is interesting to note that any minimal explanation for $[\![x \neq v]\!]$ under a partial assignment $[\![x_1 \neq v_1]\!] \wedge \ldots \wedge [\![x_n \neq v_n]\!]$ is a prime implicate [Reiter and de Kleer, 1987] of the constraint C containing $[\![x \neq v]\!]$ and some subset of $\{[\![x_1 = v_1]\!], \ldots, [\![x_k = v_k]\!]\}$.

```
mdd_explain(G, var, val)
    reachT = mark_reachT(G, var, val) % Find the set of nodes that can reach true.
    explanation = {}
    queue = {G.root}
    while(queue ≠ ∅)
        for(node in queue)
            for(e ∈ node.out_edges)
                if(e.var ≠ var and e.end ∈ reachT)
                    explanation ∪:= (e.var, e.val)
        nqueue = {} % Record nodes of interest on the next level.
        for(node in queue)
            for(e ∈ node.out_edges)
                if (e.var == var and e.val == val)
                        or (e.var ≠ var and (e.var, e.val) ∉ explanation))
                    nqueue ∪:= e.end
        queue = nqueue
    return explanation

mark_reachT(G, var, val) )
    reachT = {T} % Reset the set of nodes that can reach true.
    queue = {T.in_edges} % Reset the queue of nodes to be processed.
    for(edge in queue)
        if(edge.begin ∈ reachT) continue
        if(edge.var == var)
            if(edge.val == val)
                reachT ∪:= {edge.begin}
                queue ∪:= edge.begin.in_edges
        else if(G.status[edge] == alive and edge.begin ∉ reachT)
            reachT ∪:= {edge.begin}
            queue ∪:= {edge.begin.in_edges}
    return reachT
```

Figure 3.5: Non-incremental MDD explanation. Extended from Subbarayan [2008].

should note that explanations are only required to be generated during the computation of a nogood from failure, not during propagation, hence not every inference will need to be explained. Once we are using incremental propagation, the overhead of constructing minimal explanations is relatively even higher.

It is difficult to see how to generate a minimal explanation incrementally, since the minimality relies on examining the whole MDD.[2] Thus we give up on minimality and instead search for a *sufficiently small* reason without exploring the whole graph.

In order to achieve this, we make two observations. First, an edge being killed is most likely to have effects in the nearby levels; an edge at level j can kill a node at

[2]Note that references to incrementality in this section are not referring to the re-use of information between executions, but instead to the traversal of the graph starting from only the edges to be explained, and progressively expanding as needed.

```
mdd_inc_explain(G, var, val)
    kfa = {} % edges killed from above
    kfb = {} % edges killed from below
    for(edge in G.edges(var, val))
        % Split possible supports
        if(killed(edge, above))
            kfa ∪:={edge}
        else
            kfb ∪:={edge}
    % Explain all those killed from below
    return explain_down(kfb)
    % And all those killed from above
            ∪ explain_up(kfa)
```

Figure 3.6: Top-level wrapper for incremental explanation.

level $j+2$ only if it is the final support to a node at level $j+1$, which in turn remains the only support for a node at level $j+2$. If we are searching for an explanation for the death of an edge, it is most likely to be near the edge being explained. This is particularly the case for constraints which are *local* in nature, where the possible values for x_j are most strongly constrained by the values of variables in nearby levels. Second, if the cause of the propagation is far from the killed node, this may indicate the presence of a narrow *cut* in the graph, which eliminates a large set of nodes. The goal of the incremental algorithm is to search the section of the graph where the explanation is likely to be, but follow chains of propagation to hopefully find any narrow cuts (which provide explanation for an entire subgraph).

Pseudo-code explaining the inference $var \neq val$ is given in Figures 3.6 and 3.7. The code makes use of function killed_below to check if a node has been killed from below (and similarly for killed_above). In practice, the results of these functions are cached to avoid recomputation. The functions explain_down and explain_up keep track of pending nodes of the next level which may be required to be explained. We omit code to explain failure, which is similar.

The algorithm first records the reason for the removal of each edge. We then traverse the graph from all the edges defining a removed value $var = val$ depending on how they were killed. For those killed from below we search breadth-first for edges below that were killed by domain reduction, whose endpoint was not also killed from below. They are added to the reason for the removal of $var = val$. We then traverse the edges which are not already part of the reason and add their child edges to check in the next level. Pending edges are edges whose end node may be

```
killed_below(G,node)
    % node is killed below if
    for(edge in node.out_edges)
        s = G.status[edge]
        if(s ∈ {alive, above})
            % No outgoing edge is alive
            % or killed from above
            return false;
    return true
```

```
explain_down(kfb)
    reason = {}
    % Traverse the MDD downwards, breadth first
    while(¬is_empty(kfb))
        % Scan the current level for edges
        % that will need explaining.
        pending = {}
        for(e in kfb)
            % For each edge requiring explanation
            if(G.status[e] = dom and
                ¬killed_below(G, e.end))
                % There is no later explanation,
                % so add (e.var, e.val) to the reason.
                reason ∪:={(e.var, e.val)}
            else
                pending ∪:={e}
        next = {}
        % Collect the edges that haven't been
        % explained at this level.
        for(e in pending)
            if((e.var, e.val) ∉ reason)
                % If e is not explained already
                % collect its outgoing edges
                next ∪:= e.end.out_edges
        % Continue with the next layer of edges.
        kfb = next
    return reason
```

Figure 3.7: Pseudo-code for incremental explanation of MDDs. $killed\_above$ and explain_up act in exactly the same fashion as $killed\_below$ and $explain\_down$, but in opposite directions.

required to be explained on the next level, but it can happen that before the current level is finished they are already explained. Hence the two pass approach.

A greedier algorithm which just tried to find a "close" reason why $var = val$ has been removed would stop the search whenever it reached an edge killed by domain reduction. On first sight this might seem to be preferable, as it traverses less of the MDD and gives a more "local" explanation. Our experiments showed two deficiencies: in many cases this killed edge may be redundant as it is explained by other edges killed higher up that are still required to be part of the explanation for other reasons; and it failed to find "narrow cuts" in the MDD which lead to more reusable explanations.

Example 3.3. *Consider the explanation of $x_0 \neq 0$ determined in Example 3.1. The edge $(x_0, 0, 1, 12)$ is marked as* **below**, *so it is added to kfb. In* **explain_down** *we add 12 as a pending node. We then insert $(x_1, 1, 12, 13)$ and $(x_1, 0, 12, 15)$ into $next$ and restart the* ***while*** *loop. Nodes 13 and 15 become pending and in the next iteration of the* ***while*** *loop kfb is $\{(x_2, 1, 13, 14), (x_2, 1, 15, 16)\}$. The first edge is killed by domain and its end node 14 is not killed from below so we add $(x_2, 1)$ to reason. For the second edge node 16 is killed from below, so $(x_2, 1, 15, 16)$ is a pending edge. In the second* ***for*** *loop over kfb we determine that is already explained by $reason$. The algorithm terminates with the reason $\{(x_2, 1)\}$. This becomes the clause $\neg[\![x_2 = 1]\!] \rightarrow \neg[\![x_0 = 0]\!]$.* □

Example 3.4. *Unfortunately, these explanations are not guaranteed to be minimal. Consider again the constraint demonstrated in Example 3.1, but instead with $x_3 \neq 1$ fixed first, and $x_0 \neq 0$ fixed later. This kills nodes $\{15, 16\}$ from below, and nodes $\{5, 6, 12, 13, 14, 17\}$ killed from above. In order to explain $x_2 \neq 1$, we must determine reasons for the edges $(x_2, 1, 13, 14)$ and $(x_2, 1, 15, 16)$. Explaining $(x_2, 1, 13, 14)$ gives us $\{x_0 \neq 0\}$. As $(x_2, 1, 15, 16)$ was killed from below, we also add $x_3 \neq 1$ to the reason, even though $x_0 \neq 0$ already explains this edge.*

The algorithm mdd_inc_explain is $O(|G|)$ for a single execution, no better than the non-incremental explanation in the worst case. However, if the constraint is reasonably local in nature, significantly fewer edges will be explored – if an explanation e contains variables V_e, the algorithm will explore at most those edges between $min(V_e)$ and $max(V_e)$.

3.2.3 Shortening Explanations for Large Domains

Both the non-incremental and incremental algorithms for MDD explanation collect explanations of the form $(\wedge\neg[\![x_i = v_{ij}]\!]) \rightarrow \neg[\![x = v]\!]$. These are guaranteed to be correct, and, in the non-incremental case, minimal. But they may be very large since for a single variable x_i with large initial domain $D(x_i)$ we may have up to $|D(x_i)| - 1$ literals involved.

A first simplification is to replace any subexpression $\wedge_{d\in D(x_i), d\neq d'}\neg[\![x_i = d]\!]$ by the equivalent expression $[\![x_i = d']\!]$. This shortens explanation clauses considerably without weakening them. But it does not occur frequently. A second simplification is to replace $\wedge_{d\in S}\neg[\![x_i = d]\!]$ by $\neg[\![x_i \leq l-1]\!] \wedge [\![x_i \leq u]\!] \wedge \wedge_{d\in S\cap[l\,..\,u]}\neg[\![x_i = d]\!]$ where $l = \min(D(x_i) - S)$ and $u = \max(D(x_i) - S)$ are the least and greatest values of x_i consistent with the formula. Again this can sometimes shorten clauses considerably, but sometimes is of no benefit.

Finally we can choose to weaken the explanation. Suppose that in the current state $D(x_i) = d'$, that is x_i is fixed to d', then we can choose to replace $\wedge_{d\in S}\neg[\![x_i = d]\!]$, where $|S| > 1$ and $d' \notin S$ by $[\![x_i = d']\!]$. This shortens the explanation, but weakens it.

Example 3.5. *Consider a state where the following explanation has been generated:*

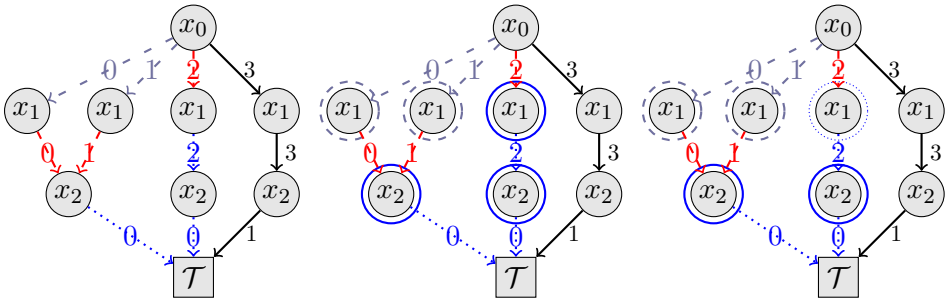
$$[\![x \neq 2]\!] \wedge [\![x \neq 3]\!] \rightarrow [\![y \neq 1]\!]$$

We can resolve this with the implicit clauses $[\![x = 5]\!] \rightarrow [\![x \neq 2]\!]$ *and* $[\![x = 5]\!] \rightarrow [\![x \neq 3]\!]$ *to give:*

$$[\![x = 5]\!] \rightarrow [\![y \neq 1]\!]$$

If $\mathcal{D}(x) = \{5\}$ *at the time of inference, we can replace the original explanation with this smaller explanation.* □

While we could perform this as a postprocess by first creating an explanation and then weakening it, doing so will make the explanations far from minimal. Hence we need to adjust the explanation algorithms so that as soon as they collect (x_i, e) and (x_i, e') in a reason, when in the current state $x_i = d'$ we in effect add all of $(x_i, e''), e'' \in D(x_i) - \{d'\}$ to the reason being generated (which will simplify to a single literal $[\![x_i = d']\!]$ in the explanation).



(a) Constraint (b) Without weakening (c) Inline weakening

Figure 3.8: Explaining the inference $x_2 \neq 0$. The incremental explanation algorithm will generate the explanation $x_0 \neq 2 \wedge x_1 \neq 0 \wedge x_1 \neq 1$. This can then be shortened to $x_0 \neq 2 \wedge x_1 = 3$. If weakening is performed during explanation, $x_1 \neq 0 \wedge x_1 \neq 1$ will immediately be shortened, and the edge $x_0 = 2$ will never be reached, yielding the explanation $x_1 = 3$.

Example 3.6. *Consider the MDD state shown in Figure 3.8(a) after the external inferences that* $x_1 \neq 0$, $x_1 \neq 1$, *and* $x_0 \neq 2$. *The two leftmost* x_1 *nodes are killed from below, while the third* x_1 *node is killed from above. In explaining the inference* $x_2 \neq 0$, *the incremental explanation algorithm starts at the edges to be explained, then collects* $x_1 \neq 0$ *and* $x_1 \neq 1$ *as values that must remain removed. Since the edge* $x_1 = 2$ *has not yet been explained, the algorithm continues, fixing* $x_0 \neq 2$. *We can then shorten this explanation to* $x_1 = 3 \wedge x_0 \neq 2$. *However, if we weaken the explanation during construction, we detect that* $x_1 \neq 0 \wedge x_1 \neq 1$ *can be weakened to* $x_1 = 3$, *which eliminates the remaining* $x_1 \neq 2$ *edge, giving us a final explanation of* $x_1 = 3 \rightarrow x_2 \neq 0$. □

3.3 Experimental Results

Experiments were conducted on a 3.00GHz Core2 Duo with 2 GB of RAM running Ubuntu GNU/Linux 8.10. Our solver is a modified version of MiniSAT2 (release 070721), augmented with MDD propagators. Explanations are constructed on demand during conflict analysis, and added to the clause database as learned clauses.

We compare a number of variations of our solver: `base` propagation and non-incremental explanation; `ip` is the incremental propagation approach described herein, with non-incremental explanation; `+w` denotes a method with explanation weakening; and `+e` denotes incremental explanations. $\texttt{dec}_{tse}$ is the standard Tseitin decomposition described in Chapter 2, and $\texttt{dec}_{dc}$ is a domain consistent decomposition that is described in detail in Chapter 4. All times are given in seconds.

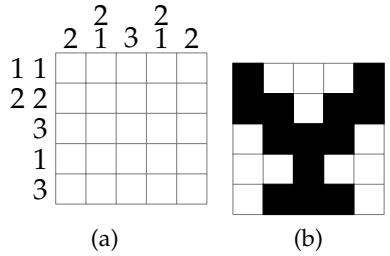


Figure 3.9: (a) An example nonogram puzzle, (b) the corresponding solution.

3.3.1 Nonograms

Nonograms [Ueda and Nagao, 1996] are a set of puzzles that have been studied both in terms of constraint programming, and in their own right, and a number of standalone solvers have been designed to solve these problems. A nonogram consists of an $n \times m$ matrix of blocks which may or may not be filled. Each row and column is marked with a sequence of numbers $[n_0, n_1, ..., n_k]$. This constraint indicates that there must be a sequence of n_0 filled squares, followed by one or more empty squares, followed by n_1 filled squares, and so on. Nonogram solvers are often used to assist puzzle design – rather than finding a single solution, a solver is used to determine uniqueness of a solution.

In all cases, the model is constructed introducing a Boolean variable for each square in the matrix, and converting each row and column constraint into a DFA, then expanding the DFA into a BDD.

An example nonogram is given in Figure 3.9a. The $[2, 2]$ next to the second row indicates that there must be a block of 2 filled blocks, followed by a gap, then another 2 filled squares. This constraint is converted into a DFA. The solution to this puzzle is given in Figure 3.9b.

The nonogram puzzle instances are taken from Wolter [b], which compares 15 different solvers for the problem on a 2.6GHz AMD Phenom quad-core processor with 8Gb of memory. These solvers either find two distinct solutions, or prove that there is a unique solution. Two solvers (PBNSolve and BGU) are listed as solving all but two instances within 30 minutes, taking a total of 97.08s and 236.34s respectively for the solved instances; all other solvers fail to solve at least three of the instances within the time limit.

The results in Tables 3.1 to 3.3 compare various approaches: the best solver from Wolter [b] PBNSOLVE 1.09, GECODE 3.10, and our solvers. The tables show

Without learning							
Problem	PBNSOLVE		GECODE		Seq		
	time	fails	time	fails	base	ip	fails
1_dancer	**0.0**	0	**0.0**	0	0.02	0.03	0
6_cat	**0.0**	0	**0.0**	0	0.05	0.04	0
21_skid	**0.0**	0	**0.0**	0	0.04	0.04	0
27_bucks	**0.0**	2	**0.0**	2	0.06	0.06	2
23_edge	**0.0**	22	**0.0**	25	0.03	0.03	26
2413_smoke	**0.0**	7	**0.0**	8	0.04	0.05	9
16_knot	**0.0**	0	**0.00**	0	0.09	0.08	0
529_swing	**0.0**	0	0.01	0	0.16	0.16	0
65_mum	**0.00**	22	0.01	22	0.11	0.11	23
7604_dicap	**0.01**	0	**0.01**	0	0.25	0.25	0
1694_tragic	**0.03**	193	0.11	255	0.24	0.19	256
1611_merka	**0.00**	25	0.02	13	0.29	0.27	14
436_petro	**0.06**	246	69.20	106919	34.75	15.73	106920
4645_m_and_m	**0.07**	180	0.65	428	0.80	0.38	429
3541_signed	**0.03**	146	8.43	6484	4.95	1.75	6485
803_light	0.43	995	—	—	—	—	—
6574_forever	3.94	147112	4.94	30900	2.91	1.45	30901
10810_center	8.43	277046	**0.03**	2	0.28	0.28	3
2040_hot	0.89	2508	—	—	—	—	—
6739_karate	0.87	9959	53.41	170355	34.06	13.00	170356
8098_domIII	11.55	208689	—	—	368.90	243.70	8351050
2556_flag	0.50	22184	3.06	16531	1.95	0.67	16532
2712_lion	6.67	44214	—	—	—	—	—
10088_marley	—	—	—	—	—	—	—
9892_nature	—	—	—	—	—	—	—
12548_sierp	—	—	—	—	—	—	—
$\sum$	—	—	—	—	—	—	—

Table 3.1: Unique-solution performance results on hard nonogram instances from Wolter [b], using solvers without learning.

the average time (over 25 runs) in seconds and the number of failures in the search for each instance. The sums of each column are given in row $\sum$. Since the MDDs are BDDs in this case weakening (Section 3.2.3) is not applicable. Note that base and ip perform exactly the same search. We use two search strategies: (Seq) filling in the matrix in order from left-to-right and top-to-bottom which is also used by GECODE; and (VSIDS) using activity based VSIDS search [Zhang et al., 2001] which concentrates on the exploring decisions that have been most active in contributing to failure.

Table 3.1 compares the non-learning approaches. Note that VSIDS search is only applicable with learning since activity is derived from learning. The results here show that specialized code PBNSolve (which is not based on constraint programming) is highly competitive. Interestingly, PBNSolve is listed in Wolter [b]

Problem	Without learning			With learning				
	Seq			Seq				
	base	ip	fails	base	ip	fails	ip+e	fails
1_dancer	0.02	0.03	0	0.03	0.03	0	0.02	0
6_cat	0.05	0.04	0	0.04	0.05	0	0.05	0
21_skid	0.04	0.04	0	0.05	0.05	0	0.04	0
27_bucks	0.06	0.06	2	0.06	0.05	2	0.06	2
23_edge	0.03	0.03	26	0.03	0.03	18	0.03	22
2413_smoke	0.04	0.05	9	0.04	0.05	7	0.05	8
16_knot	0.09	0.08	0	0.08	0.08	0	0.08	0
529_swing	0.16	0.16	0	0.16	0.15	0	0.16	0
65_mum	0.11	0.11	23	0.12	0.11	22	0.11	22
7604_dicap	0.25	0.25	0	0.26	0.25	0	0.24	0
1694_tragic	0.24	0.19	256	0.22	0.21	141	0.19	123
1611_merka	0.29	0.27	14	0.29	0.28	13	0.27	10
436_petro	34.75	15.73	106920	0.53	0.38	3068	0.56	5173
4645_m_and_m	0.80	0.38	429	0.40	0.31	130	0.29	128
3541_signed	4.95	1.75	6485	0.40	0.32	337	0.32	425
803_light	—	—	—	0.37	0.24	1585	0.19	1064
6574_forever	2.91	1.45	30901	0.08	**0.07**	207	**0.07**	258
10810_center	0.28	0.28	3	0.27	0.27	2	0.28	2
2040_hot	—	—	—	1.12	0.77	4708	0.72	5527
6739_karate	34.06	13.00	170356	0.67	0.48	4525	0.40	3717
8098_domIII	368.90	243.70	8351050	8.37	7.09	147444	6.17	130704
2556_flag	1.95	0.67	16532	0.18	**0.17**	179	**0.17**	389
2712_lion	—	—	—	9.97	7.26	39193	6.19	29673
10088_marley	—	—	—	—	—	—	—	—
9892_nature	—	—	—	1.01	0.68	5346	0.60	5456
12548_sierp	—	—	—	9.21	**6.97**	55994	7.45	53558
$\sum$	—	—	—	—	—	—	—	—

Table 3.2: Unique-solution performance results on hard nonogram instances from Wolter [b] using a sequential search strategy.

as solving nature in 68s, where here it exceeds the 10 minute time limit; the other reported runtimes match the observed results. Gecode and our approaches have the same sequential search. Clearly incremental propagation is advantageous over the base approach in terms of speed.

Table 3.2 compares our algorithms with learning. Clearly learning makes an enormous difference on these benchmarks. First, note that even base is competitive with the best reported solution. Next, the results show that incremental explanation is clearly beneficial, although it can increase search space because it creates non-minimal explanations. While the domain consistent decomposition dramatically outperformed the standard Tseitin decomposition, the incremental propagators were substantially faster (despite requiring, in some cases, substantially more search).

With learning									
Problem	VSIDS								
	$\texttt{dec}_{tse}$	fails	$\texttt{dec}_{dc}$	fails	base	ip	fails	ip+e	fails
1_dancer	0.03	9	0.03	0	0.03	0.03	0	0.03	0
6_cat	0.05	46	0.05	0	0.05	0.05	0	0.05	0
21_skid	0.05	41	0.05	0	0.05	0.05	0	0.05	0
27_bucks	0.08	39	0.08	2	0.07	0.07	2	0.07	2
23_edge	0.03	37	0.03	13	0.03	0.04	10	0.03	10
2413_smoke	0.06	282	0.06	5	0.06	0.06	4	0.05	3
16_knot	0.18	551	0.12	0	0.09	0.09	0	0.09	0
529_swing	0.40	1186	0.24	0	0.18	0.18	0	0.18	0
65_mum	0.23	899	0.16	19	0.12	0.12	11	0.12	8
7604_dicap	2.27	16749	0.42	0	0.29	0.28	0	0.27	0
1694_tragic	0.58	2371	0.33	82	0.25	0.24	107	0.21	102
1611_merka	1.80	6923	0.50	16	0.32	0.29	11	0.30	11
436_petro	0.72	6183	0.20	111	0.13	0.13	21	0.13	64
4645_m_and_m	4.62	10201	0.71	230	0.48	0.34	146	0.33	131
3541_signed	0.66	1878	0.57	102	0.84	0.55	531	0.35	292
803_light	0.42	1923	0.19	82	0.14	0.13	40	**0.12**	39
6574_forever	0.10	618	0.09	97	0.08	**0.07**	128	**0.07**	199
10810_center	1.68	9852	0.44	3	0.31	0.30	5	0.31	5
2040_hot	20.68	40411	0.76	213	0.63	0.43	221	**0.35**	141
6739_karate	2.60	13171	0.33	544	0.19	0.17	150	**0.15**	92
8098_domIII	0.69	9460	0.56	3592	0.16	0.13	2089	**0.11**	1652
2556_flag	0.26	174	0.23	12	0.18	0.19	25	0.18	16
2712_lion	18.07	62531	6.24	6297	2.67	1.53	6940	**0.34**	898
10088_marley	45.89	134859	12.61	11058	5.08	2.25	7034	**0.76**	2218
9892_nature	18.19	75179	3.00	5967	0.98	**0.58**	2401	0.65	3516
12548_sierp	220.90	345854	11.51	15904	19.84	11.65	42281	7.61	37063
$\sum$	341.24	741427	39.51	44349	33.25	19.95	62157	**12.91**	46462

Table 3.3: Unique-solution performance results on hard nonogram instances from Wolter [b] using a VSIDS search strategy.

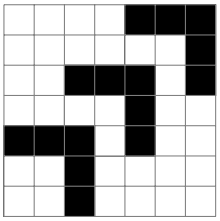


Figure 3.10: An example of the *domino logic* problem set, with $n = 3$.

Since the survey benchmarks were very easy for the MDD propagators we also experimented on a hard artificial class of nonogram problems: *domino logic*, or n-Dom problems, described at Wolter [a]. These are very hard to solve, no solver on the website can solve instances beyond size $n = 16$. These instances are constructed from n identical rotated V shapes, illustrated in Figure 3.10. Comparative results are shown in Tables 3.4 to 3.6. As before, incremental propagation is ben-

Without learning							
n	PBNSOLVE		GECODE		Seq		
	time	fails	time	fails	base	ip	fails
05	**0.00**	121	0.01	163	0.04	0.04	163
06	0.04	718	0.11	2K	0.10	0.08	2K
07	0.34	8K	1.70	29K	0.98	0.76	29K
08	2.14	30K	29.89	435K	16.59	11.88	435K
09	17.09	209K	—	—	362.63	247.50	8351K
10	169.89	1745K	—	—	—	—	—
11	—	—	—	—	—	—	—

Table 3.4: Unique solution performance results for non-learning solvers on *domino logic* nonograms. None of these solvers solve any problem of size greater than 10.

With learning					
n	Seq				
	base	ip	fails	ip+e	fails
05	0.03	0.02	60	0.04	76
06	0.04	0.04	328	0.04	341
07	0.10	0.08	1959	0.08	1825
08	0.48	0.38	11622	0.54	16326
09	8.37	7.04	147444	6.12	130704
10	91.63	79.42	1068666	87.96	1174063
11	—	—	—	—	—
12	—	—	—	—	—
13	—	—	—	—	—
14	—	—	—	—	—
15	—	—	—	—	—
16	—	—	—	—	—
17	—	—	—	—	—
18	—	—	—	—	—
19	—	—	—	—	—
20	—	—	—	—	—
$\sum$	—	—	—	—	—

Table 3.5: Unique solution performance results on the *domino logic* nonogram instances using a sequential search strategy.

eficial, and learning is vital. The BDDs for these constraints are very narrow (between 2 and 6 nodes), so explanation generation accounts for only 1% of execution time; differences between the execution time of ip and ip+e are due to differences in search. While ip+e seems to perform slightly worse than ip using a sequential search method, it appears to drive VSIDS consistently towards better search decisions, reducing time and backtracks by up to 50%.

With learning									
n	VSIDS								
	dec$_{tse}$	fails	dec$_{dc}$	fails	base	ip	fails	ip+e	fails
05	0.04	64	0.04	41	0.05	0.03	34	0.04	41
06	0.04	281	0.04	137	0.04	**0.03**	135	0.04	108
07	0.09	1330	0.08	454	**0.04**	0.05	330	**0.04**	345
08	0.26	3774	0.26	1895	**0.06**	**0.06**	733	**0.06**	670
09	0.67	9460	0.56	3592	0.14	0.14	2089	**0.10**	1652
10	4.41	44680	1.14	6651	0.33	0.23	4571	**0.15**	2729
11	7.97	58414	2.40	11804	0.53	0.39	6565	**0.28**	4855
12	17.76	107155	4.46	19019	1.38	0.98	15133	**0.70**	10937
13	183.80	888493	13.28	41529	2.33	1.68	24351	**1.06**	15289
14	75.24	331334	24.39	63129	5.59	3.88	51167	**2.44**	32402
15	366.03	1127704	67.59	148260	7.86	5.40	65226	**3.14**	36638
16	—	—	82.88	147708	18.03	12.40	123398	**5.76**	59959
17	—	—	183.28	276533	68.32	50.48	381688	**12.38**	109134
18	—	—	392.91	445572	101.31	74.19	500366	**30.83**	226933
19	—	—	—	—	118.16	83.57	538742	**65.66**	395421
20	—	—	—	—	384.99	293.43	1341157	**124.45**	606265
$\sum$	—	—	—	—	709.16	526.94	3055685	**247.13**	1503378

Table 3.6: Unique solution performance results on the *domino logic* nonogram instances using a VSIDS search strategy.

3.3.2 Nurse Scheduling

The second set of experiments uses nurse scheduling benchmarks from Section 6.2 of Brand et al. [2007], where nurses are rostered to day shifts, evening shifts, night shifts and days off. In model 1, each nurse must work 1 or 2 night shifts in every 7 days, 1 or 2 evening shifts, 1 to 5 day shifts and 2 to 5 days off. In model 2, nurses must work 1 or 2 night shifts every 7 days, and 1 or 2 days off every 5 days (which makes day and evening shifts indistinguishable). In both models, a nurse cannot work a second shift within 12 hours of the first. The constraints are encoded as a single `regular` constraint per nurse and a `global_cardinality` [Beldiceanu et al., 2010] constraint per shift, converted to MDDs. We use the 50 instances of 28 day schedules used in Brand et al. [2007] for each model with a 5 minute time limit, plus the next 50 instances from the N30 dataset, available at Vanhoucke and Maenhout. Results for both GECODE and the non-learning solvers are omitted, as they were unable to solve any instances in 5 minutes.

Tables 3.7 and 3.8 show the results on the nurse scheduling benchmarks using sequential search (assigning each nurse for day 1, then each nurse for day 2, etc.), first fail search (picking the nurse/day pair with the smallest remaining domain, breaking ties according to the sequential search), and VSIDS search. The first line is

for direct comparison: it gives the average time and fails for problems solved by all solvers using the given search strategy. The second line gives number of problems solved by each solver and the average solving time and average failures for these solved problems. Comparing to the best results from Brand et al. [2007] (which used first fail and a 100 second time limit) our base solver solves more instances (24 versus 9, and 32 versus 8).[3]

Comparing base versus ip we see that incremental propagation is usually beneficial. For the problems with more backtracking required on average we can see that the incremental propagation can be substantially faster than non-incremental propagation. Incremental explanation for these problems usually reduces the number of problems that can be solved.

The results show that weakening can be beneficial even with the very small domains of this benchmark. Weakening improves on almost all examples for model 2, except first fail with incremental explanation. Although it requires more search it is almost always faster and sometimes more robust.

On these problems, although the decompositions propagate slower than the learning propagators, they solve approximately as many instances (and in some cases, several more). This appears to be due to a lack of re-usable explanations from the `global_cardinality` coverage constraints; the intermediate variables introduced by the Tseitin decomposition turn out to be critical to proving unsatisfiability of these instances.

Cardinality Constraints

As observed above, while for many of these constraints using the direct MDD propagators is beneficial, they seem to perform quite poorly on cardinality and `global_cardinality` constraints. For a cardinality constraint, there is no ordering dependence amongst the variables.

To prove that $\sum_{i\in[0,n-1]} x_i \geq k$ is unsatisfiable, we must prove that for any subset of size k, at least one x_i will be false. When using only external variables in explanations, we must prove that each of the $\binom{|X|}{k}$ subsets is unsatisfiable separately.

[3]The experiments from Brand et al. [2007] are run on a Pentium 4 3.20GHz machine with 1Gb RAM.

Search	dec$_{tse}$	fails	dec$_{dc}$	fails
Seq	27.72	7320.72	1.64	80.33
	37 / 29.06	7839.46	49 / 6.04	341.86
FF	18.59	8482.04	35.29	3510.08
	45 / 28.30	9419.73	30 / 33.67	3160.20
VSIDS	48.04	32816.63	5.67	2479.00
	29 / 47.54	46032.17	73 / 20.25	7364.01

Search	base	fails	ip	fails	weak	fails
Seq	0.63	330.64	**0.45**	330.75	0.53	435.36
	49 / 7.52	6059.65	**49 / 5.63**	6449.96	49 / 6.70	7212.41
FF	3.00	4126.67	**1.81**	4090.25	11.76	13043.75
	41 / 25.76	19191.93	41 / 12.68	19721.29	39 / 28.97	27757.59
VSIDS	2.15	1513.85	1.00	1941.26	**0.96**	1318.30
	73 / 10.84	5981.86	71 / 5.10	4879.04	73 / 10.95	6858.56

Search	ip+e	fails	ip+ew	fails
Seq	**0.45**	527.89	0.55	664.97
	47 / 1.21	1191.04	47 / 3.98	2134.45
FF	2.87	5493.79	10.88	6211.62
	38 / 10.78	14856.13	37 / 18.59	16945.57
VSIDS	1.90	4681.74	2.23	4406.41
	72 / 6.54	6647.18	**73 / 7.10**	6838.68

Table 3.7: Nurse sequencing, multi-sequence constraints, model 1.

Search	dec$_{tse}$	fails	dec$_{dc}$	fails
Seq	27.46	50336.09	0.67	85.32
	47 / 27.46	50336.09	**73 / 8.24**	1757.15
FF	11.90	12347.71	18.79	5677.76
	43 / 18.44	28476.44	32 / 21.49	5912.94
VSIDS	64.96	130351.85	1.93	2565.57
	59 / 67.47	146394.32	86 / 9.49	9746.43

Search	base	fails	ip	fails	ip+w	fails
Seq	0.33	134.62	**0.26**	186.34	0.27	105.72
	70 / 3.26	4956.57	70 / 1.85	5011.67	70 / 1.31	3021.19
FF	0.35	446.67	**0.23**	338.14	0.29	645.76
	41 / 14.84	19189.46	41 / 6.98	15314.46	41 / 15.23	27240.39
VSIDS	0.52	512.09	0.39	652.13	**0.33**	462.50
	86 / 4.44	3421.27	87 / 6.02	8687.74	**87 / 4.10**	5821.41

Search	ip+e	fails	ip+ew	fails
Seq	**0.26**	146.60	0.28	147.11
	69 / 4.17	5669.68	69 / 5.70	10942.51
FF	0.26	487.05	0.35	869.48
	37 / 0.54	1332.59	39 / 7.14	20791.82
VSIDS	0.63	1706.44	0.42	958.93
	84 / 2.38	4574.18	85 / 0.65	1186.12

Table 3.8: Nurse sequencing, multi-sequence constraints, model 2

However, BDD-based decompositions will have O($|X|k$) variables representing the subformula $x_{j,m} \leftrightarrow \sum_{i \in [j,n-1]} \geq m$.

Search	$\texttt{dec}_{tse}$	fails	$\texttt{dec}_{dc}$	fails
Seq	11.80	5829.35	13.71	3312.96
	53 / 14.84	6905.91	52 / 13.71	3312.96
FF	26.61	17069.40	41.62	13679.20
	41 / 31.34	25962.78	31 / 46.30	18364.61
VSIDS	90.21	47841.00	6.88	4307.39
	47 / 92.86	63109.64	87 / 15.45	9527.08

Search	base	fails	ip	fails	ip+w	fails
Seq	1.93	2569.54	**0.78**	2569.52	0.97	2867.52
	59 / 11.49	15209.07	**59 / 8.15**	16476.14	58 / 8.25	12604.74
FF	15.37	19275.84	4.39	14776.28	3.62	11057.56
	41 / 39.08	53043.24	40 / 17.37	46627.28	43 / 20.77	41139.37
VSIDS	2.79	8108.16	**0.78**	3595.05	1.69	6971.91
	89 / 4.91	15260.85	88 / 2.95	11894.49	**89 / 3.89**	13432.61

Search	ip+e	fails	ip+ew	fails
Seq	0.83	2906.08	0.80	2746.98
	58 / 4.61	11179.53	57 / 4.80	8355.67
FF	3.38	10949.88	**2.99**	9540.04
	42 / 17.58	42307.26	**45 / 15.57**	40421.04
VSIDS	6.89	13072.77	2.69	7032.84
	88 / 7.14	21284.11	88 / 2.31	8433.73

Table 3.9: Nurse sequencing, multi-sequence constraints with decomposed cardinality, model 1.

An alternative approach to representing cardinality constraints is the decomposition described in Abío et al. [2011]. As with the conventional Tseitin transformation, we first build the BDD, then introduce a fresh literal for each internal node in the BDD. However, since cardinality constraints are monotone, for each node $n(v, t, f)$, we only need to introduce the following clauses:

$$\neg [\![t]\!] \rightarrow \neg [\![n]\!]$$

$$\neg [\![f]\!] \wedge \neg v \rightarrow \neg [\![n]\!]$$

We constructed a hybrid model, where each MDD-based `global_cardinality` constraint was replaced with a set of these decomposed cardinality constraints. Tables 3.9 and 3.10 give results for this revised model for the nurse scheduling problem.

In all but one case, the model with decomposed cardinality constraints performs at least as well as the pure MDD model. In many cases, and particularly those using a VSIDS search strategy, the decomposed model solved at least 10 more instances within the time limit than the original model.

Search	$\texttt{dec}_{tse}$	fails	$\texttt{dec}_{dc}$	fails
Seq	2.27	6317.41	4.52	4426.08
	67 / 3.39	9382.16	70 / 4.31	4215.79
FF	9.92	26579.59	18.34	15589.24
	43 / 11.43	31169.86	31 / 28.16	26006.00
VSIDS	60.19	142594.46	2.55	6778.20
	95 / 61.11	144397.73	97 / 3.86	9873.85

Search	base	fails	ip	fails	ip+w	fails
Seq	1.38	4600.08	0.76	4600.00	**0.32**	2709.27
	74 / 7.02	12032.43	75 / 7.87	14762.08	**79 / 7.93**	14581.53
FF	18.01	24512.06	2.49	9614.76	2.32	8041.00
	43 / 29.53	60819.26	46 / 25.41	62212.98	48 / 13.54	39973.25
VSIDS	0.64	4230.98	1.13	6850.29	0.44	4570.63
	98 / 1.40	7862.24	97 / 1.17	7529.79	99 / 2.88	14351.43

Search	ip+e	fails	ip+ew	fails
Seq	0.66	4340.62	0.37	3028.14
	72 / 4.33	9021.74	77 / 11.67	18447.27
FF	**1.53**	6223.18	2.02	7620.06
	41 / 15.38	45446.54	**54 / 29.92**	74996.72
VSIDS	0.45	4112.98	**0.36**	3727.76
	98 / 3.16	13384.15	**99 / 0.81**	7173.22

Table 3.10: Nurse sequencing, multi-sequence constraints with decomposed cardinality, model 2.

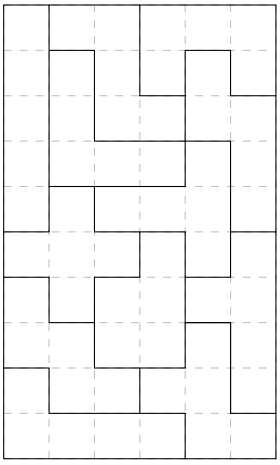


Figure 3.11: A solution for the 6×10 Pentomino problem.

3.3.3 Pentominoes

For another set of experiments we consider the pentomino problems, which involve placing a set of 5-block shapes in such a way as to fill a given area. The most common variant of the puzzle is to place 12 of these shapes, which may be rotated or reflected, inside a rectangle with an area of 60 units (one of $3 \times 20, 4 \times 15, 5 \times$

	Without learning					With learning			
Size	gecode	fails	base	ip	fails	$\texttt{dec}_{tse}$	fails	$\texttt{dec}_{dc}$	fails
3x20	19.87	36K	33.64	7.63	36K	5.65	21K	103.99	11K
4x15	338.60	649K	546.86	131.41	651K	212.86	333K	—	—
5x12	—	—	—	519.81	2482K	—	—	—	—
6x10	—	—	—	—	—	—	—	—	—
$\sum$	—	—	—	—	—	—	—	—	—

	With learning								
Size	base	ip	fails	ip+w	fails	ip+e	fails	ip+ew	fails
3x20	9.68	4.99	11K	4.02	10K	3.26	10K	**2.57**	9K
4x15	300.91	172.90	380K	165.41	382K	128.85	379K	**97.58**	353K
5x12	—	730.34	1571K	761.40	1657K	590.18	1656K	**497.88**	1586K
6x10	—	—	—	—	—	—	—	—	—
$\sum$	—	—	—	—	—	—	—	—	—

Table 3.11: Time to find all solutions for pentominoes, with FD model and no symmetry breaking.

$12, 6 \times 10$). A model of the pentomino problems using the `regular` constraint is described in Lagerkvist [2008].

Tables 3.11 to 3.14 compare the performance of our explaining MDD propagators with the conventional `regular` constraints of Lagerkvist [2008] (implemented in GECODE 3.1.0). We tested both the finite domain and Boolean models of the pentomino problems, using a time limit of 20 minutes. All approaches use the same search strategy as in Lagerkvist and Pesant [2008] which is slightly different to the default strategy in the Gecode model.

First incremental propagation is always better than non-incremental propagation, and indeed the difference without learning is quite substantial. Next, we find that incremental explanations provide a significant improvement over non-incremental explanations. On the finite-domain model, weak explanations provide a significant improvement in propagation speed with relatively little increase in search – the combination of the two techniques is on all instances the best algorithm, and significantly outperforms the conventional `regular` constraints. While the base solver is not as good as GECODE on the FD model, once we add learning and weakening our results are substantially better. The decompositions once again perform uniformly worse than the incremental propagators. Note that all the variants of the learning solver have very similar failure counts; the difference in performance is due primarily to faster propagation and explanation algorithms.

Without learning						With learning			
Size	gecode	fails	base	ip	fails	$\texttt{dec}_{tse}$	fails	$\texttt{dec}_{dc}$	fails
3x20	5.22	9K	9.20	2.29	9K	3.50	14K	45.75	5K
4x15	131.78	239K	209.41	50.70	239K	115.30	193K	—	—
5x12	464.75	788K	653.59	173.04	789K	730.27	925K	—	—
6x10	—	—	—	447.97	1840K	—	—	—	—
$\sum$	—	—	—	674.00	2877223	—	—	—	—

With learning									
Size	base	ip	fails	ip+w	fails	ip+e	fails	ip+ew	fails
3x20	4.84	2.62	4K	2.34	4K	1.58	4K	**1.35**	4K
4x15	106.60	62.70	128K	49.64	116K	45.23	129K	**34.76**	121K
5x12	379.78	233.76	465K	205.10	444K	183.23	481K	**148.44**	448K
6x10	949.09	558.72	1176K	511.00	1106K	480.99	1235K	**390.51**	1111K
$\sum$	1440.31	857.80	1774K	768.08	1670K	711.03	1850K	**575.06**	1684K

Table 3.12: Time to find all solutions for pentominoes, with FD model and symmetries removed.

a	b	l	a	t	e
m	a	i	d	e	n
a	b	s	e	n	t
s	e	l	l	e	r
s	l	e	e	t	y

Figure 3.12: A solution to the 5×6 `crossword` problem with the `lex` dictionary, used in [Cheng and Yap, 2010].

3.3.4 Other Problems

MDDs have previously been considered [Cheng and Yap, 2010] as a way of encoding `table` constraints. Where the constraints we have considered thus far constrain many variables with small domains, extensional tables are generally over few variables with larger domains, and less sharing of nodes. We now discuss several benchmarks commonly used for testing extensional table constraints.

Crossword

The `crossword` problems used in [Cheng and Yap, 2010] define an $m \times n$ grid of variables, and each row and column is constrained to be an element of a specified dictionary. These problems are fairly pathological with respect to learning MDD propagators; the produced MDDs are extremely *wide* – some propagators have an average width of ∼2700 nodes, rather than 10–30 for most of the `regular` constraints we have considered – and have very little sharing between nodes. As such, explanation generation is very expensive, and explanations generated are

Without learning						With learning			
Size	gecode	fails	base	ip	fails	$\texttt{dec}_{tse}$	fails	$\texttt{dec}_{dc}$	fails
3x20	5.32	36K	2.66	1.82	36K	1.70	11K	2.52	7K
4x15	88.68	649K	47.67	32.50	651K	64.71	363518	110.71	202K
5x12	346.42	2478K	185.73	127.98	2482K	366.80	1931K	710.71	901K
6x10	907.59	5998K	477.58	325.84	6008K	—	—	—	—
$\sum$	1348.01	9161K	713.64	488.14	9176K	—	—	—	—

With learning					
Size	base	ip	fails	ip+e	fails
3x20	1.00	0.78	9K	**0.72**	9K
4x15	41.17	33.41	398K	**28.27**	424K
5x12	176.49	144.59	1677K	**120.74**	1749K
6x10	437.29	353.48	4089K	**299.71**	4277K
$\sum$	655.95	532.26	6173K	**449.44**	6460K

Table 3.13: Time to find all solutions for pentominoes, with Boolean model and no symmetry breaking.

Without learning						With learning			
Size	gecode	fails	base	ip	fails	$\texttt{dec}_{tse}$	fails	$\texttt{dec}_{dc}$	fails
3x20	1.39	874K	0.96	0.58	9K	0.86	6K	1.12	3K
4x15	33.77	239K	22.46	13.40	239K	30.33	157K	38.35	63K
5x12	113.62	788K	72.96	45.26	789K	138.09	677K	212.05	259K
6x10	288.91	1838K	179.21	112.14	1840K	430.94	1960K	748.13	703K
$\sum$	437.69	2873K	275.59	171.38	2877K	600.22	2800K	999.65	1028K

With learning					
Size	base	ip	fails	ip+e	fails
3x20	0.51	0.41	4K	**0.33**	4K
4x15	13.87	10.38	117K	**9.65**	129K
5x12	52.02	41.03	442K	**36.14**	462K
6x10	128.40	100.56	1088K	**89.15**	1136K
$\sum$	194.80	152.38	1651K	**135.27**	1730K

Table 3.14: Time to find all solutions for pentominoes, with Boolean model and symmetries removed.

very large and not reusable – ∼1000 literals/nogood rather than 10–100 for other problems in this chapter. Indeed, the number of backtracks for the learning solver on these instances is identical to the solver without learning; none of the generated nogoods ever prunes the search space. Since maintaining these nogoods is pure overhead, the learning solver is up to an order of magnitude slower than the non-learning solver on some `crossword` instances.

Renault

The Renault–Megane car configuration problems [Amilhastre et al., 2002, van Dongen et al., 2008] are another benchmark commonly used to test extensional con-

straint representations. While constraints in the `renault` instances have a large extensional representation, they – in contrast to the `crossword` dictionaries – produce very compact MDD representations (the largest having ∼240 nodes). These MDDs are small enough that the solver is driven immediately to a solution, independent of the choice of explanation algorithm. However, when run without learning, the variable domains are large enough that the default sequential search strategy takes more than 10 minutes on all instances.

3.4 Conclusion

In this chapter we have defined an MDD propagation with explanation. We introduced an incremental propagation algorithm for MDDs using watch flags, and an incremental approach to explaining propagation for MDD constraints. The incremental propagation algorithm is significantly better than approaches starting from the root, at least on the kind of MDDs with large arity and low width appearing in the problems we study. Incremental explanation often improves on non-incremental explanation particularly when using activity based search where the non-minimality of the resulting explanations is not so critical. The resulting system provides the state-of-the-art solution to nonogram puzzles.

4

Decomposable Negation Normal Form

As illustrated in chapters 2 and 3, MDDs can be used to construct propagators for arbitrary finite-domain constraints. While MDDs can produce concise representations (and hence efficient propagators) for a variety of useful constraints, for constraints without a convenient sequential structure – such as the `grammar` constraint – the size of the MDD may be exponential in the number of variables.

In these cases, we would like a method for representing constraints which can concisely encode a larger class of functions, while still permitting efficient analysis. In this chapter we consider *Smooth, Decomposable Negation Normal Form* (`s-DNNF`) [Darwiche, 2001], which can produce polynomial encodings of a wider range of functions (notably including context-free grammars, or `grammar` constraints) while still permitting linear-time testing of satisfiability. Given the recent development of *sentential decision diagrams* [Darwiche, 2011], which can be automatically constructed in a similar fashion to BDDs, it seems likely that `s-DNNF` will be an increasingly convenient constraint representation.

In this chapter we investigate how to construct explaining propagators for `s-DNNF` circuits, and compare them with the only existing approach we are aware of for handling such circuits in constraint programming systems, decomposing the circuits using a form of Tseitin transformation [Tseitin, 1968].

The remainder of this chapter is structured as follows. In Section 4.1, we describe the `s-DNNF` circuit representation for constraints. Then in Section 4.2, we outline existing decomposition-based methods used for using `s-DNNF` constraints.

In Section 4.3, we give algorithms for performing propagation on constraints expressed as s−DNNF circuits; then in Section 4.4, we describe both minimal and greedy algorithms for explanation construction. In Section 4.6 we compare these methods on GRAMMAR-based models for shift scheduling problems, as well as randomly generated forklift scheduling problems. Finally, we conclude in Section 4.7.

4.1 Smooth Decomposable Negation Normal Form

A circuit in *Negation Normal Form (NNF)* is a propositional formula using connectives $\{\wedge, \vee, \neg\}$, such that $\neg$ is only applied to variables. While NNF normally defines functions over Boolean variables, it can be readily adapted to non-binary domains by permitting leaves of the form $[\![x_i = v_j]\!]$ for each value in $D(x_i)$. Hence the Boolean variable b is represented by leaves $[\![b = 0]\!]$ and $[\![b = 1]\!]$ corresponding directly to $\neg b$ and b. As we are concerned with constraints over finite-domain variables, we consider circuits in this class of *multi-valued* NNF.

The rest of the presentation ignores bounds literals $[\![x_i \leq v_j]\!]$. We can extend the algorithms herein to directly support bounds literals $[\![x_i \leq v_j]\!]$ but it considerably complicates their presentation. They (and their negations) can of course be represented with disjunctive nodes e.g. $\left[\!\left[\vee_{v' \leq v_j} [\![x_i = v']\!]\right]\!\right]$.

We shall use vars to denote the set of variables involved in an NNF circuit, defined as:

$$\begin{aligned} \mathsf{vars}([\![x_i = v_j]\!]) &= \{x_i\} \\ \mathsf{vars}([\![\bigvee N]\!]) &= \bigcup_{n' \in N} \mathsf{vars}(n') \\ \mathsf{vars}([\![\bigwedge N]\!]) &= \bigcup_{n' \in N} \mathsf{vars}(n') \end{aligned}$$

It is difficult to analyse NNF circuits in the general case – even determining satisfiability is NP-hard. However, restricted subclasses of NNF, described in Darwiche and Marquis [2002], permit more efficient analysis. In this chapter, we are concerned with *decomposability* and *smoothness*.

Decomposability requires that for any node of the form $\phi = [\![\bigwedge N]\!]$, any two children n_i, n_j must satisfy $\mathsf{vars}(n_i) \cap \mathsf{vars}(n_j) = \emptyset$ – that is, children of a conjunction cannot have any shared dependencies. Similarly, *smoothness* requires that for any node $\phi = [\![\bigvee N]\!]$, any two children n_i, n_j must satisfy $\mathsf{vars}(n_i) = \mathsf{vars}(n_j)$.

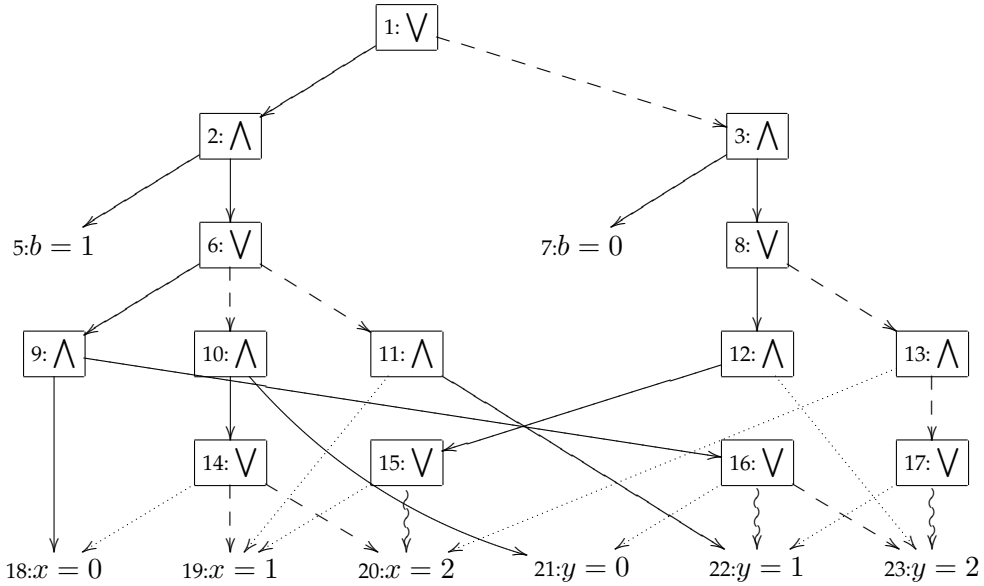


Figure 4.1: An example s−DNNF graph for $b \leftrightarrow x + y \leq 2$

.

Smooth Decomposable Negation Normal Form (s−DNNF) is the set of circuits of the form

$$
\begin{aligned}
\phi \ &\rightarrow [\![x_i = v_j]\!] \\
&\mid [\![\bigvee N]\!] \textbf{ iff } \forall_{n_i,n_j \in N, n_i \neq n_j} \ \mathsf{vars}(n_i) = \mathsf{vars}(n_j) \\
&\mid [\![\bigwedge N]\!] \textbf{ iff } \forall_{n_i,n_j \in N, n_i \neq n_j} \ \mathsf{vars}(n_i) \cap \mathsf{vars}(n_j) = \emptyset
\end{aligned}
$$

We represent a s−DNNF circuit as a graph G with literal leaves, and-nodes and or-nodes with children their subformulae. We assume $G.root$ is the root of the graph and $n.parents$ are the parent nodes of a node n.

Example 4.1. *An s−DNNF for the constraint $b \leftrightarrow x + y \leq 2$ where $D_{init}(b) = \{0, 1\}$ and $D_{init}(x) = D_{init}(y) = \{0, 1, 2\}$ is shown in Figure 4.1. Ignore the different styles of edges for now. It is smooth, e.g. all of nodes 9,10,11,12,13 have* vars $= \{x, y\}$*, and it is decomposable, e.g. for each such node the left child has* vars $= \{x\}$ *and the right child has* vars $= \{y\}$. □

	$\texttt{decomp}_{\texttt{tt}}$	$\texttt{decomp}_{\texttt{dc}}$
	$\langle 1 \rangle$	
1:	$\neg\langle 1\rangle \vee \langle 2\rangle \vee \langle 3\rangle$	$\emptyset$
2:	$(\neg\langle 2\rangle \vee [\![b = 1]\!]) \wedge (\neg\langle 2\rangle \vee \langle 6\rangle)$	$\neg\langle 2\rangle \vee \langle 1\rangle$
3:	$(\neg\langle 3\rangle \vee [\![b = 0]\!]) \wedge (\neg\langle 3\rangle \vee \langle 8\rangle)$	$\neg\langle 3\rangle \vee \langle 1\rangle$
5:	$\emptyset$	$\neg [\![b = 1]\!] \vee \langle 2\rangle$
6:	$\neg\langle 6\rangle \vee \langle 9\rangle \vee \langle 10\rangle \vee \langle 11\rangle$	$\neg\langle 6\rangle \vee \langle 2\rangle$
7:	$\emptyset$	$\neg [\![b = 0]\!] \vee \langle 3\rangle$
8:	$\neg\langle 6\rangle \vee \langle 9\rangle \vee \langle 10\rangle \vee \langle 11\rangle$	$\neg\langle 6\rangle \vee \langle 2\rangle$
	$\dots$	$\dots$
18:	$\emptyset$	$\neg [\![x = 0]\!] \vee \langle 9\rangle \vee \langle 14\rangle$
	$\dots$	$\dots$

Table 4.1: Clauses produced by the decomposition of the graph in Fig. 4.1

4.2 DNNF Decomposition

Previous methods for working with these constraints (implicitly in Quimper and Walsh [2006] and explicitly in Jung et al. [2008]) transformed the circuit into a set of clauses by introducing a new Boolean variable for each node.

For each node $n = [\![\mathbf{op}\ N]\!]$, we introduce a new Boolean variable $\langle n \rangle$. We then introduce the following clauses

$$[\![\bigvee N]\!] : \neg\langle n\rangle \vee \bigvee_{n_i \in N} \langle n_i\rangle$$

$$[\![\bigwedge N]\!] : \bigwedge_{n_i \in N} \neg\langle n\rangle \vee \langle n_i\rangle$$

and set the variable $\langle G.root\rangle$ to true.

For the domain consistent encoding, we also introduce for $n \in N \setminus \{G.root\}$:

$$\bigvee_{n_i \in n.parents} \langle n_i\rangle \vee \neg\langle n\rangle$$

Example 4.2. *Consider the graph shown in Figure 4.1. The clauses generated by the decomposition of this constraint are shown in Table 4.1.* $\texttt{decomp}_{\texttt{tt}}$ *gives the clauses generated by the basic encoding.* $\texttt{decomp}_{\texttt{dc}}$ *gives the additional clauses produced by the domain-consistent encoding.* □

```
dnnf_propagate(G, X, D)
    clear_cache()
    reachable := prop_mark(D, G.root)
    if(¬reachable) return false
    supported := prop_collect(G.root)
    for(x_i ∈ X)
        for(v_j ∈ D(x_i))
            if([[x_i = v_j]] ∉ supported)
                infer([[x_i ≠ v_j]])
    return true
```

4.3 DNNF Propagation

In this section, we describe algorithms for propagating constraints encoded as s−DNNF circuits.

4.3.1 Propagation from the root

Consider an s−DNNF circuit G over variables X. dnnf_propagate(G, X, D) enforces domain consistency over G with domain D. It consists of three stages. First, it determines which nodes may be both true and reachable from $G.root$ under the current partial assignment. If the root is not possibly true the propagator fails, there are no solutions. Second it collects in $supported$ which literals $[\![x_i = v_j]\!]$ participate in solutions by being part of nodes that are true and reachable. Third, it propagates that any unsupported literals must be false.

Marking the reachable nodes (prop_mark) simply traverses the s−DNNF circuit marking which nodes are reachable, and storing in a cache whether they may be true ($alive$) given the current domain D. Each node is visited at most once.

Collecting the literals (prop_collect) that appear in solutions simply traverses the s−DNNF circuit restricted to the nodes which are reachable and true, and returns all literals encountered. Each true node is visited at most once.

Example 4.3. *Imagine we are propagating the s−DNNF shown in Figure 4.1 when* $D(b) = \{0\}$ *and* $D(x) = \{2\}$*. The marking stage marks nodes* $\{5, 2, 18, 9, 19, 11\}$ *as dead and the rest alive. The collection visits nodes* 1, 3, 7, 8, *12, 13, 15, 17, 20, 22, 23 and collects* $b = 0$, $x = 2$, $y = 1$ *and* $y = 2$*. Propagation determines that* $y \neq 0$*.* □

```
prop_mark(D, node)
    c := lookup(node)
    if(c ≠ NOTFOUND) return c
    switch(node)
        case([[x_i = v_j]])
            alive := (v_j ∈ D(x_i))
        case([[⋀ N]])
            alive := ⋀_{n'∈N} prop_mark(D, n')
        case([[⋁ N]])
            alive := ⋁_{n'∈N} prop_mark(D, n')
    cache(node, alive)
    return alive
```

```
prop_collect(node)
    c := lookup(node)
    if(c == true)
        % Ensure the node is collected
        % only once.
        cache(node, false)
        switch(node)
            case([[x_i = v_j]])
                supported := {[[x_i = v_j]]}
            case([[ op N]])
                supported := ⋃_{n'∈N} prop_collect(n')
    else
        return {}
```

4.3.2 Incremental propagation

Propagation from the root can be expensive; it must traverse the entire `s-DNNF` circuit each time the propagator executes. In many cases very little will have changed since the last time the propagator was executed. We can instead keep track of which nodes in the `s-DNNF` circuit are reachable and possibly true by just examining the part of the circuit which may have changed starting from leaves which have changed.

inc_propagate$(changes, G, X)$ propagates the `s-DNNF` circuit G over variables X given change to domains $changes$ which are literals $[\![x_i = v_j]\!]$ that have become false since the last propagation. The algorithm maintains for each node whether it is dead: unreachable or false with the current domain, and for each node which parents rely on it to keep them possibly true ($node.watched\_parents$) and for each node which children rely on this node to keep them reachable ($node.watched\_children$). In the first phase the algorithm visits a set of nodes kfb which are "killed from below", i.e. became false because of leaf information. `And` nodes are killed from below if one of their children becomes killed, while `or`-nodes are killed from below if all their children are killed. The first phase also records killed `and`-nodes in kfa ("killed from above") since we have to mark their other children as possibly unreachable. If the root is killed from below the propagator returns failure.

The second phase visits nodes in kfa and determines if they kill child nodes since they are the last alive parent, in which case the child node is added to kfa. A killed literal node ensures that we propagate the negation of the literal.

```
inc_propagate(changes, G, X)
    kfb := changes
    kfa := {}
    % Handle nodes that were killed due to dead children.
    for(node ∈ kfb)
        for(parent ∈ node.watched_parents)
            switch(parent)
                case([[⋀ N]])
                    if(dead[parent]) continue
                    dead[parent] := true
                    parent.killing_child := node % For greedy explanation.
                    kfb ∪:= {parent}
                    kfa := {parent} % Handle other children
                case([[⋁ N]])
                    if(∃n' ∈ N s.t. ¬dead[n'])
                        % A support still remains – update the watches.
                        node.watched_parents \:= {parent}
                        n'.watched_parents ∪:= {parent}
                    else
                        % No supports – kill the node.
                        dead[parent] := true
                        parent.killed_above := false
                        kfb ∪:= {parent}
    if(G.root ∈ kfb) return false
    % Downward pass.
    for(node ∈ kfa)
        switch(node)
            case([[x_i = v_j]])
                infer([[x_i ≠ v_j]])
                continue
        for(child ∈ node.watched_children)
            if(∃n' ∈ child.parents s.t ¬dead[n'])
                node.watched_children \:= {child}
                n'.watched_children ∪:= {child}
            else
                dead[child] := true
                kfa ∪:= {child}
                child.killed_above := true
    return true
```

During propagator construction, $watched\_parents$ and $watched\_children$ are initialised to $\emptyset$. For each node n, we then pick one parent p and add n to $p.watched\_children$ – p is now supporting n from above. For `or`-nodes, we then pick one child c, and add n to $c.watched\_parents$ – since n is satisfiable so long as any child is alive, it must be satisfiable so long as c is not killed. In the case of an `and`-node, however, we must add n to $watched\_parents$ of *each* of its children, as n must be killed if any children die.

Example 4.4. *Imagine we are propagating the* s−DNNF *graph of Figure 4.1. The policy for initial watches is illustrated in Figure 4.1, where edges for initially watched parents are solid or dotted, and edges for initially watched children are solid or dashed.*

Suppose we set $D(x) = \{2\}$*. The changes are* $[\![x = 0]\!]$ *and* $[\![x = 1]\!]$*. Initially* $kfb = \{18, 19\}$*. Then 9 is added to* kfb *and* kfa *with killing child 18, and similarly 11 is added to* kfb *and* kfa *with killing child 19. Because* 6 *is a watched parent of 9, it is examined and the watched parents of 10 is set to 6. In the second phase examining node 9 we set 16 as dead and add it to* kfa*. Examining 11 we look at its child 22 and set node 16s watched children to include 22. Examining node 16 we set 10s watched children to include 21, and 17s watched children to include 22. No propagation occurs.*

Now suppose we set $D(b) = \{0\}$*. The changes are* $[\![b = 1]\!]$*. Initially* $kfb = \{5\}$ *and this causes 2 to be added to* kfb *and* kfa *and the killing child set to 5. Examining 2 causes the watched parent of 3 to be set to 1. In the second phase examining 2 causes 6 to be added to* kfa*, which causes 10 to be added to* kfa*, which causes 14 and 21 to be added to* kfa*. Examining 14 adds 20 to the watched children on 15. Examining 21 we propagate that* $y \neq 0$*.* □

4.4 Explaining DNNF Propagation

In this section, we describe several algorithms for computing explanations for s−DNNF circuits. Given the computational cost of computing explanations for s−DNNF circuits, it is beneficial to both compute explanations lazily, and store computed explanations in the clause database (so they can be re-used, rather than recomputed).

4.4.1 Minimal Explanation

The explanation algorithm is similar in concept to that used for BDDs and MDDs. To explain $[\![x \neq v]\!]$ we assume $[\![x = v]\!]$ and hence make the s−DNNF unsatisfiable. A correct explanation is (the negation of) all the values for other variables which are currently false ($varQ$). We then progressively remove assignments (unfix literals) from this explanation while ensuring the constraint as a whole remains unsatisfiable. We are guaranteed to create a *minimal explanation* $\bigwedge_{l \in expln} \neg l \rightarrow [\![x \neq v]\!]$ since removing any literal l' from the $expln$ would mean $G \wedge \bigwedge_{l \in expln - \{l'\}} \neg l \wedge x = v$ is satisfiable. This is essentially the same process used by the ROBUSTXPLAIN algorithm

```
sdnnf_explain(¬ [[x = v]], G, X, D)
    for(x_i ∈ X \ {x}, v_j ∈ D(x_i)) unlock([[x_i = v_j]])
    unlock([[x = v]])
    set_binding(G.root)
    expln := {}
    varQ := {[[x_i = v_j]] | x_i ∈ X \ {x}, v_j ∉ D(x_i)}
    for([[x_i = v_j]] ∈ varQ)
        if(binding[[[x_i = v_j]]]) expln ∪:= {[[x_i = v_j]]}
        else unlock(node)
    return expln
```

of Junker [2004]; however, the s-DNNF structure allows us to unfix assignments directly rather than repeatedly reconstructing subproblems.

Unlike (B/M)DDs, s-DNNF circuits do not have a global variable ordering that can be exploited. As such, we must update the reachability information as we progressively unfix leaf nodes. A node n is considered *binding* if n becoming satisfiable would make the root r satisfiable. $locks[n]$ denotes the number of dead children holding n dead. And nodes $[\![\bigwedge N]\!]$ start with $|N|$ locks while other nodes have 1. If n is *binding* and $locks[n] = 1$, then making any children satisfiable will render r satisfiable.

The sdnnf_explain algorithm initialises locks and then unlocks all nodes which are true for variables other than in the explained literal $[\![x = v]\!]$, and unlocks the explained literal. This represents the state of the current domain D except that we set $D(x) = \{v\}$. All nodes which may be true with the explained literal true will have 0 locks. The algorithm then marks the root as binding using set_binding. If the locks on the node are 1, then set_binding marks any locked children as also binding. The algorithm then examines each literal in $varQ$. If the literal is binding then clearly setting it to true will allow the root to become true, hence it must remain in the explanation. If not it can be removed from the explanation. We unfix the literal or equivalently unlock the node. We chain unlocking up as nodes reach zero locks, we unlock their parent nodes. Any node with just one lock which is binding, then makes its locked children binding.

Example 4.5. *To create a minimal explanation for the propagation of $y \neq 0$ of Example 4.3 we initialize the locks using* init_locks *which sets the locks to 2 for each and node, and 1 for each other node. We unlock the literals which are in the current domain, for variables other that y, that is $b = 0$ and $x = 2$. Unlocking $b = 0$ reduces the locks on 7 to 0, and hence unlocks 3, reducing its locks to 1. Unlocking $x = 2$ reduces the locks on 13 to 1, and 14*

```
init_locks(G)
    for(node ∈ G)
        switch(node)
            case([[⋀ N]])
                locks[node] := |N|
            case([[⋁ N]])
                locks[node] := 1
            case([[x_i = v_j]])
                locks[node] := 1
```

```
unlock(node)
    if(locks[node] == 0) return
    locks[node] -= 1
    if(locks[node] == 0)
        for(parent ∈ node.parents)
            unlock(parent)
    else if(locks[node] == 1 ∧ binding[node])
        for(n' ∈ node.children s.t. locks[n'] > 0)
            set_binding(n')
```

```
set_binding(node)
    if(binding[node]) return
    switch(node)
        case([[ op N]])
            for(n' ∈ N s.t. locks[n'] > 0)
                set_binding(n')
```

and 15 to 0. Unlocking 14 and 15 reduces the locks on 10 and 12 to 1. We then unlock the propagated literal $y = 0$. This reduces the locks on 10 and 16 to 0. Unlocking 16 reduces the locks on 9 to 1. Unlocking 10 causes 6 to unlock which reduces the locks on 2 to 1. We now set the root as binding. Since it has 1 lock we set its children 2 and 3 as binding. Since node 2 has one lock, binding it sets the child 5 as binding, but not 6 (since it has zero locks). Binding 3 has no further effect. Finally traversing $varQ = \{[\![b = 1]\!], [\![x = 0]\!], [\![x = 1]\!]\}$ adds $[\![b = 1]\!]$ to the explanation since it is binding. Since $x = 0$ is not binding it is unlocked, which unlocks 9. Since $x = 1$ is not binding it is unlocked, which sets the locks of 11 to 1 but has no further effect. The explanation is $b \neq 1 \rightarrow y \neq 0$ is minimal. □

4.4.2 Greedy Explanation

Unfortunately, on large circuits, constructing a minimal explanation can be expensive. For these cases, we present a greedy algorithm for constructing valid, but not necessarily minimal, explanations.

This algorithm is shown as sdnnf_greedy_explain. It relies on additional information recorded during execution of inc_prop to record the cause of a node's death, and follows the chain of these actions to construct an explanation. node.killed_above indicates whether the node was killed by death of parents – if true, we add the node's parents to the set of nodes to be explained; otherwise, we add one (in the case of conjunction) or all (for disjunction) children to the explanation queue. If a node n is a conjunction that was killed due to the death of a child, $n.killing\_child$

```
sdnnf_greedy_explain((x ≠ v), G, X)
    explQ := [[x = v]].parents
    expln := {}
    for(node ∈ explQ)
        if(node.killed_above)
            explQ ∪:= node.parents
        else
            switch(node)
                case([[x_i = v_j]])
                    expln ∪:= {[[x_i = v_j]]}
                case([[⋀ N]])
                    explQ ∪:= {node.killing_child}
                case([[⋁ N]])
                    explQ ∪:= {N}
    return expln
```

indicates the child that killed node n – upon explanation, we add this node to the explanation queue.

Example 4.6. *Explaining the propagation $y \neq 0$ of Example 4.4 proceeds as follows. Initially $explQ = \{10, 16\}$. Since 10 was killed from above we add 6 to $explQ$, similarly 16 adds 9. Examining 6 we add 2 since it was killed from above. Examining 9 we add $x = 0$ to $expln$ as the killing child. Examining 2 we add $b = 1$ to $expln$ as the killing child. The explanation is $b \neq 1 \wedge x \neq 0 \rightarrow y \neq 0$. This is clearly not minimal.* □.

4.4.3 Explanation Weakening

Explanations derived from `s-DNNF` circuits can often be very large. This causes overhead in storage and propagation. It can be worthwhile to weaken the explanation in order to make it shorter. This also can help direct propagation down the same paths and hence give more reusable nogoods. Conversely the weaker nogood may be less reusable since it is not as strong.

We can shorten an explanation $\wedge L \rightarrow l$ as follows. Suppose there are at least two literals $\{[\![x_i \neq v]\!], [\![x_i \neq v']\!]\} \subseteq L$. Suppose also that at the time of explanation $D(x_i) = \{v''\}$ (where clearly $v'' \neq v$ and $v'' \neq v'$). We can replace all literals about x_i in L by the literal $[\![x_i = v'']\!]$. This shortens the explanation, but weakens it. This is analogous to the MDD explanation weakening described in the previous chapter.

For greedy explanation, we perform weakening as a postprocess. However for minimal explanation, weakening as a postprocess can result in explanations that are far from minimal. Hence we need to adjust the explanation algorithm so that for a variable x_i, we first count the number of nodes $[\![x_i = v_j]\!]$ that are binding. If

in the current state $D(x_i) = \{v'\}$ and there are at least 2 binding nodes we add $[\![x_i = v']\!]$ to the explanation and progress to x_{i+1}; otherwise, we process the nodes as usual.

Example 4.7. *Consider again the constraint given in Example 4.3. Assume we set* $D(b) = \{1\}$ *and* $D(x) = 2$*. This kills* $[\![x = 0]\!]$, $[\![x = 1]\!]$ *and* $[\![b = 0]\!]$ *– we then run* inc_propagate, *and discover that* $y \neq 1$ *and* $y \neq 2$*.*

Suppose an explanation is requested for $y \neq 1$*. We start running* sdnnf_explain, *determining the set of binding and locked nodes. At this point, nodes 18 and 19 are both binding, so* $[\![x \neq 0]\!]$ *and* $[\![x \neq 1]\!]$ *are both added to the explanation, giving* $[\![x \neq 0]\!] \wedge [\![x \neq 1]\!] \wedge [\![b \neq 0]\!] \rightarrow [\![y \neq 1]\!]$*.*

However, as $D(x) = \{2\}$ *at the time of explanation, we can instead replace* $[\![x \neq 0]\!] \wedge [\![y \neq 1]\!]$ *with* $[\![x = 2]\!]$*, giving an explanation* $[\![x = 2]\!] \wedge [\![b \neq 0]\!] \rightarrow [\![y \neq 1]\!]$*.* □

4.5 Relationship between MDD and s−DNNF algorithms

The s−DNNF propagation algorithms are very similar to the corresponding algorithms for MDDs. If an MDD is converted into a corresponding s−DNNF circuit (by representing each edge (x, v_i, s, d) explicitly as $([\![x = v_i]\!] \wedge d)$) the non-incremental propagation algorithm will behave exactly as for the MDD. The incremental propagation algorithm also behaves similarly, however propagates slightly slower (as an edge in an MDD has at most one parent; whereas and-nodes in a s−DNNF circuit may have arbitrarily many).

Although it operates in a different fashion, the minimal explanation algorithm – assuming it unfixes leaves in increasing order – will generate the same explanation as the corresponding MDD algorithm, as it follows the same per-variable progressive relaxation process. The minimal explanation algorithm given in this chapter is more flexible, as variables can be unfixed in any order; however, what constitutes a *good* ordering for constructing explanations (for either MDDs or s−DNNFs) remains an open question.

The incremental explanation algorithm given in Chapter 3 operates level-by-level to reduce the size of generated explanations; the greedy algorithm given in this chapter does not have this additional information, and will often produce larger explanations. It may be possible to take advantage of the decomposable

nature of the s-DNNF to construct an algorithm that behaves similarly to the incremental explanation for MDDs; however, we have not explored this in detail.

Explanation weakening operates in an identical fashion for both MDDs and s-DNNFs; however, as mentioned above, the greedy algorithm does not keep track of sufficient global information to perform inline weakening, so performs weakening as a postprocess.

4.6 Experimental Results

Experiments were conducted on a 3.00GHz Core2 Duo with 2 GB of RAM running Ubuntu GNU/Linux 8.10. The propagators were implemented in CHUFFED, a state-of-the-art lazy clause generation [Ohrimenko et al., 2009] solver. All experiments were run with a 1 hour time limit.

We consider two problems that involve grammar constraints that can be expressed using s-DNNF circuits. For the experiments, decomp denotes propagation using the domain consistent decomposition described in Section 4.2 (which was slightly better than the simpler decomposition), base denotes propagation from the root and minimal explanations, ip denotes incremental propagation and minimal explanations, +g denotes greedy explanations and +w denotes explanation weakening.

4.6.1 Shift Scheduling

Shift scheduling, a problem introduced in Demassey et al. [2006], allocates n workers to shifts such that (a) each of k activities has a minimum number of workers scheduled at any given time, and (b) the overall cost of the schedule is minimized, without violating any of the additional constraints:

- An employee must work on a task (A_i) for at least one hour, and cannot switch tasks without a break (b).
- A part-time employee (P) must work between 3 and 5.75 hours, plus a 15 minute break.
- A full-time employee (F) must work between 6 and 8 hours, plus 1 hour for lunch (L), and 15 minute breaks before and after.

- An employee can only be rostered while the business is open.

These constraints can be formulated as a `grammar` constraint as follows:

$$
\begin{array}{lllllll}
S & \rightarrow & RP^{[13,24]}R \mid RF^{[30,38]}R & & & & \\
F & \rightarrow & PLP & & P & \rightarrow & WbW \\
W & \rightarrow & A_i^{[4,\ldots]} & & A_i & \rightarrow & a_iA_i \mid a_i \\
L & \rightarrow & llll & & R & \rightarrow & rR \mid r
\end{array}
$$

This `grammar` constraint can be converted into `s-DNNF` as described in Quimper and Walsh [2006]. Note that some of the productions for P, F and A_i are annotated with restricted intervals – while this is no longer strictly context-free, it can be integrated into the graph construction with no additional cost.

The coverage constraints and objective function are implemented using the monotone BDD decomposition described in Abío et al. [2011].

Table 4.2 compares our propagation algorithms versus the domain consistent decomposition [Jung et al., 2008] on the shift scheduling examples of Quimper and Walsh [2006]. Instances $(2, 2, 10)$ and $(2, 4, 11)$ are omitted, as no solvers proved the optimum within the time limit. Generally any of the direct propagation approaches require less search than a decomposition based approach. This is slightly surprising since the decomposition has a richer language to learn nogoods on. But it accords with earlier results for BDD propagation; the Tseitin literals tend to confuse activity based search making it less effective. The non-incremental propagator `base` is too expensive, but once we have incremental propagation (`ip`) all methods beat the decomposition. Clearly incremental explanation is not so vital to the execution time as incremental propagation, which makes sense since we only explain on demand, so it is much less frequent than propagation. Both weakening and greedy explanations increase the search space, but only weakening pays off in terms of execution time.

4.6.2 Forklift Scheduling

As noted in Katsirelos et al. [2009], the shift scheduling problem can be more naturally (and efficiently) represented as an NFA. However, for other grammar constraints, the corresponding NFA can (unsurprisingly) be exponential in size relative to the arity.

In order to evaluate these methods on grammars which do not admit a tractable regular encoding, we present the *forklift scheduling problem*

A forklift scheduling problem is a tuple (N, I, C), where N is the number of stations, I is a set of items and C is a cost for each action. Each item $(i, source, dest) \in I$ must be moved from station $source$ to station $dest$. These objects must be moved using a forklift. The possible actions are:

$move_j$ Move the forklift to station j.

$load_i$ Shift item i from the current station onto the forklift tray.

$unload_i$ Unload item i from the top of the forklift tray at the current station.

$idle$ Do nothing.

Items may be loaded and unloaded at any number of intermediate stations, however they must be unloaded in a last-in first-out (LIFO) order.

The LIFO behaviour of the forklift can be modelled with the grammar:

$$\begin{aligned} S &\rightarrow W \mid WI \\ W &\rightarrow WW \\ &\mid move_j \\ &\mid load_i\ W\ unload_i \\ I &\rightarrow idle\ I \mid idle \end{aligned}$$

Note that this grammar does not prevent item i from being loaded multiple times, or enforce that the item must be moved from $source$ to $dest$. To enforce these constraints, we define a DFA for item $(i, source, dest)$ with 3 states for each station:

$q_{k,O}$ Item at station k, forklift at another station.

$q_{k,U}$ Forklift and item both at station k, but not loaded.

$q_{k,L}$ Item on forklift, both at station k.

With start state $q_{source,O}$ and accept states $\{q_{dest,O}, q_{dest,U}\}$. We define the transition function as follows (where $\perp$ represents an error state):

δ	$move_k$	$move_j, j \neq k$	$load_i$	$load_j, j \neq i$	$unload_i$	$unload_j, j \neq i$
$q_{k,O}$	$q_{k,U}$	$q_{k,O}$	$\perp$	$q_{k,O}$	$\perp$	$q_{k,O}$
$q_{k,U}$	$q_{k,U}$	$q_{k,O}$	$q_{k,L}$	$q_{k,U}$	$\perp$	$q_{k,U}$
$q_{k,L}$	$q_{k,L}$	$q_{j,L}$	$\perp$	$q_{k,L}$	$q_{k,U}$	$q_{k,L}$

A `regular` constraint (which is transformed into an MDD) is used to encode the DFA for each item.

Experiments with forklift sequencing use randomly generated instances with cost 1 for $load_j$ and $unload_j$, and cost 3 for $move_j$. The instance n-i-v has n stations and i items, with a planning horizon of v. The instances are available at `ww2.cs.mu.oz.au/~ggange/forklift`.

The results for forklift scheduling are shown in Table 4.3. They differ somewhat for those for shift scheduling. Here the `base` propagator has no search advantage over the decomposition and is always worse, presumably because the interaction with the DFA side constraints is more complex, which gives more scope for the decomposition to use its intermediate literals in learning. Incremental propagation `ip` is similar in performance to the decomposition. It requires substantially less search than `base` presumably because the order of propagation is more closely tied to the structure of `s-DNNF` circuit, and this creates more reusable nogoods. For forklift scheduling weakening both dramatically reduces search and time, and greedy explanation has a synergistic effect with weakening. The best version `ip+gw` is significantly better than the decomposition approach.

4.7 Conclusion

In this chapter we have defined an s-DNNF propagator with explanation. We define non-incremental and incremental propagation algorithms for s-DNNF circuits, as well as minimal and greedy approaches to explaining the propagations. The incremental propagation algorithm is significantly better than non-incremental approach on our example problems. Greedy explanation usually improves on non-incremental explanation, and weakening explanations to make them shorter is usually worthwhile. The resulting system provides state-of-the-art solutions to problems encoded using grammar constraints.

Table 4.2: Comparison of different methods on shift scheduling problems.

Inst.	decomp		base		ip		ip+w		ip+g		ip+gw	
	time	fails	time	fails	time	fails	time	fails	time	fails	time	fails
1,2,4	9.58	21284	17.38	28603	6.89	18041	9.05	26123	**2.59**	7827	6.70	14834
1,3,6	41.28	73445	96.47	99494	44.11	96801	56.32	103588	**39.77**	115166	80.01	128179
1,4,6	18.70	23250	7.41	9331	3.08	6054	2.74	5758	**1.27**	4234	4.04	9406
1,5,5	5.14	17179	3.26	4871	2.25	8820	3.20	15253	1.72	9939	**1.16**	5875
1,6,6	2.11	3960	1.39	1275	**0.88**	2551	1.12	3293	1.46	5806	0.97	3428
1,7,8	84.48	124226	159.16	273478	50.68	99574	**27.78**	85722	90.92	262880	106.09	250338
1,8,3	1.44	5872	5.37	8888	2.74	6083	2.53	5974	**0.47**	1599	1.02	3216
1,10,9	270.98	373982	1886.15	2389076	309.33	682210	**75.39**	158492	790.55	1802971	170.42	415286
2,1,5	0.37	1217	0.50	653	0.24	221	0.50	1405	**0.19**	710	0.22	624
2,3,6	240.14	162671	136.88	94966	195.79	181709	158.07	153738	**83.65**	159623	87.43	89192
2,5,4	95.90	160104	70.44	72447	36.50	74236	**21.28**	39374	87.26	186018	206.94	360892
2,6,5	99.20	130621	154.47	127314	116.23	163864	123.29	199502	214.24	380586	**64.26**	87175
2,8,5	58.67	136001	253.70	294527	63.53	118504	**38.83**	87444	116.11	221235	113.11	168101
2,9,3	13.61	37792	31.62	41817	**13.21**	28161	14.71	29910	32.67	74192	14.81	23530
2,10,8	590.73	507418	325.27	224429	**97.09**	133974	110.78	159988	162.03	224753	293.49	389813
Geom.	25.21	45445.09	35.46	40927.05	16.12	30816.80	**14.77**	32380.06	16.61	44284.41	17.79	36937.70

Table 4.3: Comparison of different methods on a forklift scheduling problems.

Inst.	decomp		base		ip		ip+w		ip+g		ip+gw	
	time	fails	time	fails	time	fails	time	fails	time	fails	time	fails
3-4-14	**0.58**	4962	2.00	4966	1.52	5912	1.30	3820	1.00	6069	0.80	4392
3-5-16	10.98	42421	46.19	53789	35.40	45486	15.32	28641	22.72	42023	**9.19**	30219
3-6-18	318.55	492147	687.69	611773	380.09	458177	223.06	289221	275.31	454268	**124.10**	279207
4-5-17	36.60	83241	142.77	146131	77.52	99027	43.94	72511	60.75	112160	**20.42**	53643
4-6-18	358.47	587458	704.20	643074	379.09	437797	251.67	331946	410.26	719219	**124.39**	283560
4-7-20	—	—	—	—	—	—	3535.74	3640783	—	—	**1858.79**	3057492
5-6-20	1821.55	2514119	—	—	—	—	1922.73	1894107	2521.49	3374187	**1220.28**	1893025
Geom.	—	—	—	—	—	—	118.80	176102.11	—	—	**65.65**	164520.95

Part II

Combinatorial Optimization for Document Composition and Diagram Layout

A wide range of document layout problems involve arranging a set of document elements on a (generally bounded) canvas, subject to various constraints amongst elements, and between elements and the page. While they differ in concrete constraints and the configuration space, they tend to be highly combinatorial in nature, having a search space which grows rapidly with the problem size. While some restricted problems admit polynomial-time algorithms, many of these configuration problems are NP-hard. Even the problem of selecting column widths to minimize the height of a table is NP-hard [Anderson and Sobti, 1999]. Given the modest size of most real-world layout problems, many of these problems can readily be solved using conventional combinatorial optimization techniques.

In Part II, we apply combinatorial optimization techniques to compute optimal solutions for several document composition and diagram layout problems. We present models for k-level graph layout, table layout and guillotine-based document layout. We also present a set of techniques for handling complex disjunctive constraints in cases where the layout is to be directly manipulated by the user, rather than generated autonomously.

5

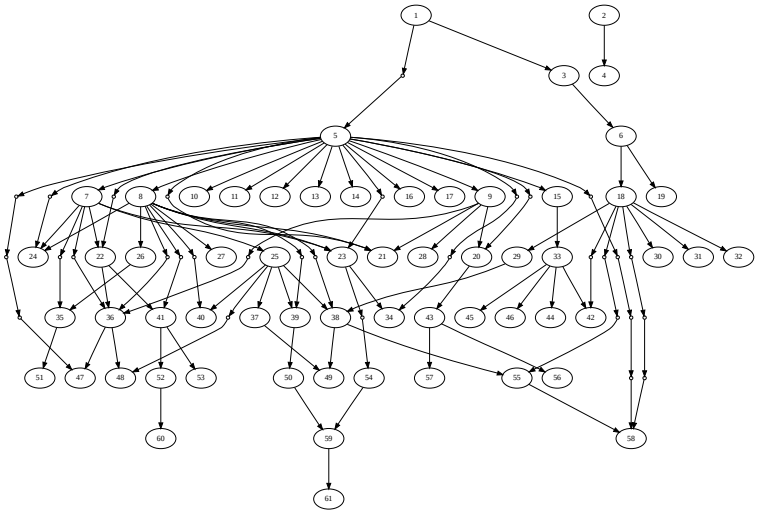
k-level Graph Layout

A *hierarchical network diagram* is a representation of a graph, where each vertex of the graph is assigned to one of a set of horizontal *layers*, and edges connect nodes on different layers (preferably directed downwards). The MDDs presented in Chapter 3 are an ideal example – the nodes are aligned in layers according to the tested variable, and each edge connects to a node lower in the graph.

The standard approach for drawing hierarchical network diagrams is a three phase approach due to Sugiyama et al. [1981] in which (a) nodes in the graph are assigned levels producing a k-level graph; (b) nodes are assigned an order so as to minimize edge crossings in the k-level graph; and (c) the edge routes and node positions are computed. There has been considerable research into step (b) which is called *k-level crossing minimization*. Unfortunately this step is NP-hard even for two layers ($k = 2$) where the ordering on one layer is given [Garey and Johnson, 1983]. Thus, research has focussed on developing heuristics to solve it. In practice a common approach is to iterate through the levels, re-ordering the nodes on each level using heuristic techniques such as the barycentric method [Di Battista et al., 1999], however other more global heuristics have been developed [Matuszewski et al., 1999]. We consider instead the application of combinatorial optimization techniques to find optimal solutions to the k-level crossing minimization problem.

An alternative to performing crossing minimization in phase (b) is *k-level planarization* problem. This was introduced by Mutzel [1996] and is the problem of finding the minimal set of edges that can be removed which allow the remain-

Figure 5.1: Graphviz heuristic layout for the *profile* example graph.



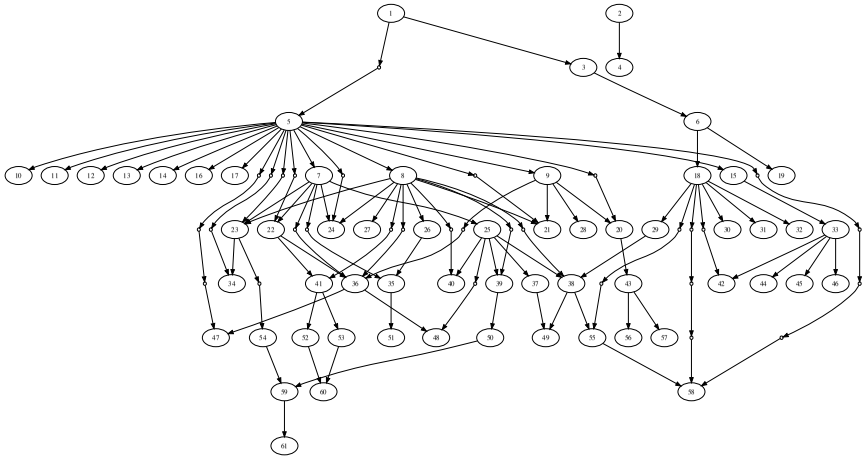
ing edges in the k-level graph to be drawn without any crossings. Mutzel gave a motivating example where maximizing planar subset gave a layout which was perceived as having fewer crossings than minimum crossing layout, despite actually having 41% *more* crossings. While in some sense simpler than k-level crossing minimization (since the problem is tractable for $k = 2$ with one side fixed) it is still NP-hard for $k > 2$ (by reduction from HAMILTONIAN-PATH) [Eades and Whitesides, 1994]. A disadvantage of k-level planarization is that it does not take into account the number of crossings that the non-planar edges generate and so a poor choice of which edges to remove can give rise to unnecessary edge crossings.

Here we introduce a combination of the two approaches we call *k-level planarization and crossing minimization*. This minimizes the weighted sum of the number of crossings *and* the number of edges that need to be removed to give a planar drawing. We believe that this can give rise to nicer drawings than either k-level planarization or k-level crossing minimization while providing a natural generalization of both.

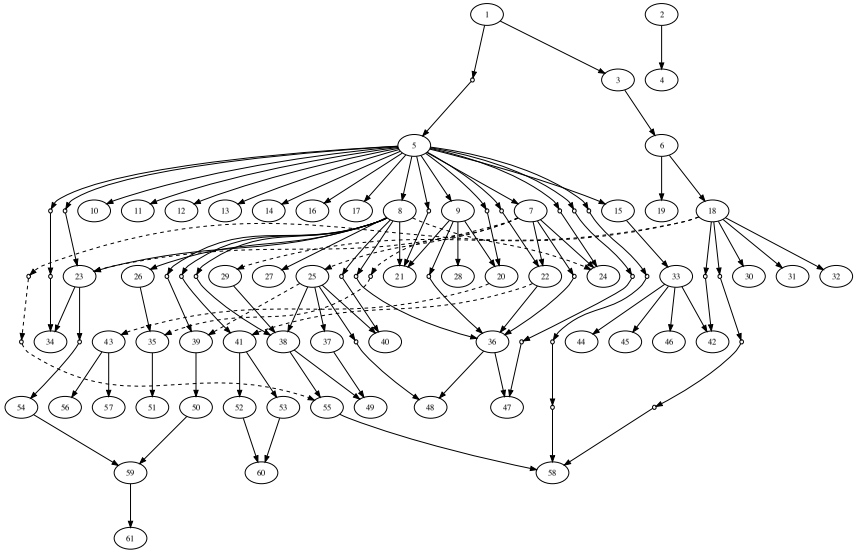
As some evidence for this consider the drawings shown in Figures 5.1 and Figure 5.2 of the example graph `profile` from the GraphViz gallery [Gansner and North, 2000]. Figure 5.1 shows the layout from GraphViz using its heuristic for edge crossing minimization. It has 54 edge crossings and requires removal of 17 edges to become planar.

Figure 5.2: Different layouts of the *profile* example graph.

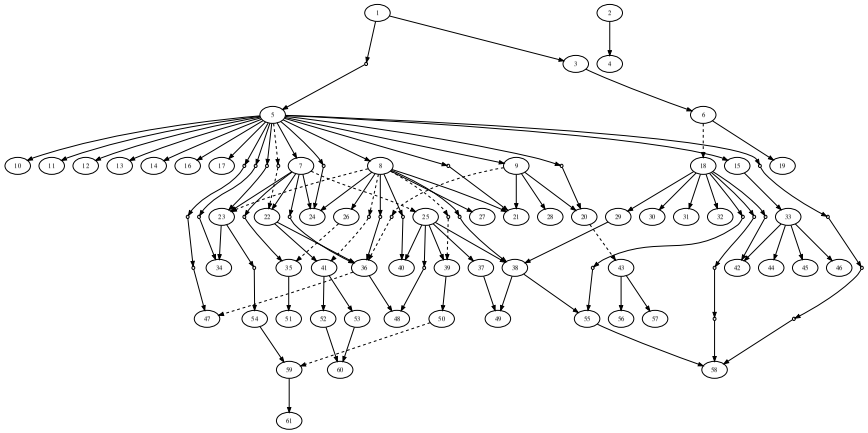
(a) Crossing minimization



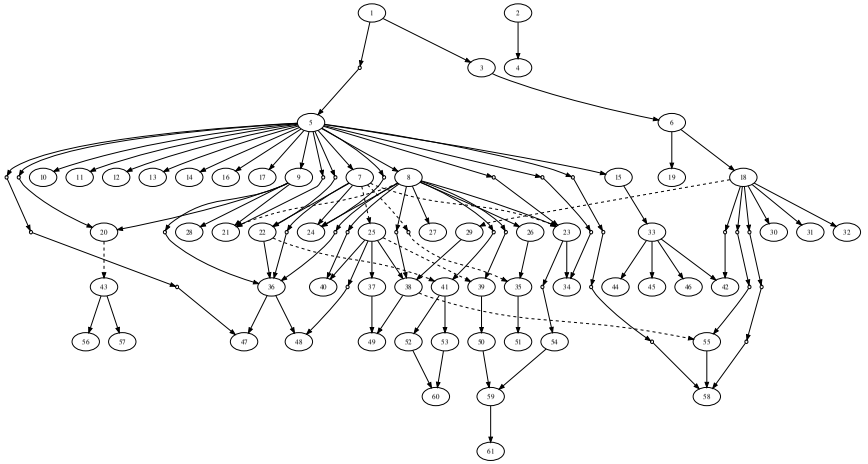
(b) Maximum planar subset



(c) Minimize crossings, then maximize planar subset



(d) Maximum planar subset, then crossing minimization



The layout resulting from minimizing edge crossings is shown in Figure 5.2(a). It has 38 crossings, significantly less than the heuristic layout, and requires 13 edge deletions. The layout resulting from maximizing the planar subgraph is shown in Figure 5.2(b) with deleted edges dotted. It requires only 9 edges to be deleted but has 81 crossings. The layout clearly shows that maximizing the planar subgraph in isolation is not enough, leading to many unnecessary crossings.

The combined model allows us to minimize both crossings and edge deletions for planarity simultaneously. Figure 5.2(c) shows the result of minimizing crossings and then maximizing the planar subset. It yields 38 crossings and 11 edge deletions. Figure 5.2(d) shows the results of of maximizing the planar subset and the minimize crossings. It yields 9 edge deletions and 57 edge crossings, a substantial improvement over the maximal planar subgraph layout of Figure 5.2(b).

We believe these combined layouts illustrate that some combination of minimal edge crossing and minimal edge deletions for planarity can in some cases lead to better layout than either individually. However, this cannot be evaluated in general without some method for computing suitable layouts. Particularly for complex, hybrid objective functions of this kind, it is not obvious how to design an algorithm to generate these layouts; and it is not ideal to expend considerable effort designing a heuristic before knowing whether the given aesthetic criterion is sensible. It seems worthwhile, then, to develop generic techniques that allow easier exploration of different objectives.

Apart from introducing these combined layouts, this chapter has two main technical contributions. The first is to give a binary program for the combined k-level planarization and crossing minimization problem. By appropriate choice of the weighting factor this model reduces to either k-level planarization or k-level crossing minimization. Our basic model is reasonably straightforward but we use some tricks to reduce symmetries, handle leaf nodes in trees and improve bounds for edge cycles.

Our second technical contribution is to evaluate performance of the binary program using both a generic MIP solver and a generic SAT solver. While MIP techniques are not uncommon in graph drawing the use of SAT techniques is quite unusual. Our reason for considering MIP is that MIP is well suited to combinatorial optimization problems in which the linear relaxation of the problem is close to

the original problem. However this does not seem true for k-level planarization and/or k-level crossing minimization. Hence it is worth investigating the use of other generic optimization techniques.

We find that modern SAT solving with learning, and modern MIP solvers (which now have special routines to handle SAT style models) are able to handle the k-level planarization and crossing minimization problems and their combination for quite large k, meaning that we can solve step (b) to optimality. They are fast enough to find the optimal ordering of nodes on all layers for graphs with hundreds of nodes in a few seconds, so long as the graph is reasonably narrow (less than 10 nodes on each level) and for larger graphs they find reasonable solutions within one minute.

The significance of our research is twofold. First it provides a benchmark for measuring the quality of heuristic methods for solving k-level crossing minimization and/or k-level planarization. Second, the method is practical for small to medium graphs and leads to significantly fewer edge crossings involving fewer edges than is obtained with the standard heuristic approaches. As computers increase in speed and SAT solving and MIP solving techniques continue to improve we predict that optimal solution techniques based on MIP and SAT will replace the use of heuristics for step (b) in layout of hierarchical networks.

Furthermore, our research provides support for the use of generic optimization techniques for exploring different aesthetic criteria. The use of generic techniques allows easy experimentation with, for instance, our hybrid objective function. As another example rather than k-level planarization we might wish to minimize the total number of edges involved in crossings. This is simple to do with generic optimization. Another advantage of generic optimization techniques is that they also readily handle additional constraints on the layout, such as placing some nodes on the outside or clustering nodes together.

The most closely related work is on the use of MIP and branch-and-bound techniques for solving k-level crossing minimization. Jünger and Mutzel [1997] compared heuristic methods for two layer crossing minimization with a MIP encoding solved using a specialized branch-and-cut algorithm to solve to optimality. They found that the MIP encoding for the case when one layer is fixed is practical for reasonably sized graphs. In another paper, Jünger et al. [1997] gave a 0-1 model for k-level crossing minimization and solved it using a generic MIP solver. They found

that at that time MIP techniques were impractical except for quite small graphs. We differ from this in considering planarization as well and in investigating SAT solvers. Randerath et al. [2001] gave a partial-MAXSAT model of crossing minimization, however did not provide any experiments. We show that SAT solving with learning, and more recent MIP solvers (which now have special routines to handle SAT style models) are now practical for reasonably sized graphs.

Also related is Mutzel [1996] which describes the results of using a MIP encoding with branch-and-cut for the 2-level planarization problem. Here we give a binary program model for k-level planarization and show that SAT with learning and modern MIP solvers can solve the k-level planarization problem for quite large k. We use a similar model to that of Jünger and Mutzel but examine both MIP and SAT techniques for solving it.

The chapter is organized as follows. In the next section we give our model for combined planarity and crossing minimization. In Section 5.2 we show how to improve the model by taking into account graph properties. In Section 5.3 we give results of experiments comparing the different measures, and finally in Section 5.4 we conclude.

5.1 Model

A general framework for generating layouts of hierarchical data was presented by Sugiyama et al. [1981]. This proceeds in three stages. First, the vertices of the graph are partitioned into horizontal layers. Then, the ordering of vertices within these horizontal layers is permuted to reduce the number of edge crossings. Finally, these layers are positioned to straighten long edges and minimize edge length. Our focus is on the second stage of this process – permuting the vertices on each layer.

Consider a graph with nodes divided into k layers, with edges restricted to adjacent layers, ie. edges from layer i to $i+1$. Denote the nodes in the $k-th$ layer by $nodes[k]$, and the edges from layer k to layer $k+1$ by $edges[k]$. For a given edge e, denote the start and end nodes by $e.s$ and $e.d$ respectively.

The combined model for maximal planar subgraph and crossing minimization is defined by the binary program:

$$\text{min} \quad \textstyle\sum_{k\in levels} C \sum_{e,f\in edges[k]} c_{(e,f)} + P \sum_{e\in edges[k]} r_e \tag{5.1}$$

s.t.

$$\textstyle\bigwedge_{k\in levels} \bigwedge_{i,j,k\in nodes[k]} l_{(i,j)} \wedge l_{(j,k)} \rightarrow l_{(i,k)} \tag{5.2}$$

$$\textstyle\bigwedge_{k\in levels} \bigwedge_{e,f\in edges[k]} c_{(e,f)} \leftrightarrow l_{(e.s,f.s)} \oplus l_{(e.d,f.d)} \tag{5.3}$$

$$\textstyle\bigwedge_{k\in levels} \bigwedge_{e,f\in edges[k]} r_e \vee r_f \vee \neg c_{(e.f)} \tag{5.4}$$

The operation $x \oplus y$ denotes the operator XOR, and is equivalent to $(x \vee y) \wedge (\neg x \vee \neg y)$. The variable $l_{(i,j)}$ indicates node i is before j in the level ordering. The variable $c_{(e,f)}$ indicates that edge e crosses edge f. The variable r_e indicates that edge e is deleted to make the graph planar. The constants C and P define the relative weights of crossing minimization and edge deletion for planarity. The 3-cycle constraints of Equation 5.2 ensures that the order variables are assigned to a consistent ordering. Equation 5.3 defines the edge crossings variables in terms of the ordering: the edges cross if the relative order of the start and end nodes are reversed. It is encoded in clauses as

$$\begin{aligned} &c_{(e,f)} \vee l_{(e.s,f.s)} \vee \neg l_{(e.d,f.d)}, \quad && c_{(e,f)} \vee \neg l_{(e.s,f.s)} \vee l_{(e.d,f.d)}, \\ &\neg c_{(e,f)} \vee l_{(e.s,f.s)} \vee l_{(e.d,f.d)}, \quad && \neg c_{(e,f)} \vee \neg l_{(e.s,f.s)} \vee \neg l_{(e.d,f.d)}. \end{aligned}$$

The planarity requirement is encoded in Equation 5.4 which states that for each pair either one is removed, or they don't cross. The combined model uses $O(k.(e^2+n^2))$ Boolean variables and is $O(k.(n^3+e^2))$ in size. As mentioned in Chapter 2, the SAT model handles optimization problems by solving a sequence of satisfiability problems with progressively restricted objective values. Sorting networks provide a convenient interface for this; if the current solution of a minimization problem has value k, we can search for a better solution by asserting $\neg o_k$ (where o_k is the k^{th} output of the sorting network) and re-solving.

We can convert this clausal model to a MIP binary program by converting each clause $b_1 \vee \cdots b_l \vee \neg b_{l+1} \vee \cdots \vee \neg b_m$ to the linear constraint $b_1 + \cdots + b_l - b_{l-1} - \cdots - b_m \geq m - l + 1$.

Long edges are handled by adding intermediate nodes in the levels that the long edges cross and breaking the edge into components. For crossing minimization each of these new edges is treated like an original edge. For the minimal deletion of edges each component edge in a long edge e is encoded using the same deletion variable r_e.

By adjusting the relative weights for crossing C, and planarization P, we can create and evaluate new measures of clarity of the graph. With $C = 1+\sum_{k\in levels} |edges[k]|$ and $P = 1$ we first minimize crossings, then minimize edge deletions for planarity. With $C = 1$ and $P = \sum_{k\in levels} |edges[k]|^2$ we first minimize edge deletions and then crossings. While we limit our evaluation to lexicographic orderings, other choices of C and P can be used to express different combined objectives.

5.2 Additional Constraints

While the basic model described in Section 5.1 is sufficient to ensure correctness, finding the optimum still requires a great deal of search. We can modify the model to significantly improve performance.

First note that we add symmetry breaking by fixing the order of the first two nodes appearing on the same level. If the graph to be layed out has more symmetries than this left-to-right symmetry we could use this to fix more variables (although we don't do this in the experiments). Next, we can improve edge crossing minimization by using as an upper bound the number of crossings in a heuristic layout. We could also use heuristic solutions to bound planarity but doing so requires computing how many edges need deletion, which is non-trivial.

5.2.1 Cycle Parity

Healy and Kuusik introduced the vertex-exchange graph [Healy and Kuusik, 1999] for analyzing layered graphs. Each edge in the vertex-exchange graph corresponds to a potential crossing in the initial graph; each node corresponds to a pair of nodes within a level.

Consider the graph shown in Figure 5.3(a), its vertex-exchange graph is shown in Figure 5.3(b). Note there are two edges (ab, de) corresponding to the two pairs

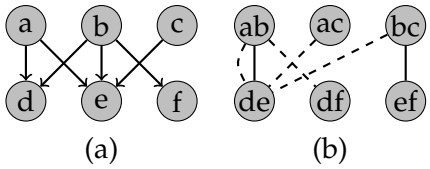


Figure 5.3: (a) A graph, with an initial ordering (b) The corresponding vertex-exchange graph.

$((a,d),(b,e))$ and $((a,e),(b,d))$. Edges corresponding to crossings in Figure 5.3(a) are shown as solid, the rest are dashed.

For any given cycle in the vertex exchange graph, permuting nodes within a layer will maintain parity in the number of crossings in the cycle. For cycles with an odd number of crossings, this means that at least one of the pairs of edges in the cycle will be crossing. This can be represented by the clause $\bigvee_{(e,f)\in cycle} c_{(e,f)}$. When finding the maximal planar subgraph, we then know that at least one edge involved in the cycle must be removed from the subgraph. Similarly since the cycle is even in length we know that not all edges can cross, represented by $\bigvee_{(e,f)\in cycle} \neg c_{(e,f)}$. Both these constraints can be added to the model.

A special case of cycle parity is the $K_{2,2}$ subgraph. This subgraph always produces exactly one crossing, irrespective of the relative orderings of the nodes in the subgraph. When minimizing crossings, the corresponding $c_{(e,f)}$ variables need not be included in the objective function, which considerably simplifies the problem structure. Note that, for example, a $K_{3,3}$ subgraph contains 9 $K_{2,2}$ subgraphs, and each of the 9 $c_{e,f}$ variables arising can be omitted from the problem. For the experiments we add constraints for cycles of length 6 or less, since the larger cycles did not improve performance.

5.2.2 Leaves

It is not difficult to prove that if a node on layer k has m child leaf nodes (unconnected to any other node) on layer $k+1$, then all of these leaf nodes can be ordered together.

Consider the partial layout illustrated in Figure 5.4, where each node 1,2,3 and 4 is a leaf node with no outgoing arcs. If we place a node f in between nodes 1,2,3 and 4 (as illustrated) there is always at least as good a crossing solution by placing

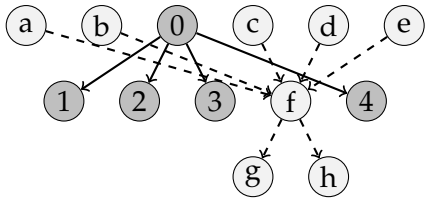


Figure 5.4: A partial layout with respect to some leaf nodes 1,2,3,4

f either before or after all of them. Here since there are 2 parents before 0 and 3 after, f should be placed after 4, leading to 8 crossings rather than the 9 illustrated.

Similarly maximizing planarity always requires that all edges to siblings left of f be removed or all edges from parents before 0, and all edges to siblings right of f or all edges from parents after 0. An optimal solution always results by either deleting all edges to leaf nodes (which makes the leaf positions irrelevant), or ordering f after all leaves and deleting all edges from parents before 0, or ordering f before all leaves and deleting all edges from parents after 0.

Since there is no benefit in splitting leaf siblings we can treat them as a single node, but note we must appropriately weight the edge resulting, since it represents multiple crossings and multiple edge deletions.

Let N be a set of m leaf nodes from a single parent node i. We replace N by a new node j', and replace all edges $\{(i,j) \mid j \in N\}$ by the single edge (i,j'). We replace each m terms $c_{((i,j),f)}, j \in N$ in the objective function by one term $m \times c_{((i,j'),f)}$ and replace the set of m terms $r_{(i,j)}, j \in N$ in the objective by the term $m \times r_{(i,j')}$.

5.3 Experimental Results

We tested the binary model on a variety of graphs, using the pseudo-Boolean constraint solver MiniSAT+[Eén and Sörensson, 2006], and the Mixed Integer Programming solver CPLEX12.0. All experiments were performed on a 3.0GHz Xeon X5472 with 32 GB of RAM running Debian GNU/Linux 4.0. We ran for a maximum of 60s, and all times are given in seconds.

We compared 4 different objective functions:

- crossing minimization: $C = 1$, $P = 0$;
- maximal planar subgraph $C = 0$, $P = 1$;

Problem	Crossing minimization					Maximal planar subgraph			
	graphviz	MIP		SAT		MIP		SAT	
	cross/plan	best	solved	best	solved	best	solved	best	solved
crazy	3 / **1**	**2**	0.04	**2**	**0.01**	**1**	0.03	**1**	**0.01**
datastruct	**2** / 2	**2**	**0.00**	**2**	**0.00**	**1**	0.02	**1**	**0.00**
fsm	**0** / **0**	**0**	**0.00**	**0**	**0.00**	**0**	**0.00**	**0**	**0.00**
lion_share	7 / 3	**4**	**0.04**	**4**	0.11	**2**	0.05	**2**	**0.02**
profile	54 / 17	**38**	**6.81**	54	—	12	—	**9**	**5.39**
switch	**20** / **17**	**20**	0.75	**20**	**0.64**	**17**	—	**17**	**34.49**
traffic_lights	**0** / **0**	**0**	**0.00**	**0**	**0.00**	**0**	**0.00**	**0**	**0.00**
unix	3 / **1**	**2**	0.05	**2**	**0.01**	**1**	0.04	**1**	**0.02**
world	50 / 16	**46**	—	50	—	15	—	**13**	—

Table 5.1: Time to find and prove the minimal crossing layout and maximal planar subgraph for Graphviz examples using MIP and SAT.

- crossing minimization then maximal planar subgraph $C = 1+\sum_{k\in levels} |edges[k]|$, $P = 1$; and
- maximal planar subgraph then minimize crossings $C = 1$, $P = \sum_{k\in levels} |edges[k]|^2$.

To compare speed and effectiveness of the model we ran it on two sets of graphs. The first set of graphs are all the hierarchical network diagrams appearing in the GraphViz gallery [Gansner and North, 2000]. The second set of graphs are random graphs of k-levels with n nodes per level and a fixed edge density of 20% (that is each node is connected on average to 20% of the nodes on the next layer); these do not include any long edges. Problem class gk_n is a suite of 10 randomly generated instances with k levels and n nodes per level.

Table 5.1 shows the results of minimizing edge crossings and maximizing planar subgraphs with MIP and SAT solvers, as well as the crossings resulting in the Graphviz heuristic layout for graphs from the GraphViz gallery. We use the best variation of our model for each solver: for the MIP solver this is with all improvements described in the previous section, while for the SAT solver we omit the leaf optimization since it slows down the solver. For each solver we show the solution, with least edge crossings or minimal number of edges deleted for planarity, found in 60s and the time to prove optimality or '—' if it was not proved optimal. The results show that for realistic graphs we can find better solutions than the heuristic method, even when there are very few crossings. The best found crossing and edges deleted for planarization and instances solved to optimality/time combination are highlighted in bold. We can find optimal crossing solutions for 9 out of

Problem	Crossing minimization					Maximal planar subgraph			
	graphviz	MIP		SAT		MIP		SAT	
	best	best	solved	best	solved	best	solved	best	solved
g3_7	41 / 29	**35**	**10 / 0.00**	**35**	10 / 0.02	**18**	**10 / 0.01**	**18**	10 / 0.02
g3_8	103 / 59	**86**	**10 / 0.03**	**86**	10 / 0.10	**31**	10 / 0.15	**31**	**10 / 0.04**
g3_9	204 / 87	**195**	**10 / 0.07**	**195**	10 / 6.67	**63**	10 / 4.60	**63**	**10 / 0.12**
g3_10	399 / 126	**373**	**10 / 0.97**	380	8 / 14.44	**91**	5 / 15.53	**91**	**10 / 3.37**
g4_7	70 / 44	**55**	**10 / 0.01**	**55**	10 / 0.04	**32**	10 / 0.11	**32**	**10 / 0.04**
g4_8	187 / 93	**169**	**10 / 0.09**	**169**	10 / 0.68	**61**	10 / 3.83	**61**	**10 / 0.16**
g4_9	351 / 130	**342**	**10 / 0.89**	345	8 / 13.69	**94**	6 / 26.85	**94**	**10 / 2.82**
g4_10	703 / 209	**681**	**9 / 2.37**	—	—	161	—	**152**	—
g5_7	101 / 61	**95**	**10 / 0.03**	**95**	10 / 0.11	**47**	10 / 0.43	**47**	**10 / 0.08**
g5_8	284 / 125	**245**	**10 / 0.32**	**245**	10 / 4.01	**93**	10 / 25.46	**93**	**10 / 2.78**
g5_9	474 / 192	**450**	**10 / 0.99**	—	5 / 30.47	139	1 / 35.37	**138**	**3 / 47.46**
g6_7	141 / 82	**131**	**10 / 0.03**	**131**	10 / 0.18	**57**	10 / 1.51	**57**	**10 / 0.18**
g6_8	357 / 154	**324**	**10 / 0.53**	**324**	10 / 13.34	112	4 / 11.64	**111**	**10 / 28.81**
g6_9	684 / 252	**637**	**10 / 3.28**	—	—	**190**	—	197	—
g7_7	159 / 91	**148**	**10 / 0.08**	**148**	10 / 0.58	**67**	10 / 3.91	**67**	**10 / 0.78**
g7_8	390 / 183	**366**	**10 / 0.72**	372	6 / 12.35	**134**	**1 / 47.75**	140	—
g7_9	813 / 293	**786**	**9 / 12.54**	—	—	238	—	**233**	—
g8_7	249 / 131	**235**	**10 / 0.16**	**235**	10 / 4.06	**92**	8 / 13.38	**92**	**10 / 4.91**
g8_8	466 / 224	**431**	**10 / 1.51**	—	1 / 10.73	**154**	—	165	—
g9_7	269 / 150	**238**	**10 / 0.30**	**238**	10 / 2.60	**108**	7 / 17.72	**108**	**8 / 15.18**
g9_8	572 / 259	**541**	**10 / 2.95**	—	3 / 28.71	**197**	—	200	—
g10_7	334 / 172	**304**	**10 / 0.27**	**304**	10 / 9.67	**119**	**8 / 24.05**	121	4 / 19.39
g10_8	733 / 302	**661**	**10 / 10.68**	—	—	**216**	—	225	—

Table 5.2: Time to find and prove the minimal crossing layout and maximal planar subgraph, using MIP and SAT for random examples.

ten examples, and maximal planar subgraph solutions for 7 out of 10 examples. The MIP approach is clearly superior for minimizing edge crossings, while SAT is superior for maximizing planarity.

Table 5.2 shows the results of crossing minimization and maximal planar subgraph for the second data set of random graphs using the MIP and SAT solver. The table shows: the total number of crossings when the graphs are laid out using GraphViz then for each solver: the total number of crossings or edge deletions in the best solutions found in 60s for the suite (a '—' indicates that for at least one instance the method found no solution better than the Graphviz bound in 60s) and the number of instances where optimal solutions were found and proved and the average time to prove optimality.

The results are in accord with those for the first dataset and show that the MIP solver can almost always find optimal minimal crossing solutions within this time bound (only two instances failed). The Graphviz solutions can be substantially improved, the best solutions found have 10-20% fewer crossings.

Problem	Crossing then planarization				Planarization then crossing			
	MIP		SAT		MIP		SAT	
	best	solved	best	solved	best	solved	best	solved
crazy	**(2, 1)**	**0.07**	**(2, 1)**	10.51	**(1, 2)**	**0.08**	**(1, 2)**	0.31
datastruct	**(2, 1)**	**0.02**	**(2, 1)**	0.26	**(1, 2)**	**0.03**	**(1, 2)**	0.23
fsm	**(0, 0)**	**0.00**	**(0, 0)**	**0.00**	**(0, 0)**	**0.00**	**(0, 0)**	**0.00**
lion_share	**(4, 3)**	**0.15**	**(4, 3)**	3.55	**(2, 5)**	**0.52**	**(2, 5)**	0.60
profile	**(38, 11)**	**29.96**	(281, 34)	—	**(12, 66)**	—	(13, 145)	—
switch	**(20, 17)**	**1.36**	**(20, 17)**	3.61	**(17, 20)**	—	**(17, 20)**	—
traffic_lights	**(0, 0)**	**0.00**	**(0, 0)**	**0.00**	**(0, 0)**	**0.00**	**(0, 0)**	0.01
unix	**(2, 1)**	**0.07**	**(2, 1)**	10.52	**(1, 2)**	**0.09**	**(1, 2)**	0.32
world	**(47, 14)**	—	(108, 19)	—	(18, 79)	—	**(15, 106)**	—

Table 5.3: Time to find and prove optimal mixed objective solutions for Graphviz examples using MIP and SAT.

Problem	Crossing then planarization				Planarization then crossing			
	MIP		SAT		MIP		SAT	
	best	solved	best	solved	best	solved	best	solved
g3_7	**(35, 22)**	**10 / 0.01**	**(35, 22)**	10 / 0.04	**(18, 39)**	**10 / 0.02**	**(18, 39)**	10 / 0.04
g3_8	**(86, 41)**	**10 / 0.13**	**(86, 41)**	10 / 0.37	**(31, 102)**	10 / 0.31	**(31, 102)**	**10 / 0.21**
g3_9	**(195, 78)**	**10 / 0.48**	**(195, 78)**	10 / 3.09	**(63, 231)**	9 / 10.31	**(63, 231)**	**10 / 1.04**
g3_10	**(373, 115)**	**10 / 2.67**	(564, 166)	2 / 40.99	(91, 444)	2 / 13.84	**(91, 419)**	**9 / 10.64**
g4_7	**(55, 35)**	**10 / 0.05**	**(55, 35)**	10 / 0.23	**(32, 58)**	**10 / 0.21**	**(32, 58)**	10 / 0.23
g4_8	**(169, 76)**	**10 / 0.57**	**(169, 76)**	10 / 1.58	**(61, 223)**	10 / 6.66	**(61, 223)**	**10 / 1.91**
g4_9	**(342, 116)**	**10 / 1.66**	(418, 162)	5 / 33.21	**(94, 386)**	3 / 31.03	(95, 382)	**8 / 14.76**
g4_10	**(681, 195)**	**9 / 8.17**	(1249, 338)	—	**(158, 933)**	—	(160, 928)	—
g5_7	**(95, 55)**	**10 / 0.08**	**(95, 55)**	10 / 0.35	**(47, 104)**	10 / 0.76	**(47, 104)**	**10 / 0.53**
g5_8	**(245, 108)**	**10 / 0.66**	**(245, 108)**	10 / 8.38	(95, 269)	4 / 29.48	**(94, 290)**	**8 / 25.24**
g5_9	**(450, 174)**	**10 / 3.83**	(694, 210)	1 / 42.43	**(142, 612)**	—	(146, 656)	—
g6_7	**(131, 64)**	**10 / 0.25**	**(131, 64)**	10 / 1.98	**(57, 153)**	10 / 2.74	**(57, 153)**	**10 / 2.32**
g6_8	**(324, 136)**	**10 / 1.16**	(357, 150)	6 / 22.50	**(112, 419)**	**2 / 22.62**	(117, 413)	2 / 31.29
g6_9	**(637, 228)**	**10 / 8.15**	(1353, 513)	—	**(192, 881)**	—	(212, 967)	—
g7_7	**(148, 83)**	**10 / 0.34**	**(148, 83)**	10 / 23.94	**(67, 168)**	10 / 10.66	**(67, 168)**	**10 / 10.59**
g7_8	**(366, 159)**	**10 / 3.00**	(454, 236)	2 / 20.16	**(136, 472)**	—	(148, 500)	—
g7_9	**(778, 255)**	**8 / 18.96**	(1372, 481)	—	**(236, 1031)**	—	(258, 1303)	—
g8_7	**(235, 116)**	**10 / 0.50**	**(235, 116)**	10 / 14.06	**(92, 272)**	5 / 15.54	(93, 277)	**8 / 22.09**
g8_8	**(431, 195)**	**10 / 5.06**	(641, 345)	1 / 33.37	**(154, 552)**	—	(182, 639)	—
g9_7	**(238, 123)**	**10 / 0.77**	(241, 126)	9 / 25.00	**(108, 260)**	**6 / 16.29**	(112, 283)	2 / 57.22
g9_8	**(541, 229)**	**10 / 6.17**	(981, 464)	—	**(198, 757)**	—	(216, 871)	—
g10_7	**(304, 144)**	**10 / 1.59**	(329, 201)	7 / 33.19	**(119, 362)**	**4 / 32.01**	(126, 415)	1 / 58.29
g10_8	**(661, 256)**	**9 / 15.15**	(1216, 546)	—	**(199, 832)**	—	(224, 987)	—

Table 5.4: Time to find and prove optimal mixed objective solutions for random examples using MIP and SAT.

For maximal planar subgraph, in contrast to edge crossings, the SAT solver is better than the MIP solver, although as the number of levels increases the advantage decreases.

Tables 5.3 and 5.4 show the results for the mixed objective functions: minimizing crossings then maximizing planar subgraph and the reverse. For minimizing

crossings first MIP dominates as before, and again is able to solve almost all problems optimally within 60s. For the reverse objective SAT is better for the small instances, but suffers as the instances get larger. This problem is significantly harder than the minimizing crossings first.

Results not presented demonstrate that the improvements presented in the previous section make a substantial difference. The elimination of $K_{2,2}$ cycles is highly beneficial to both solvers. Constraints for larger cycles can have significant benefit for the MIP solver but rarely benefit the SAT solver. The leaf optimization is good for the MIP solver, but simply slows down the SAT solver. We believe this is because it complicates the MiniSAT+ translation of the objective function to clauses. Overall the optimizations improve speed by around 2-5$\times$. They allow 6 more instances to find optimal solutions for minimizing crossing, 5 for maximal planar subgraph, 19 for crossing minimization then maximal planar subgraph, and 9 for maximal planar subgraph then crossing minimization.

It is also possible to solve these combined objectives using a staged optimization procedure; for example, first minimizing crossings, then maximizing the planar subset subject to the minimum number of crossings. On the instances we tested, the performance of MIP was similar for either the combined or staged objectives. Surprisingly, the SAT solver performed considerably worse using the staged procedure than with the combined objective. Indeed, one instance took 3s to solve the combined objective problem, but the staged procedure took 127s to solve just the second stage. The reason for this dramatic difference is unclear, and would be worth investigating.

5.4 Conclusion

This chapter demonstrates that maximizing clarity of heirarchical network diagrams by edge crossing minimization or maximal planar subgraph or their combination can be solved optimally for reasonable sized graphs using modern SAT and MIP software. Using this generic solving technology allows us to experiment with other notions of clarity combining or modifying these notions. It also gives us the ability to accurately measure the effectiveness of heuristic methods for solving these problems.

6

Table Layout

TABLES are provided in virtually all document formatting systems and are one of the most powerful and useful design elements in current web document standards such as (X)HTML [HTML Working Group, 2002], CSS [Bos et al., 1998] and XSL [Clark and Deach, 1998]. For on-line presentation it is not practical to require the author to specify table column widths at document authoring time since the layout needs to adjust to different width viewing environments and to different sized text since, for instance, the viewer may choose a larger font. Dynamic content is another reason that it can be impossible for the author to fully specify table column widths. This is an issue for web pages and also for variable-data printing (VDP) in which improvements in printer technology now allow companies to cheaply print material which is customized to a particular recipient. Good automatic layout of tables is therefore needed for both on-line and VDP applications and is useful in many other document processing applications since it reduces the burden on the author of formatting tables.

However, automatic layout of tables that contain text is computationally expensive. Anderson and Sobti [1999] have shown that table layout with text is NP-hard. The reason is that if a cell contains text then this implicitly constrains the cell to take one of a discrete number of possible configurations corresponding to different numbers of lines of text. It is a difficult combinatorial optimization problem to find which combination of these discrete configurations best satisfies reasonable layout requirements such as minimizing table height for a given width.

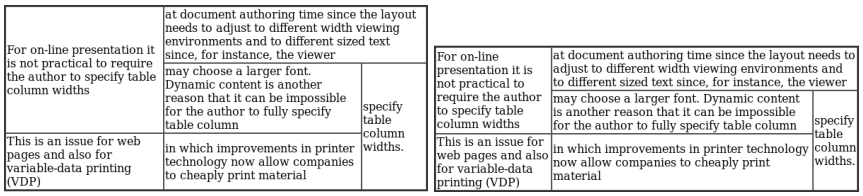


Figure 6.1: Example table comparing layout using the Mozilla layout engine, Gecko (on the left) with the minimal height layout (on the right).

Table layout research is reviewed by Hurst, Li and Marriott [Hurst et al., 2009]. Starting with Beach [1985], a number of authors have investigated automatic table layout from a constrained optimization viewpoint and a variety of approaches for table layout have been developed. Almost all approaches use heuristics and are not guaranteed to find the optimal solution. They include methods that use a desired width for each column and scale this to the actual table width [Raggett et al., 1999, Borning et al., 2000, Badros et al., 1999], methods that use a continuous linear or non-linear approximation to the constraint that a cell is large enough to contain its contents [Anderson and Sobti, 1999, Beaumont, 2004, Hurst et al., 2005, 2006a, Lin, 2006], a greedy approach [Hurst et al., 2005] and an approach based on finding a minimum cut in a flow graph [Anderson and Sobti, 1999].

In this chapter we are concerned with complete techniques that are guaranteed to find the optimal solution. While these are necessarily non-polynomial in the worst case (unless P=NP) we are interested in finding out if they are practical for small and medium sized table layout. Even if the complete techniques are too slow for normal use, it is still worthwhile to develop complete methods because these provide a benchmark with which to compare the quality of layout of heuristic techniques. For example, while Gecko (the layout engine used by the Firefox web browser) is the most sophisticated of the HTML/CSS user agents whose source code we've seen, the generated layouts can be considerably suboptimal even for small tables. Figure 6.1 shows a 3 by 3 table laid out using the Mozilla layout engine, and the corresponding minimum height layout. Notice that the top-left and bottom-right cells span two rows, and the top-right cell spans two columns.

We know of only three other papers that have looked at complete methods for table layout with breakable text. The first is a branch-and-bound algorithm described in Wang and Wood [1997], which finds a layout satisfying linear designer constraints on the column widths and row heights. However it is only complete

in the sense that it will find a feasible layout if one exists and is not guaranteed to find an optimal layout that, say, minimizes table height.[1] The second is detailed in a recent paper by Bilauca and Healy [2010]. They give two MIP based branch-and-bound based complete search methods for simple tables. Bilauca and Healy have also presented an updated model [Bilauca and Healy, 2011], which was developed after the material in this chapter was published.

The first contribution of this chapter is to present three new techniques for finding a minimal height table layout for a fixed width. All three are based on generic approaches for solving combinatorial optimization problems that have proven to be useful in a wide variety of practical applications.

The first approach uses an $A^\star$ based approach (see, e.g., Russell and Norvig [2002]) that chooses a width for each column in turn. Efficiency of the $A^\star$ algorithm crucially depends on having a good lower bound for estimating the minimum height for any full table layout that extends the current layout. We use a heuristic that treats the remaining unfixed columns in the layout as if they are a single merged column each of whose cells must be large enough to contain the contents of the unfixed cells on that row. The other key to efficiency is to prune layouts that are not *column-minimal* in a sense that it is possible to reduce one of the fixed column widths without violating a cell containment constraint while keeping the same row heights.

The second and third approaches are both constraint programming models. The second is a fairly direct encoding of the problem, introducing width and height variables for each cell; we evaluate this model with both a conventional finite-domain constraint solver, and a lazy clause generation solver. The third is a modified lazy clause generation model which avoids introducing cell variables, constraining pairs of row and column variables directly.

The second contribution of this chapter is to provide an extensive empirical evaluation of these three approaches as well as the two MIP-based approaches of Bilauca and Healy [2010]. We first compare the approaches on a large body of tables collected from the web. This comprised more than 2000 tables that were hard to solve in the sense that the standard HTML table layout algorithm did not find the minimal height layout. Most methods performed well on this set of examples

[1] One could minimize table height by repeatedly searching for a feasible solution with a table height less than the best solution so far.

and solved almost all problems in less than 1 second. We then stress-tested the algorithms on some large artificial table layout examples.

The rest of this chapter is organized as follows. In Section 6.1 we give a formal definition of the problem. In Section 6.2, we describe an A* method for table layout. In Sections 6.3 and 6.4 we give an initial constraint programming model, and a revised model to take advantage of lazy clause generation solvers. In Section 6.5, we describe existing integer programming models for the problem, and in Section 6.6 we give an empirical evaluation of the described methods. Finally, in Section 6.7 we conclude.

6.1 Background

We assume throughout this chapter that the table of interest has n columns and m rows. A *layout* (w, h) for a table is an assignment of widths, w, to the columns and heights, h, to the rows where w_c is the width of column c and h_r the height of row r. We make use of the width and height functions:

$$\begin{array}{rclcrcl} wd_{c_1,c_2}(w) & = & \sum_{c=c_1}^{c_2} w_c, & & wd(w) & = & wd_{1,n}(w), \\ ht_{r_1,r_2}(h) & = & \sum_{r=r_1}^{r_2} h_r, & & ht(h) & = & ht_{1,m}(h) \end{array}$$

where ht and wd give the overall table height and width respectively.

The designer specifies how the grid elements of the table are partitioned into logical elements or *cells*. We call this the *table structure*. A *simple* cell spans a single row and column of the table while a *compound cell* consists of multiple grid elements forming a rectangle, i.e. the grid elements span contiguous rows and columns.

If d is a cell we define $rows(d)$ to be the rows in which d occurs and $cols(d)$ to be the set of columns spanned by d. We let

$$\begin{array}{ll} bot(d) = \max rows(d), & top(d) = \min rows(d), \\ left(d) = \min cols(d), & right(d) = \max cols(d). \end{array}$$

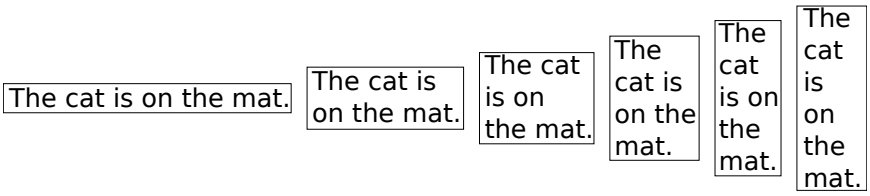


Figure 6.2: Example minimal text configurations.

and, letting $Cells$ be the set of cells in the table, for each row r and column c we define

$$rcells_c = \{d \in Cells \mid right(d) = c\},$$
$$cells_c = \{d \in Cells \mid c \in cols(d)\},$$
$$bcells_r = \{d \in Cells \mid bottom(d) = r\}$$

$cells_c$ is the set of cells spanning column c. $rcells_c$ is the set of cells with column-spans ending ending at column c; similarly, $bcells_r$ is the set of cells with row-spans ending at row r.

Each cell d has a minimum width, $minw(d)$, which is typically the length of the longest word in the cell, and a minimum height $minh(d)$, which is typically the height of the text in the cell.

The table's *structural constraints* are that each cell is big enough to contain its content and at least as wide as its minimum width and as high as its minimum height.

The *minimum-height table layout* problem [Anderson and Sobti, 1999] is, given a table structure, content for the table cells and a maximum width W, to find an assignment to the column widths and row heights such that the structural constraints are satisfied, the overall width is no greater than W, and the overall height is minimized.

For simplicity, we assume that the minimum table width is wide enough to allow the structural constraints to be satisfied. Furthermore, we do not consider nested tables nor do we consider designer constraints such as columns having fixed ratio constraints between them.

6.1.1 Minimum configurations

The main decision in table layout is how to break the lines of text in each cell. Different choices give rise to different width/height cell configurations. Cells have

a number of *minimal configurations* where a minimal configuration is a pair (w, h) s.t. the text in the cell can be laid out in a rectangle with width w and height h but there is no smaller rectangle for which this is true. That is, for all $w' \leq w$ and $h' \leq h$ either $h = h'$ and $w = w'$, or the text does not fit in a rectangle with width w' and height h'. These minimum configurations are *anti-monotonic* in the sense that if the width increases then the height will never increase. For text with uniform height with W words (or more exactly, W possible line breaks) there are up to W minimal configurations, each of which has a different number of lines. In the case of non-uniform height text there can be no more than $O(W^2)$ minimal configurations. Figure 6.2 illustrates the minimal configurations for a cell containing the text "The cat is on the mat".

A number of algorithms have been developed for computing the minimum configurations of the text in a cell [Hurst et al., 2009]. Here we assume that these are pre-computed and that

$$configs_d = [(w_1, h_1), ..., (w_{N_d}, h_{N_d})]$$

gives the width/height pairs for the minimal configurations of cell d sorted in increasing order of width. We will make use of the function $minheight(d, w)$ which gives the minimum height $h \geq minh(d)$ that allows the cell contents to fit in a rectangle of width $w \geq minw(d)$. This can be readily computed from the list of configurations. The variable cw_d (resp. ch_d) represents the selected width (height) for cell d.

The mathematical model of the table layout problem can be formalized as:

find w and h that minimize $ht(h)$ subject to

$$\forall d \in Cells. \quad (cw_d, ch_d) \in configs_d \wedge \qquad (1)$$

$$\forall d \in Cells. \quad wd_{left(d),right(d)}(w) \geq cw_d \wedge \qquad (2)$$

$$\forall d \in Cells. \quad ht_{top(d),bot(d)}(h) \geq ch_d \wedge \qquad (3)$$

$$wd(w) \leq W \qquad (4)$$

In essence, automatic table layout is the problem of finding *minimal configurations for a table*: i.e. minimal width / height combinations in which the table can be laid out. One obvious necessary condition for a table layout (w, h) to be a minimal

configuration is that it is impossible to reduce the width of any column c while leaving the other row and column dimensions unchanged and still satisfy the structural constraints. We call a layout satisfying this condition *column-minimal*.

We now detail three algorithms for solving the table layout problem. All are guaranteed to find an optimal solution but in the worst case may take exponential time.

6.2 A* Algorithm

The first approach uses an $A^\star$ based approach [Russell and Norvig, 2002] that chooses a width for each column in turn. A partial layout (w, c) for a table is a width w for the first $c-1$ columns. The algorithm starts from the empty partial layout ($c = 1$) and repeatedly chooses a partial layout to extend by choosing possible widths for the next column.

Partial layouts also have a penalty p, which is a lower bound on the height for any full table layout that extends the current partial layout. The partial layouts are kept in a priority queue and at each stage a partial layout with the smallest penalty p is chosen for expansion. The algorithm has found a minimum height layout when the chosen minimal-penalty partial layout has $c = n + 1$ (and is therefore a total layout). The code is given in function complete-A*-search(W) where W is the maximum allowed table width. For simplicity we assume W is greater than the minimum table width. (The minimum table width can be determined by assigning each column its min_c width from possible-col-widths, or can equivalently be derived from the corresponding maximum positions also used in that function.)

Given widths w for columns $1, \ldots, c-1$ and maximum table width of W, the function possible-col-widths(c,w,W) returns the possible widths for column c that correspond to the width of a minimal configuration for a cell in c and which satisfy the minimum width requirements for all the cells in d and still satisfy the minimum width requirements for columns $c+1, \ldots, n$ and allow the table to have width W.

Efficiency of an $A^\star$ algorithm usually depends strongly on how tight the lower bound on penalty is, i.e., how often (and how early) the heuristic informs us that we can discard a partial solution because all full table layouts that extend that partial

layout will either have a height greater than the optimal height, or have height greater or equal to some other layout that isn't discarded.

We use a heuristic that treats the remaining unfixed columns in the layout as if they are a single merged column each of whose cells must be large enough to contain the contents of the unfixed cells on that row. We approximate the contents by a lower bound of their area. The function compute-approx-row-heights(w,h,c,W) does this, returning the estimated (lower bound) row heights after laying out the area of the contents of columns $c+1,\ldots,n$ in a single column whose width brings the table width to W. Compound cells that span multiple rows, and positions in the table grid that have no cell, use a very simple lower bound of zero.

A* methods often store the set of previously expanded nodes, to avoid repeatedly expanding the same partial solutions. In this case the cost of maintaining this set is relatively expensive, as the encoding of a partial solution must keep track of the minimum height of each row as well as the start position of certain column spans. Given that isomorphic states are encountered infrequently in these problems, we avoid storing this set of *closed* nodes.

We instead present the following method for discarding partial solutions. Partial layouts which must lead to a full layout which is not column minimal are not considered. If the table has no compound cells spanning multiple rows then any partial layout that is not column minimal for the columns that have been fixed can be discarded because row heights can only increase in the future and so the layout can never lead to a column-minimal layout. This no longer holds if the table contains cells spanning multiple rows as row heights can decrease and so a partial layout that is not column minimal can be extended to one that is column minimal. However, it is true that if the cells spanning multiple rows are ignored, i.e. assumed to have zero content, when determining if the partial layout is column minimal then partial layouts that are not column minimal can be safely discarded. The function weakly-column-minimal(w,c) does this by checking that none of the columns $1,\ldots,c$ can be narrowed without increasing the height of a row, ignoring compound cells spanning multiple rows.

In our implementation of complete-A*-search, the iteration over possible widths works from maximum v downwards, stopping once the new partial solution is either known not to be column minimal or (optionally) once the penalty exceeds

a certain maximum penalty which should be an upper bound on the minimum height. Our implementation computes a maximum penalty at the start, by using a heuristic search based on [Hurst et al., 2005].

Creating a new partial layout is relatively expensive (see below), so this early termination is more valuable than one might otherwise expect. However, the cost of this choice is that this test must be done before considering the height lower bounds for future cells (the remaining-area penalty), since the future penalty is at its highest for maximum v.

For the implementation of compute-approx-row-heights, note that D_{free} and $area_r$ don't depend on w or $h0$, and hence may be precalculated; while w may be stored in cumulative form, and $W - w_{1,c}$ is independent of r constant ; so that the loop body can run in constant time, plus the cost of a single call to $minheight$ (this can be made constant with O(W) space overhead, or O(log($|configs_d|$)) otherwise). h_{fix} denotes the height of any newly introduced cells terminating at the current (r, c) position; note that there is at most one such cell. h_{free} denotes the computed lower bound for the unfixed cells in the row. The lower bound for a row r is then the maximum of $h0_r$, the height of previously fixed cells, any newly fixed cells, and the lower bound for the as-yet unfixed cells in the row.

6.3 A CP model for table layout

A Zinc [Marriott et al., 2008] model is given below. Each cell d has a configuration variable $f[d]$ which chooses the configuration (cw, ch) from an array of tuples $cf[d]$ of (width, height) configurations defining $configs_d$. Note that $t.1$ and $t.2$ return the first and second element of a tuple respectively. The important variables are: w, the width of each column, and h, the height of each row. These are constrained to fit each cell, and so that the maximum width is not violated.

```
int: n; % number of columns
int: m; % number of rows
int: W; % maximal width
set of int: Cells; % numbered cells
array[Cells] of 1..m: top;
array[Cells] of 1..m: bot;
array[Cells] of 1..n: left;
array[Cells] of 1..n: right;
array[Cells] of array[int] of tuple(int,int): cf;
```

possible-col-widths(c,w,W)
  $min_c := \max_{d \in rcells_c} \{minw(d) - wd_{left(d),c-1}(w)\}$
  for($c' := n$ down to $c+1$)
    $w_{c'} := \max_{d \in lcells_{c'}} \{minw(d) - wd_{c'+1,right(d)}(w)\}$
  $max_c := W - wd_{1,c-1}(w) - wd_{c+1,n}(w)$
  for($d \in rcells_d$)
    $widths_d := \{w_k - wd_{left(d),c-1}(w) | (w_k, h_k) \in configs_d\}$
    $widths_d := \{v \in widths_d \mid min_c \leq v \leq max_c\}$
  return $(\bigcup_{d \in rcells_d} widths_d)$

weakly-column-minimal(w,c)
  for($r := 1$ to m)
    $D_r := \{d \in Cells \mid right(d) \leq c$ and $rows(d) = \{r\}\}$
    $h_r := \max_{d \in D_r} \{minheight(d, wd_{left(d),right(d)}(w))\}$
  for($c' \in \{1, \ldots, c\}$)
    $cm :=$ false
    for($d \in rcells_{c'}$ s.t. $|rows(d)| = 1$)
      if(minheight$(d, wd_{left(d),c'}(w) - \epsilon) > h_{bot(d)}$)
        $cm := true$
        break
    if($\neg cm$) **return** false
  return true

compute-approx-row-heights(w,$h0$,c,W)
  for($r \in \{1, \ldots, m\}$)
    $D_{fix} := \{d \in Cells \mid right(d) = c$ and $bot(d) = r\}$
    if($D_{fix} = \emptyset$) $h1 := 0$
    else $h_{fix} := \max_{d \in D_{fix}} \{ minheight(d, wd_{left(d),right(d)}(w)) - ht_{top(d),r-1}(h) \}$
    $D_{free} := \{d \in Cells \mid c < right(d)$ and $rows(d) = \{r\}\}$
    $area_r := \sum_{d \in D_{free}} area(d)$
    if($area_r = 0$) $h_{free} := 0$
    else $area_r/(W - wd_{1,c})$
    $h := h[h_r \mapsto \max\{h0_r, h_{fix}, h_{free}\}]$
  return h

complete-A*-search(W)
  create a new priority-queue q
  add $(0, -1, [c \mapsto 0 \mid c = 1..n], [r \mapsto 0 \mid r = 1..m])$ to q
  while(true)
    remove lowest priority state $(p, -c, w, h)$ from q
    if($c = n+1$) **return** (w, h)
    $widths_c :=$ possible-col-widths(c, w, W)
    for($v \in widths_c$ **s.t.** weakly-column-minimal$(w[c \mapsto v], c)$)
      $w' := w[c \mapsto v]$
      $h' :=$ compute-approx-row-heights(w,h,c,W)
      add $(ht(h'), -(c+1), w', h')$ to q

Figure 6.3: An A* algorithm for table layout.

```
array[Cells] of var int: f; % cell configurations
array[1..n] of var int: w;  % column widths
array[1..m] of var int: h;  % row heights

constraint forall(d in Cells)(
      % constraint (2)
  sum(c in left[d]..right[d])(w[c])>=cf[d][f[d]].1
  /\  % constraint (3)
  sum(r in top[d]..bot[d])(h[r])>=cf[d][f[d]].2
);
      % constraint (4)
constraint sum(c in 1..n)(w[c]) <= W;
solve minimize sum(r in 1..m)(h[r]);
```

The Zinc model does not enforce column minimality of the solutions, but solutions will be column minimal because of the optimality condition.

The reason that we thought that lazy clause generation might be so effective for the table layout problem is the small number of key decisions that need to be made. While there may be $O(nm)$ cells each of which needs to have an appropriate configuration determined for it, there are only n widths and m heights to decide. These variables define all communication between the cells. Hence if we learn nogoods about combinations of column widths and row heights there are only a few variables involved, and these nogoods are likely to be highly reusable. We can see the benefit of nogood learning by comparing the constraint programming model, with and without learning.

Example 6.1. *Consider laying out a table of the form*

aa		aa			
	aa		aa	aa	
		aa	aa		
			aa	aa	
				aa	aa
			aa		aa

where each aa entry can have two configurations: **wide** *two characters wide and one line high, or* **high** *one character wide and two lines high (so* `cf[d] = [(2,1),(1,2)]`*). Assume the remaining cells have unique configuration (1,1), and there is a maximal table width of 9, and a maximal table height of 9. Choosing the configuration of cell (1,1) as* **wide** (`f[(1,1)] = 1`) *makes $w_1 \geq 2$, similarly if cell (1,3) is* **wide** *then $w_3 \geq 2$. The*

effect of each decision in terms of propagation is illustrated in the implication graph in Figure 6.4 Now choosing the configuration of cell (2,2) as **wide** *makes* $w_2 \geq 2$ *and then propagation on the sum of column widths forces each of the remaining columns to be at most width 1:* $w_4 \leq 1, w_5 \leq 1, w_6 \leq 1$. *Then* $w_4 \leq 1$ *means* $h_2 \geq 2$, $h_3 \geq 2$, $h_4 \geq 2$ *and* $h_6 \geq 2$ *since we must pick the second configuration for each of the cells in column 4. These height constraints together violate the maximal height constraint. Finite domain propagation backtracks undoing the last decision and sets the configuration of (2,2) as* **high** (`f[(2,2)] = 2`)*, forcing* $h_2 \geq 2$. *Choosing the configuration of (2,5) as* **wide** *makes* $w_5 \geq 2$ *and then propagation on the sum of column widths forces each of the remaining columns to be at most width 1:* $w_2 \leq 1, w_4 \leq 1, w_5 \leq 1$. *Again* $w_4 \leq 1$ *means the maximal height constraint is violated. So search undoes the last decision and sets the configuration of (2,5) as* **high**.

Let's contrast this with lazy clause generation. After making the first three decisions the implication graph is shown in Figure 6.4. The double boxed decisions reflect making the cells (1,1), (1,3) and (2,2) wide. The consequences of the last decision are shown in dashed boxes. Lazy clause generation starts from the nogood $h_2 \geq 2 \wedge h_3 \geq 2 \wedge h_4 \geq 2 \wedge h_6 \geq 2 \rightarrow false$ *and replaces* $h_6 \geq 2$ *using its explanation* $f[(6,4)] = 2 \rightarrow h_6 \geq 2$ *to obtain* $h_2 \geq 2 \wedge h_3 \geq 2 \wedge h_4 \geq 2 \wedge f[(6,4)] = 2 \rightarrow false$. *This process continues until it arrives at the nogood* $w_4 \leq 1 \rightarrow false$ *which only has one literal from the last decision level. This is the 1UIP nogood. It will immediately backjump to the start of the search (since the nogood does not depend on any other literals at higher decision levels) and enforce that* $w_4 \geq 2$. *Search will again make cell (1,1)* **wide**, *and on making cell (1,3)* **wide** *it will determine* $w_3 \geq 2$ *and consequently that* $w_2 \leq 1$, $w_5 \leq 1$ *and* $w_6 \leq 1$ *which again causes violation of the maximal height constraint. The 1UIP nogood is* $w_1 \geq 2 \wedge w_3 \geq 2 \rightarrow fail$, *so backjumping removes the last choice and infers that* $w_3 \leq 1$ *which makes* $h_1 \geq 2$ *and* $h_3 \geq 2$.

Note that the lazy clause generation completely avoids considering the set of choices (1,1), (1,3) and (2,5) **wide** *since it already fails on setting (1,1) and (1,3)* **wide**. *This illustrates how lazy clause generation can reduce search. Also notice that in the implication graph the consequences of a configuration choice only propagate through width and height variables, and hence configuration choices never appear in nogoods.* □

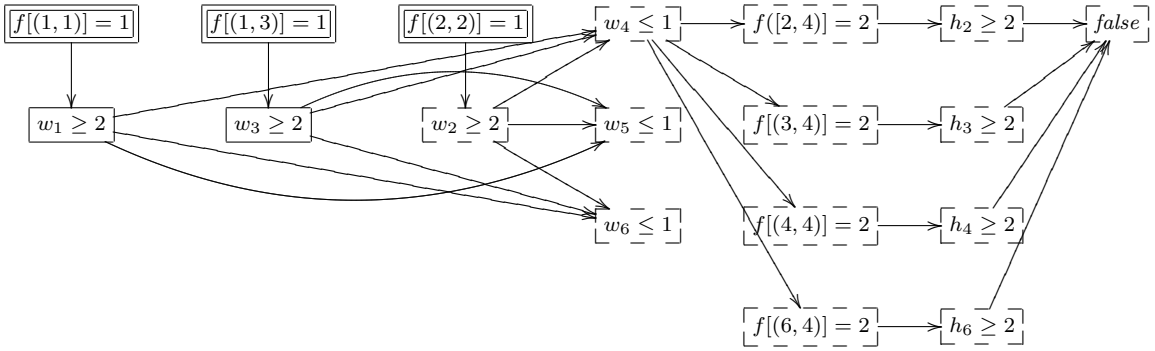


Figure 6.4: An implication graph for searching the layout problem of Example 6.1

6.4 Cell-free CP model

The CP model described in the previous section introduces O(mn) configuration variables in order to represent cell shapes; in practice, the solver spends a significant percentage of runtime updating these variables.

However, we don't necessarily need to explicitly represent each cell in the model. Consider a cell spanning a single row r and single column c with a set of configurations $C = \{(w_1, h_1), \ldots, (w_k, h_k)\}$, the configurations ordered according to increasing width (or, equivalently, decreasing height). Consider two adjacent configurations, (w_i, h_i) and (w_{i+1}, h_{i+1}).

If the cell is in a configuration $c \leq i$, then $h[r] \geq h_i$. If the cell is in a configuration $c \geq i + 1$, then $w[c] \geq w_{i+1}$. By introducing these constraints for each pair of configurations, we can avoid the need to introduce configuration variables for each cell. The revised model is given below.

```
constraint forall(d in Cells)(
     % minimum height
     sum(c in left[d]..right[d])(w[c]) >= cf[d][0].1
  /\ forall(i in 2..length(cf[d]))(
         % boundary between pairs of configurations
         sum(c in left[d]..right[d])(w[c]) >= cf[d][i].1
      \/ sum(r in top[d]..bot[d])(h[r]) >= cf[d][i-1].2
     )
     % minimum width
  /\ sum(r in top[d]..bot[d])(h[r]) >= cf[d][length(cf[d])].2
);
```

In the case of non-compound tables, this is simplified to:

```
constraint forall(d in Cells)(
     % minimum height
```

```
      w[left[d]] >= cf[d][0].1
  /\ forall(i in 2..length(cf[d]))(
           % boundary between pairs of configurations
           w[left[d]] >= cf[d][i].1
        \/ h[top[d]] >= cf[d][i-1].2
      )
      % minimum width
  /\ h[top[d]] >= cf[d][length(cf[d])].2
);
```

As lazy clause generation solvers introduce literals for variable bounds, each of these constraints becomes a single propositional clause. This can be handled efficiently by the underlying SAT solver, and avoids the cost of constantly updating the domains of the (now non-existent) cell variables.

Example 6.2. *Consider the aa cell described in Example 6.1, which has configurations* $\{(1,2),(2,1)\}$*, spanning the single column* c *and single row* r*. The constraints generated for the cell are*

```
    w[r] >= 1
/\ (w[c] >= 2 \/ h[r] >= 2)
/\ h[c] >= 1
```

□

6.5 Mixed Integer Programming

Bilauca and Healy [2010] consider using mixed integer programming (MIP) to model the table layout problem. They consider two models for the simple table layout problem and do not consider compound cells, i.e. row and column spans. Their basic model BMIP uses 0–1 variables (`cellSel`) for each possible configuration, to model the integer configuration choice `f` used in the CP model. This basic model can be straightforwardly extended to handle compound cells.

Their improved model adds redundant constraints on the column widths to substantially improve MIP solving times for harder examples. They compute the minimum width (`minW`) for each column as the maximum of the minimum widths of the cells in the column, and the minimum height (`minH`) for each row analogously. They then compute the set of possible column widths (`colWSet`) for each column from those configurations in the column which have at least width `minW` and height `minH`. Note this improvement relies on the fact that there are no column

spans. While in practice this usually does provide the set of possible widths in any column-minimal layout, in general the "improved model" is incorrect.

Example 6.3. *Consider laying out a* 2×2 *table with cell configurations* $\{(1,3),(3,1)\}$ *for the top left cell, and* $\{(2,2)\}$ *for the remaining cells, with a width limit of 5. The minimal height of the first row is 2. The minimal width of the first column is 2. Hence none of the configurations of the top left cell are both greater than the minimal height and minimal width. The possible column widths for column 1 are then computed as* $\{2\}$*. The only layout is then choosing configuration* $(1,3)$ *for the top left cell, giving a total height of 5. Choosing the other configuration leads to a total table height of 4.*

We can fix this model by including in the possible column widths the smallest width configuration for each cell in that column which is less than the minimum row height. For the example above this means the possible columns widths become $\{2,3\}$. Bilauca and Healy [2010] give an OPL model of their "improved model", to contrast it with the CP model above we give a corresponding corrected Zinc model MIP.

```
int: m; % number of rows
int: n; % number of columns
int: W; % maximal width
array[1..m,1..n] of set of tuple(int,int): cf;

array[1..n] of int: minW = [ max(r in 1..m)
  (min(t in cf[r,c])(t.1)) | c in 1..n ];
array[1..m] of int: minH = [ max(c in 1..n)
  (min(t in cf[r,c])(t.2)) | r in 1..m ];
array[1..n] of set of int: colWset =
  [ { t.1 | r in 1..m, t in cf[r,c] where
    t.1 >= minW[c] /\ (t.2 >= minH[r] \/
    t.1 == min({ u.1 | u in cf[r,c] %FIX
                where u.2 < minH[r] })) }
  | c in 1..n];

array[1..n] of array[int] of var 0..1: colSel =
  [ [ d:_ | d in colWset[c] ] | c in 1..n ];
array[1..n,1..m] of array[int,int] of var 0..1:
  cellSel =
  array2d(1..n,1..m,[ [ t:_ | t in cf[r,c] ]
                    | r in 1..m, c in 1..n ]);
array[1..n] of var int: w;
array[1..m] of var int: h;

constraint forall(r in 1..m, c in 1..n)(
```

```
  sum(t in cf[r,c])(cellSel[r,c][t]) = 1 /\
  sum(t in cf[r,c])(cellSel[r,c][t] * t.1) <= w[c] /\
  sum(t in cf[r,c])(cellSel[r,c][t] * t.2) <= h[r]);
constraint forall(c in 1..n)(
  sum(d in colWset[c])(colSel[c][d]) = 1 /\
  w[c] = sum(d in colWset[c])(colSel[c][d] * d));

constraint sum(c in 1..n)(w[c]) <= W;
solve minimize sum(r in 1..m)(h[r]);
```

Note that Bilauca and Healy also consider a CP model of the problem, but this is effectively equivalent to the MIP model because they do not make use of variable array indices (`element` constraints) that FD solvers support and which allow stronger propagation than that of the 0–1 encoding.

6.6 Evaluation

We compare different approaches to optimal table layout: the A* algorithm of Section 6.2; the two constraint programming models of Section 6.3, namely the basic CP implementation without learning (CP-W), the model using lazy clause generation to provide learning (CP) and the cell-free model eliminating separate cell variables (CP_{cf}); and BMIP the basic MIP model of Bilauca and Healy [2010], and MIP the (corrected) improved model of Bilauca and Healy [2010] described in Section 6.5. For the CP approaches, both CP-W and CP_{seq} use a sequential search that chooses a cell that has the smallest height configuration remaining of all unfixed cells and tries to set it to that minimal height. For the lazy clause generation solver CP_{vsids} and CP_{cf} both use the default activity based search.

The $A^\star$ algorithm is written in the high-level declarative programming language Mercury [Somogyi et al., 1996]. Notes in Section 6.2 give some idea of what optimizations have or haven't been applied to the source code of the implementation shown in these timings.

For the constraint programming approaches we used the CHUFFED lazy clause generation solver (which can also be run without nogood generation). CHUFFED is a state-of-the-art CP solver, which scored the most points in all categories of the 2010 MiniZinc Challenge[2] which compares CP solvers. Since CHUFFED does not

[2] `http://www.g12.csse.unimelb.edu.au/minizinc/challenge2010/results2010.html`

currently support Zinc, we created the model using the C++ modelling capabilities of CHUFFED. The resulting constraints are identical to that shown in the model.

For the MIP approach, we used a script to construct a mixed integer programming model for each table, identical to that created by the Zinc model (and the (corrected) original OPL model of Bilauca and Healy), which was solved using CPLEX 12.1.

We first evaluated the various approaches using a large corpus of real-world tables. This was obtained by collecting more than 10,000 web pages using a web crawler, extracting non-nested tables (since we have not considered how to handle nested tables efficiently), resulting in over 50,000 tables. To choose the goal width for each table, we laid out each web page for three viewport widths (760px, 1000px and 1250px) intended to correspond to common window widths. Some of the resulting table layout problems are trivial to find solutions for; we retained only those problems that our $A^\star$ implementation took at least a certain amount of time to solve. (Thus, the choice may be slightly biased against our $A^\star$ solver, insofar as some other solver might find different examples to be hard.) This left 2063 table layout problems in the corpus. We split the corpus into **web-compound** and **web-simple** based on whether the table contained compound cells or not.

Table 6.1 shows the results of the different methods on the **web-simple** examples. The table shows the number of tables laid out optimally for various time limits up to 10 seconds. Note that the last row indicates the number of instances *not* solved within the time-limit. They show that in practice for simple tables all of the methods are very good, and able to optimally layout almost all tables very quickly. The worst method is CP-W and the evaluation clearly shows the advantage of learning for constraint programming. We find, like Bilauca and Healy [2010], that the improved MIP model MIP while initially slower is more robust than basic model BMIP. Overall CP_{cf} is the most robust approach never requiring more than 0.1 second on any example. However, the performance of the A* method is surprisingly good given the relative simplicity of the approach in comparison to the sophisticated CPLEX and CHUFFED implementations and the use of Mercury rather than C or C++.

Table 6.2 shows the results of the different methods on the **web-compound** examples. We compare all the previous algorithms except for MIP since it is not ap-

time (s)	CP-W	CP$_{seq}$	CP$_{vsids}$	CP$_{cf}$	BMIP	MIP	A*
≤ 0.01	1018	1183	1097	**1264**	1009	774	961
≤ 0.10	1064	1267	1217	**1271**	1160	1076	1162
≤ 1.00	1103	**1271**	1260	**1271**	1221	1223	1259
≤ 10.00	1120	**1271**	**1271**	**1271**	1261	1270	1269
> 10.00	151	**0**	**0**	**0**	10	1	2

Table 6.1: Number of instances from the **web-simple** data-set solved within each time limit.

time (s)	CP-W	CP$_{seq}$	CP$_{vsids}$	CP$_{cf}$	BMIP	A*
≤ 0.01	708	738	695	**749**	702	630
≤ 0.10	721	767	760	**771**	760	751
≤ 1.00	734	775	774	778	**787**	**787**
≤ 10.00	742	778	780	782	790	**792**
> 10.00	50	14	12	10	2	**0**

Table 6.2: Number of instances from the **web-compound** data-set solved within each time limit.

plicable when there are compound cells. The results are somewhat different to the simple tables. While CP$_{cf}$ is still fastest for the easier tables, all the CP approaches have difficulty with some of the harder compound tables. The $A^\star$method appears to be the most robust method on these instances. The poor performance of the CP approaches appears to be due to the large number of symmetric solutions in cases where the height of a set of rows is dominated by a row-span (or equivalently width of a set of columns).

Given the relatively similar performance of the approaches on the real-world tables we decided to "stress-test" the approaches on some harder artificially constructed examples. We only used simple tables so that we could compare with MIP. Table 6.3 shows the results. We created tables of size $m \times n$ each with k configurations by taking text from the Gutenberg project edition of The Trial [Kafka, 1925, 2005] k words at a time, and assigning to a cell all the layouts for that k words using fixed width fonts. For the experiments we used $k = 6$. We compare different versions of the layout problem by computing the minimum width $minw$ of the table as the sum of the minimal column widths, and the maximal width $maxw$ of the table as the sum of the column widths resulting when we choose the minimal height for each row. The *squeeze* s for table is defines as $(W - minw)/maxW$. We compare the table layout for 5 different values of squeeze. Obviously with a squeeze of 0.0 or 1.0 the problem is easy, the interesting cases are in the middle.

	s	CP$_{seq}$	CP$_{vsids}$	CP$_{cf}$	BMIP	MIP	A*
10×10	0.00	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.02
	0.25	3.29	0.02	**0.00**	0.16	0.97	0.07
	0.50	0.3	0.01	**0.00**	0.22	0.56	0.04
	0.75	0.07	0.02	**0.00**	0.55	1.18	0.08
	1.00	**0.00**	**0.00**	**0.00**	0.01	0.01	0.00
20×20	0.00	0.02	0.01	**0.00**	0.01	0.04	0.19
	0.25	—	0.86	**0.05**	27.18	28.65	36.15
	0.50	—	0.07	**0.00**	188.27	163.86	10.76
	0.75	—	0.28	**0.01**	43.83	40.07	58.75
	1.00	0.02	0.01	**0.00**	0.04	0.08	0.00
30×30	0.00	0.04	0.03	**0.00**	0.04	0.07	1.53
	0.25	—	254.47	**8.07**	—	253.08	—
	0.50	—	0.38	**0.02**	—	—	—
	0.75	—	9.4	**0.2**	—	—	—
	1.00	0.04	0.04	**0.00**	0.10	0.18	0.01
40×40	0.00	0.09	0.06	**0.01**	0.07	0.20	3.78
	0.25	—	—	—	—	—	—
	0.50	—	1.11	**0.02**	—	—	—
	0.75	—	216.67	**3.4**	—	—	—
	1.00	0.09	0.05	**0.01**	0.19	0.34	0.02

Table 6.3: Results for artificially constructed tables. Times are in seconds.

The harder artificial tables illustrate the advantages of the conflict directed search of CP$_{vsids}$ and CP$_{cf}$. On the **simple** and **artificial** instances, CP$_{cf}$ is uniformly the best method, generally 1–2 orders of magnitude faster than CP$_{vsids}$, and up to 4 orders of magnitude faster than other methods.

The difference in behavior between the real-world and artificial tables may be due to differences in the table structure. The tables in the **web-simple** and **web-compound** corpora tend to be narrow and tall, with very few configurations per cell – the widest table has 27 columns, compared with 589 rows, and many cells have only one configuration. On these tables, the greedy approach of picking the widest (and shortest) configuration tends to quickly eliminate tall layouts. The **artificial** corpus, having more columns and more configurations (but without the symmetries of the **compound** instances), requires significantly more search to prove optimality; in these cases, the learning and conflict-directed search of CP$_{vsids}$ and CP$_{cf}$ provides a significant advantage.

6.7 Conclusion

Treating table layout as a constrained optimization problem allows us to use powerful generic approaches to combinatorial optimization to tackle these problems. We have given three new techniques for finding a minimal height table layout for a fixed width: the first uses an $A^\star$ based approach while the second approach uses pure constraint programming (CP) and the third uses lazy clause generation, a hybrid CP/SAT approach. We have compared these with two MIP models previously proposed by Bilauca and Healy.

An empirical evaluation against the most challenging of over 50,000 HTML tables collected from the Web showed that all methods can produce optimal layout quickly.

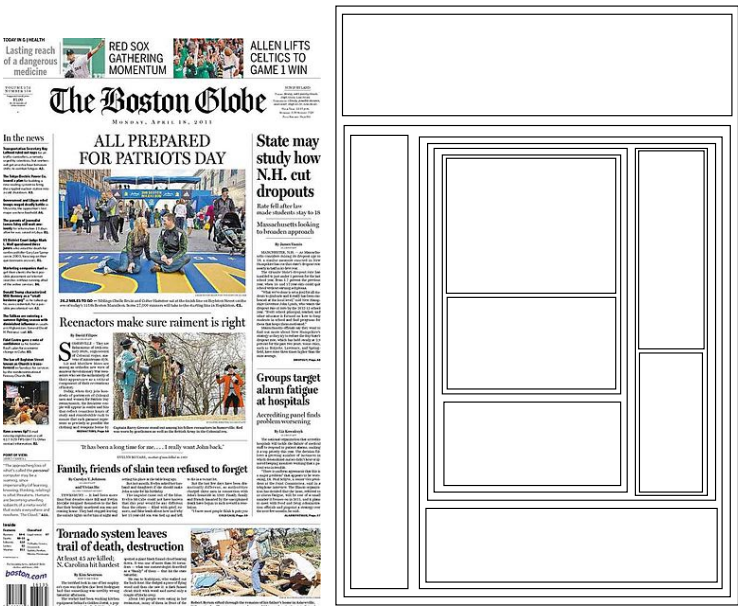
The $A^\star$-based algorithm is more targeted than the constraint-programming approaches: while the $A^\star$ algorithm did well on the web-page-like tables for which it was designed, we would expect that more generic constraint-programming approaches would be a safer choice for other types of large tables. This turned out to be the case for the large artificially constructed tables we tested, where the approach using lazy clause generation was significantly more effective than the other approaches; however, the lazy clause generation approach performed poorly in cases with many overlapping row- or column-spans.

All approaches can be easily extended to handle constraints on table widths such as enforcing a fixed size or that two columns must have the same width. Handling nested tables, especially in the case cell size depends in a non-trivial way on the size of tables inside it (for example when floats are involved) is more difficult, and is something we plan to pursue.

7

Guillotine-based Text Layout

GUILLOTINE-BASED page layout is a method for document layout, commonly used by newspapers and magazines, where each region of the page either contains a single article, or is recursively split either vertically or horizontally. The newspaper page shown in Figure 7.1(a) is an example of a guillotine-based layout where Figure 7.1(b) shows the series of cuts used to construct this layout.



(a) (b)

Figure 7.1: (a) Front page of The Boston Globe, together with (b) the series of cuts used in laying out the page. Note how the layout uses fixed width columns.

Surprisingly, there appears to have been relatively little research into algorithms for automatic guillotine-based document layout. We assume that we are given a sequence of articles $A_1, A_2, \ldots, A_n$ to layout. The precise problem depends upon the page layout model [Hurst et al., 2009].

- The first model is vertical scroll layout in which layout is performed on a single page of fixed width but unbounded height: this is the standard model for viewing HTML and most web documents. Here the layout problem is to find guillotine layout for the articles which minimizes the height for a fixed width.

- The second model is horizontal scroll layout in which there is a single page of fixed height but unbounded width. This model is well suited to multicolumn layout on electronic media. Here the layout problem is to find guillotine layout for the articles which minimizes the width for a fixed height.

- The final model is layout for a sequence of articles in fixed height and width pages. Here the problem is to find a guillotine layout which maximises the prefix of the sequence of articles $A_1, A_2, \ldots, A_k$ that fit on the (first) page (and then subsequently for the second, third, ... page).

We are interested in two variants of these problems. The easier variant is *fixed-cut* guillotine layout. Here we are given a guillotining of the page and an assignment of articles to the rectangular regions on the page. The problem is to determine how to best layout each article so as to minimize the overall height or width. The much harder variant is *free* guillotine layout. In this case we need to determine the guillotining, article assignment and the layout for each article so as to minimize overall height or width.

The main contribution of this chapter is to give polynomial-time algorithms for optimally solving the fixed-cut guillotine layout problem and a dynamic programming based algorithm for optimally solving the free guillotine layout. While our algorithm for free guillotine layout is exponential (which is probably unavoidable since the free guillotine layout problem is NP-Hard (see Section 7.1), it can layout up to 13 articles in a few seconds (up to 18 if the articles must use columns of a fixed width).

Our automatic layout algorithms support a novel interaction model for viewing documents such as newspapers or magazines on electronic media. In this model we use free guillotine layout to determine the initial layout. We can fine tune this layout using fixed-cut guillotine layout in response to user interaction such as changing the font size or viewing window size. Using the same choice of guillotining ensures the basic relative position of articles remains the same and so the layout does not change unnecessarily and disorient the reader. An example of this is shown in Figure 7.2. However, if at some point the choice of guillotining leads to a very bad layout, such as articles that are too wide or narrow or too much wasted space, then we can find a new guillotining that is close to the original guillotining, and re-layout using this new choice.

Guillotine-based constructions have been considered for a variety of document composition problems. Photo album composition approaches [Atkins, 2008] have a fixed document size, and must construct an aesthetically pleasing layout while maintaining the aspect ratio of images to be composed.

A number of heuristics have been developed for automated newspaper composition [González et al., 1999, Strecker and Hennig, 2009] which also focus on constructing layouts for a fixed page-width. The first approach [González et al., 1999] considers only a single one column configuration per article, and lays out all articles to minimize height in a fixed number of columns. The second approach [Strecker and Hennig, 2009] breaks the page into a grid and considers up to 8 configurations on grid boundaries per article, It focuses on choosing which articles to place in a fixed page size, using a complex objective based on coverage. Both approaches make use of local search and do not find optimal solutions.

Hurst [2009] suggested solving the fixed-cut guillotine layout problem by solving a sequence of one-dimensional minimisation problems to determine a good layout recursively. This approach was fast but not guaranteed to find an optimal layout.

A closely related problem to these is the guillotine stock-cutting problem. Given an initial rectangle, and a (multi-)set S of smaller rectangles with associated values, the objective is to find a cutting pattern which gives the set $S' \subseteq S$ with maximum value. This in some sense a harder form of the third model we discuss above. A number of exact [Christofides and Whitlock, 1977, Christofides and Hadjiconstanti-

nou, 1995] and heuristic [Alvarez-Valdés et al., 2002] methods have been proposed for the guillotine stock-cutting problem. This differs from the guillotine layout problem in that each leaf region has a single configuration, rather than a (possibly large sized) disjoint set of possible configurations. It does not appear that these approaches scale to the size of problem we consider.

The remainder of this chapter is structured as follows. In Section 7.1, we give a formal definition of the guillotine layout problem. Then, in Section 7.2 we give bottom-up and top-down algorithms for solving the fixed guillotine layout problem, and in Section 7.3 for the free guillotine layout problem. In Section 7.4 we describe an algorithm for updating layouts. In Section 7.5 we present an experimental evaluation of the described algorithms, and in Section 7.6 we conclude.

7.1 Problem Statement

In the rest of the chapter will focus on finding a guillotine layout which minimizes the height for a fixed width. It is straightforward to modify our algorithms to find a guillotine layout which minimizes the width for a fixed height: we simply swap height and widths in the input to the algorithms.

We can also use algorithms for minimising height to find a guillotine layout maximising the number of articles in a fixed size page. For a particular subsequence $A_1, .., A_k$ we can use the algorithm to compute the minimum height h_k for laying them out in the page width. We simply perform a linear or binary search to find the maximum k for which h_k is less than the fixed page height. We can use the area of the articles' content to provide an initial upper bound on k.

The main decision in the fixed-cut guillotine layout is how to break the lines of text in each article. Different choices give rise to different width/height configurations. Each article has a number of *minimal configurations* where a minimal configuration is a pair (w, h) such that the content in the article can be laid out in a rectangle with width w and height h but there is no smaller rectangle for which this is true. That is, for all $w' \leq w$ and $h' \leq h$ either $h = h'$ and $w = w'$, or the content does not fit in a rectangle with width w' and height h'.

Typically we would like the article to be laid out with multiple columns. One way of doing this is to allow the configuration to take any width and to compute the

number of columns and their width based on the width of the configuration. We call this *article dependent* column layout. In this case for text with uniform height with W words (or more exactly, $W - 1$ possible line breaks), there are up to W minimal configurations, each of which has a different number of lines. In the case of non-uniform height text, there can be no more than $O(W^2)$ minimal configurations.

The other way of computing the columns is to compute the width and number of columns based on the page width and then each article is laid out in a configuration of one, two, three etc column widths. This is, for instance, the approach used in Figure 7.1. We call this *page dependent* column layout. In this case the number of different configurations is much less and is simply the number of columns on the page.

We assume the minimal configurations for an article A are given as a discrete list of allowed configurations $C(A) = [\ (w_0, h_0), \ldots, (w_k, h_k)\ ]$, ordered by increasing width (and decreasing height). In the algorithms described in the following sections, we refer to the i^{th} entry of an ordered list L with $L[i]$ (adopting the convention that indices start at 0), and concatenate lists with $+\!+$. For a configuration c, we use $\mathsf{w}(c)$ to indicate the width, and $\mathsf{h}(c)$ for the height. Note that we can choose to exclude configurations that are too narrow or too wide.

A guillotine cut is represented by a tree of cuts, where each node has a given height/width configuration. A leaf node $\textsc{cell}(A)$ in the tree holds an article A. An internal node is either: $\textsc{vert}(X, Y)$, where X and Y are its child nodes, representing a vertical split with articles in X to the left and articles in Y to the right; or $\textsc{horiz}(X, Y)$, representing a horizontal split with articles in X above and articles in Y below. Given a chosen configuration for each leaf node we can determine the configuration of each internal nodes as follows:

If $c(X) = (w_x, h_x)$ is the chosen configuration for X and $c(Y) = (w_y, h_y)$ is the chosen configuration for Y, then define

$$\mathsf{vert}((w_x, h_x), (w_y, h_y)) = (w_x + w_y, \max(h_x, h_y))$$
$$\mathsf{horiz}((w_x, h_x), (w_y, h_y)) = (\max(w_x, w_y), h_x + h_y)$$

and let

$$c(\text{VERT}(X, Y)) = \mathsf{vert}(c(X), c(Y))$$
$$c(\text{HORIZ}(X, Y)) = \mathsf{horiz}(c(X), c(Y)).$$

The *fixed-cut guillotine layout problem* for fixed width w is given a fixed tree T, determine the configuration of leaf nodes (and internal nodes) such that $c(T) = (w_r, h_r)$ where $w_r < w$ and h_r is minimized.

The *free guillotine layout problem* for fixed width w is given a set of articles S determine the guillotine cut T for S and configurations of leaf nodes (and internal nodes) such that $c(T) = (w_r, h_r)$ where $w_r < w$ and h_r is minimized.

We note that the free guillotine layout problem is NP-hard.

Theorem 1. *The decision problem for* FREE-GUILLOTINE *is NP-complete*

Proof. Consider an instance $\langle\{n_1, \ldots, n_k\}, d\rangle$ of BIN-PACKING [Garey and Johnson, 1979]. We construct a free guillotine layout instance with $w = d$, and leaves $L_1 \ldots L_n$ with configurations $C(L_k) = \{(n_k, 1)\}$.

Assume there is a solution to this guillotine instance with height h. Compute the y-coordinate of each leaf node. For each node L_j at y-coordinate i, we allocate n_j to bin i. As leaves cannot overlap, the sum of leaf widths for a given y-value is at most d; therefore, this constructs a valid bin-packing solution with h bins.

Assume there is a bin-packing solution $\{\{n_{1,1}, n_{1,2}, \ldots\}, \ldots, \{n_{h,1}, n_{h,2}, \ldots\}\}$. We can construct a guillotine layout with tree:

$$\mathsf{horiz}(\mathsf{vert}(L_{1,1}, \mathsf{vert}(L_{1,2}, \ldots)), \mathsf{horiz}(\ldots, \mathsf{vert}(L_{h,1}, \mathsf{vert}(L_{h,2}, \ldots))))$$

Each horizontal layer has width at most d, and there are exactly h layers. Therefore the instance of BIN-PACKING has a solution with h bins iff the instance of FREE-GUILLOTINE has a solution of height h. As BIN-PACKING is strongly NP-complete, FREE-GUILLOTINE layout must also be strongly NP-hard. Given a fixed tree of cuts and (w, h) for each leaf, we can check that the layout is valid and has height at most h by making a linear walk through the tree; therefore, FREE-GUILLOTINE is in NP. As FREE-GUILLOTINE is in NP, and is NP-hard, FREE-GUILLOTINE is NP-complete. □

Figure 7.2: Example of (a) a possible guillotine layout, and (b) the same layout adapted to a narrower display width.

7.2 Fixed-cut Guillotine Layout

We will first look at solving the fixed-cut guillotine layout problem. This is a restricted form of guillotine layout, where the tree of cuts is specified, and the algorithm must pick a configuration for each article which leads to the minimum height layout. Fixed-cut guillotine layout is useful in circumstances such as online newspapers, where the layout should remain consistent, but must adapt to changes in display area. An example of this is given in Figure 7.2. It might also be useful in semi-automatic document authoring tools that support guillotine layout.

7.2.1 Bottom-up construction

Dynamic programming is a natural approach to tackle minimum height guillotine layout problems since each sub problem of the guillotine layout is again a (smaller) minimum height guillotine layout problem. The only real choices that arise in fixed-cut layout are where to place the vertical split between X and Y in a vertical cut $\textsc{vert}(X, Y)$ in order to obtain the minimal height. Rather than searching for a best vertical cut, we solve this problem in the bottom-up construction by computing the *list* of minimal configurations, $C(X)$ for each subtree X of T.

Consider a node $\textsc{vert}(X, Y)$, where $C(X)$ is the list of minimal configurations for X and $C(Y)$ is the list of minimal configurations for Y. To construct the list of minimal configurations for $\textsc{vert}(X, Y)$ we iterate across the configurations $C(X)$ and $C(Y)$. Given a minimal configuration $\mathsf{vert}(C(X)[i], C(Y)[j])$, we can find the next next minimal configuration that is wider (and shorter). If $C(X)[i]$ is taller than

```
join_vert(CX,CY,w)
    C := ∅
    i := 0
    j := 0
    while (i ≤ |CX| ∧ j ≤ |CY|)
        (w_x, h_x) := CX[i]
        (w_y, h_y) := CY[j]
        if(w_x + w_y > w) break
        C := C ++[ vert(CX[i], CY[j]) ]
        if(h_x > h_y) i := i + 1
        else if(h_x < h_y) j := j + 1
        else
            i := i + 1
            j := j + 1
    return C
```

Figure 7.3: Algorithm for constructing minimal configurations for a vertical split from minimal configurations for the child nodes.

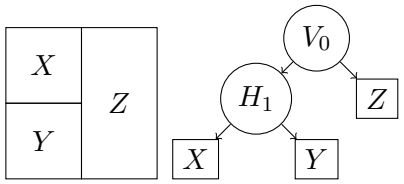


Figure 7.4: Cut-tree for Example 7.1.

$C(Y)[j]$, we can only construct a shorter configuration by picking a shorter configuration for X. In fact, since any narrower configuration for Y will be strictly taller than $C(X)[i]$ (otherwise $\mathsf{vert}(C(X)[i], C(Y)[j])$ would not be minimal), and any shorter configuration will be strictly wider, the next minimal configuration is exactly $\mathsf{vert}(C(X)[i+1], C(Y)[j])$. We can use similar reasoning for the cases where $C(X)[i]$ is shorter than $C(Y)[j]$. Since $\textsc{vert}(C(X)[0], C(Y)[0])$ is the narrowest minimal configuration, we can construct all minimal configurations by performing a linear scan over $C(X)$ and $C(Y)$. Pseudo-code for this is given in Figure 7.3.

Example 7.1. *Consider a problem with 3 articles $\{X, Y, Z\}$ having configurations $C(X) = C(Y) = [\ (1,2), (2,1)\ ], C(Z) = [\ (1,3), (2,2), (3,1)\ ]$, and the tree of cuts shown in Figure 7.4.*

Consider finding the optimal layout for $w = 3$. First we must construct the minimal configurations for the node marked H_1. We start by picking the narrowest configurations for X and Y, giving $C(H_1) = [\ (1,4)\ ]$. We then need to select the next narrowest con-

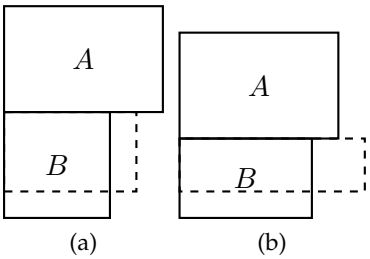


Figure 7.5: (a) In this case, the initial configurations of A and B do not form a minimal configuration. (b) Even though A has no further configurations, we can construct additional minimal configurations by picking a shorter configuration for B.

figuration from either X or Y. Since both have the same width, we then join both $(2,1)$ configurations, to give $C(H_1) = [\ (1,4),(2,2)\ ]$.

We then construct the configurations for V_0. We again select the narrowest configurations, $C(H_1)[0]$ and $C(Z)[0]$, giving $C(V_0) = [\ (2,4)\ ]$. Since $C(H_1)[0]$ is taller, we select the next configuration from H_1. Combining $C(H_1)[1]$ with $C(Z)[0]$ gives us $C(V_0) = [\ (2,4),(3,3)\ ]$. Since $w = 3$, we can terminate at this point, giving $(3,3)$ as the minimal configuration. If w were instead 4, we would combine $C(H_1)[1]$ with $C(Z)[1]$, giving the new configuration $(4,2)$. □

Constructing the minimal configurations for $\text{HORIZ}(X,Y)$ is exactly the dual of the vertical case. From a minimal configuration constructed from $C(X)[i]$ and $C(Y)[j]$, we can construct a new minimal configuration by picking the narrowest of $C(X)[i+1]$ and $C(Y)[j+1]$. The only additional complexity is that (a) $\text{HORIZ}(C(X)[0], C(Y)[0])$ is not guaranteed to be a minimal configuration, and (b) we must keep producing configurations until both children have no more successors, rather than just one. These cases are illustrated in Figure 7.5. Pseudo-code for this is given in Figure 7.6, and the overall algorithm is in Figure 7.7.

Consider a cut $\text{VERT}(X,Y)$ with children X and Y. Given $C(X)$ and $C(Y)$, the algorithm described in Figures 7.3 to 7.7 computes the configurations for $C(\text{VERT}(X,Y))$ in $\text{O}(|C(X)| + |C(Y)|)$, yielding at most $|C(X)| + |C(Y)|$ configurations (and similarly for $\text{HORIZ}(X,Y)$). Given a set of leaf nodes S, we construct at most $\sum_{A \in S} |C(A)|$ configurations at any node. As we perform this step $|S|-1$ times, this gives a worst-case time complexity of $O(|S| \sum_{A \in S} |C(A)|)$ for the bottom-up construction.

An advantage of the bottom-up construction method is that, if we record the lists of constructed configurations, we can update the layout for a new width in

```
join_horiz(CX,CY,w)
    C := ∅
    minW := max(w(CX[0]), w(CY[0]))
    i := arg max_{i'} w(CX[i']) s.t. w(CX[i']) ≤ minW
    j := arg max_{j'} w(CY[j']) s.t. w(CY[j']) ≤ minW
    while (i ≤ |CX| ∨ j ≤ |CY|)
        C := C ++[ horiz(CX[i], CY[j]) ]
        (w_x, h_x) := CX[i + 1]
        (w_y, h_y) := CY[j + 1]
        if(j + 1 = |CY| ∨ w_x > w_y) i := i + 1
        else if(i + 1 = |CX| ∨ w_x < w_y) j := j + 1
        else
            i := i + 1
            j := j + 1
    return C
```

Figure 7.6: Algorithm for producing minimal configurations for a horizontal split from child configurations. While the maximum width is included as an argument for consistency, we don't need to test any of the generated configurations, since the width of the node is bounded by the width of the input configurations.

```
fixguil_BU(T,w)
    switch (T)
        case CELL(A):
            return C(A)
        case VERT(T_1, T_2):
            return join_vert(fixguil_BU(T_1, w),
                             fixguil_BU(T_2, w), w))
        case HORIZ(T_1, T_2):
            return join_horiz(fixguil_BU(T_1, w),
                              fixguil_BU(T_2, w), w))
```

Figure 7.7: Algorithm for constructing the list of minimal configurations for a fixed set of cuts.

$O(\log |C| + |T|)$ time by performing a binary search on configurations of the root node using the new width, then follow the tree of child configurations (or $O(|T|)$ time if we use $O(w)$ space to construct a lookup table).

7.2.2 Top-down dynamic programming

We also consider a top-down dynamic programming approach, where subproblems are expanded only when required for computing the optimal solution. Consider a subproblem layout(HORIZ(X, Y), w). Using a top-down method, we need only to calculate subproblems layout(X, w) and layout(Y,w), rather than all configurations for the current node. The difficulty is in the case of vertical cuts, as we cannot determine directly how much of the available width should be allocated to X or Y. As such, we must compute layout(X, w') and layout(Y, $w - w'$) for the set of possible cut positions w'.

A top down dynamic programming solution is almost a direct statement of the Bellman equations as a functional program, with caching to avoid repeated computation. The main difficulty is the requirement to examine every possible width when determining the best vertical split. Psuedo-code is given in Figure 7.8, where lookup(T, w) looks in the cache to see if there is an entry $(T, w) \mapsto c$ and returns c if so, or NOTFOUND if not; and cache(T, w, c) adds an entry $(T, w) \mapsto c$ to the cache.

This algorithm is outlined in Figure 7.8. Note that for simplicity we ignore the case where there is no layout of tree T with width $\leq w$. This can be easily avoided by adding an artificial configuration $(0, \infty)$ to the start of the list of configurations for each article A.

While the algorithm finds the optimal solutions quite quickly for fixed trees, there are a number of improvements to this basic algorithm which will also be useful for the free layout problem (Section 7.3).

Restricting vertical split positions

The algorithm given in Figure 7.8, on a vertical split, must iterate over all possible values of w' to find the optimal cut position. Let w^T_{min} indicate the narrowest possible configuration for T. Since we are only interested in feasible layouts, for a node VERT(T_1, T_2) we need only consider cut positions in $[w^{T_1}_{min}, w - w^{T_2}_{min}]$. We

```
fixguil_TD(T,w)
    c := lookup(T,w)
    if c ≠ NOTFOUND return c
    switch (T)
        case CELL(A):
            c := C(A)[i] where i is maximal s.t. w(C(A)[i]) ≤ w
        case HORIZ(T1, T2):
            c := horiz(fixguil_TD(T1, w), fixguil_TD(T2, w))
        case VERT(T1, T2):
            c := (0, ∞)
            for w' = 0..w
                c' := vert(fixguil_TD(T1, w'), fixguil_TD(T2, w − w'))
                if (h(c') < h(c)) c := c'
    cache(T,w,c)
    return c
```

Figure 7.8: Pseudo-code for the basic top-down dynamic programming approach, returning the minimal height configuration $c = (w_r, h_r)$ for tree T such that $w_r \leq w$.

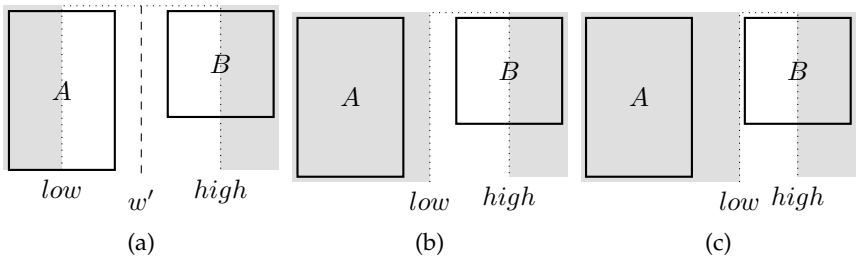


Figure 7.9: Illustration of using a binary chop to improve search for the optimal cut position. If $h_A^{w'} > h_B^{w-w'}$ as shown in (a), we cannot improve the solution by moving the cut to the left. Hence we can update (b) $low = w'$. Since B will retain the same configuration until the cut position exceeds $w - w_B^{w-w'}$, we can (c) set $low = w - w_B^{w-w'}$.

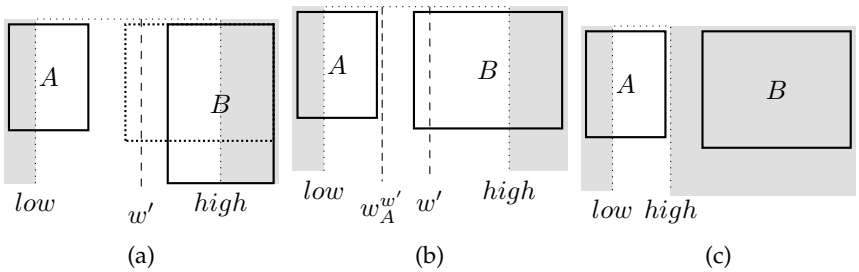


Figure 7.10: If the optimal layout for fixguil_TD(A, w') has width smaller than w', then we may lay out B in all the available space, using $w - w_A^{w'}$, rather than $w - w'$. If B is still taller than A, we know the cut must be moved to the left of $w_A^{w'}$ to find a better solution.

can improve this by using a binary cut to eliminate regions that cannot contain the optimal solution, keeping track of the range $low..high$ where the optimal cut is.

Consider the cut shown in Figure 7.9(a). Let $h_A^{w'} = \text{fixguil\_TD}(A, w')$ and $h_B^{w-w'} = \text{fixguil\_TD}(B, w - w')$. In this case, $h_A^{w'} > h_B^{w-w'}$. As the resulting configuration has height $\mathbf{max}(h_A^{w'}, h_B^{w-w'})$, the only way we can reduce the overall height is by adopting a shorter configuration for A – by moving w' further to the right. Normally we would set $low = w'$ as shown in Figure 7.9(b). In fact, we can move low to $\mathbf{max}(w', w - w_B^{w-w'})$ as shown in Figure 7.9(c), since moving w' right cannot increase the overall height until B shifts to a narrower configuration.

We can improve this further by observing that, if configurations are sparse, we may end up trying multiple cuts corresponding to the same configuration. If we construct a layout for A with cut position w', but A does not fill all the available space (so $w_A^{w'} < w'$), we can use that additional space to lay out B. If B is still taller than A (as shown in Figure 7.10), we know that the cut can be shifted to the left of $w_A^{w'}$, rather than just w'.

The case for $\text{VERT}(T_1, T_2)$ in Figure 7.8 can then be replaced with the following:

$c := (0, \infty)$
$low := w_{min}^{T_1}$
$high := w - w_{min}^{T_2}$
while $(low \leq high)$
  $w' := \left\lfloor \frac{low+high}{2} \right\rfloor$
  $c_1 :=$ fixguil_TD(T_1,w')
  $c_2 :=$ fixguil_TD(T_2,$w - \mathsf{w}(c_1)$)
  $c' :=$ vert(c_1, c_2)
  if $(\mathsf{h}(c') < \mathsf{h}(c))$ $c := c'$
  if $(\mathsf{h}(c_1) \leq \mathsf{h}(c_2))$ $high := \mathsf{w}(c_1) - 1$
  if $(\mathsf{h}(c_1) \geq \mathsf{h}(c_2))$ $low := \max(w' + 1, w - \mathsf{w}(c_2))$

Example 7.2. *Consider again the problem described in Example 7.1. The root node is a vertical cut, so we must pick a cut position. Since $w_{min}^{H_1} = w_{min}^{Z} = 1$, the cut must be in the range $[1, 2]$.*

We choose the initial cut as $w' = 1$. The sequence of calls made is as follows:

$f(V_0, 3)$
  $w' = 1$

$$
\begin{aligned}
&f(H_1, 1)\\
&\quad f(X, 1)\\
&\qquad \rightarrow (1, 2)\\
&\quad f(Y, 1)\\
&\qquad \rightarrow (1, 2)\\
&\quad \rightarrow (1, 4)\\
&f(Z, 2)\\
&\quad \rightarrow (2, 2)\\
&\rightarrow (3, 4)
\end{aligned}
$$

The best solution found so far is $(3, 4)$*. Since the height of* H_1 *is greater than the height of* Z*, we know an improved solution can only be to the right of the current cut. We update* $low := 2$*, and continue:*

$$
\begin{aligned}
&w' = 2\\
&\quad f(H_1, 2)\\
&\qquad f(X, 2)\\
&\qquad\quad \rightarrow (2, 1)\\
&\qquad f(Y, 2)\\
&\qquad\quad \rightarrow (2, 1)\\
&\qquad \rightarrow (2, 2)\\
&\quad f(Z, 1)\\
&\qquad \rightarrow (1, 3)\\
&\quad \rightarrow (3, 3)\\
&\rightarrow (3, 3)
\end{aligned}
$$

Finding the optimal solution at $w' = 2$*, giving configuration* $(3, 3)$*.* □

7.3 Free Guillotine Layout

In this section we consider the more difficult problem of free guillotine layout. Given a set of leaves (say, newspaper articles), we want to construct the optimal tree of cuts such that all leaves are used, and the overall height is minimized. Both the top-down and bottom-up construction methods given in the last section for fixed-cut guillotine layout can be readily adapted to solving the free layout problem.

The structure of the bottom-up algorithm remains largely the same. To compute the minimal configurations for a set S', we try all binary partitionings of S' into S'' and $S' \setminus S''$. We then generate the configurations for $\text{VERT}(S' \setminus S'', S'')$ and $\text{HORIZ}(S' \setminus S'', S'')$ as for the fixed problem. However, we must then eliminate any non-minimal configurations that have been generated. This is done by merge, which merges two sets of minimal configurations. Pseudo-code for this process is given in Figure 7.11. As we need to generate all configurations for all $2^{|S|}$ subsets of S, we construct the results for subsets in order of increasing size.

For the top-down method, at each node we want to find the optimal layout for a given set S and width w. To construct the solution, we try all binary partitions of S. Consider a partitioning into sets S' and S''. As there are a large number of symmetric partitionings, we enforce that the minimal element of S must be in S'. We then try laying out both $\text{VERT}(S', S'')$ and $\text{HORIZ}(S', S'')$, picking the best result.

Pseudo-code for the top-down dynamic programming approach is given in Figure 7.12. The structure of the algorithm is very similar to that for the fixed layout problem, except it now includes additional branching to choose binary partitions of S and try both cut directions. As before, w^S_{min} indicates the narrowest feasible width for laying out S. This is calculated by taking the the widest minimum configuration width for any node in S.

7.3.1 Bounding

The dynamic program as formulated has a very large search space. We would like to reduce this by avoiding exploring branches containing strictly inferior solutions. We can improve this if we can calculate a lower bounds $\mathsf{lb}(S, w)$ on the height of any configuration for S in width w. If h_{max} is the best height so far and $\mathsf{lb}(S, w) \geq h_{max}$, we know the current state cannot be part of any improved optimal solution, so we can simply cut-off search early with the current bound. This is a form of bounded dynamic programming [Puchinger and Stuckey, 2008].

For the minimum-height guillotine layout problem, we compute the minimum area used by some configuration of each leaf. This allows us to determine a lower bound on the area required for laying out the set of articles S. Since any valid layout must occupy at least $\mathsf{area}(S)$, a layout with a fixed width of w will have a height of at least $\left\lceil \frac{\mathsf{area}(S)}{w} \right\rceil$.

```
freeguil_BU(S,w)
  for(c ∈ {2,...,|S|})
    for(S' ⊆ S, |S'| = c)
      C(S) := ∅
      e := min i ∈ S'
      for(S'' ⊂ S' \ {e})
        C(S') := merge(C(S'),
                       join_horiz(C(S' \ S''), C(S''), w))
        C(S') := merge(C(S'),
                       join_vert(C(S' \ S''), C(S''), w))
  return C(S)[|C(S)| − 1]

merge(CX, CY)
  C := []
  i := 0
  j := 0
  while (i ≤ |CX| ∧ j ≤ |CY|)
    (w_x, h_x) := CX[i]
    (w_y, h_y) := CY[j]
    if(w_x < w_y)
      if(h_x > h_y)
        C := C ++[ CX[i] ]
        i := i + 1
      else j := j + 1
    else if(w_x > w_y)
      if(h_x < h_y)
        C := C ++[ CY[j] ]
        j := j + 1
      else i := i + 1
    else % w_x = w_y
      if(h_x ≤ h_y) j := j + 1
      else i := i + 1
  while (i ≤ |CX|)
    C := C ++[ CX[i] ]
    i := i + 1
  while (j ≤ |CY|)
    C := C ++[ CY[j] ]
    j := j + 1
  return C
```

Figure 7.11: Pseudo-code for a bottom-up construction approach for the free guillotine-layout problem for articles S. The configurations $C(S')$ for $S' \subseteq S$ are constructed from those of $C(S' \setminus S'')$ and $C(S'')$ where $S' \setminus S''$ and S'' are non empty and the first set is lexicographically smaller than the second.

```
freeguil_TD(S,w)
    c := lookup(S,w)
    if c ≠ NOTFOUND return c
    if (S = {A})
        c := C(A)[i] where i is maximal s.t. w(C(A)[i]) ≤ w
    else
        e := min(S)
        c := (0, ∞)
        for S' ⊂ S \ {e}
            L := {e} ∪ S'
            R := S \ L
            % Try a horizontal split
            c' := horiz(freeguil_TD(L, w), freeguil_TD(R, w))
            if(h(c') ≤ h(c)) c := c'
            % Find the optimal vertical split
            low := w^L_min
            high := w − w^R_min
            while(low ≤ high)
                w' := ⌊(low+high)/2⌋
                c_l := freeguil_TD(L, w')
                c_r = freeguil_TD(R, w − w(c_l))
                c' := vert(c_l, c_r)
                if(h(c') ≤ h(c)) c := c'
                if(h(c_l) ≤ h(c_r)) high := w(c_l) − 1
                if(h(c_l) ≥ h(c_r))
                    low := max(w' + 1, w − w(c_r))
    cache(S,w,c)
    return c
```

Figure 7.12: Basic top-down dynamic programming for the free guillotine layout problem.

We can also use the area approximation to reduce the set of vertical splits that must be explored. If we have a current best height h_{max}, any cut for $\text{VERT}(X, Y)$ where $w' \leq \left\lceil \frac{\mathsf{area}(X)}{h_{max}} \right\rceil$ or $w' \geq w - \left\lceil \frac{\mathsf{area}(Y)}{h_{max}} \right\rceil$ cannot give an improved solution. Pseudo-code for the bounded dynamic programming approach is given in Figure 7.13. Note that configurations are now given as a triple (w_i, h_i, e_i), where $e_i \in \{\mathsf{true}, \mathsf{false}\}$ indicates whether the configuration is exact ($e_i = \mathsf{true}$), or a lower bound ($e_i = \mathsf{false}$). We use $\mathsf{e}(c)$ to extract the third component of a configuration. The algorithm is structured similarly to the previous top-down algorithm given in Figure 7.12; but whenever we select a new partitioning or cut, we first use the area approximation to test if the selected subproblem could produce an improved solution. If the current subproblem requires more area than is available, we terminate immediately. For the horizontal cut at (1), we adjust the maximum height for L according to the minimum height for R (since, if $h_L + h_R \leq h_{max}$ then $h_L \leq h_{max} - h_R$). We apply similar reasoning at (2) to check if a vertical cut is feasible. At (3), we establish bounds on the vertical cut positions as described above. Note that, at (4), if the left subproblem exceeds the available height, we don't compute the optimum for the right subproblem, and take the area approximation instead. A final optimization is to note that if we find a configuration c which has height equal to the lower bound $\left(\mathsf{h}(c) = \left\lceil \frac{\mathsf{area}(S)}{w} \right\rceil\right)$ we can immediately return this solution.

7.4 Updating Layouts

In the interaction model proposed in the introduction, we suggested using a fixed-cut layout to re-layout an article during user interaction, until the current fixed-cut leads to a very bad layout. A layout can be considered bad for two reasons. The first reason is that current choice of guillotining does not allow a layout for the desired width while a different choice of guillotining will. The second reason is that the choice of guillotine leads to a non-compact layout and so to a page height that is unnecessarily large. In the case that the current fixed-cut leads to bad layout we wish to modify the guillotining to give a layout close to the current layout.

First, we must determine how bad a layout can be before we re-layout the document.

```
freeguil_bTD(S,w,h_max)
  c := lookup(S,w)
  if(c ≠ NOTFOUND)
    if(e(c) ∨ h(c) ≥ h_max) return c
  % If the bound is greater than
  % h_max, we can stop early
  if (⌈area(S)/w⌉ ≥ h_max) c := (w, ⌈area(S)/w⌉, false)
  if (S = {A})
    i := maximal i' s.t. w(C(A)[i']) ≤ w
    c := (w(c'), h(c'), h(c') ≤ h_max) where c' = C(A)[i]
  else
    e := min(S)
    c := (0, ∞)
    for S' ⊂ S \ {e}
      L := {e} ∪ S'
      R := S \ L
      A_l := area(L)
      A_r := area(R)
      % Try a horizontal split   (1)
      c_l := freeguil_bTD(L, w, h_max − ⌈A_r/w⌉)
      c_r := freeguil_bTD(R, w, h_max − h(c_l))
      if(h(c_l) + h(c_r) ≤ h(c))
        c := (max(w(c_l), w(c_r)), h(c_l) + h(c_r), e(c_l) ∧ e(c_r))
        if(h(c) = ⌈area(S)/w⌉) break
      % Ensure a vertical split is feasible   (2)
      if(w^L_min + w^R_min > w) continue
      h^vert_min := max(⌈A_l/(w − w^R_min)⌉, ⌈A_r/(w − w^L_min)⌉)
      if(h^vert_min > h_max)
        if (h^vert_min < hc) c := (w, h^vert_min, false)
        continue
      % Find the optimal vertical split   (3)
      low := max(w^L_min, ⌈A_l/h_max⌉)
      high := w − max(w^R_min, ⌈A_r/h_max⌉) + 1
      while(low < high)
        w' := ⌊(low+high)/2⌋
        c_l := freeguil_bTD(L, w', h_max)
        if(h(c_l) ≥ h_max)   (4)
          c_r := (w − w(c_l), ⌈A_r/(w − w(c_l))⌉, false)
        else
          c_r := freeguil_bTD(R, w − w(c_l), h_max)
        if(max(h(c_l), h(c_r)) ≤ h(c))
          c := (w(c_l) + w(c_r), max(h(c_l), h(c_r)), e(c_l) ∧ e(c_r))
          if(h(c) ≤ h_max) h_max := h(c)
        if(h(c_l) ≤ h(c_r)) high := h(c_l) − 1
        if(h(c_l) ≥ h(c_r)) low := max(w' + 1, w − w(c_r))
        if(h(c) = ⌈area(S)/w⌉) break for
  cache(S,w,c)
  return c
```

Figure 7.13: Pseudo-code for the bounded top-down dynamic programming approach. Note that while bounding generally reduces search, if a previously expanded state is called again with a more relaxed bound, we may end up partially expanding a state multiple times.

Given a set of articles S, we can precompute the optimal layout for a set of given widths W using `freeguil_BU` or `freeguil_TD`. We can then build a piecewise linear approximation $\mathsf{approx\_height}(S, w)$ to the minimal height for free layout of S for width w for all possible widths. However, as illustrated in Figure 7.15, the optimal layout is generally very close to the area bound for the documents we have been considering. As such, we can use the simpler approximation $\mathsf{approx\_height}(S, w) = \left\lceil \frac{\mathsf{area(S)}}{w} \right\rceil$. We use this function to determine when to change guillotine cuts during user interaction. Assume the current layout of S is T, then if $\mathsf{layout}(T, w) > \alpha \times \mathsf{approx\_height}(S, w)$ we know that the fixed-cut is giving poor layout. We use $\alpha = 1.1$.

When we are generating a new guillotine cut for S, we want to ensure that the new layout is "close" to the current cut T. Our approach is to try and change the guillotining only at the bottom of the current cut T. Define the tree height of a tree T as follows:

$$\mathsf{theight}(\textsc{cell}(A)) = 0$$
$$\mathsf{theight}(\textsc{vert}(T_1, T_2)) = \max(\mathsf{theight}(T_1), \mathsf{theight}(T_1)) + 1$$
$$\mathsf{theight}(\textsc{horiz}(T_1, T_2)) = \max(\mathsf{theight}(T_1), \mathsf{theight}(T_1)) + 1$$

We first try to modify only subtrees with tree height 1 (that is parents of leaf nodes). If that fails to improve the current layout enough we modify subtrees of tree height 2, etc. Psuedo-code for the interactive layout problem is given in Figure 7.14.

7.5 Experimental Results

To evaluate the methods described in Sections 7.2 to 7.3, we required a set of documents suitable for guillotine layout. To construct this data-set, we randomly select a set of n of articles from the REUTERS-21578 news corpus [reu], then use a modified version of the binary search described in Hurst et al. [2006b] to determine the set of available configurations for each article. All times are given in seconds, and all experiments are run with a time limit of 600 seconds. Times given are averages over 10 instances of each problem size.

```
interact(T,w)
    c := layout(T,w)
    S := articles in T
    k := 1
    while(h(c) > α × approx_height(S, w))
        c := relayout(T,w,k)
        k := k + 1
    return c

relayout(T,w,k)
    c := lookup(T,w)
    if(c ≠ NOTFOUND) return c
    if(theight(T) ≤ k)
        S := set of articles appearing in T
        return freeguil_TD(S,w)
    switch (T)
        case CELL(A):
            c := C(A)[i] where i is maximal s.t. w(C(A)[i]) ≤ w
        case HORIZ(T1, T2):
            c := horiz(relayout(T1, w, k), relayout(T2, w, k))
        case VERT(T1, T2):
            c := (0, ∞)
            for(w' = 0..w)
                c' := vert(relayout(T1, w', k), relayout(T2, w − w', k))
                if (h(c') < h(c)) c := c'
    cache(T,w,c)
    return c
```

Figure 7.14: Pseudo-code for the basic top-down dynamic programming re-layout, where we can change configuration for subtrees with tree height less than or equal to k.

n	td	td+b	bu
10	**0.00**	**0.00**	**0.00**
20	**0.00**	**0.00**	**0.00**
30	0.02	0.01	**0.00**
40	0.03	0.02	**0.00**
50	0.06	0.05	**0.00**
60	0.13	0.10	**0.00**
70	0.18	0.14	**0.00**
80	0.29	0.22	**0.00**
90	0.41	0.33	**0.00**

(a)

n	td	td+b	bu
10	**0.00**	**0.00**	**0.00**
20	**0.00**	**0.00**	**0.00**
30	0.01	**0.00**	**0.00**
40	0.01	**0.00**	**0.00**
50	0.02	0.01	**0.00**
60	0.03	0.01	**0.00**
70	0.03	0.01	**0.00**
80	0.04	0.01	**0.00**
90	0.05	0.02	**0.00**

(b)

Table 7.1: Results for the fixed-cut minimum-height guillotine layout problem with (a) $w = w_{min} + 0.1(w_{max} - w_{min})$, and (b) $w = w_{min} + 300$.

For convenience in generating the dataset, we assume the use of a fixed-width font. While A-series page sizes have a $1 : \sqrt{2}$ aspect ratio, fixed-width fonts fit approximately equal number of lines as characters per line.

All experiments were conducted on a 3.00Ghz Core2 Duo with 2 GB of RAM running Ubuntu GNU/Linux 8.10. td denotes the top-down dynamic programming approach, and td+b is top-down dynamic programming with bounding. bu denotes bottom-up construction.

7.5.1 Fixed-Cut Layout

Instances for fixed-cut guillotine layout were constructed with a random tree of cuts, selecting horizontal and vertical cuts with equal probability. Initially, we selected the instance width as a linear combination of the minimum and maximum width configurations for the instance. Results given in Table 7.1(a) are constructed with $w = w_{min} + 0.1(w_{max} - w_{min})$, where w_{min} is the overall width when every article takes the narrowest feasible configuration (and similarly for w_{max}). Clearly, the top-down methods degrade quite rapidly compared to the bottom-up method. This appears to be due more to the rapidly increasing width than the increasing number of articles; the instances with 90 articles are laid out on a page that is 3000 to 5000 characters wide. This is illustrated in Table 7.1(b), where we calculated $w = w_{min} + 300$. Although the top-down methods are still distinctly slower than the bottom-up approach, they now scale far more gracefully.

n	td	td+b	bu
4	**0.00**	**0.00**	**0.00**
5	**0.00**	**0.00**	**0.00**
6	0.01	**0.00**	**0.00**
7	0.03	**0.00**	0.01
8	0.12	**0.01**	0.04
9	0.42	**0.03**	0.15
10	1.62	**0.11**	0.50
11	5.81	**0.41**	1.64
12	22.33	**1.50**	5.02
13	106.46	**6.09**	17.51
14	413.63	**24.43**	53.11
15	—	**74.84**	143.83

Table 7.2: Results for the free minimum-height guillotine layout problem. Times (in seconds) are averages of 10 randomly generated instances with n articles.

7.5.2 Free Layout

For the free layout problem, we constructed instances for each size between 4 and 15. The instance width was selected as $\left\lceil \sqrt{(1+\alpha)\mathsf{area}(S)} \right\rceil$, to approximate a layout on an A-series style page with α additional space. For these experiments, we selected $\alpha = 0.2$. Times given in Table 7.2 denote the average time for solving the 10 instances of the indicated size.

As before, td performs significantly worse than the other methods. Unlike the fixed layout problem, these instances have much narrower page widths, and the search space arises largely from the selection of binary partitions. As a result, bounding provides a substantial improvement – td+b is consistently around twice as fast as bu on these instances.

In this first experiment we did not use column-based layout. However, in practice column-based layout is preferable so as to avoid long text measures. We generated test data for page dependent column-based layouts in a similar manner to the other guillotine layouts; having selected a column width, we calculate the number of lines required for the article body, and use this to determine the dimensions given a varying number of columns. This is combined with the layout for the article title (calculated as before).

We select a column width of 38 characters, chosen as being typical of print newspapers. Page width is selected as before, then rounded up to the nearest number of columns. Results for this dataset are given in Table 7.3. The results for this case dif-

n	td	td+b	bu
4	**0.00**	**0.00**	**0.00**
5	**0.00**	**0.00**	**0.00**
6	**0.00**	**0.00**	**0.00**
7	**0.00**	**0.00**	**0.00**
8	**0.00**	**0.00**	**0.00**
9	0.01	**0.00**	0.01
10	0.04	**0.00**	0.05
11	0.12	**0.00**	0.16
12	0.39	**0.01**	0.47
13	1.23	**0.04**	1.53
14	3.85	**0.10**	4.38
15	13.10	**0.37**	15.50
16	41.65	**0.50**	48.32
17	168.19	**1.17**	157.96
18	590.76	**3.80**	442.19

Table 7.3: Results for the free minimum-height guillotine layout problem using page dependent column-based layout. Times (in seconds) are averages of 10 randomly generated instances with n articles.

fer substantially from those for the non column-based instances – since the number of possible vertical cuts is much smaller (even the large instances generally have only 4 columns) fewer subproblems need to be expanded at each node during the execution of the dynamic programming approaches. In this case, td slightly outperforms bu on small instances, but degrades more rapidly; td+b is considerably faster than either method.

7.5.3 Updating Layouts

In practice, for non page dependent column-based layouts, a fixed optimal cutting remains near-optimal over a wide range of width values. To illustrate this, we took a document with 13 articles from the set used in Section 7.5, and computed the optimal cutting for $w = 200$. Figure 7.15 shows the height given by laying out this fixed cutting using layout with widths between 40 and 200. We compare this with the height given by the area bound and the optimal layout for each width. While the fixed layout is quite close to the optimal height over a wide range of values, it begins to deviate as we decrease the viewport width. For widths 40 and 50, this fixed layout is infeasible, and we are forced to compute a new tree of cuts.

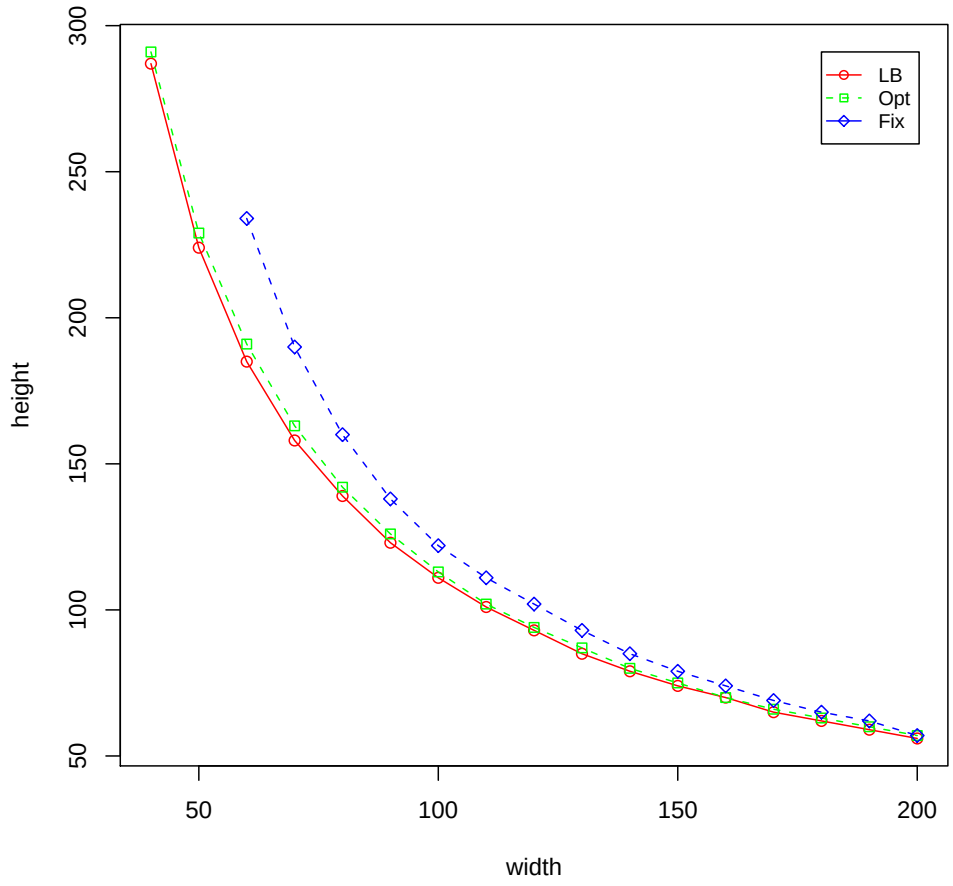


Figure 7.15: Layout heights for a 13-article document used in Section 7.5. LB is the lower bound at the given width, and OPT is the minimum height given by freeguil_bTD. For FIX, we computed the optimal layout for $w = 200$, and adjusted the layout to the desired with using layout.

To test the performance of the re-layout algorithm, we consider again the set of 13-article documents used in the previous experiment. We computed the optimal layout for page widths between 40 and 200 characters, in 5 character intervals. We compared this with adapting the fixed layout computed for $w = 40$, and progressively used relayout at each width. relayout was implemented with the bounded top-down methods for both the fixed and free components.

The average runtime for freeguil_bTD over the varying documents and widths was 7.48s. Runtime for layout was less than 0.01s in all cases, but deviated from the minimal height by up to 40%. Average runtime for relayout (with $\alpha = 1.1$) was 0.02s, and deviated from the minimal height by at most 10%. Results for page dependent column-based layout are similar. For documents with 16 articles, layout generated layouts up to 32% taller than the optimum; freeguil_bTD took 0.48s on average, compared to less than 0.01 for relayout (and $\alpha = 1.1$).

7.6 Conclusion

Guillotine-based layouts are widely used in newspaper and magazine layout. We have given algorithms to solve two variants of the automatic guillotine layout problem: the fixed cut guillotine layout problem in which the choice of guillotine cuts is fixed and the free guillotine layout problem in which the algorithm must choose the guillotining. We have shown that the fixed guillotine layout problem is solvable in polynomial time while the free guillotine layout problem is NP-Hard.

We have presented bottom-up and top-down methods for the minimum-height guillotine layout problem. For fixed-cut guillotine layout, the bottom-up method is far superior, as complexity is dependent only on the number of leaf configurations, rather than the page width; the bottom-up method can optimally layout reasonably sized graphs in real-time.

For the free guillotine layout problem, which has smaller width and larger search space, the bounded top-down method was substantially faster than the other methods. On instances with arbitrary cut positions, the bounded top-down method could solve instances with up to 13 articles in a few seconds; when restricted to page dependent column-based layouts, we can quickly produce layouts for at least 18 articles.

We did not, however, consider CP or MIP-based approaches for this problem. CP and MIP are both best suited for problems which are in some sense *flat*, in that the structure of the solution is already known; in the case of table layout, for example, the set of rows and columns is fixed, and the problem is merely to assign values to each. It is difficult to use these methods to model problems (such as guillotine layout) where the solution requires recursively constructing a tree structure. Also, in cases where dynamic programming methods are applicable, they can be quite difficult to beat using other techniques.

We have also suggested a novel interaction model for viewing on-line documents with a guillotine-based layout in which we solve the free guillotine layout problem to find an initial layout and then use the fixed cut guillotine layout to adjust the layout in response to user interaction such as changing the font size or viewing window size.

Currently our implementation only handles text. Future work will be to incorporate images.

8

Smooth Linear Approximation of Geometric Constraints

CONSTRAINT-BASED graphics originated with Sketchpad [Sutherland, 1964], one of the earliest interactive graphics applications. This was a forerunner of modern CAD software and allowed the user to set up persistent geometric constraints on objects such as fixing the length of a line or the angle between two lines which were maintained by an underlying constraint solver during subsequent user manipulation. In the almost fifty years since then, constraint-based graphics has proven useful in a number of application areas.

- Geometric constraint solving is provided in most modern CAD applications, such as Pro/ENGINEER. Such applications allow parametric modelling in which the designer can specify the design in terms of geometric constraints such as having a common endpoint or lines being parallel rather than in terms of individual object placement and dimensions. Importantly, this allows parametric re-use of components in a design.

- Constraint-based graphics is also provided in several generic diagram authoring tools, for example MicroSoft Visio or Dunnart [Dwyer et al., 2008]. Such tools often provide semi-automatic layout such as connector routing, persistent object alignment or distribution relationships, and some provide automatic layout of networks and trees.

- A final application area has been adaptive layout, in particular for GUIs. Example tools include Amulet, Madeus [Jourdan et al., 1998] and the widget

layout manager in OS X 10.7 (Lion). Here geometric constraints are used to specify the relative position and relationship of objects in the layout, allowing the precise placement to adapt to different size viewports or fonts etc.

In all of these applications, constraint solving allows the application to preserve design aesthetics, such as alignment and distribution, and structural constraints, such as containment, during manipulation of the graphic objects or during adaptation of a layout to a new context.

Unfortunately, geometric constraint solving is, in general, computationally expensive. This difficulty is compounded by the desire for real-time updating of the layout during user interaction. Thus, a wide variety of specialized constraint solving algorithms have been developed for different applications. While the algorithms are usually quite efficient this is because they are typically quite restricted in the kinds of geometric constraints that can be handled. In particular, few algorithms handle the important geometric constraint of non-overlap between objects. This is, perhaps, unsurprising, since solving non-overlap constraints is NP-hard. There is still a need for more generic geometric constraint solving algorithms that are efficient enough for interactive graphical applications.

We present a new approach to geometric constraint solving in interactive graphical applications. The approach is generic, supporting a wide variety of different geometric constraints including alignment, distribution, containment and non-overlap.

Our starting point is the set of efficient linear constraint solving techniques developed for graphical applications [Borning et al., 1997b, Marriott and Chok, 2002, Badros et al., 2001]. These minimize a linear (or sometimes a convex quadratic) objective function subject to a conjunction of linear equality and inequality constraints. They efficiently handle those geometric constraints, such as alignment, distribution and containment within a convex shape, which can be modelled as a conjunction of linear constraints. These are increasingly used in applications including widget layout in OS X 10.7 (Lion), the diagramming tool Dunnart, multimedia authoring tool Madeus and the Scwm window manager. Unfortunately, geometric constraints such as non-overlap or containment in a non-convex shape are inherently non-linear and so are currently not supported by these constraint solving techniques.

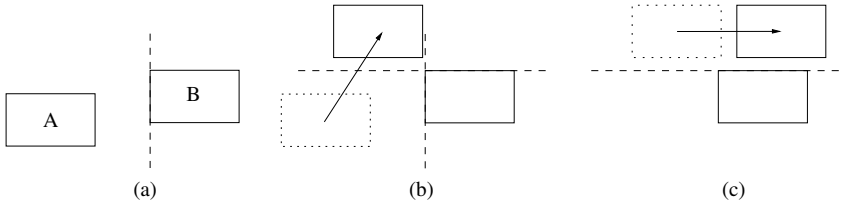


Figure 8.1: Smooth linear approximation of non-overlap between two boxes. Satisfaction of any of the constraints: *left-of*, *above*, *below* and *right-of* is sufficient to ensure non-overlap. Initially (a) the *left-of* constraint is satisfied. As the left rectangle is moved (b), it passes through a state where both the *left-of* and *above* constraint are satisfied. When the *left-of* constraint stops the movement right, the approximation is updated to *above* and (c) motion can continue.

The key to our approach is to use a linear approximation of these more difficult geometric constraints which ensures that the original geometric constraint will hold. As the solution changes the linear approximation is smoothly modified. Thus we call the technique *smooth linear approximation* (SLA). The approach is exemplified in Figure 8.1. It is worth pointing out that SLA is not designed to find a new solution from scratch, rather it takes an existing solution and continuously updates this to find a new locally optimal solution. This is why the approach is tractable and also well suited to interaction since, if the user does not like the local optimum the system has found, then they can use direct manipulation to escape the local optimum.

This chapter has four main contributions.

- The first contribution is a generic algorithm for SLA. We also give a variant of the algorithm which is *lazy* in the sense that it does not use a linear approximation for the complex geometric constraints until they are about to become violated. (Section 8.3)

- We show how SLA can be used to straightforwardly model a variety of non-linear geometric constraints: non-overlap of two boxes, minimum Euclidean distance, placement of a point on a piecewise-linear curve and containment in a non-convex polygon. We also demonstrate that SLA can model text-boxes that can vary their height and width but are always large enough to contain their textual content. (Section 8.4)

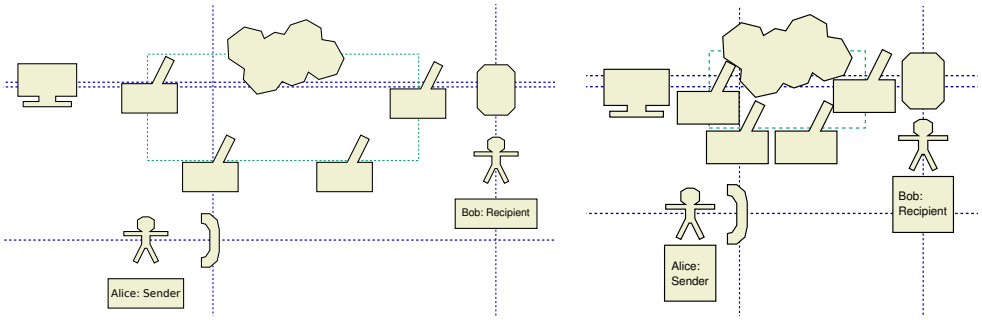
- We then explore in more detail how SLA can be used to model non-overlap of polygons. We first consider non-overlap of two convex polygons. We then investigate how to efficiently handle non-overlap of non-convex polygons which is significantly more difficult. One approach is to *decompose* each non-convex polygon into a collection of convex polygons joined by equality constraints. Unfortunately, this leads to a large number of non-overlap constraints. In our second approach, we *invert* the problem and, in essence, model non-overlap of polygons A and B by the constraint that A is contained in the region that is the complement of B. In practice this leads to substantially fewer constraints.

  However, naive use of SLA to model non-overlap of many polygons leads to a quadratic increase in the number of linear constraints since a constraint is generated between each pair of objects. This is impractical for larger diagrams. By using the Lazy SLA Algorithm together with efficient incremental object collision detection techniques developed for computer graphics [Lin et al., 1996][1] we give an approach which can scale up to larger diagrams. (Section 8.5)

- Finally, we provide a detailed empirical evaluation of these different algorithms in Section 8.6. We focus on non-overlap of multiple non-convex polygons because in practice this is most difficult class of problem to handle efficiently. (Section 8.6)

We believe the algorithms described here provide the first viable approach to handling a wide variety of non-linear geometric constraints including non-overlap of (possibly non-convex) polygons in combination with linear constraints in interactive constraint-based graphical applications. We have integrated the algorithms into the constraint based diagramming tool Dunnart. An example of using the tool to construct and modify a network diagram is shown in Figure 8.2.

[1] It is perhaps worth emphasizing that collision-detection algorithms by themselves are not enough to solve our problem. We are not just interested in detecting overlap: rather, we must ensure that objects do not overlap while still satisfying other design and structural constraints and placing objects as close as possible to the user's desired location.



(a) (b)

Figure 8.2: An example of using SLA to modify a complex constrained diagram of a communications network. Elements of the diagram are convex and non-convex objects constrained to not overlap. The mid point of the cloud and top centre of the switch objects are all constrained to lie on boundary of the rectangle (to enforce the "ring" layout). The computer to the left is horizontally aligned with the top of its switch. The tablet to the right is horizontally aligned with its switch and vertically aligned with its user and the text box. The telephone is vertically aligned with its switch and horizontally aligned with its user. The telephone user is vertically aligned with its text box. The figure illustrates two layouts (a) the original layout, and (b) a modified layout where the diagram has been shrunk horizontally and vertically. Notice how the constraints are maintained, the non-overlap constraints have become active, and text boxes have resized to have narrower width.

8.1 Related Work

Starting with Sutherland [1964], there has been considerable work on developing constraint solving algorithms for supporting direct manipulation in interactive graphical applications. These approaches fall into four main classes: propagation based (e.g. [Vander Zanden, 1996, Vander Zanden et al., 2001]); linear arithmetic solver based (e.g. [Borning et al., 1997b, Marriott and Chok, 2002, Badros et al., 2001]); geometric solver-based (e.g. [Kramer, 1992, Bouma et al., 1995, Fudos and Hoffmann, 1997, Joan-Arinyo and Soto-Riera, 1999]); and general non-linear optimization methods such as Newton-Raphson iteration (e.g. [Nelson, 1985]). However, none of these techniques support non-overlap and the other complex geometric constraints we consider here.

Hosobe [2001] describes a general purpose constraint solving architecture that handles non-overlap constraints and other non-linear constraints. The system uses variable elimination to handle linear equalities and a combination of non-linear optimization and genetic algorithms to handle the other constraints. Our approach addresses the same issue but is technically quite different, and we believe much

faster – our method doesn't require expensive non-linear optimization methods, and takes advantage of information from the solver to select configurations for disjunctive constraints, rather than requiring a separate local search phase.

Enforcement of object non-overlap has been a concern for physically-based modelling in computer graphics. The standard problem is modelling of non-penetrating freely rotating rigid bodies. Here equality constraints model joints and other connections between objects, objects have external forces working upon them (such as gravity, friction or forces imposed by the user) and objects cannot penetrate each other. The basic approach is to compute the objects' positions at successive discrete times. At each time step the simulation computes the forces on an object, computes the object velocity and then appropriately moves the objects by the small time increment to get their new position. However it may be that in the new position some of the objects collide. Fast object collision detection is used to detect this and the time step is decreased until the time at which the objects first touch is found. Repulsive forces between the touching objects are added and their force is carefully computed to ensure that the object's will not penetrate each other. Baraff [1994, 1996] has developed fast methods to compute the repulsive forces.

SLA, in particular the lazy algorithm, is quite similar to this basic approach—in some senses we are generalizing the lazy enforcement of non-overlap to other kinds of geometric constraints. What is different is that typically in physics simulation the constraints and variables model object velocities and forces, while in our approach they model object positions and dimensions. Thus, in physics simulation the user controls an object's position by applying a force to it while in our context they directly control the position and re-layout is driven by the user changing object positions. Modelling the problem at the level of object position and dimensions is, we believe, more natural for interactive diagramming and GUI applications.

We note Harada, Witkin, and Baraff [Harada et al., 1995] have investigated how to allow the continuous model of the standard physically-based modelling to allow discrete changes, such as allowing an object to pass through another object, in response to user interaction. This is something that we might consider in our work.

SLA is also similar to standard approaches in non-linear optimization in which non-linear constraints are approximated by linear constraints [Nocedal and Wright,

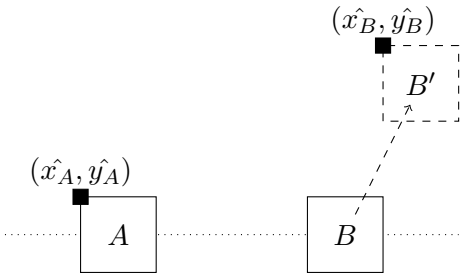


Figure 8.3: A diagram with two objects, A and B, which are constrained to be horizontally aligned. The object B is being moved to B'. The desired positions of $\{x_A, y_A, x_B, y_B\}$ are shown.

1999]. The main innovation in what we are doing is the use of a smooth transition between approximations.

8.2 Interactive Constraint-based Layout

As a variety of spatial constraints, such as alignment and distribution, can be conveniently represented as linear constraints, a number of efficient techniques have been developed for handling incremental updating of linear programs for use in graphical applications [Borning et al., 1997b, Badros et al., 2001].

Consider a diagram with a set of objects with positions $P = \{(x_1, y_1) \ldots, (x_n, y_n)\}$, and a set of linear constraints C over the object positions. If an object p is being directly manipulated, its variables (x_p, y_p) are said to be *edit variables*. We shall use E to denote the set of edit variables. Let $(\hat{x_p}, \hat{y_p})$ denote the new user-specified position for p. If p is not being directly manipulated, we define $(\hat{x_p}, \hat{y_p})$ to be the current position of p. The value $\hat{v}$ is said to be the *desired value* of v. The goal, then, is to find a solution that moves all the variables as close as possible to their respective desired values, while satisfying all the constraints C.

Example 8.1. *Consider the diagram shown in Figure 8.3. The rectangles A and B are horizontally aligned. When the user attempts to move B to the position marked B', we want to move (x_B, y_B) to the specified position, while keeping (x_A, y_A) as close to the current position as possible. The edit variables are $\{x_B, y_B\}$, which the solver wants to move to $(x_{B'}, y_{B'})$. The stay variables $\{x_A, y_A\}$ are to be kept close to their current location.* □

To find the desired solution, we need to introduce error terms δ_v^+ and δ_v^- to represent how far above and below $\hat{v}$ the current assignment is. We then minimize the

error terms to find the optimal solution. We must keep δ_v^+ and δ_v^- separate because we want to minimize $|\delta_v|$ rather than the value δ_v. However, not all differences are equally important. If variable v is being directly manipulated, it is more important to move v towards $\hat{v}$ than to keep $v' \notin E$ close to $\hat{v'}$. As such, we have an ordering over our objectives:

1. Satisfy all constraints $c \in C$.
2. Move all edit variables $v \in E$ towards $\hat{v}$.
3. Keep all non-edit variables $v' \notin E$ close to $\hat{v'}$.

While we could perform a multi-stage optimization, the easiest solution is to add weights to the error terms. Let w_e be the weight assigned to error terms for edit variables, and w_s be the weight assigned to other error variables. Generally we want $w_e \gg w_s$. With this, we can formulate the problem as a linear program:

$$\begin{aligned}
\textbf{min} \quad & \sum_{v_i \in E} w_e(\delta_{v_i}^+ + \delta_{v_i}^-) + \sum_{v_i \notin E} w_s(\delta_{v_i}^+ + \delta_{v_i}^-) \\
\textbf{s.t.} \quad & C \\
& v_1 = \hat{v_1} + \delta_{v_1}^+ - \delta_{v_1}^- \\
& v_2 = \hat{v_2} + \delta_{v_2}^+ - \delta_{v_2}^- \\
& \cdots \\
& v_n = \hat{v_n} + \delta_{v_n}^+ - \delta_{v_n}^-
\end{aligned}$$

When the desired value for an edit variable v is changed, we simply need to update the constant $\hat{v}$ then find the updated optimum. If the tableau remains feasible with the updated constants, we can use θ as the initial basic feasible solution. If the updated tableau is no longer feasible, we now have an optimal solution that must be made feasible. This is the *dual* of the normal optimization problem (moving from a feasible solution to an optimal one), and can be solved by starting from θ (which is feasible in the dual problem) and then optimizing the dual problem [Borning et al., 1997a].

One complication is the addition of constraints. When a new constraint c' is added, the current solution θ is likely to no longer be a feasible solution to the hard constraints C'. If this is the case, we must re-run Phase I of the simplex algorithm

to find a new feasible solution. However, θ is a basic solution which satisfies all constraints except c'. Once we introduce the artificial variable $a_{c'}$ for c', we can use we can use θ as our initial basic feasible solution to the revised solution, rather than discarding the current solution and re-running Phase I from scratch. Once we have restored feasibility, we then re-optimize as usual (although with $E = \emptyset$, as we want to keep all the objects at their current positions).

Example 8.2. *Consider the linear program from Example 2.1. We want to move the point* (x, y) *from* $(1, 0)$ *to* $(4, 1)$ *without violating any of the existing constraints. We introduce error terms for* x *and* y*, constructing the linear program:*

$$
\begin{aligned}
\textbf{\textit{min}}\ & w_e\delta_x^+ + w_e\delta_x^- + w_e\delta_y^+ + w_e\delta_y^- \\
\textbf{\textit{s.t.}}\ & \frac{1}{2}x + y \leq 3 \\
& x + \frac{2}{3}y \leq 4 \\
& y \leq 2 \\
& x \geq 1 \\
& x = 4 + \delta_x^+ - \delta_x^- \\
& y = 1 + \delta_y^+ - \delta_y^- \\
& x, y, \delta_x^+, \delta_x^-, \delta_y^+, \delta_y^- \geq 0
\end{aligned}
$$

Converted to standard form, this becomes:

$$
\begin{aligned}
\textbf{\textit{min}}\ & w_e\delta_x^+ + w_e\delta_x^- + w_e\delta_y^+ + w_e\delta_y^- \\
\textbf{\textit{s.t.}}\ & \frac{1}{2}x + y + s_1 = 3 \\
& x + \frac{2}{3}y + s_2 = 4 \\
& y + s_3 = 2 \\
& x - s_4 = 1 \\
& x = 4 + \delta_x^+ - \delta_x^- \\
& y = 1 + \delta_y^+ - \delta_y^- \\
& x, y, s_1, s_2, s_3, s_4, \delta_x^+, \delta_x^-, \delta_y^+, \delta_y^- \geq 0
\end{aligned}
$$

The initial solution for $(x, y) = (1, 0)$ *gives us the basis* $\{x, s_1, s_2, s_3, \delta_x^-, \delta_y^-\}$. *Assuming* $w_e = 1$, *applying substitutions for this basis gives us the tableau:*

$$\begin{aligned}
s_1 &= \frac{5}{2} - \frac{1}{2}s_4 - y \\
s_2 &= 3 - s_4 - \frac{2}{3}y \\
s_3 &= 2 - y \\
x &= 1 + s_4 \\
\delta_x^- &= 3 - s_4 + \delta_x^+ \\
\delta_y^- &= 1 - y + \delta_y^+ \\
\hline
f &= 4 - y - s_4 + 2\delta_x^+ + 2\delta_y^+
\end{aligned}$$

If we pivot on y, *the tableau becomes:*

$$\begin{aligned}
s_1 &= \frac{3}{2} - \frac{1}{2}s_4 + \delta_y^- - \delta_y^+ \\
s_2 &= \frac{7}{3} - s_4 + \frac{2}{3}\delta_y^- - \frac{2}{3}\delta_y^+ \\
s_3 &= 1 + \delta_y^- - \delta_y^+ \\
x &= 1 + s_4 \\
\delta_x^- &= 3 - s_4 + \delta_x^+ \\
y &= 1 - \delta_y^- + \delta_y^+ \\
\hline
f &= 3 - s_4 + 2\delta_x^+ + \delta_y^- + \delta_y^+
\end{aligned}$$

We then pivot on s_4:

$$\begin{aligned}
s_1 &= \frac{1}{3} + \frac{1}{2}s_2 + \frac{1}{3}\delta_y^- - \frac{4}{3}\delta_y^+ \\
s_4 &= \frac{7}{3} - s_2 + \frac{2}{3}\delta_y^- - \frac{2}{3}\delta_y^+ \\
s_3 &= 1 + \delta_y^- - \delta_y^+ \\
x &= \frac{10}{3} - s_2 + \frac{2}{3}\delta_y^- - \frac{2}{3}\delta_y^+ \\
\delta_x^- &= \frac{2}{3} + s_2 + \delta_x^+ - \frac{2}{3}\delta_y^- + \frac{2}{3}\delta_y^+ \\
y &= 1 - \delta_y^- + \delta_y^+ \\
\hline
f &= \frac{2}{3} + s_2 + 2\delta_x^+ + \frac{1}{3}\delta_y^- + \frac{1}{3}\delta_y^+
\end{aligned}$$

As there are no negative coefficients in the objective row, we terminate. This tableau corresponds to the solution $(x, y) = (\frac{10}{3}, 1)$*, which is the nearest feasible solution to* $(4, 1)$*.*

□

8.3 The SLA Algorithm

In this section we present the basic SLA algorithm and a variant which is lazy in the enforcement of constraints. We first review linear constraint solving.

8.3.1 Linear constraint solving

Typical geometric constraints provided in many constraint-based graphics applications are:

- horizontal and vertical alignment
- horizontal and vertical distribution
- horizontal and vertical ordering that keeps objects a minimum distance apart horizontally or vertically while preserving their relative ordering
- a fixed value for the position or size of an object.

Each of the above geometric relationships can be modelled as a linear constraint over variables representing the position of the objects in the diagram. For this reason, a common approach in constraint-based graphics applications is to use a constraint solver that can support linear constraints. Details of these methods are given in Chapter 2.

8.3.2 The Basic SLA Algorithm

However, not all geometric constraints are linear. The approach presented here, *smooth linear approximation* (SLA), locally approximates each non-linear constraint by a conjunction of linear constraints. As the solution changes the linear approximation is smoothly modified.

A *linear approximation* of a complex constraint c is a (possibly infinite) disjunctive set of linear configurations $\{F_0, F_1, \dots\}$ where each *configuration* F_i is a conjunction of linear constraints. We require that the linear approximation is *sound* in the sense that each linear configuration implies the complex constraint and *complete* in the sense that each solution of c is a solution of one of the linear configurations.

```
sla(C, o)
    finished := false
    while(¬finished)
        θ := minimize o subject to ⋀_{c∈C} c.config
        finished := update_configs(C, θ)
    return θ

update_configs(C, θ)
    ret := true
    for(c ∈ C)
        F := c.config
        c.update(θ)
        if(F ≠ c.config) ret := false
    return ret
```

Figure 8.4: Basic SLA Algorithm for solving sets of non-linear constraints using smooth linear approximations.

Example 8.3. *Consider the non-overlap constraint of the two boxes in Figure 8.1. To ensure that boxes A and B do not overlap, we must ensure that the boxes are separated along some axis. Equivalently, we must ensure that box A is to the left of, to the right of, above or below box B.*

Assume we want to ensure that A is to the left of B. Given that the variable (x_A, y_A) denotes the centre of A, we can enforce this with the linear constraint:

$$x_A + \frac{w_A}{2} \leq x_B - \frac{w_B}{2}$$

□

SLA works by moving from one configuration for a constraint to another, requiring that both configurations are satisfied at the point of change. This *smoothness* criteria reduces the difficulty of the problem substantially since we need to consider only configurations that are satisfied with the present state of the diagram. It also fits well with continuous updating of the diagram during direct manipulation.

The Basic SLA Algorithm is very simple and is given in Figure 8.4. In the algorithm we represent a complex constraint by an object c that has a current configuration $c.config$ and a method $c.update(\theta)$ that, given a solution θ to the current configuration, updates the configuration if necessary. To ensure smoothness, the solution θ is required to be a solution of the new configuration.

Given a set of complex constraints $\mathbf{C}$ and an objective function o to be minimized, the algorithm uses a linear constraint solver to find a minimal solution θ us-

ing the current configuration for each complex constraint. It then calls update_configs to update the current configuration for each complex constraint. If the configuration for all of the constraints remains unchanged, the algorithm terminates.

One choice in the algorithm is whether to resolve whenever a configuration is updated or to update all constraint configurations before resolving. Our current implementation and the algorithm given in Figure 8.4 uses the second approach. In practice we found little difference between the two approaches.

The algorithm is generic in:

- The choice and technique for generating the linear configurations for the complex constraint.

- How to determine if an alternative linear configuration might improve the solution.

In the next two sections we describe various choices for these operations for modelling various kinds of non-linear geometric constraints.

Assuming that the linear approximation for each complex constraint is sound, it is clear that if the Basic SLA Algorithm terminates, it will return a solution that satisfies all of the complex constraints. Proof of termination depends on how configuration updating is performed—one needs to ensure that the configurations do not cycle without actually improving the solution. By only updating configurations when the objective can be improved, we can guarantee that cycling cannot occur. If each constraint has a finite set of configurations, this is sufficient to ensure termination. Otherwise, care must be taken to ensure that there cannot be an infinite sequence of configurations with infinitesimally improving objective values.

One might hope that the solution returned by the Basic SLA Algorithm is a *global optimum* in the sense that it is a solution to the complex constraints that minimizes the objective function. However, in general this is unrealistic since for the kinds of non-linear geometric constraints we are considering it is typically NP-hard to find a global optimum, and so in any algorithm fast enough for practical use the best that one can hope for is that the solution is a *local optimum*. A reasonable choice of configuration update will provide this. In practice, local optimization is preferable for direct manipulation, as the solver will respond more predictably to

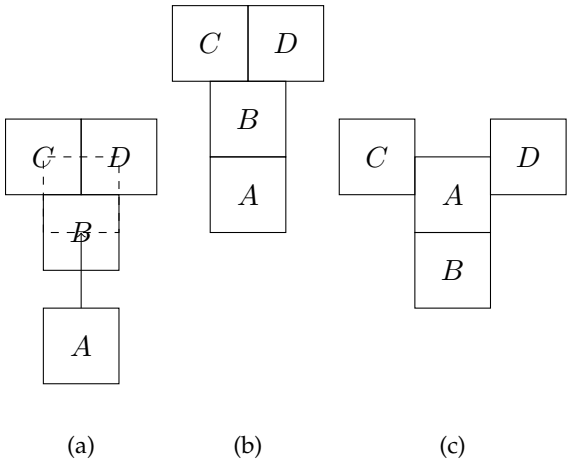


Figure 8.5: (a) A set of non-overlapping squares. A is to be moved to the dashed square. (b) The local optimum computed from the initial configurations. (c) A global optimum.

user input – as illustrated in Figure 8.5, moving to the global solution may result in sudden structural rearrangements to the diagram.

8.3.3 The Lazy SLA Algorithm

Our experience with the Basic SLA algorithm shows that if there are a large number of complex constraints it may become slow because of the large number of linear constraints in the linear solver. We now give a variant of the algorithm which is *lazy* in the sense that it does not use a linear approximation for the complex geometric constraints until they are about to become violated. This can significantly improve efficiency because it reduces the number of constraints in the linear solver.

In the lazy algorithm a complex constraint c need not be currently enforced by any linear approximation and so $c.config$ returns the "empty" linear approximation. Constraints which are being enforced by a linear approximation are said to be *enforced*. The algorithm relies on the object c representing a complex constraint having three additional methods: $c.enforced$ which returns the status of whether or not the constraint is currently enforced, $c.safe(\theta)$ checks whether or not the constraint c can be safely left unenforced with the solution θ since θ satisfies it, $c.enforce(\theta)$ which enforces c and sets $c.config$ to a configuration that satisfies θ and $c.unenforce$ which stops enforcing c and sets $c.config$ to the empty configuration. Note that $c.safe(\theta)$ is a pre-condition for $c.enforce(\theta)$.

```
lazy_sla(C, o, θ)
    finished := false
    while(¬finished)
        θ' := minimize o subject to ⋀_{c ∈ C s.t. c.enforced} c.config
        finished := update_enforced(C, θ, θ')
        if(finished)
            θ := θ'
            finished := update_configs(C, θ')
    return θ'

update_enforced(C,θ, θ')
    for(c ∈ C s.t. not c.enforced)
        if(¬c.safe(θ'))
            c.enforce(θ)
            return false
    return true

update_configs(C, θ)
    ret := true
    for(c ∈ C s.t. c.enforced)
        F := c.config
        c.update(θ)
        if(F ≠ c.config) ret := false
        else if(c.safe(θ)) c.unenforce()
    endfor
    return ret
```

Figure 8.6: Lazy SLA Algorithm for solving sets of non-linear constraints using smooth linear approximations.

The Lazy SLA Algorithm is given in Figure 8.6. The Lazy SLA Algorithm takes an additional input argument, θ, which is the current solution and which must satisfy all of the complex constraints including those that are unenforced. It is an invariant of the algorithm that θ remains set to such a solution. The procedure update_enforced ensures that the proposed solution θ' to the enforced constraints also satisfies the unenforced constraints: if it does not, an unsatisfied unenforced constraint has its status changed to enforced and the main loop is executed again. The procedure update_configs is similar to that in the Basic SLA Algorithm, and updates the current configuration in all enforced constraints. It also changes the status of enforced constraints to unenforced if θ' allows this.

Termination and correctness of the Lazy SLA Algorithm is the same as for the Basic SLA Algorithm: it will always return a solution satisfying all of the constraints and for appropriate choices of complex constraint methods will terminate and return a locally optimal solution.

```
update(θ)
    let left-of ≡ x1 + w1/2 ≤ x2 − w2/2
    let right-of ≡ x1 − w1/2 ≥ x2 + w2/2
    let above ≡ y1 − h1/2 ≥ y2 + h2/2
    let below ≡ y1 + h1/2 ≤ y2 − h2/2
    switch(config)
        case(left-of)
            if(active(left-of, θ))
                if(val(above, θ) and better(left-of, above, θ))
                    config := above
                else if(val(below, θ) and better(left-of, below, θ))
                    config := below
            return
        case(above)
            if(active(above, θ))
                if(val(right-of, θ) and better(above, right-of, θ))
                    config := right-of
                else if(val(left-of, θ) and better(above, left-of, θ))
                    config := left-of
            return
        case(right-of)
            if(active(right-of, θ))
                if(val(below, θ) and better(right-of, below, θ))
                    config := below
                else if(val(above, θ) and better(right-of, above, θ))
                    config := above
            return
        case(below)
            if(active(below, θ))
                if(val(left-of, θ) and better(below, left-of, θ))
                    config := left-of
                else if(val(right-of, θ) and better(below, right-of, θ))
                    config := right-of
            return
```

Figure 8.7: Configuration update method for non-overlapping boxes.

8.4 Examples of SLA

In this section we give a number of simple examples illustrating the power of SLA.

8.4.1 Non-overlapping boxes

We start with a very simple example: non-overlap of two boxes, i.e. axis-parallel rectangles. Conceptually the approach is straightforward: we model non-overlap of boxes R_1 and R_2 by enforcing "R_1 *left-of* R_2," "R_1 *right-of* R_2," "R_1 *above* R_2" or "R_1 *below* R_2." It should be clear that this is a sound and complete approxima-

tion to the non-overlap constraint. Figure 8.1 illustrates how SLA works with this example.

Figure 8.7 gives the configuration update method. It assumes that the rectangle R_i has variables (x_i, y_i) giving its center and w_i and h_i giving its width and height respectively. It is worth emphasizing that the size and height of the rectangles are not required to be fixed: they can be true variables whose value is computed by the solver. The update method calls the function $val(e, \theta)$ which returns the value of an expression e which may contain variables under the current solution θ. In the case e is a constraint, it tests if θ satisfies the constraint c, returning $true$ or $false$ appropriately. It also uses the Boolean function $active(c, \theta)$ which tests whether the linear constraint c is *active* at the current solution.

Intuitively a linear inequality constraint $c \equiv \sum_{i=1}^{n} a_i x_i \leq b$ is *active* if removing that constraint could lead to a better optimum solution. If $\sum_{i=1}^{n} a_i x_i < b$ the constraint is not active since it is not binding the current solution. Thus one possible definition of active is that an inequality constraint is active whenever it is *tight* in the sense that $\sum_{i=1}^{n} a_i x_i = b$. Unfortunately, while simple and easy to implement, this definition is a little bit too general. Imagine that the desired position for the two boxes is that that the upper-right corner of R_1 touches the bottom-left corner of R_2. In this case both the *left-of* and *below* constraints are tight. If we use tightness as the definition of active then the configuration update method will cycle indefinitely, alternating between these two constraints as the current configuration. However, in reality while tight, neither of these constraints are active in the sense that removing them leads to a better solution since the optimum solution is that the two corners touch.

Thus we require a definition of active that really captures how the constraint is affecting the current solution. A fundamental notion in constrained optimization, the constraint's *Lagrange multiplier*, measures exactly this. As described in Chapter 2, the Lagrange multiplier for a constraint is exactly the coefficient of the corresponding slack variable in the simplex tableau. For the purposes of this chapter, the key property is that the value of the Lagrange multiplier λ_c for a linear inequality $c \equiv \sum_{i=1}^{n} a_i x_i \leq b$ provides a measure of how "binding" a constraint is; it gives the rate of increase of the objective function as a function of the rate of increase of b. That is, it gives the cost that imposing the constraint will have on the objective

update(θ)
  if($active(config, \theta)$)
    $dx := val(x_2 - x_1, \theta)$
    $dy := val(y_2 - y_1, \theta)$
    $d := \sqrt{dx^2 + dy^2}$
    $config := \frac{dx}{d} \times (x_2 - x_1) + \frac{dy}{d} \times (y_2 - y_1) \geq r$
  return

Figure 8.8: Configuration update method for minimum Euclidean distance.

function, or conversely, how much the objective can be increased if the constraint is relaxed.[2]

Thus, intuitively, a constraint with a small Lagrange multiplier is preferable to one with a large Lagrange multiplier since it has less effect on the objective. In particular, removing a constraint with a Lagrange multiplier of 0 will not allow the objective to be improved and so the Lagrange multiplier is defined to be 0 for an inequality that is not active, i.e. if $\sum_{i=1}^{n} a_i x_i < b$. Simplex-based LP solvers, as a byproduct of optimization, compute the Lagrange multiplier of all constraints in the solver. In our example we therefore use the definition that $active(c, \theta)$ holds iff $\lambda_c \neq 0$.

The only subtlety in the configuration update method is the need to ensure that we do not get cycling behaviour resulting from repeatedly flipping between two configurations. The final part of the puzzle is the definition of the function $better(c_1, c_2, \theta)$ which essentially determines if it is worthwhile swapping active constraint c_1 for a feasible constraint c_2. The trick here is to ensure that we do not get cycling by flipping between two different configurations. This is done by temporarily adding c_2 to the constraint solver and computing λ_{c_1} and λ_{c_2} and then returning $\lambda_{c_1} > \lambda_{c_2}$ which holds if it is "better" to swap to c_2 since this will lead to a constraint with a smaller Lagrange multiplier. Computing the new Lagrange multiplier is very efficient since the current solution will still be the optimum. Note that whatever the result, the function $better$ does not permanently add c_2 to the solver.

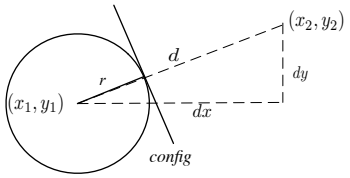


Figure 8.9: Computation of new linear approximation for minimum Euclidean instance.

8.4.2 Minimum Euclidean distance

Our next example is also quite simple: imposing a minimum Euclidean distance r between two points (x_1, y_1) and (x_2, y_2). Figure 8.8 gives the configuration update method. The update method computes a new linear approximation if the current configuration is active. This new approximation depends upon the current position of the points (x_1, y_1) and (x_2, y_2) and can be understood to be the linearization of the minimum distance constraint around the tangent point. It is illustrated in Figure 8.9.

Unlike the case for non-overlapping boxes, the new configuration for Euclidean distance must be computed dynamically since there are an infinite number of possible configurations. Note that we can use this construct to model non-overlap of two circles.

8.4.3 Point on the perimeter of a rectangle

In node-link diagrams representing metabolic pathways, one drawing convention for cycles is to place nodes in the cycle on a perimeter of an axis-aligned rectangle whose size and position adjusts to the desired position of the nodes. Again this is straightforward to encode using SLA. Figure 8.10 gives the configuration update method for the constraint that point p lies on the perimeter of rectangle R. There are four configurations: "p *on-left* R," "p *on-right* R," "p *on-top* R," and "p *on-bottom* R" which correspond to which side of the rectangle the point lies on. Clearly this is a sound and complete approximation to the original complex constraint. The code assumes that the point p has variables (x_p, y_p) giving its position and that rectangle

[2]It follows that at an optimal solution the Lagrange multiplier λ_c for an inequality cannot be negative.

update(θ)
  let *on-top* $\equiv\ y_p = y_R + \frac{h_R}{2} \wedge x_p \leq x_P + \frac{w_R}{2} \wedge x_p \geq x_P - \frac{w_R}{2}$
  let *on-bottom* $\equiv\ y_p = y_R - \frac{h_R}{2} \wedge x_p \leq x_P + \frac{w_R}{2} \wedge x_p \geq x_P - \frac{w_R}{2}$
  let *on-left* $\equiv\ x_p = x_R - \frac{w_R}{2} \wedge y_p \leq y_P + \frac{h_R}{2} \wedge y_p \geq y_P - \frac{h_R}{2}$
  let *on-right* $\equiv\ x_p = x_R + \frac{w_R}{2} \wedge y_p \leq y_P + \frac{h_R}{2} \wedge y_p \geq y_P - \frac{h_R}{2}$
  switch(*config*)
    case(*on-top*)
      if($val(x_p = x_R - \frac{w_R}{2}, \theta)$ **and** $lm(y_p = y_R + \frac{h_R}{2}, \theta) > 0)$)
        config := *on-left*
      else if($val(x_p = x_R + \frac{w_R}{2}, \theta)$ **and** $lm(y_p = y_R + \frac{h_R}{2}, \theta) > 0)$)
        config := *on-right*
      return
    case(*on-bottom*) ... analogous to *on-top*
    case(*on-left*) ... analogous to *on-top*
    case(*on-right*) ... analogous to *on-top*

Figure 8.10: Configuration update method for constraining a point to lie on a perimeter of a rectangle.

R has variables (x_R, y_R) giving its center and w_R and h_R giving its width and height respectively.

A difference from the non-overlapping box example is that each configuration is a conjunction of three linear constraints rather than a single constraint. For instance "p *on-top* R" is modelled by

$$y_p = y_R + \frac{h_R}{2} \ \wedge\ x_p \leq x_P + \frac{w_R}{2} \ \wedge\ x_p \geq x_P - \frac{w_R}{2}$$

which ensures that p is on the top side of R. The criteria for changing the configuration is that the point must be on the corner of the rectangle and that the Lagrange multiplier associated with the linear equality constraint of the current configuration indicates that the change would be beneficial. For instance if the current configuration is *on-top* and p is at the top left corner then the algorithm will change to the *on-left* configuration if the Lagrange multiplier of $y_p = y_R + \frac{h_R}{2}$ is strictly positive since a strictly positive Lagrange multiplier means that reducing the height of the rectangle would reduce the objective function and so it would be beneficial to allow p to move down the side. We use the function $lm(c, \theta)$ to compute the Lagrange multiplier for constraint c at the current solution θ.

It is straightforward to generalize this constraint to one that allows the point to lie on the perimeter of an arbitrary convex polygon. It should be clear that the approximation is sound and complete.

Figure 8.11: SLA of containment within a non-convex polygon using a dynamic maximal convex polygon. Initially containment in the non-convex polygon is approximated by containment in the rectangle C_1. When the point is moved and reaches a boundary of C_1 the approximation is updated to the rectangle C_2.

8.4.4 Containment within a non-convex polygon

Containment within a polygon is a useful geometric constraint. Forcing a point to lie inside a convex polygon is naturally modelled using linear constraints but containment within a non-convex polygon cannot be modelled with a conjunction of linear constraints. SLA is well-suited to modelling containment of a point p within a non-convex polygon P: we simply approximate P by a (possibly infinite) set of convex polygons $C_1, C_2, ...$ s.t. $C_i \subset P$ and that $\bigcup_i C_i = P$. Containment within P is soundly and completely approximated by containment in one of the C_i. There are many possible ways to choose the convex polygons.

The first approach we explored is to choose the C_i to be the set of maximal convex polygons that lie inside P. They are computed dynamically and are allowed to overlap. The algorithm updates the configuration corresponding to C_i whenever p lies on a boundary of C_i which is not a boundary of P. It computes a new maximal convex polygon C_j inside P that strictly contains p. The process is illustrated in Figure 8.11.

The disadvantage of this approach is that it is not simple to compute the new maximal convex polygon. It is also quite an expensive operation. We have therefore explored another approach which is suitable so long as the non-convex polygon P is rigid (i.e. the shape and orientation is fixed) and simple (i.e. with no crossed edges). The approach is to decompose P into triangular regions $T_1, ..., T_n$. Such decompositions are standard in computer graphics and there are a number of algorithms for partitioning simple polygons into triangles [Seidel, 1991, Chazelle, 1991],

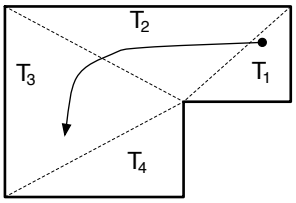


Figure 8.12: SLA of containment within a non-convex polygon using triangular decomposition. The figure shows the triangular decomposition of the non-convex polygon from Figure 8.11. The original approximation is containment in triangle T_1. As the point moves and touches a triangle boundary this is updated to containment in triangle T_2 and then triangle T_3.

of varying complexity. Our implementation uses a simple approach described by O'Rourke [1998].

Each triangle T_i gives rise to a different configuration in which p is constrained to lie inside T_i. Containment within T_i is enforced using three linear constraints, one for each side of the triangle. When the point p is on the boundary of T and the side it is on s is common to triangle T' then the approximation will be updated to containment within T' if the constraint corresponding to s is active. This is illustrated in Figure 8.12. In the case p is on a vertex and the corresponding two sides are active and can be updated, the side with the highest Lagrange multiplier is chosen.

8.4.5 Textboxes

Variable height text boxes are provided in most graphical editors and presentation software. These are axis-aligned rectangles whose width is specified by the user and whose height expands/shrinks to fit the text. When using a graphical editor with constraints the natural generalization of variable height textboxes are rectangles whose width or height can vary but which are always large enough to contain their textual content.

Textboxes are equivalent to the table cells discussed in Chapter 6; they have a finite number of *minimal layouts* where a minimal layout is a pair (w, h) such that the text in the textbox can be laid out in a rectangle with width w and height h but there is no smaller rectangle for which this is true. That is, for all $w' \leq w$ and $h' \leq h$ either $h = h'$ and $w = w'$, or the text does not fit in a rectangle with width w' and

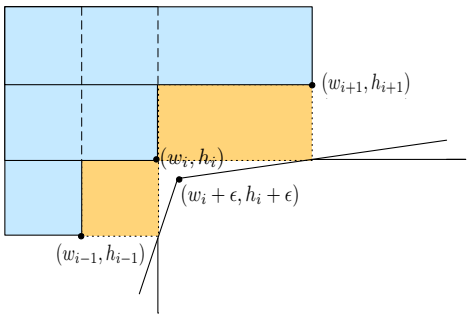


Figure 8.13: Linear approximations for textboxes. The minimal layouts (w_i, h_i) are marked with bullet points. The feasible solutions for the layout are points below and to the left of the shaded region.

height h'. For simplicity we assume that these minimum layouts are *anti-monotonic* in the sense that if the width increases then the height will never increase—this is almost invariably true in practice.

Assume that the textbox T has width w and height h. Requiring T to be large enough to contain its textual content is equivalent to requiring that $w \geq w_i$ and $h \geq h_i$ for one of its minimal layouts (w_i, h_i). SLA can be used to move between the different choices of minimal layouts. The only catch is that because the constraints $w \geq w_i$ and $h \geq h_i$ are at right-angles the solution tends to stick in the minimal layout $w = w_i$ and $h = h_i$. To smooth the transition to adjacent configurations we "flatten" this by adding a small constant ϵ to w_i and h_i. Assuming that the next narrower minimal layout is (w_{i-1}, h_{i-1}) and the next wider layout is (w_{i+1}, h_{i+1}) the actual linear approximation we use is shown in Figure 8.13. This approximation is clearly no longer complete, but remains sound and does not appear to cause any numerical stability issues.

The configuration update method is shown in Figure 8.14. It moves between adjacent minimal layouts when the current width and height allow this. Note that the geometry of the minimal layouts ensures that the conditions on at most one of the if statements can hold. In practice the minimal layouts need not be computed all at once but dynamically as needed. Efficient methods for computing minimal layouts are surveyed in Hurst et al. [2009]. We use the binary search algorithm described in Hurst et al. [2006c].

update(θ)
let $[(w_1, h_1), ..., (w_n, h_n)]$ be the minimal layouts ordered by increasing width.
let $wb_i \equiv w \geq w_i$ for $i = 1..n$
let $hb_i \equiv h \geq h_i$ for $i = 1..n$
let $wc_1 \equiv wb_1$
let $wc_i \equiv wb_i \wedge (w_i + \epsilon - w_{i-1}) \times (h - h_{i-1}) \leq (h_i + \epsilon - h_{i-1}) \times (w - w_{i-1})$ for $i = 2..n$
let $hc_n \equiv hb_n$
let $hc_i \equiv hb_i \wedge (w_{i+1} - w_i - \epsilon) \times (h - h_i - \epsilon) \leq (h_{i+1} - h_i - \epsilon) \times (w - w_i - \epsilon)$ for $i = 1..n-1$
let i be s.t. $config \equiv wc_i \wedge hc_i$
if$(active(wb_i, \theta)$ **and** $i > 1$ **and** $lm(wb_i, \theta) > 0)$
  $config := wc_{i-1} \wedge hc_{i-1}$
else if$(active(hb_i, \theta)$ **and** $i < n$ **and** $lm(hb_i, \theta) > 0)$
  $config := wc_{i+1} \wedge hc_{i+1}$
return

Figure 8.14: Configuration update method for requiring a textbox to be large enough to contain its textual content.

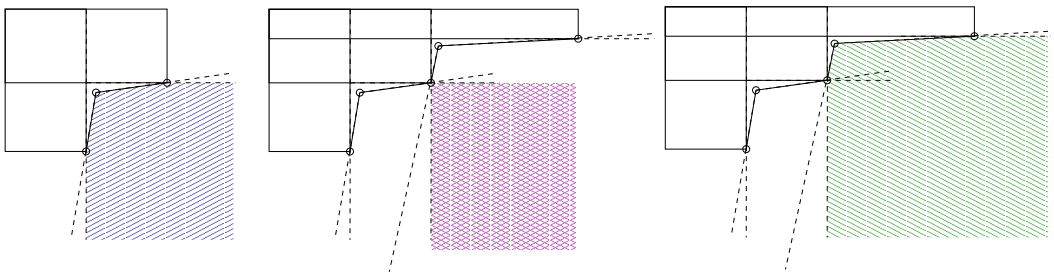


Figure 8.15: Approximation of textbox configurations. Two different configurations, and the intermediate stage are marked; the shaded region is the set of legal values for the width and height. Note that at any point where a transition between configurations may occur, the point satisfies both configurations.

8.5 Non-Overlap of Polygons

In the previous section we saw how to model non-overlap of two boxes and of two circles. In this section we consider non-overlap of convex and non-convex polygons. We start by considering non-overlap of two convex polygons. We restrict our attention to rigid polygons since this means that the slope of the polygon's edges are fixed which allows us to use a linear constraint to model a point being on, above or below the edge.

8.5.1 Non-overlap of two convex polygons

The obvious approach to handle non-overlap of two convex polygons P and Q is to choose an edge of P for which the corresponding line separates the two polygons

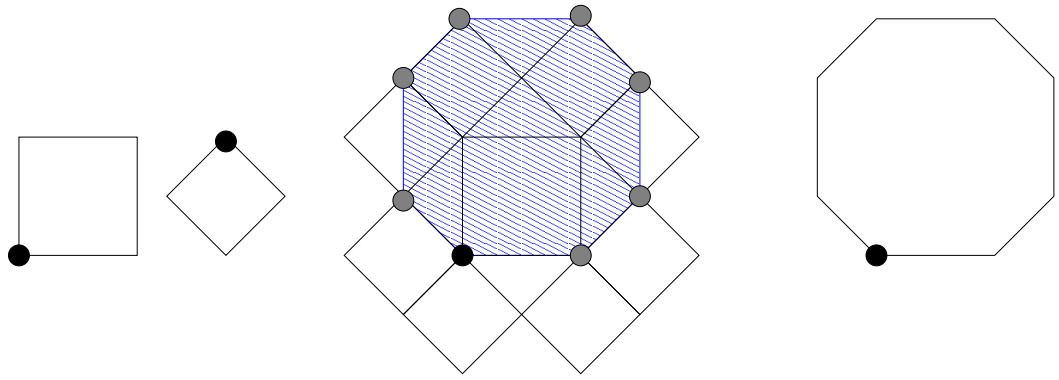


Figure 8.16: A unit square S and unit diamond D and their Minkowski difference $S \oplus -D$. The local origin points for each shape are shown as circles.

and add a constraint that the closest point on Q to this line remains on the other side of the line. This is the direct generalization of the approach used for non-overlap of boxes. When the linear approximation is updated we need to move to the appropriate adjacent edge on P and compute the new closest point on Q. Conceptually this is what we do. However, our implementation is simplified by using the Minkowski difference, denoted by $P \oplus -Q$, of the two polygons P and Q to essentially pre-compute the closest point. Given some fixed point p_Q in Q and p_P in P the *Minkowski difference* is the polygon M such that the point $p_Q - p_P$ (henceforth referred to as the *query point*) is inside M iff P and Q intersect.

For convex polygons, it is possible to "walk" one polygon around the boundary of the second; the vertices of the Minkowski difference consist of the offsets of the second polygon at the extreme points of the walk. It follows that the Minkowski difference of two convex polygons is also convex. An example of the Minkowski difference of two convex polygons is given in Figure 8.16 while an example of a non-convex Minkowski sum is shown in Figure 8.19.

There has been considerable research into how to compute the Minkowski difference of two polygons efficiently.[3] Optimal $O(n + m)$ algorithms for computing the Minkowski difference of two convex polygons with n and m vertices have been known for some time [Ghosh, 1990, O'Rourke, 1998]. Until recently calculation of the Minkowski difference of non-convex polygons decomposed the polygons into convex components, constructed the convex Minkowski difference of each pair, and

[3]More precisely, research has focused on the computation of their Minkowski sum since the Minkowski difference of A and B is simply the Minkowski sum of A and a reflection of B.

```
update(θ)
  let [(x_0, y_0), ..., (x_{n-1}, y_{n-1})] be the offset from (x_P, y_P) of the vertices of M in clockwise order.
  let c_i ≡ (x_{i+1 mod n} − x_i) × (y_Q − y_i − x_P) ≥ (y_{i+1 mod n} − y_i) × (x_Q − x_i − x_P) for i = 0..n
  let i be s.t. config ≡ c_i
  if(active(c_i, θ))
    if(val(c_{(i−1)modn}, θ) and better(c_i, c_{(i−1)modn}, θ))
      config := c_{(i−1)modn}
    else if(val(c_{(i+1)modn}, θ) and better(c_i, c_{(i+1)modn}, θ))
      config := c_{(i+1)modn}
  return
```

Figure 8.17: Configuration update method for non-overlap of two rigid polygons P and Q with Minkowski difference M computed using the reference points $p_Q \equiv (x_Q, y_Q)$ in Q and $p_P \equiv (x_P, y_P)$ in P.

took the union of the resulting differences. More recently direct algorithms have appeared based on convolutions of a pair of polygons [Ramkumar, 1996, Flato, 2000].

We can model non-overlap of convex polygons P and Q by the constraint that the query point is *not* inside their Minkowski difference, M. As the Minkoskwi difference of two convex polygons is a convex polygon, it is straightforward to model non-containment in M: it is a disjunction of single linear constraints, one for each side of M, specifying that the query point lies on the outside of that edge.

The approximation is sound and complete. It is also relatively simple and efficient to update the approximation as the shapes are moved. If the constraint corresponding to the current edge is active then we move to an adjacent edge whenever this is feasible and strictly reduces the associated Lagrange multiplier. We note that the Minkowski difference only needs to be computed once. The actual linear approximation we use is shown in Figure 8.17; it is very similar to the code in Figure 8.7.

8.5.2 Overlap of two non-convex polygons

We now extend our technique for handling non-overlap of convex polygons to the case when one or both of the polygons are non-convex. We restrict attention to simple polygons without internal holes. When using SLA we cannot move an object from outside a polygon to inside an internal hole of a polygon in any case.

Probably the most obvious approach is to decompose each non-convex polygon into a union of convex polygons which are constrained to be joined together (either

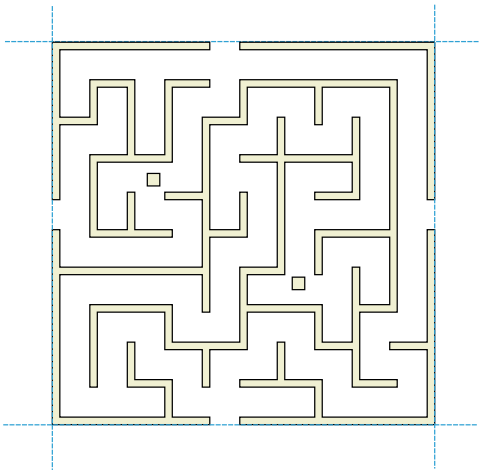


Figure 8.18: Example diagram with complex non-convex polygons.

using equality constraints, or simply using the same variables to denote the shared position), and add a non-overlap constraint for each pair of polygons.

This *decomposition* based method is relatively simple to implement since there are a number of well explored methods for convex partitioning of polygons, including Greene's dynamic programming method [Greene, 1983] for optimal partitioning. However, it has a potentially serious drawback: in the worst case, even the optimal decomposition of a non-convex polygon will have a number of convex components that is linear in the number of vertices in the polygon. This means that in the worst case the non-overlap constraint for a pair of non-convex polygons with n and m vertices will lead to $\Omega(nm)$ non-overlap constraints between the convex polygons.

Consider the shapes shown in Figure 8.18. There are 4 non-convex polygons (each starting from the corners of the maze), and two fixed convex polygons within the maze. Each non-convex polygon is aligned with its neighbours on the edge of the maze. There are 12, 2, 6, and 34 components of the non-convex objects in the NW, NE, SE, and SW respectively. The non-overlap of the NW and SE corner objects requires 72 non-overlap constraints to encode in the decomposition based approach, but indeed these two objects will never interact due to the other constraints.

As illustrated by the maze example, in reality most of these $\Omega(nm)$ non-overlap constraints are redundant and unnecessary. An alternative approach is to use our earlier observation that we can model non-overlap of convex polygons P and Q by

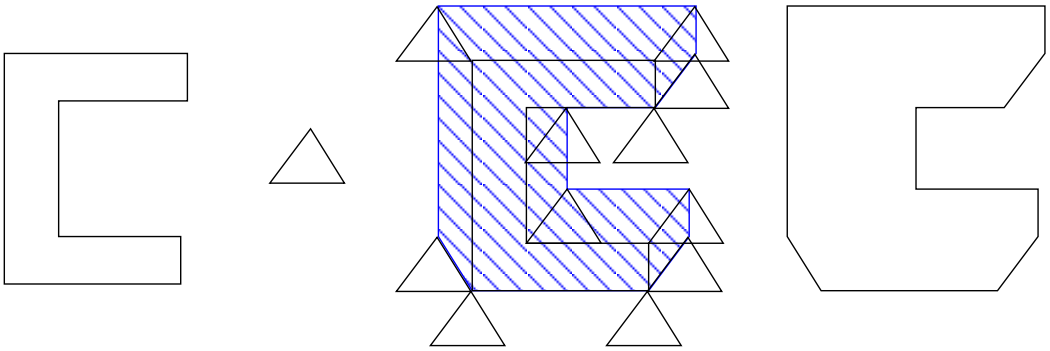


Figure 8.19: The Minkowski difference of a non-convex and a convex polygon. From left, A, B, extreme positions of A and B, $A \oplus -B$.

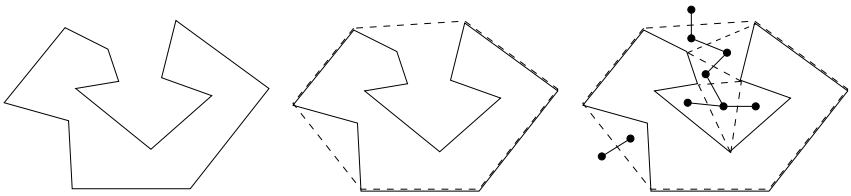


Figure 8.20: A non-convex polygon, together with convex hull, decomposed pockets and adjacency graphs. From a given region, the next configuration must be one of those adjacent to the current region.

the constraint that the query point is *not* inside their Minkowski difference, M. This remains true for non-convex polygons, although the Minkowski difference may now be non-convex. An example Minkowski difference for a non-convex polygon is shown in Figure 8.19.

In our second approach we pre-compute the Minkowski difference M of the two, possibly non-convex, polygons P and Q and then decompose the space not occupied by the Minkowski polygon into a union of convex regions, $R_1, .., R_m$. We have that the query point is not inside M iff it is inside one of these convex regions. Thus, we can model non-overlap by a disjunction of linear constraints, with one for each region R_i, specifying that the query point lies inside the region. We call this the *inverse* approach.

These regions cover the region outside the polygon's convex hull, the non-convex *pockets* where the boundary of the polygon deviates from the convex hull, and the *holes* inside the polygon. The key to the inverse approach is that whenever the query point is not overlapping with the polygon, it must be either outside the convex hull of the polygon (as in the convex case), or inside one of the pockets or holes. If each pocket and hole is then partitioned into convex regions, it is pos-

sible to approximate the non-overlap of two polygons with either a single linear constraint (for the convex hull) or a convex containment (for a pocket or hole). An example is shown in Figure 8.20. Note that in practice we can ignore holes in the polygon since with SLA it is impossible to reach them.

The configuration update algorithm is virtually identical to that detailed in Section 8.4.4. We use the approach in which pockets and holes are decomposed into triangular regions.

One of the advantages of the inverse approach is that, in most cases, particularly when the pairs of polygons are distant, the two polygons are treated as convex. It is only when the polygons are touching, and the query point lies upon the opening to a *pocket* that anything more complex occurs.

8.5.3 Overlap of multiple polygons

However, naive use of SLA to model non-overlap of many polygons leads to a quadratic growth in the number of linear constraints since at least one constraint is generated between each pair of objects (and potentially more in the case of non-convex polygons). Our experience suggests that this is impractical for larger diagrams and was the reason for developing the Lazy SLA Algorithm. The key to efficiency is to use Lazy SLA together with efficient incremental object collision detection techniques developed for computer graphics.

We have investigated two variants of this idea which differ in the meaning of overlap and hence the definition of the method $c.safe$ in the Lazy SLA Algorithm. The first variant tests the intersection of the polygons, so $c.safe$ holds if the two polygons do not strictly intersect. While this addresses the problem of having many constraints in the solver, $O(n^2)$ constraint checks must be performed during each update. We then augmented this with a bounding-box based detection step. If the bounding boxes do not strictly overlap, $c.safe$ holds; otherwise, we perform the normal intersection test.

Implementation relies on an efficient method for determining if the bounding boxes of the polygons overlap. Determining if n 2-D bodies overlap is a well studied problem and numerous algorithms and data structures devised including Quad/Oct-trees [Samet, 1990], and dynamic versions of structures such as range,

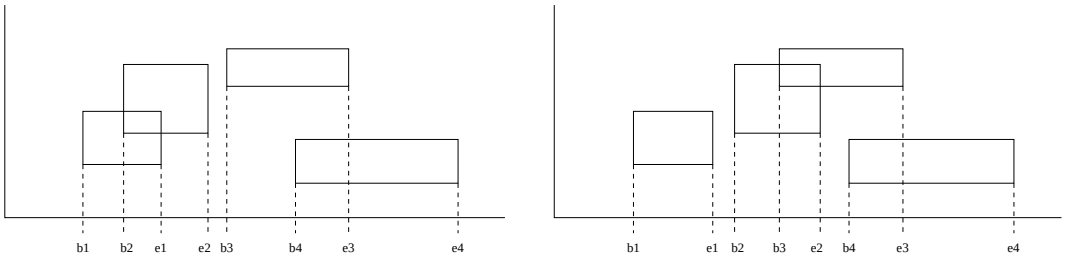


Figure 8.21: The sorted list of endpoints is kept to facilitate detection of changes in intersection. As the second box moves right, b_2 moves to the right of e_1, which means that boxes 1 and 2 can no longer intersect. Conversely, endpoint e_2 moves to the right of b_3, which means that boxes 2 and 3 may now intersect.

segment and interval-trees [Chiang and Tamassia, 1992]. The method we have chosen to use is an adaptation of that presented in Lin et al. [1996].

The algorithm is based, as with most efficient rectangle-intersection solutions, on the observation that two rectangles in some number of dimensions will intersect if and only if the span of the rectangles intersect in every dimension. Thus, maintaining a set of intersecting rectangles is equivalent to maintaining (in two dimensions) two sets of intersecting intervals.

The algorithm acts by first building a sorted list of rectangle endpoints, and marking corresponding pairs to denote whether or not they are intersecting in either dimension. While this step takes, in the worst case, $O(n^2)$ time for n rectangles, it is in general significantly faster. As shapes are moved, the list must be maintained in sorted order, and intersecting pairs updated. This is done by using insertion sort at each time-step, which will sort an almost sorted list in $O(n)$ time.

In order to use the Lazy SLA Algorithm we must also provide a definition for the method $c.enforce(\theta)$ which chooses a configuration to enforce the constraint c that two polygons do not overlap. This is done by trying the configurations in turn, choosing the first configuration that is satisfied by θ.

A change in intersection is registered only when a left and right endpoint of different bounding boxes swap positions. If a left endpoint is shifted to the left of a right endpoint, an intersection is added if and only if the boxes are already intersecting in all other dimensions. If a left endpoint is shifted to the right of a right endpoint, the pair cannot intersect. (See Figure 8.21)

8.6 Evaluation

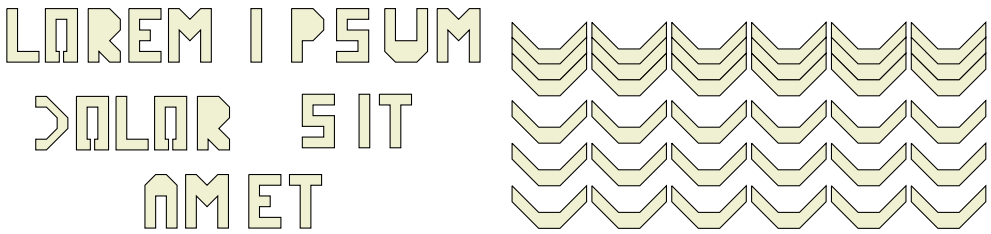
We have implemented all of the algorithms described in the chapter. They were implemented using the Cassowary linear inequality solver [Badros et al., 2001], included with the QOCA constraint solving toolkit [Marriott et al., 1998]. Non-convex Minkowski difference calculation was implemented using the Minkowski_sum_2 CGAL package produced by Wein [Wein, 2006].

All of the applications of SLA described in Section 8.4 work well and are more than fast enough for interactive graphical applications. For this reason in the experimental evaluation described in this section we focus on evaluating the performance of the algorithms proposed for non-overlap of many non-convex polygons because this is the most complex and potentially expensive geometric constraint handled by SLA. We evaluate whether they are fast enough to support incremental update during direct manipulation since this is the most demanding requirement.

We implemented both the decomposition and direct approach for handling non-overlap of non-convex polygons. The decomposition was handled using Greene's dynamic programming algorithm [Greene, 1983]. For the decomposition approach, both the eager and lazy variants were implemented; however only the lazy variants of the direct approach were tested. Each variant was tested with (of course) exactly the same input sequence of interaction.

While much consideration has been given to the implementation of Simplex-based constraint solvers, they are still vulnerable to floating-point inaccuracy. This can particularly be a problem when repeatedly solving highly constrained problems with non-integer coefficients, such as occur with non-overlap and text layout. As such, we provide results with both double-precision arithmetic, which is significantly faster but vulnerable to numerical stability issues, and an exact rational representation using GMP (`gmplib.org`).

Two experiments were conducted. Both involved direct manipulation of diagrams containing a large number of non-convex polygons some of which were linked by alignment constraints. We focused on non-convex polygons because all of our algorithms will be faster with convex polygons than non-convex. The experimental comparison of the approaches were run on an Intel Core2 Duo E8400 with 4GB RAM.



(a) (b)

Figure 8.22: Diagrams for testing. (a) Non-overlap of polygons representing text. The actual diagram is constructed of either 2 or 3 repetitions of this phrase. (b) The top row of shapes is constrained to align, and pushed down until each successive row of shapes is in contact.

The first experiment measured the time taken to solve the constraint system for a particular set of desired values during direct manipulation of the diagram containing non-convex polygons representing letters, shown in Figure 8.22(a). Individual letters were selected and moved into the center of the diagram in turn. The results are given in Table 8.1(a). Note that the eager variants were terminated after 30 minutes – they were far too slow to be usable in practice. It is interesting to note that, in this instance, the solver was faster using rationals than using doubles; this appears to be due to numerical instability causing the simplex solver to converge slowly.

In order to further explore scalability, a diagram of tightly fitting U-shapes, Figure 8.22(b), of a varying number of rows was constructed, and the top row pushed through the lower layers. Results are given in Figure 8.1(b).

The performance of the eager decomposition approach clearly highlights the need for the Lazy SLA algorithm; the lazy version could be solved between 5 and 100 times faster.

The results clearly demonstrate that the direct approach is significantly faster than the decomposition approach. When using a floating point representation, any of the lazy variants could be used for direct manipulation. Even when using exact numerical representation, the direct approach, combined with bounding-box based collision detection, could be solved quickly enough to facilitate direct manipulation without noticeable delay.

Rational	Decomp.			Direct	
	Eager†	Lazy	Lazy+BB	Lazy	Lazy+BB
Ave Time	1510.36	392.00	24.52	58.73	12.21
Max Time	720184.00	982.90	123.09	247.07	41.00
Ave Cycle	2.84	1.18	2.06	1.14	1.14
Max Cycle	27	5	11	6	6
Double	Decomp.			Direct	
	Eager†	Lazy	Lazy+BB	Lazy	Lazy+BB
Ave Time	2428.26	6.27	1.56	1.08	0.89
Max Time	169708.00	68.53	10.56	5.84	33.66
Ave Cycle	3.948	1.37	1.61	1.17	1.18
Max Cycle	15	16	13	6	18

(a) Text diagram

Rational	Decomp.			Direct	
Rows	Eager	Lazy	Lazy+BB	Lazy	Lazy+BB
5	365.87	71.08	18.38	18.74	10.01
6	623.04	100.86	23.80	25.86	12.23
7	1080.01	133.83	29.64	36.65	15.90
8	1545.00	164.58	35.82	46.10	18.37
9	2698.42	206.31	40.84	59.02	23.47
10	3830.43	246.33	47.44	72.38	25.18
Double	Decomp.			Direct	
Rows	Eager	Lazy	Lazy+BB	Lazy	Lazy+BB
5	37.00	0.98	0.80	0.67	0.66
6	96.43	1.29	1.03	0.83	0.83
7	121.53	1.82	1.30	1.15	1.07
8	215.59	2.62	1.46	1.27	1.25
9	316.91	3.64	1.78	1.65	1.50
10	528.65	4.67	2.03	1.95	1.65

(b) U-shaped polygons

Table 8.1: Experimental results. For the text diagram, we show the average and maximum time to reach a stable solution, and the average and maximum number of solving cycles (iterations of the **repeat while** loop in Figure 8.6) to stabilize. Experiments marked with † were terminated after 30 mins. For the U-shaped polygon test we show average time to reach a stable solution as the number of rows increases. All times are in milliseconds.

8.7 Conclusions

We have described a new generic approach to geometric constraint solving in interactive graphical application that we call smooth linear approximation (SLA). We believe it is the first viable approach to handling a wide variety of non-linear geometric constraints in combination with linear constraints in interactive constraint-based graphical applications.

A particular focus of the chapter has been handling non-overlap of (possibly non-convex) polygons. We presented two possible approaches for handling non-overlap of non-convex polygons and have shown that the direct method (which models non-overlap of polygons A and B by the constraint that A is contained in the region that is the complement of B) is significantly faster than decomposing each non-convex polygons into a collection of adjoining convex polygons.

We have also shown that the direct method can be sped up by combining it with traditional collision-detection techniques in order to lazily add the non-overlap constraint only when the bounding boxes of the polygons overlap. This is capable of solving non-overlap of large numbers of complex, non-convex polygons rapidly enough to allow direct manipulation, even when combined with other types of linear constraints, such as alignment constraints.

9

Conclusion

In this thesis we have developed two types of generic propagators for lazy clause generation solvers, that can be used to enforce a variety of global constraints, and developed models for a variety of diagram and document composition problems. We have also presented a modelling technique for supporting complex geometric constraints in an interactive constraint-based diagram system.

In Chapter 3, we introduced new algorithms for propagating constraints expressed as Multi-valued Decision Diagrams. To support their use in lazy clause generation solvers, we also developed several algorithms for explaining inferences generated by these propagators. We evaluated these propagators (and explanation algorithms) using several problems with a variety of `regular` and `sequence` constraints. In Chapter 4, we adapted these methods to constraints represented as `s-DNNF` circuits. This representation, a superset of MDDs, is slightly more expensive for analysis, but allows a polynomial representation for several classes of constraints that require an exponential number of nodes to construct as an MDD. We compared these `s-DNNF` propagators to previously published results on shift scheduling using a domain-consistent decomposition. In all cases, at least one of the `s-DNNF` propagators outperformed the decomposition; incremental propagation with explanation weakening was the best overall method on these problems. We also presented results for a forklift scheduling problem.

In both of these cases, incremental/greedy explanations were often superior, sometimes because they resulted in reduced search, and in other cases because minimal explanations were too expensive to compute. However, we were unable to

find any method that was uniformly better on all problems. A direction of definite interest, then, is to determine what characteristics of problem or constraint determine which explanation algorithm will be beneficial, either statically, or dynamically during search. Also, these explanation algorithms operate on a constraint in isolation. An interesting path for further work is to find a way of constructing a better (smaller, or more re-usable) global explanation taking into account the set of propagators involved in a conflict.

The intent of developing these techniques is to take advantage of modern Boolean reasoning techniques without losing the ability to reason about the high-level structure of the given problem. While we have considered two approaches for representing arbitrary constraints, there may be alternative representations which can more concisely encode certain classes of constraints, or allow more efficient analysis. Also, these propagators communicate only through clauses relating to the externally visible variables; there may be better ways of connecting the low-level Boolean representation with the higher-level propagators.

We have also applied combinatorial optimization techniques to solve a variety of document composition and layout problems. In Chapter 5 we developed SAT- and MIP-based models for constructing layouts for k-layered graphs with minimum crossings, and with a maximal planar subgraph. The MIP-based model was consistently able to solve larger crossing minimization problems than the SAT model; however for the planar subset problem, the SAT model was superior. Almost all the collected graphs from the GraphViz gallery could be solved within one minute, as could most of the random graphs. These models were extended to handle combined objective functions – solving first crossing minimization then planarization, and planarization then crossing minimization. For crossing minimization then planarization, the MIP-based model was again superior, only failing to solve 4 instances within one minute. For planarization then crossing minimization, the MIP model was able to quickly find good solutions, but the SAT model was able to prove optimality of substantially more instances.

The models in Chapter 5 only address the second phase of the Sugiyama layout process, and assumes nodes have already been assigned to layers. It would be interesting to construct a model for solving the optimal *complete* k-layered crossing minimization problem; that is, given a graph and a number of layers, find the

layering *and* ordering within layers which minimizes the number of crossings. It would also be of interest to apply similar techniques to other graph layout problems, such as optimal connector routing for graphs with rectangular (or otherwise area-consuming) vertices.

We have presented several models for computing minimal-height table layouts, both with and without column- and row-spans. These models used a variety of solver technologies, including integer programming, constraint programming and $A^\star$ search. As expected, the constraint-programming method without learning was substantially inferior to the other methods. All the other methods were able to solve almost all the scraped HTML tables in under 10 seconds; however, the cell-free lazy clause generation model consistently outperformed all the other models, solving all the HTML tables in under 0.1 seconds, and performing 1–2 orders of magnitude faster on the artificially generated tables. These models assume the input is text, so we can pre-compute a discrete set of possible configurations. Further work involves extending these models to handle a wider range of content; tables including sub-tables, images and other floating elements.

For the guillotine layout problem, we developed several methods for both the fixed and free variants. As the recursive structure of the problems (particularly the free layout problem) rendered them unsuitable for conventional constraint solvers, we developed bottom-up and top-down dynamic programming approaches. We also applied bounding techniques to improve the performance of the top-down approaches. For the fixed layout problem, the bottom-up method was far superior, and could quickly construct optimal solutions to large instances; adding bounding resulted in only a small improvement to top-down performance. The poor performance of the top-down methods is likely due to the large width and sparse solution space of the fixed layout problem. With the free layout problem, however, bounded top-down dynamic programming is approximately twice as fast as bottom-up construction. For instances with a column-based layout, the bounded top-down method is generally 1–2 orders of magnitude faster, as we can often cut off search early when we find a solution that is guaranteed to be optimal. As for the table layout, these guillotine layout methods are limited to text, where we precalculate the set of possible article configurations; eventually we would like to extend this to handle a wider class of media. Also, many articles are available in

multiple forms – with or without header images, and with more verbose body text. It would be interesting to explore related composition and pagination problems with different objective functions – trying to maximise some measure of "niceness" for the overall layout, rather than just minimizing height.

We have described Smooth Linear Approximation (SLA), a method for integrating complex geometric constraints into a constraint-based layout system by constructing and updating local linear approximations to the constraints. We demonstrated the use of this modelling technique by implementing a variety of constraints including flexible text-boxes, Euclidian separation constraints, and non-overlap of boxes, convex polygons and non-convex polygons. In many of these cases, maintaining the approximation in the solver is expensive, and many of the constraints are not binding (particularly in the case of non-overlap, with $\mathrm{O}(n^2)$ constraints, at most $\mathrm{O}(n)$ of which may be binding). As such, we developed the lazy SLA algorithm, and demonstrate that it is capable of maintaining non-overlap of a large number of complex polygons together with alignment constraints quickly enough to permit direct manipulation.

The current technique for handling polygon non-overlap constraints relies on having fixed size and orientation of the polygons. Handling re-sizing of convex polygons should be relatively simple; a more challenging task is to handle changes in orientation (such as rotation) or other kinds of deformation. Also of interest is the application of these techniques to other kinds of constraints.

As more and more of our reading moves online, it is increasingly necessary to provide dynamic publications which can adapt to individual readers and display devices. This requires the development of new tools and methods for automatically composing layouts for diverse classes of documents; these layout tasks are a ready supply of increasingly hard combinatorial problems. In turn, this motivates the development of new methods for improving the performance of combinatorial optimization techniques. Conversely, the continuing improvements to combinatorial optimization techniques allow us to solve practical instances of increasingly hard problems, and encourage us to attempt problems that were previously considered intractable. We hope that this cyclical feedback will result in considerable benefits for practitioners of both fields.

Bibliography

Reuters-21578, Distribution 1.0. `http://www.daviddlewis.com/resources/testcollections/reuters21578`.

K. Aardal, G. L. Nemhauser, and R. Weismantel. *Handbooks in Operations Research and Management Science: Discrete Optimization*. Elsevier, Burlington, MA, 2005.

I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. BDDs for pseudo-Boolean constraints - revisited. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing*, volume 6695 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2011.

T. Achterberg, T. Berthold, T. Koch, and K. Wolter. Constraint integer programming: A new approach to integrate CP and MIP. In *Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5015 of *Lecture Notes in Computer Science*, pages 6–20. Springer, 2008.

R. Alvarez-Valdés, A. Parajón, and J. M. Tamarit. A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems. *Computers & OR*, 29(7):925–947, 2002.

J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artificial Intelligence*, 135(1-2): 199–234, 2002.

R. J. Anderson and S. Sobti. The table layout problem. In *SCG '99: Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, pages 115–123, New York, NY, USA, 1999. ACM Press.

R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

C. B. Atkins. Blocked recursive image composition. In *Proceedings of the 16th International Conference on Multimedia*, pages 821–824, 2008.

G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, pages 73–82, New York, Nov. 1999. ACM.

G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Trans. CHI.*, 8(4):267–306, 2001.

D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH '94 Conference Proceedings*, pages 23–32, New York, 1994. ACM.

D. Baraff. Linear-time dynamics using Lagrange multipliers. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 137–146. ACM Press New York, NY, USA, 1996.

R. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world sat instances. pages 203–208, 1997.

R. J. Beach. *Setting tables and illustrations with style*. PhD thesis, University of Waterloo, 1985.

N. Beaumont. Fitting a table to a page using non-linear optimization. *Asia-Pacific Journal of Operational Research*, 21(2):259–270, 2004.

N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In M. Wallace, editor, *Proceedings of the 10th International conference on Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, pages 107–122. Springer-Verlag, 2004.

N. Beldiceanu, M. Carlsson, and J. Rampon. Global Constraint Catalog 2nd Edition. SICS Technical Report T2010:07, Swedish Institute of Computer Science, 2010.

R. Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

M. Bilauca and P. Healy. A new model for automated table layout. In *Proceedings of the 10th ACM Symposium on Document Engineering*, DocEng '10, pages 169–176. ACM, 2010.

M. Bilauca and P. Healy. Building table formatting tools. In *Proceedings of the 2011 ACM Symposium on Document Engineering*, pages 13–22. ACM, 2011.

A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications: Algorithm details. Technical Report 97-

06-01, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, July 1997a.

A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, pages 87–96, New York, Oct. 1997b. ACM.

A. Borning, R. Lin, and K. Marriott. Constraint-based document layout for the web. *Multimedia Systems*, 8(3):177–189, 2000.

B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2. W3C Working Draft, Jan. 1998. `http://www.w3.org/TR/WD-css2/`.

W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. Geometric constraint solver. *Computer-Aided Design*, 27(6):487–501, 1995.

S. Brand, N. Narodytska, C. Quimper, P. Stuckey, and T. Walsh. Encodings of the SEQUENCE constraint. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 210–224. Springer, 2007.

R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505, pages 174–177. Springer, 2009.

R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986a. doi: http://doi.ieeecomputersociety.org/10.1109/TC.1986.1676819.

R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986b.

D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. In *Proceedings of the 40th Annual Design Automation Conference*, pages 830–835. ACM, 2003.

B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6(1):485–524, 1991.

K. Cheng and R. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *14th International Conference on Principles and Process of Constraint Programming*, volume 5202 of *LNCS*, pages 509–523. Springer, 2008.

K. C. K. Cheng and R. H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

N. Christofides and E. Hadjiconstantinou. An exact algorithm for orthogonal 2-d cutting problems using guillotine cuts. *European Journal of Operational Research*, 83(1):21–38, 1995.

N. Christofides and C. Whitlock. An algorithm for two-dimensional cutting problems. *Operations Research*, pages 30–44, 1977.

J. Clark and S. Deach. Extensible Stylesheet Language (XSL), Version 1.0. World Wide Web Consortium Working Draft. `http://www.w3.org/TR/WD-xsl`, 1998.

M. Codish and M. Zazon-Ivry. Pairwise cardinality networks. In *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *LNCS*, pages 154–172. Springer, 2010.

S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

T. G. Crainic and J.-M. Rousseau. Multicommodity, multimode freight transportation: A general modeling and algorithmic framework for the service network design problem. *Transportation Research Part B: Methodological*, 20(3):225 – 242, 1986.

R. Damiano and J. Kukula. Checking satisfiability of a conjunction of BDDs. In *Proceedings of the 40th Design Automation Conference*, pages 818–823. ACM, 2003.

G. Dantzig. Discrete-variable extremum problems. *Operations Research*, pages 266–277, 1957.

G. Dantzig. *Linear programming and extensions*. The Rand Corporation, 1963.

A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 819–826. IJCAI/IAAI, 2011.

A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4):315–333, 2006.

G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

B. Dutertre and L. De Moura. The YICES SMT solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL($\mathcal{T}$). In *CAV*, pages 81–94, 2006.

T. Dwyer, K. Marriott, and M. Wybrow. Dunnart: A constraint-based network diagram authoring tool. In *Proceedings of the 16th International Symposium on Graph Drawing*, volume 5417 of *LNCS*, pages 420–431. Springer, 2008.

R. G. Dyson and A. S. Gregory. The cutting stock problem in the flat glass industry. *Operational Research Quarterly*, 25(1):41–53, 1974.

P. Eades and S. Whitesides. Drawing graphs in two layers. *Theoretical Computer Science*, 131(2):361–374, 1994.

N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

E. Flato. Robust and efficient construction of planar Minkowski sums. Master's thesis, School of Computer Science, Tel-Aviv University, 2000.

I. Fudos and C. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics (TOG)*, 16(2):179–216, 1997.

G. Gange, V. Lagoon, and P. Stuckey. Fast set bounds propagation using BDDs. In *Proceedings of the 18th European Conference on Artificial Intelligence*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 505–509. IOS Press, 2008.

G. Gange, P. Stuckey, and V. Lagoon. Fast set bounds propagation using a BDD-SAT hybrid. *Journal of Artificial Intelligence Research*, 38:307–338, 2010.

E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software: practice and experience*, 30(11):1203–1233, 2000.

M. Garey and D. Johnson. *Computers and intractability*. W. H. Freeman, 1979.

M. Garey and D. Johnson. Crossing number is np-complete. *SIAM Journal On Algebraic And Discrete Methods*, 4(3):312–316, 1983.

I. Gent, I. Miguel, and N. Moore. Lazy explanations for constraint propagators. In *12th Annual Symposium on Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2010.

P. Ghosh. A solution of polygon containment, spatial planning, and other related problems using Minkowski operations. *Comput. Vision Graph. Image Process.*, 49 (1):1–35, 1990.

R. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

R. Gomory. An algorithm for the mixed integer problem. Technical report, 1960.

J. González, J. Merelo, P. Castillo, V. Rivas, and G. Romero. Optimizing web newspaper layout using simulated annealing. In *Engineering Applications of Bio-Inspired Artificial Neural Networks*, volume 1607 of *Lecture Notes in Computer Science*, pages 759–768. Springer Berlin / Heidelberg, 1999.

D. Greene. The decomposition of polygons into convex parts. *Computational Geometry*, pages 235–259, 1983.

M. Harada, A. Witkin, and D. Baraff. Interactive physically-based manipulation of discrete/continuous models. In *SIGGRAPH '95 Conference Proceedings*, pages 199–208. ACM, 1995.

P. Hawkins and P. Stuckey. A hybrid BDD and SAT finite domain constraint solver. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages*, volume 3819 of *LNCS*, pages 103–117. Springer-Verlag, 2006.

P. Healy and A. Kuusik. The vertex-exchange graph: A new concept for multi-level crossing minimisation. In *Proceedings of the 7th International Symposium on Graph Drawing*, volume 1731 of *LNCS*, pages 205–216. Springer, 1999.

H. Hosobe. A modular geometric constraint solver for user interface applications. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, pages 91–100. ACM, 2001.

HTML Working Group. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). `http://www.w3.org/TR/xhtml1/`, 2002.

N. Hurst. *Better Automatic Layout of Documents*. PhD thesis, Monash University, Department of Computer Science, May 2009.

N. Hurst, K. Marriott, and P. Moulder. Towards tighter tables. In *Proceedings of Document Engineering, 2005*, pages 74–83, New York, 2005. ACM.

N. Hurst, K. Marriott, and D. Albrecht. Solving the simple continuous table layout problem. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 28–30. ACM, 2006a.

N. Hurst, K. Marriott, and P. Moulder. Minimum sized text containment shapes. In *DocEng '06: Proceedings of the 2006 ACM Symposium on Document engineering*, pages 3–12. ACM, 2006b.

N. Hurst, K. Marriott, and P. Moulder. Minimum sized text containment shapes. pages 3–12. ACM Press New York, NY, USA, 2006c.

N. Hurst, W. Li, and K. Marriott. Review of automatic document formatting. In *Proceedings of the 9th ACM symposium on Document engineering*, pages 99–108. ACM, 2009.

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 111–119. ACM, 1987. doi: 10.1145/41625.41635.

R. Joan-Arinyo and A. Soto-Riera. Combining constructive and equational geometric constraint-solving techniques. *ACM Transactions on Graphics (TOG)*, 18(1): 35–55, 1999.

M. Jourdan, N. Layaïda, C. Roisin, L. Sabry-Ismaïl, and L. Tardif. Madeus, and authoring environment for interactive multimedia documents. In *Proceedings of the sixth ACM international conference on Multimedia*, pages 267–272. ACM, 1998.

J. C. Jung, B. P., G. Katsirelos, and T. Walsh. Two encodings of DNNF theories. *ECAI Workshop on Inference Methods Based on Graphical Structures of Knowledge*, 2008.

M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1 (1):1–25, 1997.

M. Jünger, E. K. Lee, P. Mutzel, and T. Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In *Proceedings of the 5th International Symposium on Graph Drawing*, pages 13–24, 1997.

U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 167–172. AAAI Press / The MIT Press, 2004.

F. Kafka. *The Trial*. Project Gutenberg, 1925, 2005.

N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 302–311, 1984.

N. Karmarkar and K. Ramakrishnan. Computational results of an interior point algorithm for large scale linear programming. *Mathematical Programming*, 52(1): 555–586, 1991.

G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *LNCS*, pages 873–877. Springer, 2003.

G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 390–396. AAAI Press / The MIT Press, 2005.

G. Katsirelos, N. Narodytska, and T. Walsh. Reformulating global grammar constraints. volume 5547 of *LNCS*, pages 132–147. Springer, 2009.

L. G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979. English translation: *Soviet Math. Dokl.* **20**:191–194.

G. Kramer. A geometric constraint engine. *Artificial Intelligence*, 58(1–3):327–360, Dec. 1992.

M. Lagerkvist. *Techniques for Efficient Constraint Propagation*. PhD thesis, Kungliga Tekniska; Stockholm, 2008.

M. Lagerkvist and G. Pesant. Modeling irregular shape placement problems with regular constraints. In *First Workshop on Bin Packing and Placement Constraints BPCC'08*, 2008. `http://contraintes.inria.fr/CPAIOR08/BPPC.html`.

C. Lecoutre. Optimization of simple tabular reduction for table constraints. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, volume 5202 of *LNCS*, pages 128–143. Springer, 2008.

C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2006.

O. Lhomme. Consistency techniques for numeric csps. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 232–238. Morgan Kaufmann, 1993.

O. Lhomme and J. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 405–410. AAAI Press / The MIT Press, 2005.

M. Lin, D. Manocha, and J. Cohen. Collision detection: Algorithms and applications, 1996.

X. Lin. Active layout engine: Algorithms and applications in variable data printing. *Computer-Aided Design*, 38(5):444–456, 2006.

I. Maros and G. Mitra. Simplex algorithms. In J. Beasley, editor, *Advances in Linear and Integer Programming*, Oxford Lecture Series in Mathematics and Its Applications, 4. Clarendon Press, 1996.

J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

K. Marriott and S. Chok. Qoca: A constraint solving toolkit for interactive graphical applications. *Constraints*, 7(3):229–254, 2002.

K. Marriott, S. Chok, and A. Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, pages 340–354, London, UK, 1998. Springer-Verlag.

K. Marriott, N. Nethercote, R. Rafeh, P. Stuckey, M. Garcia de la Banda, and M. Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, 2000.

C. Matuszewski, R. Schönfeld, and P. Molitor. Using sifting for k-layer straightline crossing minimization. In *Proceedings of the 7th International Symposium on Graph Drawing*, volume 1731 of *LNCS*, pages 217–224. Springer, 1999.

M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference*, pages 530–535, 2001.

P. Mutzel. An alternative method to crossing minimization on hierarchical graphs. In *Proceedings of the 4th International Symposium on Graph Drawing*, volume 1190 of *LNCS*, pages 318–333. Springer, 1996.

G. Nelson. Juno, a constraint-based graphics system. In *SIGGRAPH '85 Conference Proceedings*, pages 235–243, San Francisco, July 1985. ACM.

R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *CAV*, pages 321–334, 2005.

J. Nocedal and S. Wright. *Numerical optimization*. Springer-Verlag, 1999. ISBN 0387987932.

O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

J. O'Rourke. *Computational Geometry in C*. 2nd edition, 1998.

G. Pesant. A regular language membership constraint for finite sequences of variables. In M. Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, pages 482–495. Springer-Verlag, 2004.

J. Puchinger and P. J. Stuckey. Automating branch-and-bound for dynamic programs. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 81–89. ACM, 2008.

C. Quimper and T. Walsh. Global grammar constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *LNCS*, pages 751–755, 2006.

C. Quimper and T. Walsh. Decomposing global grammar constraints. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 590–604, 2007.

D. Raggett, A. L. Hors, and I. Jacobs. HTML 4.01 Specification, section 'Autolayout Algorithm'. `http://www.w3.org/TR/html4/appendix/notes.html\verb|#|h-B.5.2`, 1999.

G. Ramkumar. An algorithm to compute the Minkowski sum outer-face of two simple polygons. *Proceedings of the Twelfth Annual Symposium on Computational geometry*, pages 234–241, 1996.

B. Randerath, E. Speckenmeyer, E. Boros, P. Hammer, A. Kogan, K. Makino, B. Simeone, and O. Cepek. A satisfiability formulation of problems on level graphs. *Electronic Notes in Discrete Mathematics*, 9:269–277, 2001.

R. Rasmussen and M. A. Trick. Round robin scheduling – a survey. *European Journal of Operational Research*, 188(3):617–636, 2008.

J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence - Volume 1*, pages 209–215. AAAI Press, 1996.

R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 183–189. Morgan Kaufmann, 1987.

S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Prentice Hall, 2^{nd} edition, 2002.

H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl*, 1(1):51–64, 1991.

M. Sellmann. The theory of grammar constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *LNCS*, pages 530–544. Springer, 2006.

J. P. M. Silva and I. Lynce. Towards robust CNF encodings of cardinality constraints. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 483–497. Springer, 2007.

M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, second edition, 2006.

Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

N. Sörensson and A. Biere. Minimizing learned clauses. In *12th International Conference on Theory and Applications of Satisfiability Testing*, volume 5584 of *LNCS*, pages 237–243. Springer, 2009.

D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004.

A. Srinivasan, T. Ham, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Proceedings of the 1990 IEEE International Conference on Computer-Aided Design*, pages 92–95, 1990.

T. Strecker and L. Hennig. Automatic layouting of personalized newspaper pages. In *Operations Research Proceedings 2008*, pages 469–474. Springer Berlin Heidelberg, 2009.

S. Subbarayan. Efficent reasoning for nogoods in constraint solvers with BDDs. In *Proceedings of Tenth International Symposium on Practical Aspects of Declarative Languages*, volume 4902 of *LNCS*, pages 53–57. Springer-Verlag, 2008.

K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst. Man Cybern.*, 11(2):109–125, 1981.

I. Sutherland. Sketchpad: A man-machine graphical communication system. In *DAC '64: Proceedings of the SHARE design automation workshop*, pages 329–346, New York, NY, USA, 1964. ACM Press.

G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, Part 2:115–125, 1968.

N. Ueda and T. Nagao. NP-completeness results for nonogram via parsimonious reductions. Technical Report TR96-0008, Department of Computer Science, Tokyo Institute of Technology, 1996.

M. van Dongen, C. Lecoutre, and O. Roussel. Third International CSP Solver Competition. [Online, accessed Sept 2010], 2008. `http://www.cril.univ-artois.fr/CPAI08/`.

B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, Jan. 1996.

B. Vander Zanden, R. Halterman, B. Myers, R. McDaniel, R. Miller, P. Szekely, D. Giuse, and D. Kosbie. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):776–796, 2001.

M. Vanhoucke and B. Maenhout. The nurse scheduling problem (NSP). [Online, accessed Oct 2010]. `http://www.projectmanagement.ugent.be/nsp.php`.

X. Wang and D. Wood. Tabular formatting problems. In *PODP '96: Proceedings of the Third International Workshop on Principles of Document Processing*, volume 1293 of *LNCS*, pages 171–181, London, UK, 1997. Springer-Verlag.

R. Wein. Exact and efficient construction of planar Minkowski sums using the convolution method. In *Proceedings of the 14th Annual European Symposium on Algorithms*, volume 4168 of *LNCS*, pages 829–840. Springer, September 2006.

J. Wolter. Comparison of solvers on the n-dom puzzles. [Online, accessed Sept 2010], a. `http://webpbn.com/survey/dom.html`.

J. Wolter. Survey of paint-by-numbers puzzle solvers. [Online, accessed Sept 2010], b. `http://webpbn.com/survey/`.

D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189 – 208, 1967.

L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285, 2001.