Essays on equilibrium computation, MDD-based constraint programming and scheduling

Samid Hoda

April 2010

Tepper School of Business
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Algorithms, Combinatorics and Optimization

Abstract

This thesis addresses three topics: solving the Nash Equilibrium problem for two-player zero-sum games presented in extensive form, constraint programming using multivalued decision diagrams and scheduling cranes in a factory.

In the first chapter, we develop a first-order method based on a smoothing technique of Nesterov that allows us to solve problems that are several orders of magnitude larger than was possible previously.

The second chapter investigates constraint programming based on multivalued decision diagrams (MDDs). We present a systematic framework for designing filtering algorithms for MDDs as well as concrete instantiations for several different global constraints. We also discuss some ideas for primal heuristics and branching schemes using MDDs. The third chapter describes our implementation of a solver for constraint satisfaction problems where the domain-store has been replaced by MDDs. In the fourth chapter we present a case study of propagating `among` constraints using our framework and provide more evidence that MDD-based propagation can result in enormous reduction in the size of the search tree and solution time.

In the final chapter of this thesis we address the problem of scheduling a pair of cranes that share a track to best follow a production schedule. We focus on the problem of solving the optimal control problem for the trajectories and present a dynamic programming solution.

Contents

List of Figures

List of Tables

Acknowledgements

I acknowledge so and so.

The dedictation

Chapter 1

Introduction

This thesis consists of three main parts. In the first part we introduce smoothing techniques for solving the Nash equilibrium problem for two-player zero-sum sequential games. Although this problem can be solved using linear programming, the resulting formulations for ‘practical’ problems are intractable. One of our goals is to solve the Nash equilibrium problems for games that arise from ‘heads-up’ poker. For example, the payoff matrix (which appears as part of the linear programming formulation) for Texas Hold’em poker has more than 10^{18} nonzero entries.

Our approach follows a current trend of applying first-order algorithms to non-smooth optimization problems. A key feature of these algorithms is their low computational cost per iteration, which makes them particularly attractive for large problems. We adapt Nesterov’s smoothing techniques for computing approximate equilibria. We also develop two heuristics that speed up the algorithm significantly and present a matrix decomposition that provides enormous memory savings. These techniques enable us to solve problems orders of magnitude larger than the prior state-of-the-art.

The next part of this thesis studies constraint programming which the domain store has been replaced by a more descriptive data structure: a multivalued decision diagram (MDD).

A key weakness of the domain store is that it transmits a limited amount of information. It cannot account for any interaction among the variables, because any solution in the Cartesian product of the variable domains is consistent with it. This restricts the ability of the domain store to pool the results of processing individual constraints and provide a global view of the problem.

Multivalued decision diagrams (MDDs) [28] generalize binary decision diagrams (BDDs) [2, 1], which have long been used for circuit design/verification [10, 32] and very recently for optimization [7, 21, 22]. The MDD for a constraint set is essentially a more compact representation of a branching tree, obtained by superimposing isomorphic subtrees. The shape of the resulting MDD depends on the order in which one branches on the variables.

A primary research issue in applying MDDs to solving CSPs is whether there exist fast and effective propagation algorithms for constraints. Until now (to the best of our knowledge) there were MDD propagation algorithms for the following constraints: (one-sided) inequality constraints [3], `alldiff` [3], equality constraints [23], and `among` constraints. The reasoning used for designing propagation algorithms for each of the constraints seemed to be ad-hoc. We present a systematic method for extending the reasoning used to propagate constraints in the traditional domain store setting to design MDD propagation algorithms. We will demonstrate the efficacy of the method by designing MDD propagation algorithms for several important classes of constraints. We also show how this technique can be used effectively to reuse traditional filtering algorithms for domain stores.

We conclude our work with MDD-based propagation by considering a case-study for problems that consist of several `among` constraints. Such models arise in employee scheduling and production sequencing problems. We show that there are substantial improvements in search time and search tree reductions. In fact, our experiments demonstrate that the amount of propagation obtained by the MDD is substantial, even for MDDs of very small width. There are huge savings in computation time for many of the more difficult problem instances that we considered. For example, to solve one specifically hard instance, the domain store needed 1,012,562 backtracks and 1684.7 seconds of computation time, while our MDD store with maximum width of four reduced this to two backtracks and 0.04 seconds of computation time.

The final chapter is a study of a crane scheduling problem in which a list of jobs is assigned to two cranes that share a track and cannot travel past each other. Given an assignment of jobs to cranes and the sequence of jobs on each crane we solve the problem of generating an optimal space-time trajectory for both cranes via dynamic programming. The natural state space of the dynamic program is intractable and we introduce two techniques that are necessary to solve the problem using our formulation. The first technique restricts trajectories to canonical trajectories without sacrificing optimality; the second technique involves a novel state space description that represents many states implicitly as a Cartesian product of intervals. The chapter concludes with some computational experiments illustrating our algorithm.

Chapter 2

Smoothing techniques for computing Nash Equilibria of Sequential Games

2.1 Introduction

The Nash equilibria of two-person, zero-sum sequential games are the solutions to

$$\min_{\mathbf{x}\in\mathcal{X}} \max_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{y}, A\mathbf{x}\rangle = \max_{\mathbf{y}\in\mathcal{Y}} \min_{\mathbf{x}\in\mathcal{X}} \langle \mathbf{y}, A\mathbf{x}\rangle \tag{2.1}$$

where $\mathcal{X}$ and $\mathcal{Y}$ are polytopes defining the players' strategies and A is the payoff matrix [60, 30, 55, 61]. When the minimizer plays a strategy $\mathbf{x} \in \mathcal{X}$ and the maximizer plays $\mathbf{y} \in \mathcal{Y}$, the expected utility to the maximizer is $\langle \mathbf{y}, A\mathbf{x}\rangle$ and, since the game is zero-sum, the minimizer's expected utility is $\langle \mathbf{y}, -A\mathbf{x}\rangle$. Problem (2.1) can be expressed as a linear program, but the resulting formulations are prohibitively large for most interesting games. For instance, the payoff matrix A in (2.1) for limit Texas Hold'em poker has dimension $10^{14} \times 10^{14}$ and contains more than 10^{18} non-zero entries. Problems of this magnitude are far beyond the capabilities of state-of-the-art general-purpose linear programming solvers. Even solving a substantially smaller game with a $10^6 \times 10^6$ payoff matrix containing 50 million non-zeros with conventional linear programming solvers is computationally demanding both in terms of time and memory [17].

We present a novel algorithmic approach for finding approximate solutions to (2.1). To this end, we define polytopes called *treeplexes* and concentrate on solving (2.1) when $\mathcal{X}$ and $\mathcal{Y}$ are polytopes of this type. Treeplexes generalize simplexes and include as a special case the strategy sets of sequential games. Our approach follows a current trend of applying first-order algorithms to non-smooth optimization problems [27, 31, 43, 46, 47]. A key feature of these algorithms is their low computational cost per iteration, which makes them particularly attractive for large problems. We adapt Nesterov's smoothing techniques [46, 47] for approximating (2.1). In particular, we develop first-order algorithms that take $\mathcal{O}(1/\epsilon)$ iterations to compute $\mathbf{x} \in \mathcal{X}$ and $\mathbf{y} \in \mathcal{Y}$ such that

$$0 \leq \max_{\mathbf{v}\in\mathcal{Y}} \langle \mathbf{v}, A\mathbf{x}\rangle - \min_{\mathbf{u}\in\mathcal{X}} \langle \mathbf{y}, A\mathbf{u}\rangle \leq \epsilon. \tag{2.2}$$

Such a pair of strategies is called an ϵ-equilibrium.

The simplicity and the low computational cost per iteration of our algorithm enables the computation of near-equilibria for enormous sequential games. An implementation based on our approach has been successful in obtaining ϵ-equilibria for sequential games where the payoff matrix A is of size $10^8 \times 10^8$ and contains more than 10^{12} entries (Section 2.6). These games are abstracted poker games with 10^8 information sets and 10^{12} leaves in the game tree. This problem size (as measured by the number of leaves) is over four orders of magnitude larger than what can be handled by solving the linear programming formulation via conventional solvers, such as interior-point methods [17, 16]. Our implementation is a key component of several successful poker-playing computer programs for full-scale Heads-Up Texas Hold'em poker [18, 19].

This chapter is organized as follows. Section 2.2 summarizes Nesterov's smoothing technique as it applies to problem (2.1). We highlight that technique's crucial ingredient, a pair of suitable *prox-functions* for the sets $\mathcal{X}$ and $\mathcal{Y}$. Section 2.3 presents our main idea, a template for constructing suitable prox-functions for treeplexes. Section 2.4 considers the special case of *uniform treeplexes*. For these treeplexes we provide explicit bounds on the number of iterations needed for finding an ϵ-equilibrium. Sections 2.5 and 2.6 present some computational experience with an implementation based on our approach. Finally, Section 2.7 summarizes the main conclusions and discusses ideas for future work.

2.2 Smoothing techniques

Problem (2.1) can be stated as

$$\min_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}) = \max_{\mathbf{y}\in\mathcal{Y}} \phi(\mathbf{y}) \tag{2.3}$$

where

$$f(\mathbf{x}) = \max_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{y}, A\mathbf{x}\rangle \ \text{ and } \ \phi(\mathbf{y}) = \min_{\mathbf{x}\in\mathcal{X}} \langle \mathbf{y}, A\mathbf{x}\rangle.$$

The functions f and ϕ are respectively convex and concave non-smooth functions. The left-hand side of (2.3) is a standard convex minimization problem of the form

$$\bar{h} := \min\{h(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\}. \tag{2.4}$$

First-order methods for solving (2.4) are algorithms for which a search direction at each iteration is obtained using only the first-order information of h, such as its gradient or subgradient. When h is smooth with Lipschitz gradient, there is a first-order algorithm for finding a point $\mathbf{x} \in \mathcal{X}$ such that $h(\mathbf{x}) \leq \bar{h} + \epsilon$ after $\mathcal{O}(1/\sqrt{\epsilon})$ iterations [44]. When h is non-smooth, subgradient algorithms can be applied, but they have a worst-case complexity of $\mathcal{O}(1/\epsilon^2)$ iterations [20]. However, that pessimistic result is based on treating h as a *black-box* where the value and subgradient are accessed via an oracle. For non-smooth functions with a suitable max structure, Nesterov devised first-order algorithms requiring only $\mathcal{O}(1/\epsilon)$ iterations by applying a clever *smoothing technique* [46, 47]. In this paper, we adapt that smoothing technique for solving problem (2.1).

The key component of Nesterov's smoothing technique is a pair of *prox-functions* for the sets $\mathcal{X}$ and $\mathcal{Y}$. These prox-functions are used to construct smooth approximations $f_\mu \approx f$ and $\phi_\mu \approx \phi$. To obtain approximate solutions to (2.3), gradient-based algorithms can then be applied to f_μ and ϕ_μ.

Definition 2.2.1. Assume $Q \subseteq \mathbb{R}^n$ is a convex compact set. A function $d : Q \to \mathbb{R}$ is a *prox-function* if it satisfies the following properties

- d is strongly convex in Q, i.e., there exists $\sigma > 0$ such that for all $\mathbf{x}, \mathbf{y} \in Q$, and $\alpha \in [0, 1]$

$$d(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha d(\mathbf{x}) + (1-\alpha)d(\mathbf{y}) - \frac{1}{2}\sigma\alpha(1-\alpha)\|\mathbf{x}-\mathbf{y}\|^2. \tag{2.5}$$

  The largest value of the constant σ that satisfies (2.5) for a particular norm $\|\cdot\|$ is the *strong convexity modulus* of d with respect to $\|\cdot\|$. Note that the specific value of the strong convexity modulus σ depends on its associated norm $\|\cdot\|$.

- $\min\{d(\mathbf{x}) : \mathbf{x} \in Q\} = 0$.

When $d : Q \to \mathbb{R}$ is differentiable, (2.5) can be equivalently stated in either of the following two forms [45]:

$$d(\mathbf{y}) \geq d(\mathbf{x}) + \langle \nabla d(\mathbf{x}), \mathbf{y} - \mathbf{x} \rangle + \frac{1}{2}\sigma\|\mathbf{x}-\mathbf{y}\|^2 \text{ for all } \mathbf{x}, \mathbf{y} \in Q. \tag{2.6}$$

$$\langle \nabla d(\mathbf{x}) - \nabla d(\mathbf{y}), \mathbf{x} - \mathbf{y} \rangle \geq \sigma \|\mathbf{x} - \mathbf{y}\|^2 \text{ for all } \mathbf{x}, \mathbf{y} \in Q. \tag{2.7}$$

Assume $d_{\mathcal{X}}$ and $d_{\mathcal{Y}}$ are prox-functions for the sets $\mathcal{X}$ and $\mathcal{Y}$ respectively. Then for any given $\mu > 0$, the smooth approximations $f_\mu \approx f$ and $\phi_\mu \approx \phi$ are

$$f_\mu(\mathbf{x}) := \max\{\langle \mathbf{x}, A\mathbf{y} \rangle - \mu d_{\mathcal{Y}}(\mathbf{y}) : \mathbf{y} \in \mathcal{Y}\}, \quad \phi_\mu(\mathbf{y}) := \min\{\langle \mathbf{x}, A\mathbf{y} \rangle + \mu d_{\mathcal{X}}(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\}.$$

The following result of Nesterov provides the theoretical foundation of our first-order algorithms for solving (2.1). Let $D_{\mathcal{X}} := \max\{d_{\mathcal{X}}(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\}$, and let $\sigma_{\mathcal{X}}$ denote the strong convexity modulus of $d_{\mathcal{X}}$. Let $D_{\mathcal{Y}}$ and $\sigma_{\mathcal{Y}}$ be defined likewise for $\mathcal{Y}$ and $d_{\mathcal{Y}}$. The operator norm of A used below is defined as $\|A\| := \max\{\langle \mathbf{y}, A\mathbf{x} \rangle \colon \|\mathbf{x}\|, \|\mathbf{y}\| \leq 1\}$, where the norms $\|\mathbf{x}\|$, $\|\mathbf{y}\|$ are those associated with $\sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{Y}}$.

Theorem 2.2.2 (Nesterov [46, 47])**.** *There is a procedure based on the above smoothing technique that after N iterations generates a pair of points $(\mathbf{x}^N, \mathbf{y}^N) \in \mathcal{X} \times \mathcal{Y}$ such that*

$$0 \leq f(\mathbf{x}^N) - \phi(\mathbf{y}^N) \leq \frac{4\,\|A\|}{N+1}\sqrt{\frac{D_{\mathcal{X}} D_{\mathcal{Y}}}{\sigma_{\mathcal{X}}\sigma_{\mathcal{Y}}}}. \tag{2.8}$$

Furthermore, each iteration of the procedure performs some elementary operations, three matrix-vector multiplications by A, and requires the exact solution of three subproblems of the form

$$\max_{\mathbf{x} \in \mathcal{X}} \{\langle \mathbf{g}, \mathbf{x} \rangle - d_{\mathcal{X}}(\mathbf{x})\} \quad \textit{or} \quad \max_{\mathbf{y} \in \mathcal{Y}} \{\langle \mathbf{g}, \mathbf{y} \rangle - d_{\mathcal{Y}}(\mathbf{y})\}. \tag{2.9}$$

In Section 2.5, we will present an explicit algorithm as stated in Theorem 2.2.2. Before that, we first provide a method for solving the subproblems in (2.9) as these are critical steps in the algorithm. These subproblems can be phrased in terms of the conjugate of the functions $d_{\mathcal{X}}$ and $d_{\mathcal{Y}}$ [25]. The conjugate of $d : Q \to \mathbb{R}$ is the function $d^* : \mathbb{R}^n \to \mathbb{R}$ defined by

$$d^*(\mathbf{s}) := \max\{\langle \mathbf{s}, \mathbf{x} \rangle - d(\mathbf{x}) : \mathbf{x} \in Q\}.$$

If d is strongly convex and Q is compact, then the conjugate d^* is Lipschitz continuous, differentiable everywhere, and

$$\nabla d^*(\mathbf{s}) = \operatorname{argmax}\{\langle \mathbf{s}, \mathbf{x} \rangle - d(\mathbf{x}) : \mathbf{x} \in Q\}.$$

(For a detailed discussion see [25].)

For an algorithm based on Theorem 2.2.2 to be practical, the subproblems (2.9) must be solvable quickly since their solution is required three times at each iteration of the algorithm. In other words, the conjugates $d_{\mathcal{X}}^*$ and $d_{\mathcal{Y}}^*$ and their gradients $\nabla d_{\mathcal{X}}^*$ and $\nabla d_{\mathcal{Y}}^*$ should be easily computable. This motivates the following definition.

Definition 2.2.3. Assume $Q \subseteq \mathbb{R}^n$ is a compact convex set. We say that $d : Q \to \mathbb{R}$ is a *nice prox-function* for Q if it satisfies the following three conditions:

(i) d is continuous and strongly convex in Q, and differentiable in the relative interior of Q.

(ii) The conjugate d^* satisfies $d^*(\mathbf{0}) = 0$.

(iii) The conjugate function d^* and its gradient ∇d^* are easily computable.

Example 1. For the k-dimensional simplex Δ_k, the entropy function $d(\mathbf{x}) = \ln k + \sum_{i=1}^k x_i \ln x_i$, and the Euclidean distance function $d(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^k (x_i - 1/k)^2$ are nice prox-functions. Indeed, for the entropy prox-function, the gradient of the conjugate $\nabla d^*(\mathbf{s})$ is given by the closed-form expression

$$\nabla_i d^*(\mathbf{s}) = \frac{e^{s_i}}{\sum_{j=1}^k e^{s_j}},\ i = 1, \ldots, k.$$

Furthermore, as discussed in [27, 46], the entropy function has strong convexity modulus equal to one for the L^1-norm $\|\mathbf{x}\| := \sum_{j=1}^k |x_j|$.

For the Euclidean prox-function, the gradient of the conjugate $\nabla d^*(\mathbf{s})$ is given by the expression

$$\nabla_i d^*(\mathbf{s}) = (s_i - \lambda)^+,\ i = 1, \ldots, k,$$

where $\lambda \in \mathbb{R}$ is such that $\sum_{j=1}^k (s_j - \lambda)^+ = 1$. This value of λ can be found in $\mathcal{O}(k \ln k)$ steps via a binary search in the sorted components of $\mathbf{s}$. Furthermore, from (2.7) it follows that the Euclidean prox-function has strong convexity modulus equal to one for the Euclidean norm $\|\mathbf{x}\| := \sqrt{\sum_{j=1}^k x_j^2}$.

2.3 Treeplexes

This section presents the essential elements of our approach. We define the class of *treeplex* polytopes and provide a generic technique for constructing nice prox-functions for treeplexes, using as building blocks any family of nice prox-functions for simplexes. This allows us to create practical first-order algorithms based on Theorem 2.2.2 for solving the saddle-point problem (2.1) over treeplexes $\mathcal{X}$ and $\mathcal{Y}$.

A treeplex can be seen as a tree whose nodes are simplexes. The tree structure endows the treeplex with a certain kind of sequential characteristic. In particular, treeplexes include the types of polytopes that arise in the computation of Nash equilibria of sequential games. The latter is an immediate consequence of the *sequence form* formulation of Nash equilibria for sequential games, as detailed in [60, 61, 30, 55].

Definition 2.3.1. The class of treeplexes is recursively defined as follows:

- *Basic sets:* Every standard simplex $\Delta_m := \left\{\mathbf{x} \in [0,1]^m : \sum_{j=1}^m x_j = 1\right\}$ is a treeplex.
- *Cartesian product:* If $Q_1, \ldots, Q_k$ are treeplexes then $Q_1 \times \cdots \times Q_k$ is a treeplex.
- *Branching:* If $P \subseteq [0,1]^p$ and $Q \subseteq [0,1]^q$ are treeplexes and $i \in \{1, \ldots, p\}$ then

$$P \boxed{i} Q := \left\{(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{p+q} : \mathbf{x} \in P,\ \mathbf{y} \in x_i \cdot Q\right\}$$

  is a treeplex.

The Branching operation in Definition 2.3.1 has the following sequential interpretation: the vector $\mathbf{x}$ is the set of "current stage" decision variables, and the vector $\mathbf{y}$ is the set of "next stage" decision variables following the i-th current decision variable x_i. Notice that a treeplex can be written in the form $\{\mathbf{x} \geq 0 : E\mathbf{x} = \mathbf{e}\}$ for some matrix E with entries in $\{-1, 0, 1\}$ and vector $\mathbf{e}$ with entries in $\{0, 1\}$, see [60, 61].

In the sequel we will often need to compare the norm of a vector $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{p+q}$ with those of $\mathbf{x} \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q$. This requires a certain compatibility of the norms in the spaces $\mathbb{R}^p, \mathbb{R}^q$, and $\mathbb{R}^{p+q}$. Henceforth, we shall make the following mild *norm-embedding assumption:*

$$\|\mathbf{x}\| = \|(\mathbf{x}, \mathbf{0})\|,\ \|\mathbf{y}\| = \|(\mathbf{0}, \mathbf{y})\|. \tag{2.10}$$

We now present our general procedure for constructing nice prox-functions for treeplexes. The construction relies on the following *dilation* operation from convex analysis [25]. Given a compact set $K \subseteq \mathbb{R}^d$ and a function $\Phi\colon K \to \mathbb{R}$, define the set $\bar{K} \subseteq \mathbb{R}^{d+1}$ as

$$\bar{K} := \left\{(x, \mathbf{y}) \in \mathbb{R}^{d+1} : x \in [0,1],\ \mathbf{y} \in x \cdot K\right\},$$

and define the function $\bar{\Phi}\colon \bar{K} \to \mathbb{R}$ as

$$\bar{\Phi}(x, \mathbf{y}) = \begin{cases} x \cdot \Phi\left(\frac{\mathbf{y}}{x}\right) & \text{if } x > 0, \\ 0 & \text{if } x = 0. \end{cases}$$

Proposition 2.3.2. *If K is compact and Φ is continuous in K, then $\bar{\Phi}$ is continuous in $\bar{K}$. Also if $(x, \mathbf{y}) \in \bar{K}$ is such that $x > 0$ and $\nabla\Phi\left(\mathbf{y}/x\right)$ exists, then $\nabla\bar{\Phi}\left(x, \mathbf{y}\right)$ exists and*

$$\begin{aligned} \nabla_x \bar{\Phi}(x, \mathbf{y}) &= \Phi\left(\tfrac{\mathbf{y}}{x}\right) - \left\langle \nabla\Phi\left(\tfrac{\mathbf{y}}{x}\right), \tfrac{\mathbf{y}}{x} \right\rangle, \\ \nabla_{\mathbf{y}} \bar{\Phi}(x, \mathbf{y}) &= \nabla\Phi\left(\tfrac{\mathbf{y}}{x}\right). \end{aligned} \tag{2.11}$$

Proof. The continuity follows via a straightforward limiting argument: Assume $(x^i, \mathbf{y}^i), (x, \mathbf{y}) \in \bar{K}$ and $(x^i, \mathbf{y}^i) \to (x, \mathbf{y})$. If $x > 0$ then $\mathbf{y}^i/x_i, \mathbf{y}/x \in K$ and $\mathbf{y}^i/x_i \to \mathbf{y}/x$. Since Φ is continuous, we get

$$\bar{\Phi}(x^i, \mathbf{y}^i) = \Phi(\mathbf{y}^i/x^i) \to \Phi(\mathbf{y}/x) = \bar{\Phi}(x, \mathbf{y}).$$

On the other hand, if $x = 0$ then $x^i \to 0$. Consequently,

$$|\bar{\Phi}(x^i, \mathbf{y}^i)| = |x^i \Phi(\mathbf{y}^i/x^i)| \leq x^i \max\{\Phi(\mathbf{z}) : \mathbf{z} \in K\} \to 0 = \bar{\Phi}(x, \mathbf{y}).$$

Finally, the identities in (2.11) follow by applying the chain rule. □

Assume we are given a family of nice prox-functions d_m for Δ_m, $m \in \mathbb{Z}^+$. Using this family, we recursively construct functions for treeplexes as follows:

- *Basic sets:* For $Q = \Delta_m$, let $d_Q := d_m$.
- *Cartesian product:* If $Q_1, \ldots, Q_k$ are treeplexes and $Q = Q_1 \times \cdots \times Q_k$, let
$$d_Q(\mathbf{x}^1, \ldots, \mathbf{x}^k) := \sum_{i=1}^{k} d_{Q_i}(\mathbf{x}^i)$$
where $d_{Q_1}, \ldots, d_{Q_k}$ are nice prox-functions for their respective treeplexes.
- *Branching:* If $P \subseteq [0,1]^p$ and $R \subseteq [0,1]^r$ are treeplexes, $i \in \{1, \ldots, p\}$, and $Q = P \boxed{i} R$, let
$$d_Q(\mathbf{x}, \mathbf{y}) := d_P(\mathbf{x}) + \bar{d}_R(x_i, \mathbf{y}) \tag{2.12}$$
where d_P and d_R are nice prox-functions for P and R.

Theorem 2.3.3. *The functions d_Q defined above are nice prox-functions for each treeplex Q.*

To prove Theorem 2.3.3, it suffices to show that the properties of nice prox-functions are preserved for the Cartesian product and Branching steps. Since the Cartesian product step is straightforward, we concentrate on the Branching step as stated in the following proposition.

Proposition 2.3.4. *Assume $P \subseteq [0,1]^p$ and $R \subseteq [0,1]^r$ are treeplexes, $i \in \{1, \ldots, p\}$, and $Q = P \boxed{i} R$. Furthermore, assume d_P and d_R are nice prox-functions for P and R respectively and*

$$d_Q(\mathbf{x}, \mathbf{y}) := d_P(\mathbf{x}) + \bar{d}_R(x_i, \mathbf{y}).$$

Then

(i) d_Q *is continuous and strongly convex in* Q *and differentiable in the relative interior of* Q.

(ii) d_Q^* *and* ∇d_Q^* *are computable via the following expressions*

$$d_Q^*(\mathbf{u},\mathbf{v}) = d_P^*(\tilde{\mathbf{u}}) \tag{2.13}$$

$$\nabla d_Q^*(\mathbf{u},\mathbf{v}) = (\nabla d_P^*(\tilde{\mathbf{u}}), \nabla_i d_P^*(\tilde{\mathbf{u}}) \cdot \nabla d_R^*(\mathbf{v})) \tag{2.14}$$

where

$$\tilde{u}_j = \begin{cases} u_j & \text{if } j \neq i, \\ u_i + d_R^*(\mathbf{v}) & \text{if } j = i. \end{cases}$$

Proof.

(i) The continuity of d_Q in Q and the differentiability in the relative interior of Q follow from (2.12) and Proposition 2.3.2. Since d_Q is continuous in Q, to prove its strong convexity, from (2.7) it suffices to show that there exists $\sigma > 0$ such that

$$\langle \nabla d_Q(\mathbf{x},\mathbf{y}) - \nabla d_Q(\tilde{\mathbf{x}},\tilde{\mathbf{y}}), (\mathbf{x},\mathbf{y}) - (\tilde{\mathbf{x}},\tilde{\mathbf{y}}) \rangle \geq \sigma \|(\mathbf{x},\mathbf{y}) - (\tilde{\mathbf{x}},\tilde{\mathbf{y}})\|^2 \tag{2.15}$$

for all $(\mathbf{x},\mathbf{y})$ and $(\tilde{\mathbf{x}},\tilde{\mathbf{y}})$ in the relative interior of Q.

Assume $(\mathbf{x},\mathbf{y})$ and $(\tilde{\mathbf{x}},\tilde{\mathbf{y}})$ are in the relative interior of Q. Set $\mathbf{z} := \mathbf{y}/x_i$ and $\tilde{\mathbf{z}} := \tilde{\mathbf{y}}/\tilde{x}_i$. From (2.12), Proposition 2.3.2, and some elementary calculations we get

$$\begin{aligned} \langle \nabla d_Q(\mathbf{x},\mathbf{y}) - \nabla d_Q(\tilde{\mathbf{x}},\tilde{\mathbf{y}}), (\mathbf{x},\mathbf{y}) - (\tilde{\mathbf{x}},\tilde{\mathbf{y}}) \rangle &= \langle \nabla d_P(\mathbf{x}) - \nabla d_P(\tilde{\mathbf{x}}), \mathbf{x} - \tilde{\mathbf{x}} \rangle \\ &+ x_i \cdot (d_R(\mathbf{z}) - d_R(\tilde{\mathbf{z}}) + \langle \nabla d_R(\tilde{\mathbf{z}}), \tilde{\mathbf{z}} - \mathbf{z} \rangle) \\ &+ \tilde{x}_i \cdot (d_R(\tilde{\mathbf{z}}) - d_R(\mathbf{z}) + \langle \nabla d_R(\mathbf{z}), \mathbf{z} - \tilde{\mathbf{z}} \rangle) \,. \end{aligned}$$

Therefore, since d_P and d_R are strongly convex, (2.6) yields

$$\begin{aligned} \langle \nabla d_Q(\mathbf{x},\mathbf{y}) - \nabla d_Q(\tilde{\mathbf{x}},\tilde{\mathbf{y}}), (\mathbf{x},\mathbf{y}) - (\tilde{\mathbf{x}},\tilde{\mathbf{y}}) \rangle &\geq \sigma_P \|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \tfrac{1}{2}\sigma_R x_i \|\mathbf{z} - \tilde{\mathbf{z}}\|^2 + \tfrac{1}{2}\sigma_R \tilde{x}_i \|\mathbf{z} - \tilde{\mathbf{z}}\|^2 \\ &= \sigma_P \|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \sigma_R \hat{x}_i \|\mathbf{z} - \tilde{\mathbf{z}}\|^2, \end{aligned} \tag{2.16}$$

where $\hat{x}_i = \frac{x_i + \tilde{x}_i}{2}$ and $\sigma_P, \sigma_R > 0$ are the strong convexity parameters of d_P and d_R respectively.

Next, we bound the right-hand side of (2.15). Applying the triangle inequality and using the norm-embedding assumption (2.10), we get

$$\begin{aligned} \|(\mathbf{x},\mathbf{y}) - (\tilde{\mathbf{x}},\tilde{\mathbf{y}})\| &\leq \|\mathbf{x} - \tilde{\mathbf{x}}\| + \|x_i \mathbf{z} - \tilde{x}_i \tilde{\mathbf{z}}\| \\ &= \|\mathbf{x} - \tilde{\mathbf{x}}\| + \|\tfrac{1}{2}(x_i + \tilde{x}_i)(\mathbf{z} - \tilde{\mathbf{z}}) + \tfrac{1}{2}(x_i - \tilde{x}_i)(\mathbf{z} + \tilde{\mathbf{z}})\| \\ &\leq \|\mathbf{x} - \tilde{\mathbf{x}}\| + \hat{x}_i \|\mathbf{z} - \tilde{\mathbf{z}}\| + \tfrac{1}{2}|x_i - \tilde{x}_i| \|\mathbf{z} + \tilde{\mathbf{z}}\|. \end{aligned} \tag{2.17}$$

Since R is compact, the value $M := \max\{\|\boldsymbol{\zeta}\| : \boldsymbol{\zeta} \in R\}$ is finite. Therefore from (2.17) we get

$$\|(\mathbf{x},\mathbf{y}) - (\tilde{\mathbf{x}},\tilde{\mathbf{y}})\| \leq (1+M)\|\mathbf{x}-\tilde{\mathbf{x}}\| + \hat{x}_i\|\mathbf{z}-\tilde{\mathbf{z}}\|.$$

Now, by the Cauchy-Schwarz inequality,

$$\|(\mathbf{x},\mathbf{y}) - (\tilde{\mathbf{x}},\tilde{\mathbf{y}})\|^2 \leq \left((1+M)^2\frac{1}{\sigma_P} + \frac{\hat{x}_i}{\sigma_R}\right)\left(\sigma_P\|\mathbf{x}-\tilde{\mathbf{x}}\|^2 + \sigma_R\hat{x}_i\|\mathbf{z}-\tilde{\mathbf{z}}\|^2\right).$$

Since $\mathbf{x}, \tilde{\mathbf{x}} \in P \subseteq [0,1]^p$ we get

$$\|(\mathbf{x},\mathbf{y}) - (\tilde{\mathbf{x}},\tilde{\mathbf{y}})\|^2 \leq \left((1+M)^2\frac{1}{\sigma_P} + \frac{1}{\sigma_R}\right)\left(\sigma_P\|\mathbf{x}-\tilde{\mathbf{x}}\|^2 + \sigma_R\hat{x}_i\|\mathbf{z}-\tilde{\mathbf{z}}\|^2\right). \tag{2.18}$$

From (2.16) and (2.18) it follows that (2.15) holds for

$$\sigma = \frac{1}{\frac{(1+M)^2}{\sigma_P} + \frac{1}{\sigma_R}} > 0.$$

(ii) For a given vector $(\mathbf{u},\mathbf{v}) \in \mathbb{R}^{p+r}$ we have

$$\begin{aligned}
d_Q^*(\mathbf{u},\mathbf{v}) &= \sup\{\langle(\mathbf{u},\mathbf{v}),(\mathbf{x},\mathbf{y})\rangle - d_Q(\mathbf{x},\mathbf{y}) : (\mathbf{x},\mathbf{y}) \in Q\}\\
&= \sup\{\langle\mathbf{u},\mathbf{x}\rangle + \langle\mathbf{v},\mathbf{y}\rangle - d_P(\mathbf{x}) - \bar{d}_R(x_i,\mathbf{y}) : \mathbf{x} \in P,\ \mathbf{y} \in x_i \cdot R\}\\
&= \sup\{\langle\mathbf{u},\mathbf{x}\rangle - d_P(\mathbf{x}) + x_i \cdot (\langle\mathbf{v},\mathbf{z}\rangle - d_R(\mathbf{z})) : \mathbf{x} \in P,\ \mathbf{z} \in R,\ x_i > 0\}\\
&= \sup\{\langle\mathbf{u},\mathbf{x}\rangle - d_P(\mathbf{x}) + x_i \cdot d_R^*(\mathbf{z}) : \mathbf{x} \in P\}\\
&= \sup\{\langle\tilde{\mathbf{u}},\mathbf{x}\rangle - d_P(\mathbf{x}) : \mathbf{x} \in P\}\\
&= d_P^*(\tilde{\mathbf{u}}).
\end{aligned} \tag{2.19}$$

The third and fourth steps above hold by the continuity of $\bar{d}_R$ and d_P. Hence (2.13) is proven. To prove (2.14), observe that the maximizer in the second to last step in (2.19) is $\bar{\mathbf{x}} = \nabla d_P^*(\tilde{\mathbf{u}})$. Next, consider two cases depending on the value of $\bar{x}_i$. If $\bar{x}_i > 0$ then the maximizer in the third step in (2.19) is $\bar{\mathbf{z}} = \nabla d_R^*(\tilde{\mathbf{v}})$, and consequently the maximizer in the first step in (2.19) is $(\bar{\mathbf{x}}, \bar{x}_i \cdot \bar{\mathbf{z}})$. If $\bar{x}_i = 0$ then the maximizer in the first step in (2.19) is $(\bar{\mathbf{x}}, \mathbf{0})$. In either case the maximizer in the first step in (2.19) is $\nabla d_Q^*(\mathbf{u},\mathbf{v}) = (\bar{\mathbf{x}}, \bar{x}_i \cdot \bar{\mathbf{z}}) = (\nabla d_P^*(\tilde{\mathbf{u}}), \nabla_i d_P^*(\tilde{\mathbf{u}}) \cdot \nabla d_R^*(\mathbf{v}))$. □

Remark 2.3.5. We can generalize the above construction and results to weighted versions of the prox-functions. More precisely, in the Branching step, we can define $d_Q(\mathbf{x},\mathbf{y}) := w_P d_P(\mathbf{x}) + w_R \bar{d}_R(x_i,\mathbf{y})$ for some constants $w_P, w_R > 0$. We will elaborate on this idea to obtain prox-functions yielding better complexity guarantees for uniform treeplexes.

2.4 Uniform treeplexes

In this section we derive complexity results for first-order smoothing algorithms for the problem (2.1) in the special case when $\mathcal{X}$ and $\mathcal{Y}$ are *uniform* treeplexes. This special case of (2.1) covers the formulation of Nash equilibrium for instances of many interesting games. Indeed, as will be discussed in Section 2.6, uniform treeplexes naturally arise in multi-round sequential games such as poker.

Definition 2.4.1. Assume that a treeplex $Q \subseteq [0,1]^q$, an index set $I = \{i_1, \ldots, i_b\} \subseteq \{1, \ldots, q\}$, and a positive integer k are given. Define $Q_r,\ r = 1, 2, \ldots,$ as follows

- $Q_1 := Q \times \cdots \times Q$ (k times).
- $Q_{r+1} := \hat{Q}_r \times \cdots \times \hat{Q}_r$ (k times), where

$$\hat{Q}_r := Q \boxed{I} Q_r := \{(\mathbf{x}, \mathbf{y}^1, \ldots, \mathbf{y}^b) : \mathbf{x} \in Q,\ \mathbf{y}^j \in x_{i_j} \cdot Q_r,\ j = 1, \ldots, b\}.$$

We will refer to Q_r as the r-th uniform treeplex generated by Q, I, k and will sometimes write it as $\mathcal{Q}(Q, I, k, r)$.

Remark 2.4.2. Notice that the operation $\boxed{I}$ is the same as the operation $\boxed{i}$ applied b times. More precisely,

$$Q \boxed{I} Q_r = Q \boxed{i_1} Q_r \boxed{i_2} \cdots \boxed{i_b} Q_r.$$

Given a nice prox-function d_Q for Q and constants $w_r > 0,\ r = 1, 2, \ldots,$ consider the following weighted version of our previous construction of prox-functions for treeplexes.

- For $Q_1 = Q \times \cdots \times Q$ (k times) let

$$d_{Q_1}(\mathbf{x}^1, \ldots, \mathbf{x}^k) := \sum_{j=1}^{k} d_Q(\mathbf{x}^j)$$

- For $Q_{r+1} = \hat{Q}_r \times \cdots \times \hat{Q}_r$ (k times), let

$$d_{Q_{r+1}}(\mathbf{u}^1, \ldots, \mathbf{u}^k) := \sum_{j=1}^{k} d_{\hat{Q}_r}(\mathbf{u}^j),$$

  where $d_{\hat{Q}_r}$ is defined as follows

$$d_{\hat{Q}_r}(\mathbf{x}, \mathbf{y}^1, \ldots, \mathbf{y}^b) := w_r \cdot d_Q(\mathbf{x}) + \sum_{j=1}^{b} \bar{d}_{Q_r}(x_{i_j}, \mathbf{y}).$$

We now present an explicit iteration complexity bound for a first-order smoothing algorithm for the saddle-point problem (2.1), when $\mathcal{X}$ and $\mathcal{Y}$ are uniform treeplexes. As in Theorem 2.2.2, the norm of A, $\|A\|$ is the induced operator norm of A, where the underlying norms are those associated with σ_Q and $\sigma_{\tilde{Q}}$. In particular, the result below holds for any choice of norms.

Theorem 2.4.3. *Suppose A, $\mathcal{X}$, $\mathcal{Y}$, $d_{\mathcal{X}}$, and $d_{\mathcal{Y}}$ satisfy the following conditions:*

(i) $\mathcal{X} = \mathcal{Q}(Q, I, k, r) \subseteq \mathbb{R}^m$ *and* $\mathcal{Y} = \mathcal{Q}(\tilde{Q}, \tilde{I}, \tilde{k}, \tilde{r}) \subseteq \mathbb{R}^n$.

(ii) The prox-functions $d_{\mathcal{X}}, d_{\mathcal{Y}}$ are constructed as above with weights $w_j = (kM)^2(bk)^j$, $j = 1, \ldots, r-1$ and $\tilde{w}_j = (\tilde{k}\tilde{M})^2(\tilde{b}\tilde{k})^j$, $j = 1, \ldots, \tilde{r}-1$ respectively, where $b = |I|$, $\tilde{b} = |\tilde{I}|$, $M := \max\{\|\mathbf{u}\| : \mathbf{u} \in Q\}$, $\tilde{M} := \max\{\|\mathbf{u}\| : \mathbf{u} \in \tilde{Q}\}$.

Then after N iterations the procedure from Theorem 2.2.2 yields $(\mathbf{x}, \mathbf{y}) \in \mathcal{X} \times \mathcal{Y}$ such that

$$0 \leq f(\mathbf{x}) - \phi(\mathbf{y}) = \max_{\mathbf{v}\in\mathcal{Y}} \langle \mathbf{v}, A\mathbf{x}\rangle - \min_{\mathbf{u}\in\mathcal{X}} \langle \mathbf{y}, A\mathbf{u}\rangle \leq \frac{4\|A\|G}{N+1}\sqrt{\frac{D_Q D_{\tilde{Q}}}{\sigma_Q \sigma_{\tilde{Q}}}}, \tag{2.20}$$

where $G = mn(kMr)(\tilde{k}\tilde{M}\tilde{r})$.

The crux of the proof of Theorem 2.4.3 is Lemma 2.4.4, which bounds the ratio of the maximum value to the strong convexity modulus for the prox-functions for uniform treeplexes. This ratio can be seen as a measure of the prox-function's quality. Lemma 2.4.4 provides an estimate of this ratio for the prox-functions d_{Q_r} constructed above, provided the weights w_r are chosen judiciously.

Lemma 2.4.4. *Assume Q and Q_r, $r = 1, 2, \ldots$, are as in Definition 2.4.1. Let σ, σ_r, D, D_r, and M be defined as follows*

$$\sigma := \text{strong convexity modulus of } d_Q, \ \sigma_r := \text{strong convexity modulus of } d_{Q_r},$$

$$D := \max\{d_Q(\mathbf{z}) : \mathbf{z} \in Q\}, \ D_r := \max\{d_{Q_r}(\mathbf{z}) : \mathbf{z} \in Q_r\},$$

$$M := \max\{\|\mathbf{z}\| : \mathbf{z} \in Q\}, \ M_r := \max\{\|\mathbf{z}\| : \mathbf{z} \in Q_r\}.$$

(i) The strong convexity moduli σ_r of d_{Q_r}, $r = 1, 2, \ldots$ satisfy

$$\sigma_{r+1} \geq \frac{1}{\frac{k(1+M_r)^2}{w_r \sigma} + \frac{bk}{\sigma_r}}. \tag{2.21}$$

(ii) If $w_r = (kM)^2(bk)^r,\ r = 1, 2, \ldots$ *then*

$$\frac{D_r}{\sigma_r} \leq b^{2r-2}k^{2r+2}r^2M^2\frac{D}{\sigma}. \tag{2.22}$$

Proof.

(i) Let $\hat{\sigma}_r$ be the strong convexity modulus of $d_{\hat{Q}_r}$. From the construction of d_{Q_r}, it follows that $\sigma_{r+1} \geq \hat{\sigma}_r/k$. Hence it suffices to bound $\hat{\sigma}_r$. Proceeding as in the proof of Proposition 2.3.4(i), it follows that for all $\mathbf{w} = (\mathbf{x}, \mathbf{y}^1, \ldots, \mathbf{y}^b)$ and $\tilde{\mathbf{w}} = (\tilde{\mathbf{x}}, \tilde{\mathbf{y}}^1, \ldots, \tilde{\mathbf{y}}^b)$ in the relative interior of $\hat{Q}_r$ we have

$$\langle \nabla d_{\hat{Q}_r}(\mathbf{w}) - \nabla d_{\hat{Q}_r}(\tilde{\mathbf{w}}), \mathbf{w} - \tilde{\mathbf{w}} \rangle \geq w_r\sigma\|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \sigma_r \sum_{j=1}^{b} \hat{x}_{i_j}\|\mathbf{z}^j - \tilde{\mathbf{z}}^j\|^2 \tag{2.23}$$

and

$$\|\mathbf{w} - \tilde{\mathbf{w}}\| \leq (1 + M_r)\|\mathbf{x} - \tilde{\mathbf{x}}\| + \sum_{j=1}^{b} \hat{x}_{i_j}\|\mathbf{z}^j - \tilde{\mathbf{z}}^j\|, \tag{2.24}$$

where $\mathbf{z}^j = \mathbf{y}^j/x_{i_j}$, and $\tilde{\mathbf{z}}^j = \tilde{\mathbf{y}}^j/\tilde{x}_{i_j}$ for $j = 1, \ldots, b$. Applying the Cauchy-Schwarz inequality to (2.24) we get

$$\begin{aligned} \|\mathbf{w} - \tilde{\mathbf{w}}\|^2 &\leq \left(\frac{(1 + M_r)^2}{w_r\sigma} + \frac{\sum_{j=1}^{b} \hat{x}_{i_j}}{\sigma_r}\right)\left(w_r\sigma\|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \sigma_r \sum_{j=1}^{b} \hat{x}_{i_j}\|\mathbf{z}^j - \tilde{\mathbf{z}}^j\|^2\right) \\ &\leq \left(\frac{(1 + M_r)^2}{w_r\sigma} + \frac{b}{\sigma_r}\right)\left(w_r\sigma\|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \sigma_r \sum_{j=1}^{b} \hat{x}_{i_j}\|\mathbf{z}^j - \tilde{\mathbf{z}}^j\|^2\right). \end{aligned} \tag{2.25}$$

From (2.23), (2.25), and the continuity of $d_{\hat{Q}_r}$ we obtain

$$\hat{\sigma}_r \geq \frac{1}{\frac{(1+M_r)^2}{w_r\sigma} + \frac{b}{\sigma_r}},$$

which yields (2.21) since $\sigma_{r+1} \geq \hat{\sigma}_r/k$.

(ii) Let $M_r := \max\{\|\mathbf{z}\| : \mathbf{z} \in Q_r\}$. We have $M_1 \leq kM$ and $M_{r+1} \leq k(M + bM_r)$, so

$$1 + M_r \leq kM(bk)^r,\ r = 1, 2, \ldots.$$

Hence $w_r \geq \frac{(1+M_r)^2}{(bk)^r}$, and consequently (2.21) yields

$$\frac{1}{(bk)^{r+1}\sigma_{r+1}} \leq \frac{1}{b\sigma} + \frac{1}{(bk)^r\sigma_r}.$$

Therefore, since $\sigma_1 \geq \sigma/k$, it follows that

$$\frac{1}{(bk)^r\sigma_r} \leq \frac{r}{b\sigma},\ r = 1, 2, \ldots. \tag{2.26}$$

On the other hand, from the construction of Q_r and d_{Q_r} we have

$$D_1 \leq kD,\ D_{r+1} \leq k(w_r D + bD_r),\ r = 1, 2, \ldots$$

so,

$$D_r \leq kD \left((bk)^{r-1} + \sum_{j=1}^{r-1} w_j (bk)^{r-1-j} \right).$$

Thus

$$\begin{aligned} D_r &\leq kD \left((bk)^{r-1} + \textstyle\sum_{j=1}^{r-1} w_j (bk)^{r-1-j} \right) \\ &= kD \left((bk)^{r-1} + (kM)^2 \textstyle\sum_{j=1}^{r-1} (bk)^j (bk)^{r-1-j} \right) \\ &= kD(1 + (kM)^2(r-1))(bk)^{r-1} \\ &\leq krD(kM)^2(bk)^{r-1}. \end{aligned} \tag{2.27}$$

Finally, (2.22) follows by putting together (2.26) and (2.27).

□

Proof of Theorem 2.4.3. Since $\mathcal{X} = \mathcal{Q}(Q, I, k, r) \subseteq \mathbb{R}^m$, Lemma 2.4.4 yields

$$\frac{D_{\mathcal{X}}}{\sigma_{\mathcal{X}}} \leq b^{2r-2} k^{2r+2} r^2 M^2 \frac{D_Q}{\sigma_Q}.$$

In addition, a simple induction argument shows the dimension m of $\mathcal{X} = \mathcal{Q}(Q, I, k, r)$ satisfies $m = kq \cdot \frac{(bk)^r - 1}{bk-1}$. Therefore

$$\frac{D_{\mathcal{X}}}{\sigma_{\mathcal{X}}} \leq m^2 k^2 r^2 M^2 \frac{D_Q}{\sigma_Q}. \tag{2.28}$$

Similarly,

$$\frac{D_{\mathcal{Y}}}{\sigma_{\mathcal{Y}}} \leq n^2 \tilde{k}^2 \tilde{r}^2 \tilde{M}^2 \frac{D_{\tilde{Q}}}{\sigma_{\tilde{Q}}}. \tag{2.29}$$

The iteration bound (2.20) now follows from (2.8), (2.28), and (2.29). □

For the special case when the norm in $\mathbb{R}^q$ and each $\mathbb{R}^{q_r}$ is the Euclidean norm, we can sharpen the bound in Lemma 2.4.4, and thus also the bound in Theorem 2.4.3.

Lemma 2.4.5. *Assume b, M, D, D_r, σ, and σ_r, are as in Lemma 2.4.4, and the norm in $\mathbb{R}^q$ and each $\mathbb{R}^{q_r}$ is the Euclidean norm. If $w_r = kM^2k^r,\ r = 1, 2, \ldots$, then*

$$\frac{D_r}{\sigma_r} \leq b^{2r-2} k^{r+1} r^2 M^2 \frac{D}{\sigma}. \tag{2.30}$$

Proof. For the Euclidean norm we have $\sigma_{r+1} = \hat{\sigma}_r$, where $\hat{\sigma}_r$ is the strong convexity modulus of $d_{\hat{Q}_r}$. Next, we proceed to bound $\hat{\sigma}_r$ as in the proof of Lemma 2.4.4. For all $\mathbf{w} = (\mathbf{x}, \mathbf{y}^1, \dots, \mathbf{y}^b)$ and $\tilde{\mathbf{w}} = (\tilde{\mathbf{x}}, \tilde{\mathbf{y}}^1, \dots, \tilde{\mathbf{y}}^b)$ in the relative interior of $\hat{Q}_r$ the inequality (2.23) holds. Next, instead of (2.24) we can use

$$\begin{aligned}
\|\mathbf{w} - \tilde{\mathbf{w}}\|^2 &= \|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \sum_{j=1}^{b} \|x_{i_j}\mathbf{z}^j - \tilde{x}_{i_j}\tilde{\mathbf{z}}^j\|^2 \\
&\leq \|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \sum_{j=1}^{b} \left(|x_{i_j} - \tilde{x}_{i_j}|M_r + \hat{x}_{i_j}\|\mathbf{z}^j - \tilde{\mathbf{z}}^j\|\right)^2.
\end{aligned}$$

Hence, by the Cauchy-Schwarz inequality, we get

$$\begin{aligned}
\|\mathbf{w} - \tilde{\mathbf{w}}\|^2 &\leq \|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \left(\frac{M_r^2}{w_r\sigma} + \frac{b}{\sigma_r}\right)\left(w_r\sigma\|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \sigma_r\sum_{j=1}^{b}\hat{x}_{i_j}\|\mathbf{z}^j - \tilde{\mathbf{z}}^j\|^2\right) \\
&\leq \left(\frac{(1 + M_r^2)}{w_r\sigma} + \frac{b}{\sigma_r}\right)\left(w_r\sigma\|\mathbf{x} - \tilde{\mathbf{x}}\|^2 + \sigma_r\sum_{j=1}^{b}\hat{x}_{i_j}\|\mathbf{z}^j - \tilde{\mathbf{z}}^j\|^2\right).
\end{aligned} \tag{2.31}$$

Thus, the bound in Lemma 2.4.4 can be sharpened to

$$\sigma_{r+1} = \hat{\sigma}_r \geq \frac{1}{\frac{1+M_r^2}{w_r\sigma} + \frac{b}{\sigma_r}}. \tag{2.32}$$

Furthermore, in this case $M_1^2 = kM^2$ and $M_{r+1}^2 \leq k(M^2 + bM_r^2)$ which implies

$$1 + M_r^2 \leq kM^2(bk)^r.$$

Hence $w_r \geq \frac{1+M_r^2}{b^r}$, and consequently (2.32) yields

$$\frac{1}{b^{r+1}\sigma_{r+1}} \leq \frac{1}{b\sigma} + \frac{1}{b^r\sigma_r}.$$

Therefore, since $\sigma_1 = \sigma$, it follows that

$$\frac{1}{b^r\sigma_r} \leq \frac{r}{b\sigma},\ r = 1, 2, \dots. \tag{2.33}$$

On the other hand, since $D_1 = kD$ and $D_{r+1} \leq k(w_rD + bD_r)$, it follows that

$$\begin{aligned}
D_r &\leq kD\left((bk)^{r-1} + \textstyle\sum_{j=1}^{r-1} w_j(bk)^{r-1-j}\right) \\
&= kD\left((bk)^{r-1} + kM^2\textstyle\sum_{j=1}^{r-1} k^j(bk)^{r-1-j}\right) \\
&\leq kD(1 + kM^2(r-1))(bk)^{r-1} \\
&\leq k^2rDM^2(bk)^{r-1}.
\end{aligned} \tag{2.34}$$

Finally (2.30) follows by putting together (2.33) and (2.34). □

2.5 Implementation

In this section we describe an implementation to solve (2.1) based on Nesterov's *excessive gap technique* [46] and the prox-functions constructed in this paper. We present Nesterov's algorithm specialized for the problem (2.1). We also give a complexity analysis of each iteration of this algorithm when applied to games with uniform treeplexes and describe two heuristics that were incorporated in our implementation.

2.5.1 Nesterov's Excessive Gap Technique

Assume $d_{\mathcal{X}}$ and $d_{\mathcal{Y}}$ are nice prox functions for $\mathcal{X}$ and $\mathcal{Y}$ respectively. For $\mu_{\mathcal{X}}, \mu_{\mathcal{Y}} > 0$ consider the pair of problems:

$$f_{\mu_{\mathcal{Y}}}(\mathbf{x}) := \max\{\langle \mathbf{y}, A\mathbf{x}\rangle - \mu_{\mathcal{Y}} d_{\mathcal{Y}}(\mathbf{y}) : \mathbf{y} \in \mathcal{Y}\}, \;\; \phi_{\mu_{\mathcal{X}}}(\mathbf{y}) := \min\{\langle \mathbf{y}, A\mathbf{x}\rangle + \mu_{\mathcal{X}} d_Q(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\}.$$

Algorithm 3 below, due to Nesterov [46, Section 5], generates iterates $(\mathbf{x}^k, \mathbf{y}^k, \mu_{\mathcal{X}}^k, \mu_{\mathcal{Y}}^k)$ with $\mu_{\mathcal{X}}^k$, $\mu_{\mathcal{Y}}^k$ decreasing to zero and such that the following *excessive gap condition* is satisfied at each iteration:

$$f_{\mu_{\mathcal{Y}}}(\mathbf{x}) \leq \phi_{\mu_{\mathcal{X}}}(\mathbf{y}). \tag{2.35}$$

Notice that $f(\mathbf{x}) \geq \phi(\mathbf{y})$ for all $\mathbf{x} \in \mathcal{X}, \mathbf{y} \in \mathcal{Y}$. Thus if $(\mathbf{x}, \mathbf{y}, \mu_{\mathcal{X}}, \mu_{\mathcal{Y}})$ satisfy the excessive gap condition (2.35) and $\mathbf{x} \in \mathcal{X},\; \mathbf{y} \in \mathcal{Y}$, then

$$0 \leq \phi(\mathbf{y}) - f(\mathbf{x}) \leq \mu_{\mathcal{X}} D_{\mathcal{X}} + \mu_{\mathcal{Y}} D_{\mathcal{Y}}. \tag{2.36}$$

(See [46, Lemma 3.1].)

Consequently, if the iterates $(\mathbf{x}^k, \mathbf{y}^k, \mu_{\mathcal{X}}^k, \mu_{\mathcal{Y}}^k)$ satisfy (2.35), then $f(\mathbf{x}^k) \approx \phi(\mathbf{y}^k)$ when $\mu_{\mathcal{X}}^k$ and $\mu_{\mathcal{Y}}^k$ are small.

The building blocks of our Algorithm 3 are the procedures `initial` and `shrink` defined next.

By Lemma 5.1 of [46], the following procedure `initial` finds a starting point $(\mu_{\mathcal{X}}^0, \mu_{\mathcal{Y}}^0, \mathbf{x}^0, \mathbf{y}^0)$ that satisfies the excessive gap condition (2.35).

Algorithm 1. `initial`$(A, d_{\mathcal{X}}, d_{\mathcal{Y}})$

```
1. $\mu_{\mathcal{X}}^0 := \mu_{\mathcal{Y}}^0 := \frac{\|A\|}{\sqrt{\sigma_{\mathcal{X}}\sigma_{\mathcal{Y}}}}$

2. $\hat{\mathbf{x}} := \nabla d_{\mathcal{X}}^*(\mathbf{0})$
```

```
3. $\mathbf{y}^0 := \nabla d_{\mathcal{Y}}^* \left( \frac{1}{\mu_{\mathcal{Y}}^0} A\hat{\mathbf{x}} \right)$
4. $\mathbf{x}^0 := \nabla d_{\mathcal{X}}^* \left( \nabla d_{\mathcal{X}}(\hat{\mathbf{x}}) + \frac{1}{\mu_{\mathcal{X}}^0} A^{\mathrm{T}} \mathbf{y}^0 \right)$
5. Return $(\mu_{\mathcal{X}}^0, \mu_{\mathcal{Y}}^0, \mathbf{x}^0, \mathbf{y}^0)$
```

The following procedure `shrink` enables us to reduce $\mu_{\mathcal{X}}$ and $\mu_{\mathcal{Y}}$ while maintaining (2.35).

Algorithm 2. `shrink`$(A, \mu_{\mathcal{X}}, \mu_{\mathcal{Y}}, \tau, \mathbf{x}, \mathbf{y}, d_{\mathcal{X}}, d_{\mathcal{Y}})$

```
1. $\breve{\mathbf{x}} := \nabla d_{\mathcal{X}}^* \left( -\frac{1}{\mu_{\mathcal{X}}} A^{\mathrm{T}} \mathbf{y} \right)$
2. $\hat{\mathbf{x}} := (1-\tau)\mathbf{x} + \tau \breve{\mathbf{x}}$
3. $\hat{\mathbf{y}} := \nabla d_{\mathcal{Y}}^* \left( \frac{1}{\mu_{\mathcal{Y}}} A\hat{\mathbf{x}} \right)$
4. $\tilde{\mathbf{x}} := \nabla d_{\mathcal{X}}^* \left( \nabla d_{\mathcal{X}}(\breve{\mathbf{x}}) - \frac{\tau}{(1-\tau)\mu_{\mathcal{X}}} A^{\mathrm{T}} \hat{\mathbf{y}} \right)$
5. $\mathbf{y}^+ := (1-\tau)\mathbf{y} + \tau \hat{\mathbf{y}}$
6. $\mathbf{x}^+ := (1-\tau)\mathbf{x} + \tau \tilde{\mathbf{x}}$
7. $\mu_{\mathcal{X}}^+ := (1-\tau)\mu_{\mathcal{X}}$
8. Return $(\mu_{\mathcal{X}}^+, \mathbf{x}^+, \mathbf{y}^+)$
```

By Theorem 5.2 of [46], if the input $(\mu_{\mathcal{X}}, \mu_{\mathcal{Y}}, \mathbf{x}, \mathbf{y})$ to `shrink` satisfies (2.35) then so does $(\mu_{\mathcal{X}}^+, \mu_{\mathcal{Y}}, \mathbf{x}^+, \mathbf{y}^+)$ as long as τ satisfies $\tau^2/(1-\tau) \leq \mu_{\mathcal{X}}\mu_{\mathcal{Y}}\sigma_{\mathcal{X}}\sigma_{\mathcal{Y}}/\|A\|^2$.

We are now ready to describe Nesterov's Excessive Gap Technique Algorithm (EGT) specialized to (2.1).

Algorithm 3. `EGT`$(A, d_{\mathcal{X}}, d_{\mathcal{Y}})$

```
1. $(\mu_{\mathcal{X}}^0, \mu_{\mathcal{Y}}^0, \mathbf{x}^0, \mathbf{y}^0) = \texttt{initial}(A, d_{\mathcal{X}}, d_{\mathcal{Y}})$
2. For $k = 0, 1, \ldots$:
    (a) $\tau := \frac{2}{k+3}$
    (b) If $k$ is even:    // shrink $\mu_{\mathcal{X}}$
```

```
            i. $(\mu_{\mathcal{X}}^{k+1}, \mathbf{x}^{k+1}, \mathbf{y}^{k+1}) := \mathtt{shrink}(A, \mu_{\mathcal{X}}^k, \mu_{\mathcal{Y}}^k, \tau, \mathbf{x}^k, \mathbf{y}^k, d_{\mathcal{X}}, d_{\mathcal{Y}})$
            ii. $\mu_{\mathcal{X}}^{k+1} := \mu_{\mathcal{X}}^k$
        (c) If $k$ is odd:    // shrink $\mu_{\mathcal{Y}}$
            i. $(\mu_{\mathcal{Y}}^{k+1}, \mathbf{y}^{k+1}, \mathbf{x}^{k+1}) := \mathtt{shrink}(-A^{\mathrm{T}}, \mu_{\mathcal{Y}}^k, \mu_{\mathcal{X}}^k, \tau, \mathbf{y}^k, \mathbf{x}^k, d_{\mathcal{Y}}, d_{\mathcal{X}})$
            ii. $\mu_{\mathcal{Y}}^{k+1} := \mu_{\mathcal{Y}}^k$
```

By [46, Theorem 5.2], the iterates generated by procedure `EGT` satisfy (2.35). In addition, by [46, Theorem 6.3], after N iterations, Algorithm `EGT` yields points $\mathbf{x}^N \in Q_{\mathcal{X}}$ and $\mathbf{y}^N \in Q_{\mathcal{Y}}$ with

$$0 \le \max_{\mathbf{x} \in Q_{\mathcal{X}}} \langle A\mathbf{y}^N, \mathbf{x} \rangle - \min_{\mathbf{y} \in Q_{\mathcal{Y}}} \langle A\mathbf{y}, \mathbf{x}^N \rangle \le \frac{4\,\|A\|}{N} \sqrt{\frac{D_{\mathcal{X}} D_{\mathcal{Y}}}{\sigma_{\mathcal{X}} \sigma_{\mathcal{Y}}}}. \tag{2.37}$$

2.5.2 Complexity of each EGT iteration

We next give a complexity bound on the number of arithmetic operations performed in each EGT iteration. We provide our estimate in term of the size of the *game tree* in the *extensive form* representation of the sequential game. The extensive form is a full description of the game given by a tree whose nodes correspond to the possible states of the game, branches that correspond to players' moves, payoffs at the tree's leaves, and information sets. For a detailed exposition on the extensive form representation, see, e.g., [51].

We shall refer to the number of nodes in the game tree as the *size of the game tree.* We show next that for games with uniform treeplexes the total number of basic arithmetic operations in each EGT iteration is linear in the size of the game tree. To that end, notice that aside from negligible updates, two consecutive iterations in the EGT algorithm require the following operations:

(i) three matrix-vector products of the form $A\mathbf{x}$ and three of the form $A^{\mathrm{T}}\mathbf{y}$ for some $\mathbf{x}$ and $\mathbf{y}$

(ii) one calculation of the form $\nabla d_{\mathcal{X}}(\mathbf{x})$ and one of the form $\nabla d_{\mathcal{Y}}(\mathbf{y})$ for some $\mathbf{x}$ and $\mathbf{y}$

(iii) three calculations of the form $\nabla d_{\mathcal{X}}^*(\mathbf{u})$ and three of the form $\nabla d_{\mathcal{Y}}^*(\mathbf{v})$ for some $\mathbf{u}$ and $\mathbf{v}$

Hence it suffices to show that each of these operations requires a number of basic arithmetic operations that is linear in the size of the game tree.

Let `flops`($\langle$*expression*$\rangle$) denote the number of arithmetic operations needed in the calculation of $\langle$*expression*$\rangle$. We next estimate this number for each of the calculations in (i), (ii), and (iii) above.

For (i), if the payoff matrix A is represented in explicit sparse form, then $\texttt{flops}(A\mathbf{x})$ and $\texttt{flops}(A^{\mathrm{T}}\mathbf{y})$ are less than or equal to twice the number of non-zero entries in A because each of these calculations requires one scalar multiplication and at most one addition for each non-zero in A. Since the number of non-zero entries in A is bounded by the number of leaves in the game tree [60, 61], it follows that $\texttt{flops}(A\mathbf{x})$ and $\texttt{flops}(A^{\mathrm{T}}\mathbf{y})$ are linear in the size of the game tree.

For the calculations in (ii), assume $\mathcal{X} = \mathcal{Q}(Q, I, k, r) \subseteq \mathbb{R}^m$ and $\mathcal{Y} = \mathcal{Q}(\tilde{Q}, \tilde{I}, \tilde{k}, \tilde{r}) \subseteq \mathbb{R}^n$. The construction of the uniform treeplex $\mathcal{Q}(Q, I, k, r)$ and a straightforward induction argument shows that for generic $\mathbf{x} \in \mathcal{X},\ \mathbf{z} \in Q$,

$$\texttt{flops}(\nabla d_{\mathcal{X}}(\mathbf{x})) = \frac{(bk)^r - 1}{bk - 1} \cdot k \cdot \texttt{flops}(\nabla d_Q(\mathbf{z})) + m \leq m \cdot (\texttt{flops}(\nabla d_Q(\mathbf{z})) + 1).$$

Likewise, for generic $\mathbf{y} \in \mathcal{Y},\ \mathbf{w} \in \tilde{Q}$,

$$\texttt{flops}(\nabla d_{\mathcal{Y}}(\mathbf{y})) \leq n \cdot (\texttt{flops}(\nabla d_{\tilde{Q}}(\mathbf{w})) + 1).$$

Since both m and n are smaller than the size of the game tree [60, 61], it follows that $\texttt{flops}(\nabla d_{\mathcal{X}}(\mathbf{x}))$ and $\texttt{flops}(\nabla d_{\mathcal{Y}}(\mathbf{y}))$ are sublinear in the size of the game tree.

Finally, for the calculations in (iii), again assume $\mathcal{X} = \mathcal{Q}(Q, I, k, r) \subseteq \mathbb{R}^m$ and $\mathcal{Y} = \mathcal{Q}(\tilde{Q}, \tilde{I}, \tilde{k}, \tilde{r}) \subseteq \mathbb{R}^n$. An inductive argument similar to those in Section 2.4 shows that for generic $\mathbf{u} \in \mathbb{R}^m,\ \mathbf{s} \in \mathbb{R}^q$

$$\texttt{flops}(\nabla d_{\mathcal{X}}^*(\mathbf{u})) \leq m \cdot (\texttt{flops}(\nabla d_Q^*(\mathbf{s})) + 1),$$

and for generic $\mathbf{v} \in \mathbb{R}^n,\ \mathbf{t} \in \mathbb{R}^{\tilde{q}}$

$$\texttt{flops}(\nabla d_{\mathcal{Y}}^*(\mathbf{v})) \leq n \cdot (\texttt{flops}(\nabla d_{\tilde{Q}}^*(\mathbf{t})) + 1).$$

Thus both $\texttt{flops}(\nabla d_{\mathcal{X}}^*(\mathbf{u}))$ and $\texttt{flops}(\nabla d_{\mathcal{Y}}^*(\mathbf{v}))$ are sublinear in the size of the game tree.

Consequently, the overall number of arithmetic operations in each iteration of the EGT algorithm is bounded by a small factor of the size of the game tree. Furthermore, the matrix-vector multiplications $A\mathbf{x},\ A^{\mathrm{T}}\mathbf{y}$ dominate the total number of arithmetic operations.

2.5.3 Heuristics

Algorithm `EGT` has worst-case iteration-complexity $\mathcal{O}(1/\epsilon)$ and already scales to problems much larger than is possible to solve using state-of-the-art linear programming solvers (as we demonstrate in the experiments later in this paper). In this section we introduce two heuristics for further improving the speed of the algorithm, while retaining the guaranteed worst-case iteration-complexity $\mathcal{O}(1/\epsilon)$. The heuristics attempt to decrease $\mu_\mathcal{X}$ and $\mu_\mathcal{Y}$ faster than prescribed by the EGT algorithm while maintaining the excessive gap condition (2.35). This leads to overall faster convergence in practice, as our experiments will show.

Heuristic 1: Aggressive μ reduction

The first heuristic is based on the following observation: although the value $\tau = 2/(k+3)$ computed in step 2(a) of Algorithm `EGT` guarantees the excessive gap condition (2.35), this is potentially an overly conservative value. Instead we can use an adaptive procedure to choose a larger value of τ. Since we now can no longer guarantee the excessive gap condition (2.35) *a priori*, we are required to do a *posterior* verification which occasionally necessitates an adjustment in the parameter τ. In order to check (2.35), we need to compute the values of $f_{\mu_\mathcal{Y}}$ and $\phi_{\mu_\mathcal{X}}$. Observe that

$$\phi_{\mu_\mathcal{X}}(\mathbf{y}) = -\mu_\mathcal{X} d_\mathcal{X}^* \left(-\frac{1}{\mu_\mathcal{X}} A^\mathrm{T} \mathbf{y} \right)$$

and

$$f_{\mu_\mathcal{Y}}(\mathbf{x}) = \mu_\mathcal{Y} d_\mathcal{Y}^* \left(\frac{1}{\mu_\mathcal{Y}} A \mathbf{x} \right).$$

Therefore, both $f_{\mu_\mathcal{Y}}$ and $\phi_{\mu_\mathcal{X}}$ are easily computable since $d_\mathcal{X}, d_\mathcal{Y}$ are nice prox-functions by construction.

To incorporate Heuristic 1 in Algorithm `EGT` we extend the procedure `shrink` as follows.

Algorithm 4. `decrease`$(A, \mu_\mathcal{X}, \mu_\mathcal{Y}, \tau, \mathbf{x}, \mathbf{y}, d_\mathcal{X}, d_\mathcal{Y})$

```
1. $(\mu_\mathcal{X}^+, \mathbf{x}^+, \mathbf{y}^+) :=$ shrink$(A, \mu_\mathcal{X}, \mu_\mathcal{Y}, \tau, \mathbf{x}, \mathbf{y}, d_\mathcal{X}, d_\mathcal{Y})$
2. While $-\mu_\mathcal{X}^+ d_\mathcal{X}^* \left( -\frac{1}{\mu_\mathcal{X}^+} A^\mathrm{T} \mathbf{y}^+ \right) < \mu_\mathcal{Y} d_\mathcal{Y}^* \left( \frac{1}{\mu_\mathcal{Y}} A \mathbf{x}^+ \right)$    // $\tau$ is too big
    (a) $\tau := \tau/2$
    (b) $(\mu_\mathcal{X}^+, \mathbf{x}^+, \mathbf{y}^+) :=$ shrink$(A, \mu_\mathcal{X}, \mu_\mathcal{Y}, \tau, \mathbf{x}, \mathbf{y}, d_\mathcal{X}, d_\mathcal{Y})$
3. Return $(\mu_\mathcal{X}^+, \mathbf{x}^+, \mathbf{y}^+, \tau)$
```

By Theorem 4.1 of [46], when the input $(\mu_\mathcal{X}, \mu_\mathcal{Y}, \mathbf{x}, \mathbf{y})$ to `decrease` satisfies (2.35), the procedure `decrease` will halt.

Heuristic 2: Balancing and reduction of $\mu_\mathcal{X}$ and $\mu_\mathcal{Y}$

Our second heuristic is motivated by the observation that after several calls to the `decrease` procedure, one of $\mu_\mathcal{X}$ and $\mu_\mathcal{Y}$ may be much smaller than the other. This imbalance is undesirable because the larger one contributes the most to the worst-case bound given by (2.36). Hence after a certain number of iterations we perform a *balancing* step to bring these values closer together. The balancing consists of repeatedly shrinking the larger one of $\mu_\mathcal{X}$ and $\mu_\mathcal{Y}$.

We also observed that after such balancing, the values of $\mu_\mathcal{X}$ and $\mu_\mathcal{Y}$ can sometimes be further reduced without violating the excessive gap condition (2.35). We thus include a final reduction step in the balancing heuristic.

This balancing and reduction heuristic is incorporated via the following procedure. (We chose the parameter values 0.9 and 1.5 based on some initial experimentation.)

Algorithm 5. `balance`$(A, \mu_\mathcal{X}, \mu_\mathcal{Y}, \tau, \mathbf{x}, \mathbf{y}, d_\mathcal{X}, d_\mathcal{Y})$

```
1. While $\mu_\mathcal{X} > 1.5\mu_\mathcal{Y}$     // shrink $\mu_\mathcal{X}$
       $(\mu_\mathcal{X}, \mathbf{x}, \mathbf{y}, \tau) :=$ decrease$(A, \mu_\mathcal{X}, \mu_\mathcal{Y}, \tau, \mathbf{x}, \mathbf{y}, d_\mathcal{X}, d_\mathcal{Y})$
2. While $\mu_\mathcal{Y} > 1.5\mu_\mathcal{X}$     // shrink $\mu_\mathcal{Y}$
       $(\mu_\mathcal{Y}, \mathbf{y}, \mathbf{x}, \tau) :=$ decrease$(-A^\mathrm{T}, \mu_\mathcal{Y}, \mu_\mathcal{X}, \tau, \mathbf{y}, \mathbf{x}, d_\mathcal{Y}, d_\mathcal{X})$
3. While $0.9\mu_\mathcal{Y} d_\mathcal{Y}^*\left(\frac{1}{0.9\mu_\mathcal{Y}}A\mathbf{x}\right) \le -0.9\mu_\mathcal{X} d_\mathcal{X}^*\left(-\frac{1}{0.9\mu_\mathcal{X}}A^\mathrm{T}\mathbf{y}\right)$
   // decrease $\mu_\mathcal{X}$ and $\mu_\mathcal{Y}$ if possible
       (a) $\mu_\mathcal{X} := 0.9\mu_\mathcal{X}$
       (b) $\mu_\mathcal{Y} := 0.9\mu_\mathcal{Y}$
4. Return $(\mu_\mathcal{X}, \mu_\mathcal{Y}, \mathbf{x}, \mathbf{y}, \tau)$
```

We are now ready to describe the variant of `EGT` with Heuristics 1 and 2.

Algorithm 6. `EGT-2`

```
1. $(\mu_\mathcal{X}^0, \mu_\mathcal{Y}^0, \mathbf{x}^0, \mathbf{y}^0) =$ initial$(A, d_\mathcal{X}, d_\mathcal{Y})$
2. $\tau := 0.5$
```

```
3. For $k = 0, 1, \ldots$:
    (a) If $k$ is even: // Shrink $\mu_{\mathcal{X}}$
        i. $(\mu_{\mathcal{X}}^{k+1}, \mathbf{x}^{k+1}, \mathbf{y}^{k+1}, \tau) :=$ decrease$(A, \mu_{\mathcal{X}}^{k}, \mu_{\mathcal{Y}}^{k}, \tau, \mathbf{x}^{k}, \mathbf{y}^{k}, d_{\mathcal{X}}, d_{\mathcal{Y}})$
        ii. $\mu_{\mathcal{Y}}^{k+1} = \mu_{\mathcal{Y}}^{k}$
    (b) If $k$ is odd: // Shrink $\mu_{\mathcal{Y}}$
        i. $(\mu_{\mathcal{Y}}^{k+1}, \mathbf{y}^{k+1}, \mathbf{x}^{k+1}, \tau) :=$ decrease$(-A^{\mathrm{T}}, \mu_{\mathcal{Y}}^{k}, \mu_{\mathcal{X}}^{k}, \tau, \mathbf{y}^{k}, \mathbf{x}^{k}, d_{\mathcal{Y}}, d_{\mathcal{X}})$
        ii. $\mu_{\mathcal{X}}^{k+1} = \mu_{\mathcal{X}}^{k}$
    (c) If $k \mod 100 = 0$ // balance and reduce
            $(\mu_{\mathcal{X}}^{k}, \mu_{\mathcal{Y}}^{k}, \mathbf{x}^{k}, \mathbf{y}^{k}, \tau) :=$ balance$(A, \mu_{\mathcal{X}}^{k}, \mu_{\mathcal{Y}}^{k}, \tau, \mathbf{x}^{k}, \mathbf{y}^{k}, d_{\mathcal{X}}, d_{\mathcal{Y}})$
```

2.6 Computational results

We implemented Algorithm `EGT-2` in `C++` and ran the computational experiments on an IBM eServer p5 570 with 128 gigabytes of RAM and four 1.65 GHz processors. We next report some computational experiments as well as an interesting application to the design of poker-playing programs.

2.6.1 Experimental setup

We tested the algorithm on five abstractions of poker games ranging from relatively small to very large. An *abstraction* of a game is a smaller game that captures some of the main features of the original game [8, 17, 57, 16]. The approach of abstracting a game and then solving for the equilibrium of the abstracted game is a practical way of constructing good strategies for the original game [8, 17, 16, 18, 19], and is the state-of-the-art approach to generating poker-playing programs.

We chose these problems because we wanted to evaluate the algorithms on real-world instances, rather than on randomly generated games (which may not reflect any realistic setting). Table 2.1 provides the sizes of the test instances. The first three instances, `10k`, `160k`, and `RI`, are abstractions of Rhode Island Hold'em poker [57] computed using the *GameShrink* automated abstraction algorithm [17]. The first two instances are lossy (non-equilibrium preserving) abstractions, while the `RI` instance is a lossless abstraction. The `Texas` and `GS4` instances are lossy abstractions of Texas Hold'em poker [15, 18].

Name	Rows	Columns	Non-Zero Entries
`10k`	14,590	14,590	536,502
`160k`	226,074	226,074	9,238,993
`RI`	1,237,238	1,237,238	50,428,638
`Texas`	18,536,842	18,536,852	61,498,656,400
`GS4`	299,477,082	299,477,102	4,105,365,178,571

Table 2.1: Problem sizes (when formulated as a linear program) for the instances used in our experiments.

Table 2.2 provides the average time per EGT iteration of our implementation for each of the test problems both with and without the heuristics.

Name	EGT with heuristics (time in secs)	EGT without heuristics (time in secs)
`10k`	0.10	0.10
`160k`	1.28	1.20
`RI`	7.65	6.53
`Texas`	2,400	1,420
`GS4`	42,400	28,000

Table 2.2: Average CPU time per EGT iteration for the instances used in our experiments.

Due to the enormous size of the `GS4` instance, we do not include it in the experiments that compare better and worse techniques within our algorithm. Instead, we use the four smaller instances to find a good configuration of the algorithm, and we use that configuration to tackle the `GS4` instance. We then report on how well the resulting strategies on the `GS4` instance did in the AAAI-08 Computer Poker Competition.

Previously, the most effective algorithms for solving sequential games of imperfect information were based on interior-point methods applied to the linear programming formulation of the problem [8, 17]. It seems desirable to test our algorithm against state-of-the-art implementations of such methods. However, this is not particularly relevant in the context of the problems we are solving. For example, solving the relatively small game of Rhode Island Hold’em poker required 25 GB RAM using CPLEX’s interior-point method. The instance `GS4` is more than two hundred times larger. Simply *representing* such a problem in the explicit representation required by CPLEX and other interior-point solvers would require more than $80,000$ GB RAM. The memory needed for the necessary data structures, such as storing the Cholesky factorization, would increase this further. Such a requirement is far beyond the capability of current hardware. Thus, it is not even possible to compare the run-time performance of our algorithm with linear programming approaches.

2.6.2 Experimental comparison of prox functions

Our first experiment compared the relative performance of the prox functions induced by the entropy and Euclidean prox functions described in Example 1 earlier in this paper. Figure 2.1 shows the results. (Heuristics 1 and 2, described above, and the memory saving technique described later, were enabled in this experiment.) In all of the figures, the units of the vertical axis are the number of chips in the corresponding poker games.

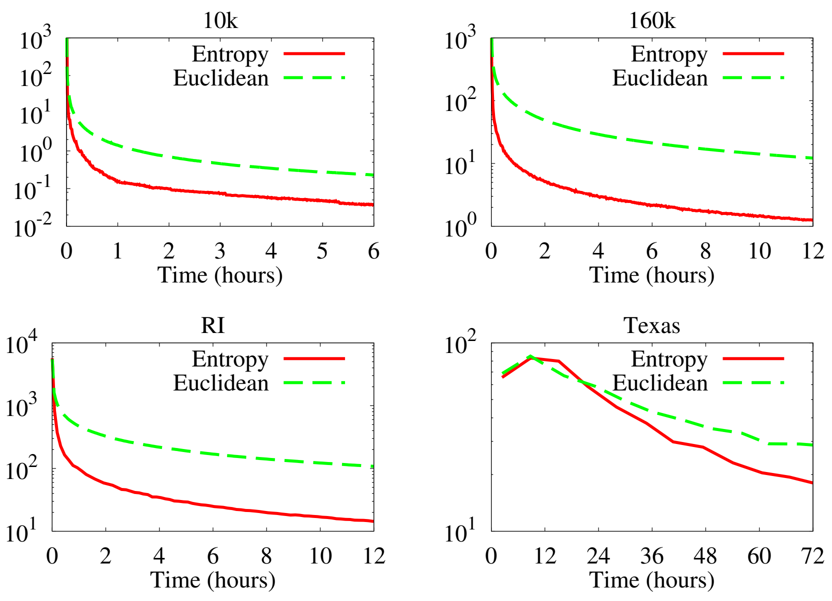


Figure 2.1: Comparison of the entropy and Euclidean prox functions. The value axis is the gap ϵ (Equation 2.2).

The entropy prox function outperformed the Euclidean prox function on all four instances. Therefore, in the remaining experiments we exclusively use the entropy prox function.

2.6.3 Experimental comparison of the heuristics

Figure 2.2 demonstrates the impact of applying Heuristic 1: *Aggressive μ reduction.* (For this experiment, Heuristic 2, was not used. The memory saving technique, also described later, was used.) On all four instances, Heuristic 1 reduced the gap significantly. On the larger instances, this reduction was an order of magnitude.

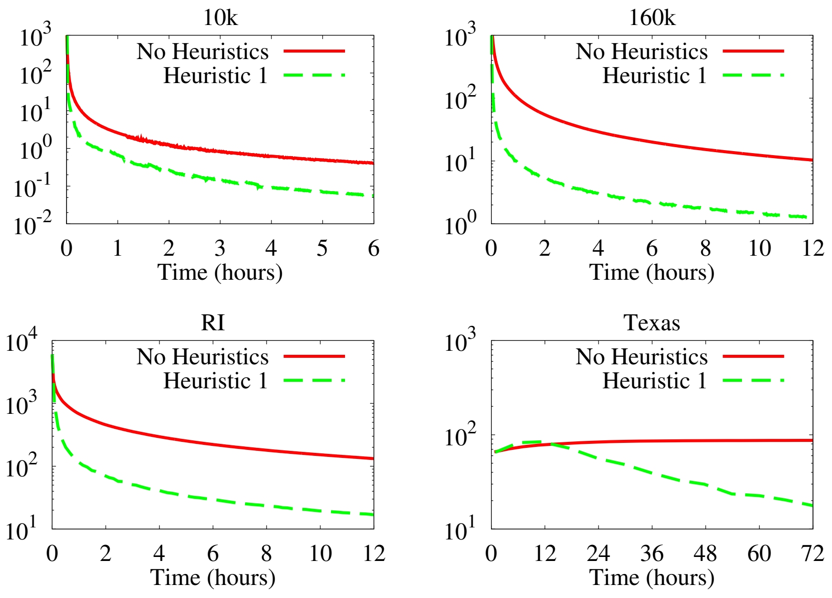


Figure 2.2: Experimental evaluation of Heuristic 1. The value axis is the gap ϵ (Equation 2.2)

Figure 2.3 demonstrates the impact of applying Heuristic 2: *Balancing and reduction of $\mu_{\mathcal{X}}$ and $\mu_{\mathcal{Y}}$.* Because Heuristic 2 is somewhat expensive to apply, we experimented with how often the algorithm should run it. (We did this by varying the constant in line 3(c) of Algorithm `EGT-2`. For example, when the figure states "10 iterations", that means that the heuristic is run once every ten iterations. In this experiment, Heuristic 1 was turned off, but the memory-saving technique, described later, was used.) Figure 2.3 shows that it is always effective to use Heuristic 2, although the frequency at which it should be applied varies depending on the instance.

2.6.4 Application to Texas Hold'em poker

Poker is a game involving elements of chance, imperfect information, and counter-speculation. Game-theoretic optimal strategies are far from straightforward, often necessitating such tactics as bluffing and

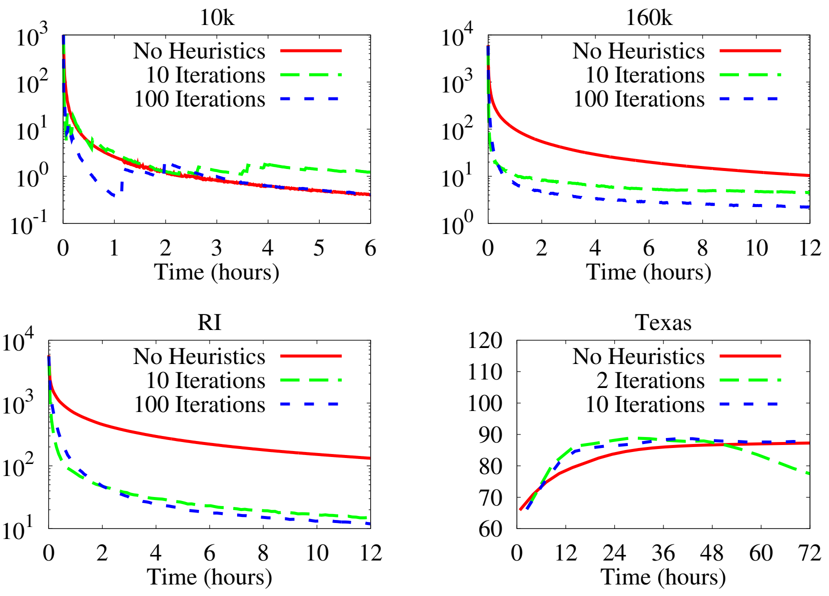


Figure 2.3: Heuristic 2 applied at different intervals. The value axis is the gap ϵ (Equation 2.2)

slow-playing. For these reasons, and others, poker has been identified as an important challenge problem for the field of artificial intelligence [9]. Just as the development of a computer program capable of beating the world's best human chess player was once seen as an important milestone, the development of a poker-playing program capable of beating the best humans is now seen as an equally important milestone.

The prox-function construction described in Section 2.3 has been instrumental in the development of some recent programs for playing Texas Hold'em poker. An important difference between different variants of Texas Hold'em is the *betting structure*. Two common betting structures are *limit*, in which players may bet a fixed amount, and *no-limit*, in which players may bet any number of their chips. Our equilibrium-finding algorithm computed the strategies for both *GS3* [18] and *Tartanian* [19], to programs that play limit and no-limit Texas Hold'em, respectively.

In 2008, the Association for the Advancement of Artificial Intelligence (AAAI) held the third annual Computer Poker Competition, where computer programs submitted by teams worldwide compete against each other. *GS4-Beta* (a subsequent version of *GS3*) placed first (out of nine) in the Limit Bankroll competition and *Tartanian* placed third (out of four) in the No-Limit competition. (*Tartanian* actually had the highest winning rate in the competition, but due to the winner determination rule for the competition, it only got third place.) This is particularly impressive given the small amount of poker-specific knowledge that was incorporated into those programs. They instead depend on an equilibrium analysis conducted by our algorithm (which in turn relies on our prox-function construction) for determining their strategies. As the developers of *GS3* and *Tartanian* point out, it is currently not feasible to solve their models using off-the-shelf linear programming solvers.

The approach used for constructing the above players is based on algorithmically creating *lossy* abstractions of the original game [15, 18, 19]. These abstractions are smaller sequential games that attempt to preserve the strategic properties of the original game. The abstracted game is then solved for an ϵ-equilibrium using the algorithm discussed in this paper. The larger the abstracted game (i.e., the finer the abstraction), the better the quality of the strategies generally is. The approach of automated abstraction followed by equilibrium finding was first used in Texas Hold'em in [16], and is nowadays used by basically all of the competitive poker-playing programs.

For the limit competition, our implementation of the EGT algorithm solved an abstracted game whose payoff matrix was $10^8 \times 10^8$. For the no-limit competition, our algorithm solved a game with payoff matrix of size $10^7 \times 10^7$. The uniform treeplexes introduced in Section 2.4 provide a perfect framework for modeling limit Texas Hold'em poker. For this game, the treeplex $Q_{\mathcal{X}}$ for the first player is a uniform treeplex. The "basic" treeplex $Q \subseteq [0,1]^{14}$ has the linear description $Q = \{\mathbf{x} \in [0,1]^{14} : E\mathbf{x} = \mathbf{e}\}$ where

$$E := \left[\begin{array}{cccccccccccccc} 1 & 1 & 1 & & & & & & & & & & & \\ & -1 & & 1 & 1 & 1 & & & & & & & & \\ & & & & & -1 & 1 & 1 & 1 & & & & & \\ & & -1 & & & & & & & 1 & 1 & 1 & & \\ & & & & & & & & & & & -1 & 1 & 1 \end{array}\right], \quad \mathbf{e} := \left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}\right].$$

The fourteen columns of E represent the possible sequence of actions that the first player can take during each betting round of the game. Each row in E encodes a simplex over three actions: *fold*, *call*, and *raise*. (The last row only allows fold and call.) The set $I = \{2, 3, 5, 6, 8, 9, 11, 12, 14\}$ indexes the sequences that do not end with a fold. Texas Hold'em is played in four rounds so $r = 4$. Finally, the value of k depends on the quality of the abstraction. The abstractions in [18] range from $k = 6$ to $k = 40$ (the k is actually different in each round). The treeplex $Q_{\mathcal{Y}}$ for the second player is also a uniform treeplex with similar characteristics.

2.6.5 Memory requirements

One particularly attractive feature of the EGT algorithm is the fact that the only operation performed on the matrix A is a matrix-vector product. As a consequence, we can exploit the problem structure to store only an *implicit* representation of the payoff matrix A. This implicit representation relies on a certain type of decomposition that is present in poker games as well as in the more general class of *games with ordered signals* [17, 15]. For example, the betting sequences that can occur in most poker games are independent of the cards that are dealt. We can decompose the payoff matrix based on these two aspects.[1]

For ease of exposition, we explain the concise representation in the context of Rhode Island Hold'em poker [57], although the general technique applies much more broadly (and we use it in our Texas Hold'em games as well). The payoff matrix A can be written as

$$A = \begin{bmatrix} A_1 & & \\ & A_2 & \\ & & A_3 \end{bmatrix}$$

where

$$\begin{aligned} A_1 &= F_1 \otimes B_1, \\ A_2 &= F_2 \otimes B_2, \text{ and} \\ A_3 &= F_3 \otimes B_3 + S \otimes W \end{aligned} \tag{2.38}$$

for much smaller matrices F_i, B_i, S, and W. The matrices F_i correspond to sequences of moves in round i that end with a fold, and S corresponds to the sequences in round 3 that end in a showdown. The matrices B_i encode the betting structures in round i, while W encodes the win/lose/draw information determined by poker hand ranks. The symbol $\otimes$ in (2.38) denotes the *Kronecker product.* Recall that the Kronecker product of two matrices $B \in \mathbb{R}^{m \times n}$ and $C \in \mathbb{R}^{p \times q}$, is

$$B \otimes C = \begin{bmatrix} b_{11}C & \cdots & b_{1n}C \\ \vdots & \ddots & \vdots \\ b_{m1}C & \cdots & b_{mn}C \end{bmatrix} \in \mathbb{R}^{mp \times nq}.$$

Given the above concise representation of A, computing $\mathbf{x} \mapsto A\mathbf{x}$ and $\mathbf{y} \mapsto A^{\mathrm{T}}\mathbf{y}$ is straightforward, and the space required is sublinear in the size of the game tree. For example, in Rhode Island Hold'em,

[1]The fact that possible betting sequences are independent of cards has also been exploited by automated abstraction algorithms, but in a totally different way [17].

the dimensions of the F_i and S matrices are 10×10, and the dimensions of B_1, B_2, and B_3 are 13×13, 205×205, and 1,774 $\times$ 1,774, respectively—in contrast to the matrix A, which is 883,741 $\times$ 883,741. Furthermore, the matrices F_i, B_i, S, and W are themselves sparse, which allows us to use the Compressed Row Storage (CRS) data structure that only stores non-zero entries.

Table 2.3 clearly demonstrates the extremely low memory requirements of the EGT algorithms when using our memory-saving technique. Most notably, on the `GS4` instance, both of the CPLEX algorithms (simplex and interior point) require more than 80,000 GB simply to *represent* the problem. In contrast, using the decomposed payoff matrix representation, the EGT algorithms require only 43.96 GB. Furthermore, in order to solve the problem, both the simplex and interior-point algorithms would require additional memory for their internal data structures. Therefore, the EGT family of algorithms with our memory-saving techniques is a significant improvement over the state-of-the-art for large-scale problems.

Name	**CPLEX IPM**	**CPLEX Simplex**	**EGT**
`10k`	0.082 GB	> 0.051 GB	0.012 GB
`160k`	2.25 GB	> 0.664 GB	0.035 GB
`RI`	25.2 GB	> 3.45 GB	0.15 GB
`Texas`	> 458 GB	> 458 GB	2.49 GB
`GS4`	$> 80,000$ GB	$> 80,000$ GB	43.96 GB

Table 2.3: Memory footprint in gigabytes of CPLEX interior-point method (IPM), CPLEX simplex, and our EGT algorithms.

The memory usage for the CPLEX simplex algorithm reported in Table 2.3 is the memory used after 10 minutes of execution (except for the `Texas` and `GS4` instances which could not run at all using either CPLEX algorithm). This algorithm's memory requirements grow and shrink during the execution depending on its internal data structures. Therefore, the number reported is a lower bound on the maximum memory usage during execution.

Although the results presented in Table 2.3 are for CPLEX, they apply to any algorithm that requires an explicit representation of the constraint matrix of the linear program. Since the only matrix operation needed by our algorithm is a matrix-vector product, we are able to use an implicit representation of the constraint matrix, as discussed above.

2.6.6 Speedup from parallelizing the matrix-vector product

Beyond our time-saving heuristics discussed earlier in this paper, we further reduce the time requirements of the matrix-vector product by parallelization. We parallelize the operation by simply partitioning the work into n pieces when n CPUs are available. The speedup we can achieve on parallel CPUs is demonstrated in Table 2.4. The instance used for this test is the `Texas` instance described above. The matrix-vector product operation scales linearly in the number of CPUs, and the time to perform one iteration of the algorithm scales nearly linearly, decreasing by a factor of 3.69 when using four CPUs.

CPUs	**matrix-vector product**		**EGT iteration**	
	time (secs)	**speedup**	**time (secs)**	**speedup**
1	278	1.00x	1,420	1.00x
2	140	1.98x	730	1.94x
3	93	2.98x	490	2.89x
4	69	4.00x	384	3.69x

Table 2.4: Effect of parallelization for the `Texas` instance.

2.7 Conclusions and future research

We developed first-order algorithms to approximate Nash equilibria of two-person zero-sum sequential games by applying Nesterov's smoothing technique to the saddle-point formulation (2.1) of the Nash equilibrium problem. The heart of our approach is a construction of nice prox-functions for the treeplex polytopes in the saddle-point formulation.

We implemented an algorithm based on our prox-functions and Nesterov's excessive gap technique. We included two novel heuristics that improve the algorithm's speed of convergence considerably. Experiments show that the algorithm based on the entropy-induced prox function is faster than the algorithm based on the Euclidean-induced prox function. For poker games and similar games, we introduced a decomposed matrix representation that reduces storage requirements drastically. Our techniques enable us to solve to near-equilibrium games that are over four orders of magnitude larger than the largest addressable previously. We also showed near-perfect speed-up from parallelization, which makes our algorithms particularly appropriate for modern multi-core architectures.

In contrast to a direct first-order approach to solve the linear programming formulation of (2.1) such as that proposed in [31], our approach automatically yields algorithms that generate feasible strategies $\mathbf{x} \in \mathcal{X}$, $\mathbf{y} \in \mathcal{Y}$ throughout execution. This is of crucial importance because points that violate the constraints defining the treeplexes $\mathcal{X}, \mathcal{Y}$ even slightly are typically meaningless strategies. In particular, unlike the iterates generated by our algorithm, the iterates generated by an infeasible algorithm would typically not yield approximate equilibria. Furthermore, the linear programming formulation of (2.1) increases the dimension of the problem substantially since it requires a new variable for each constraint in the description of the treeplexes $\mathcal{X}, \mathcal{Y}$.

In addition to our first-order smoothing approach to the problem (2.1), it is conceivable that specialized versions of other algorithmic approaches may also lead to effective algorithms for solving the saddle-point problem (2.1). For example, a specialized interior-point algorithm could use an appropriately designed iterative method to solve the system of equations at each main iteration. No such approach has been successfully developed so far.

Another approach we plan to investigate is the use of *stochastic sampling* for approximating the objective function. This has already been studied in the context of matrix games [27], although that approach was based on a different optimization algorithm. For large-scale instances, it is quite expensive to evaluate the matrix-vector product in the objective function (and in the gradient computations). Speeding up these operations, in conjunction with strong convergence guarantees, could have a significant impact in practice. These interesting alternative algorithmic approaches will be the subject of future research.

Chapter 3

MDD-based Constraint Programming

3.1 Introduction

The domain store is a fundamental tool for constraint programming, because it propagates the results of individual constraint processing. It allows the reduced domains obtained for one constraint to be passed to the next constraint for further filtering.

A weakness of the domain store, however, is that it transmits a limited amount of information. It accounts for no interaction among the variables, because any solution in the Cartesian product of the current domains is consistent with it. This restricts the ability of the domain store to pool the results of processing individual constraints and provide a global view of the problem.

To address this shortcoming, the authors of [3] proposed replacing the domain store with a richer data structure, namely a multivalued decision diagram (MDD). In this scheme, domain filtering algorithms are replaced or augmented by algorithms that refine and update the MDD to reflect each constraint. It was found that MDD-based propagation leads to substantial speedups in the solution of multiple `alldiff` constraints, in many instances reducing the search tree from a million or so nodes to a single node. The idea was extended to equality constraints in [24]. A unified node-splitting scheme for refining the MDD was proposed in [23] and applied to certain configuration problems.

This chapter is organized as follows. In Section 3.2, we provide a summary of constraint programming focusing on those aspects that we need. We then motivate the key ideas of MDD-based propagation. Section 3.3 provides a formal background on MDD and MDD-based propagation. In Section 3.4 we present a systematic method for extending traditional domain store filtering techniques to MDD filtering techniques.

The following section applies this framework to design propagation algorithms for many of the fundamental global constraints in constraint programming. We extend this technique to provide a systematic way of reusing domain store propagators and provide several alternatives for embedding this technique within a constraint solving system. We then show that *all* the specialized algorithms presented, both old and new, can be understood as more efficient implementations of the technique of reusing currently existing domain store propagators.

In Section 3.6 we present a short note on the complexity of our framework. We note that iterating to a fixed-point requires a number of iterations that is bounded by the number of edges in the MDD. By providing sufficient conditions on the strength of filtering we show that certain domain propagation techniques in our framework will achieve MDD consistency in polynomial time.

Finally, Section 3.7 describes some methods for using MDDs to augment branching strategies as well as how to incorporate MDDs in primal heuristics for solving both constraint satsification and optimization problems.

3.2 Constraint Programming Preliminaries

Given a variable x, the *domain* of x is the set of values that can be assigned to x, and is denoted by $D(x)$. In this work we only consider variables with finite domains. Generalizing to finite sequences of variables $X = (x_1, x_2, \ldots, x_k)$, the declared domain of solutions is given by the Cartesian product of the domains of the variables in X, that is, $D(x) = D(x_1) \times \cdots \times D(x_k)$. A *constraint* C on X is defined as a subset of $D(X)$. A tuple $(d_1, \ldots, d_k) \in C$ is a *solution* to C and also say that $(d_{1,} \ldots, d_k)$ *satisfies* C. A value $d \in D(x_i)$ has *support in* C (or is *consistent with respect to* C) if it belongs to some tuple in C; otherwise d is *unsupported in* C (or is *inconsistent with respect to* C). The constraint C is *inconsistent* if it does not contain a solution, that is, it is the empty set; otherwise, C is *consistent*.

A *constraint satisfaction problem*, or *CSP*, is defined by a finite sequence of variables $\mathcal{X} = (x_1, x_2, \ldots, x_n)$, together with a finite set of constraints $\mathcal{C}$, where each constraint $C \in \mathcal{C}$ is defined over a subsequence of variables $\mathrm{scope}(C) \subseteq \mathcal{X}$. The goal is to find an assignment $x_i = d_i$ with $d_i \in D(x_i)$ for $i = 1, \ldots, n$, such that all that constraints are satisfied. The assignment is called a *feasible solution* to the CSP.

The solution process of constraint programming interleaves *constraint propagation* (or *propagation* in short), and *search*. The search process effectively enumerates all possible variable-value combinations. The search process continues until a feasible solution is found or proves that no feasible solution exists. We say that this process constructs a *search tree*. Each node in the tree has a declared domain which is a subset of its parent's domain. To reduce the exponential number of combinations, *constraint propagation* is applied to each node of the search tree: given the current domains and a constraint C, remove domain values that are inconsistent with C. This is repeated for all constraints until no more domain values can be removed. The removal of inconsistent domain values is called *filtering*.

Of course we would like filtering algorithms to remove as many inconsistent values as possible. However, this goal needs to be balanced against 'speed' (that is, time complexity or efficiency) since filtering algorithms are typically applied at each node of the search tree. Indeed, conventional wisdom tells us that reducing the search tree through enhanced processing at at each node often does not justify the additional overhead. In the cases when filtering a constraint C removes *all* inconsistent values from the domain with respect to C, we say that the filtering algorithm makes C *domain consistent*. Formally, a constraint C on variables $x_1, \ldots, x_k$ is called *domain consistent* if for each variable x_i and each value $d_i \in D(x_i)$ $i \in \{1, \ldots, k)\}$, there exists a value $d_j \in D(x_j)$ for all $j \neq i$ such that $(d_1, \ldots, d_k) \in C$. For historical reasons, domain consistency is also referred to as *hyper-arc consistency* or *generalized-arc consistency*.

Establishing domain consistency for *binary constraints* (constraints defined on two variables) is inexpensive. In general, this is not the case for higher arity constraints since the naive approach requires time that is exponential in the number of variables. However, for some constraints it is possible to establish domain consistency much more efficiently by exploiting the underlying structure of the constraint.

The idea of constraint propagation presented above can be generalized so that one propagates a constraint through a *constraint store*: a datastructure that pools the results of individual constraint processing. When the next constraint is processed (filtered), the constraint store is in effect processed along with it. Notice that propagating the results from processing one constraint to the other constraints is a mechanism that allows a solver to (partially) exploit the global structure induced by the set of constraints of a CSP, that is, it approximates the goal of processing all the constraints simultaneously.

In current practice, the constraint store is normally a *domain store*: constraints are processed by filtering algorithms that remove values from the variables' domains and the reduced domains are the starting point for filtering the next constraint. A domain store also guides branching in a natural way. When branching on a variable, one can simply split the domain in the current domain store.

For more information on constraint programming we refer the reader to [4] and [14].

3.3 MDDs and MDD-Based Constraint Solving

Multivalued decision diagrams (MDDs) [28] generalize binary decision diagrams (BDDs) [2, 1], which have long been used for circuit design/verification [10, 32] and very recently for optimization [7, 21, 22]. The MDD for a constraint set is essentially a more compact representation of a branching tree, obtained by superimposing isomorphic subtrees. The shape of the resulting MDD depends on the order in which one branches on the variables.

Formally, an *ordered MDD* is a directed acyclic graph whose nodes are partitioned into n (possibly empty) subsets or *layers* $L_1, \ldots, L_{n+1}$, where the layers $L_1, \ldots, L_n$ corresponding respectively to variables $x_1, \ldots, x_n$. L_1 contains a single *top* node **T**, and L_{n+1} contains two *bottom* nodes **0** and **1**. The *width* of the MDD is the maximum number of nodes in a layer, or $\max_{i=1}^{n}\{|L_i|\}$.

All edges of the MDD are directed from an upper to a lower layer; that is, from a node in some L_i to a node in some L_j with $i < j$. For our purposes it is convenient to assume (without loss of generality) that each edge connects two adjacent layers. Let $L(v)$ denote the layer of the node v. Each edge out of layer i is labeled with an element of the domain $D(x_i)$ of x_i, and no label occurs more than once on the edges leaving any given node. The set $E(p, q)$ of edges from node p to node q may contain multiple edges, and we denote each with its label.

An edge with label v leaving a node in layer i represents an assignment $x_i = v$. Each path in the MDD from **T** to **0** or **1** can be denoted by the edge labels $v_1, \ldots, v_n$ on the path and is identified with the assignment $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$. The MDD as a whole therefore represents a pseudoboolean function f for which $f(v_1, \ldots, v_n)$ has the value 1 when $v_1, \ldots, v_n$ is a path from **T** to **1**, and 0 when it is a path from **T** to **0**.

It is clear that any pseudoboolean function of finite-domain variables $x_1, \ldots, x_n$ can be represented by an MDD. Any constraint set with finite-domain variables can likewise be represented by an MDD, because it defines a pseudoboolean function that maps every assignment to its variables $x_1, \ldots, x_n$ to true or false.

For our purposes, it is convenient to generate only the portion of an MDD that contains paths from **T** to **1**. The resulting MDD represents assignments to $x_1, \ldots, x_n$ for which $f(x_1, \ldots, x_n) = 1$. A path $v_1, \ldots, v_n$ is *feasible* for a given constraint C if setting $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$ satisfies C. Constraint

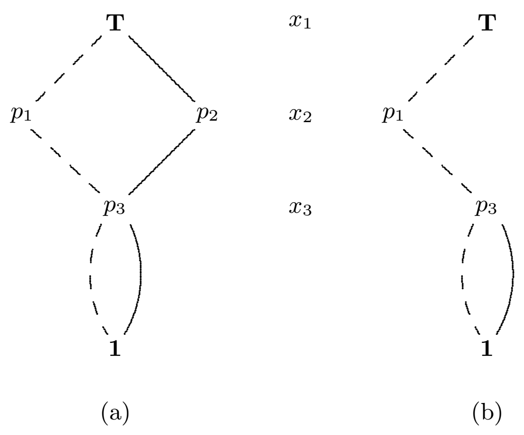


Figure 3.1: (a) MDD for $x_1 = x_2$. (b) MDD after processing for $\texttt{among}((x_1, x_2), \{1\}, 0, 1)$

C is feasible on an MDD if the MDD contains a feasible path for C.

A constraint C is *consistent* on a given MDD if every edge of the MDD lies on some feasible path. Thus consistency is achieved when all redundant edges (i.e., edges on no feasible path) have been removed. Domain consistency for C is equivalent to consistency on an MDD of width one that represents the variable domains. That is, it is equivalent to consistency on an MDD in which each layer L_i contains a single node p_i, and $E(p_i, p_{i+1}) = D(x_i)$ for $i = 1, \dots, n$. We will refer to this MDD as M_n.

A very simple example illustrates the advantage of MDD-based propagation. Suppose that a constraint satisfaction problem contains the constraints $x_1 = x_2$ and

$$\texttt{among}(\{x_1, x_2, x_3\}, \{1\}, 0, 1), \tag{3.1}$$

where the domain of each x_i is $\{0, 1\}$. The constraint (3.1) requires that at most one of the variables x_1, x_2, x_3 take the value 1. It is clear that we must have $x_1 = x_2 = 0$, and yet neither constraint allows any domain filtering. Suppose, however, we create the MDD of Figure 3.1(a) to represent the constraint $x_1 = x_2$ (this is actually a *binary* decision diagram because the variable domains are binary). An edge leaving a node in the x_i layer of the MDD indicates that $x_i = 0$ (dashed edge) or $x_i = 1$ (solid edge). The four paths from the top node to the bottom node indicate the four solutions of $x_1 = x_2$, namely $(x_1, x_2, x_3) = (0, 0, 0), (0, 0, 1), (1, 1, 0), (0, 0, 1)$.

Now if we process the MDD to reflect constraint (3.1), we can immediately delete two solid edges to obtain the MDD in Fig. 3.1(b), because they lie on no path that satisfies (3.1). This not only curtails branching by reducing the domains of x_1 and x_2 to $\{0\}$, but it creates a more restrictive MDD that can be passed along to any other constraints in the problem for further processing.

The MDD of Fig. 3.1 represents $x_1 = x_2$ exactly, but this is not typical of practice, because exact MDD representations of a constraint can grow exponentially with the number of variables. Rather, we start with a simple MDD that permits all solutions and *refine* it each time a constraint is processed. Refinement is accomplished by adding some nodes and edges to the MDD so as to exclude solutions that violate the constraint.

The basic operation of refinement is *node-splitting*, in which the edges entering a given node are partitioned into equivalence classes, and ideally the node is split into one copy for each equivalence class. The set of outgoing edges for each copy is the same as the set of outgoing edges of the original node. We note that determining the equivalence classes may be costly to compute in practice, in which case an approximation of equivalence is used. We take care that the *width* of the MDD (maximum number of nodes in a layer) remains within a fixed bound. When splitting a node we merge equivalence classes when necessary in order to respect this restriction. The resulting MDD is a relaxation in the sense that it may fail to exclude all assignments that violate the constraint, but it is a much stronger relaxation than a domain store. A principled approach to node refinement in MDDs is introduced in [23].

We also update the MDD by deleting some edges that can be part of no solution, an operation that generalizes conventional domain filtering. We will refer to it as *MDD filtering*. This can lead to further reduction of the MDD, if after the removal of the edge some other edges no longer have a path to $\mathbf{1}$ or can no longer be reached by a path from the root.

In the example of Fig. 3.1, the variables have the same ordering in the MDD as in the `among` constraint for which it is filtered. This is not true in general. The MDD is normally processed with several constraints, which may imply different natural orderings. It is impossible for the MDD ordering to be optimal for every constraint. Therefore, we designed our algorithms to be valid for an arbitrary ordering of the variables in the MDD.

A MDD-based constraint solver is based on *propagation* and *search* just as traditional CSP solvers, but the domain filtering process at each node of the tree is replaced or supplemented by an MDD refinement and filtering process. This requires that additional time be invested at each node, violating the constraint programmer's maxim that it is better to process many nodes than spend much time at each one. This maxim, however, evolved in a context in which domain stores propagated limited information. If more information can be transmitted to the next constraint, then it may be worth investing more time to obtain this information.

3.4 A Framework for MDD Propagation

A primary research issue in applying MDDs to solving CSPs is whether there exist fast and effective propagation algorithms for constraints. Indeed, there can be a jump in complexity when trying to design filtering algorithms that are *complete*, that is, achieve domain and MDD consistency respectively. For example, in [3] the authors demonstrate that although the constraint `alldiff` has a polynomial-time algorithm that achieves domain consistency it is NP-hard to achieve MDD consistency on an MDD of polynomial size.

Until now (to the best of our knowledge) there were MDD propagation algorithms for the following constraints: (one-sided) inequality constraints [3], `alldiff` [3], equality constraints [23], and `among` constraints. The reasoning used for designing propagation algorithms for each of the constraints seemed to be ad-hoc. In this section we will present a systematic method for extending the reasoning used to propagate constraints in the traditional domain store setting to design MDD propagation algorithms. We will demonstrate the efficacy of the method by designing MDD propagation algorithms for several important classes of constraints.

We start by presenting the MDD inequality propagator [3] using the language of the general framework and then proceed to the formal definitions and present further examples.

3.4.1 An inequality propagator

Suppose that we want to propagate an inequality over a separable function:

$$\sum_{j \in J} f_j(x_j) \leq b.$$

We can propagate such a constraint on an MDD by performing shortest-path computations where the length of an edge e is simply $f_{L(\mathrm{tail}(e))}(e)$. Recall that each edge e is identified with a domain value corresponding to the variable on the layer of the tail of the edge, that is, $L(\mathrm{tail}(e))$. With each node r in the MDD we will compute and store (d_T, d_1), where d_T is the length of the shortest-path from the root **T** to the node, and d_1 is the length of the shortest-path from the node to the sink **1**.

We can delete an edge e from the MDD if and only if every path through the edge e has a length greater than b, that is, if and only if

$$d_T(\mathrm{tail}(e)) + f_{L(\mathrm{tail}(e))}(e) + d_1(\mathrm{head}(e)) > b.$$

This inequality propagator achieves MDD consistency as an edge e is always removed unless there exists a feasible solution to the inequality that supports it. We can also use the path length information to refine a node. For a node s we can partition the incoming edges $E^{in}(s)$ into different classes according to the value of

$$d_T(\text{tail}(e)) + f_{L(\text{tail})}(e) + d_1(s).$$

We can compute d_T for each node using a single top-down pass through the MDD, and similarly we can compute d_1 for each node using a single bottom-up pass through the MDD. Now we will frame these computations explicitly in terms of the operators used in the general framework.

Given a node s, let $E^{in}(s)$ denote its set of incoming edges. Let $\text{tail}(e)$ denote the tail of the edge e (the head of the edges in $E^{in}(s)$ is always s). Then

$$d_T(s) = \min\{d_T(\text{tail}(e)) + f_{L(s)-1}(e) \colon e \in E^{in}(s)\}.$$

Suppose our inequality is labeled as the constraint C. In our general framework the equation for $d_T(s)$ above is an instatiation of

$$I^C(s) = \bigoplus\{d_T(\text{tail}(e)) \otimes e \colon e \in E^{in}(s)\}$$

where for constraint C and information d_T,

- $I^a \otimes e = I^a + f_{L(\text{tail}(e))}(e)$, and
- $I^a \oplus I^b = \min\{I^a, I^b\}$.

Similarly, given a node s, let $E^{out}(s)$ denote its set of outgoing edges. Let $\text{head}(e)$ denote the head of the edge e (the tail of the edges in $E^{out}(s)$ is always s). Then

$$d_1(s) = \min\{d_1(\text{head}(e)) + f_{L(s)}(e) \colon e \in E^{out}(s)\}.$$

This is an instantiation of

$$I^C(s) = \bigoplus\{d_T(\text{head}(e)) \otimes e \colon e \in E^{out}(s)\}$$

where for constraint C and information d_1,

- $I^a \otimes e = I^a + f_{L(\text{tail}(e))}(e)$, and
- $I^a \oplus I^b = \min\{I^a, I^b\}$.

3.4.2 The General Framework

Suppose we are given a constraint C and want to compute some information I^C that is 'local' to each node in the MDD. This information can be used for filtering or refining. The computation is local in the sense in that it must be computable using the information available from its neighbors (incoming or outgoing). This is not much of a restriction since the information at each node may encode information about predecessors, successors, incoming or outgoing paths.

Let I^C be the set whose elements encode the local node information for a constraint C. Let E be the set of edges of the MDD. Then we need to define two operations (parameterized by the constraint and the type of information):

- $\otimes\colon I^C \times E \to I^C$, and
- $\oplus\colon I^C \times I^C \to I^C$.

A top-down pass of our scheme will look a shortest-path computation from the root and computes $I^C(s)$ for a node s only after the information for all its predecessors has been computed by setting

$$I^C(s) = \bigoplus \{d_T(\mathrm{tail}(e)) \otimes e \colon e \in E^{in}(s)\}.$$

Similarly, a bottom-up pass of our scheme will compute $I^C(s)$ for a node s after the information for all its successors has been computed by setting

$$I^C(s) = \bigoplus \{d_T(\mathrm{head}(e)) \otimes e \colon e \in E^{out}(s)\}.$$

A top-down (bottom-up) pass of the MDD visits each edge exactly once and so the passes themselves involve an amount of work that is linear in the size of the MDD (modulo the work required to compute $\oplus$ and $\otimes$ at each node).

Next, we present several instantiations of this framework to produce MDD propagation algorithms for different constraints. To simplify our presentation we will assume that each variable x_i is represented by layer i in the MDD.

3.4.3 Propagating $x_i = x_j$

We will focus on the top-down pass since the bottom-up pass will be similar. Without loss of generality assume that $i < j$. We define

$$I(s) \otimes e = \begin{cases} \emptyset, & L(s) < i \vee L(s) \geq j \\ e, & L(s) = i \\ I(s), & i < L(s) < j \end{cases}$$

and $I^a \oplus I^b = I^a \cup I^b$.

We delete an edge $e \in E(s,t)$ where $L(s) = j$ if $e \notin I(s)$.

It is easy to see that a single top-down and bottom-up pass will achieve MDD consistency. Applying this filtering scheme to an MDD of width one results in the traditional filtering applied to domain stores.

We can also refine the MDD using the information stored at each node. Say we are processing the MDD in a top-down pass and we encounter a node s with $|I(s)| = k > 1$. Then we can split s into $s^1, \ldots, s^k$ so that $|I(s^i)| = 1$ for all s^i. Observe that if we do this as part of our top-down pass then we will create several disjoint paths from x_i to x_j where each path corresponds to a single value in the domain of x_i. This type of refinement may be too extreme; in general, we want to bucket values intelligently, for example, by forming approximate equivalence classes.

Once we refine a node we recompute its information prior to processing its children. This scheme generalizes easily to propagating $f_i(x_i) = f_j(x_j)$ for functions f_i and f_j.

3.4.4 Propagating $x_i \neq x_j$

We will focus on the top-down pass since the bottom-up pass will look almost exactly the same. Without loss of generality assume that $i < j$. We define

$$I(s) \otimes e = \begin{cases} \emptyset, & L(s) < i \vee L(s) \geq j \\ e, & L(s) = i \\ I(s), & i < L(s) < j \end{cases}$$

and $I^a \oplus I^b = I^a \cap I^b$.

We delete an edge $e \in E(s,t)$ where $L(s) = j$ if $e \in I(s)$. Applying this scheme to an MDD of width one results in the traditional filtering applied to domain stores.

We can also refine the MDD using the information stored at each node. Say we are processing the MDD in a top-down pass and we are processing a node s for which the $\oplus$ operator results in $I(s) = \emptyset$ but for which $I(r) \neq \emptyset$ for some incoming neighbors, say $I(r_i) \neq \emptyset$ for $i = 1, \ldots, k$. Then we can split s into $s^1, \ldots, s^k$ so that $|I(s^i)| \neq \emptyset$ for all s^i, $i = 1, \ldots, k$. Observe that if we do this as part of our top-down pass then we will create several disjoint paths from x_i to x_j where each path corresponds to a single value in the domain of x_i. Again, this type of refinement may be too extreme; in general, we want to bucket values intelligently.

This scheme generalizes easily to propagating $f_i(x_i) \neq f_j(x_j)$ for functions f_i and f_j.

3.4.5 Propagating $x_i < x_j$

We will focus on the top-down pass since the bottom-up pass will look almost exactly the same. Without loss of generality assume that $i < j$. We define

$$I(s) \otimes e = \begin{cases} \emptyset, & L(s) < i \vee L(s) \geq j \\ e, & L(s) = i \\ I(s), & i < L(s) < j \end{cases}$$

and $I^a \oplus I^b = I^a \cup I^b$.

We delete an edge $e \in E(s,t)$ where $L(s) = j$ if $e \leq \min\{I(s)\}$. Notice that we only need to pass interval (bounds) information to filter an edge, and in fact, for (one-sided) inequalities we only need one of the boundaries. With this observation, we define

$$I(s) \otimes e = \begin{cases} \emptyset, & L(s) < i \vee L(s) \geq j \\ [e, e]\,, & L(s) = i \\ I(s), & i < L(s) < j \end{cases}$$

and $I^a \oplus I^b = [\min\{I^a, I^b\}, \max\{I^a, I^b\}]$. Let $\partial^-[a,b] = a$ and $\partial^+[a,b] = b$. Then we delete an edge from $e \in E(s,t)$ where $L(s) = j$ if $e \leq \partial^-(s)$. A single top-down and bottom-up pass achieves MDD consistency. Applying this scheme to an MDD of width one results in the traditional filtering applied to domain stores.

This scheme generalizes easily to propagating $f_j(x_j) \prec f_i(x_i) \prec f_{j'}(x_j)$ where $\prec$ is any total order on the common codomains of the functions $f_i, f_j, f_{j'}$. To achieve MDD consistency for a two-sided inequality we need a single top-down and bottom-up pass.

3.4.6 Propagating the `All-Different` Constraint

The `alldiff` constraint is one of the most commonly used global constraints in practical constraint programming models. It arises often in sequencing and scheduling problems which are problem domains for which constraint programming has been extremely effective. The `all-different` constraint `alldiff`$(x_1, \dots, x_n)$ requires that the variables $x_1, \dots, x_n$ take distinct values.

We can frame the `alldiff` propagator presented in [3] in our framework. First, we summarize that presentation. To each node u we attach four pieces of information for each `alldiff` constraint C: ImpliedUp, ImpliedDown, AvailUp, and AvailDown. Without loss of generality assume that $\mathrm{scope}(C) \subseteq \{x_i, x_{i+1}, \dots, x_j\}$. Then

$$\mathrm{ImpliedUp}_C(s) \otimes e = \begin{cases} \emptyset, & L(s) < i \vee L(s) \geq j \\ \mathrm{ImpliedUp}_C(s) \cup e, & \mathrm{var}(s) \in \mathrm{scope}(C) \\ \mathrm{ImpliedUp}_C(s), & i \leq L(s) < j \wedge \mathrm{var}(s) \notin \mathrm{scope}(C) \end{cases}$$

and $\mathrm{ImpliedUp}^a_C \oplus \mathrm{ImpliedUp}^b_C = \mathrm{ImpliedUp_C}^a \cap \mathrm{ImpliedUp_C}^b$.

We delete an edge $e \in E(s,t)$ where $L(s) \in \mathrm{scope}(C)$ if $e \in I(s)$. Next for AvailUp we have

$$\mathrm{AvailUp}_C(s) \otimes e = \begin{cases} \emptyset, & L(s) < i \vee L(s) \geq j \\ \mathrm{AvailUp}_C(s) \cup e, & \mathrm{var}(s) \in \mathrm{scope}(C) \\ \mathrm{AvailUp}_C(s), & i \leq L(s) < j \wedge \mathrm{var}(s) \notin \mathrm{scope}(C) \end{cases}$$

and $\mathrm{AvailUp}^a_C \oplus \mathrm{AvailUp}^b_C = \mathrm{AvailUp}^a_C \cup \mathrm{AvailUp}^b_C$. Now given some node s, consider the set of variables $X_s = \mathrm{scope}(C) \cap \{x_k \colon k < L(s)\}$. If $|X_s| = |\mathrm{AvailUp}_C(s)|$ (that is, X_s forms a Hall Set) the values in $\mathrm{AvailUp}_C(s)$ cannot be assigned to any variables not in X_s. This allows us to delete edges e with $\mathrm{tail}(e) = s$ such that $e \in \mathrm{AvailUp}_C(s)$.

Notice that ImpliedUp and AvailUp are computed during a top-down pass of the MDD. ImpliedDown and AvailDown are computed similarly during a bottom-up pass. In [3] the authors use this node information to design a refining scheme.

3.4.7 Propagating Two-sided Inequality Constraints

This is a generalization of the equality propagator described in [23]. Suppose we are given an inequality constraint $lb \leq \sum_{j \in J} f_j(x_j) \leq ub$. Let $m = \min\{J\}$ and $M = \max\{J\}$. We will store two pieces of information per node s: P_T, the set of all path lengths from **T** to s (computed during the top-down pass) and P_1, the set of all path lengths from s to **1** (computed during the bottom-up pass). Then during the top-down pass we set

$$P_T(s) \otimes e = \begin{cases} 0, & L(s) < m \vee L(s) \geq M \\ \bigcup\limits_{v \in P_T(s)} (v + e), & \text{otherwise} \end{cases}$$

and $I^a \oplus I^b = I^a \cup I^b$. The operations for P_1 are defined similarly. We can delete an edge e if and only if

$$\forall v_T \in P_T(\text{tail}(e)),\, \forall v_1 \in P_1(\text{head}(e))\colon\, v_T + e + v_1 \notin [lb, ub].$$

Notice that once we delete even a single edge, the information stored at all predecessors and successors becomes ‘stale’ and the information for these nodes must be recomputed in order to guarantee filtering that achieves MDD consistency. This follows by noting that deleting an edge results in path lengths that are a subset of the path lengths that existed when the edge is accounted for. Thus the filtering condition above with the stale information is weaker than it would be with the updated information. In other words, not updating node information as edges are deleted may result in an MDD is a valid relaxation of the MDD that would arise by updating the information.

However, we will achieve MDD consistency if every time we delete an edge we update the node information for all predecessors and successors and repeat this filtering and updating until we reach a fixed-point. This follows since the filtering condition above is both necessary and sufficient for an edge to be supported by a feasible solution.

If all the variable domains are binary and if $f_j(x_j)$ are restricted to a bounded set (e.g, $f_j(x_j) \in \{-1, 0, 1\}$ for all $j \in J$) then

- the maximum number of filtering iterations required until we reached a fixed point is at most the number of edges in the initial MDD, and
- computing $\oplus$, $\otimes$ and testing an edge can be done in polynomial time and space (relative to the size of the MDD).

It follows that iterating this algorithm until we reached a fixed point will achieve MDD consistency in polynomial time. This type of reasoning is a special case of a general principle that will be discussed in detail in Section 3.6.

3.4.8 Propagating `among` Constraints

The `among` constraint is a basic global constraint; it restricts the number of variables that can be assigned a value from a specific subset of domain values. Formally, if $X = (x_1, \ldots, x_q)$ is a sequence of variables, S a set of domain values, $0 \leq \ell \leq u \leq q$ constants then the constraint

$$\texttt{among}(X, S, \ell, u) = \{(d_1, \ldots, d_q) \colon d_i \in D(x_i) \, \forall i \in \{1, \ldots, q\}, \, \ell \leq |\{i \in \{1, \ldots, q\} \colon d_i \in S\}| \leq u\}.$$

We can reduce propagating $\texttt{among}(X, S, \ell, u)$ to propagating a two-sided separable inequality constraint,

$$\ell \leq \sum_{x_i \in X} f_i(x_i) \leq u,$$

where

$$f_i(v) = \begin{cases} 1, & v \in S \\ 0, & \text{otherwise}. \end{cases}$$

Notice that because each $f_i(\cdot) \in \{0, 1\}$ it follows that we can achieve MDD consistency in polynomial time.

However, this filtering is too slow in practice. Instead, we simply propagate bounds information. That is, we can use the inequality propagator for the pair of inequalities (reasoning on the shortest and longest path lengths). Explicitly, let $\mathrm{SP}(r, s)$ be the length of a shortest path from r to s, and $\mathrm{LP}(r, s)$ the length of a longest path, where the length is defined by $f_i(x_i)$ for all $x \in X$ (and $f_i = 0$ for all $x \notin X$). Then if node r is in a layer corresponding to a variable in X, we filter an edge in $e \in E(r, s)$ if

$$\mathrm{LP}(T, r) + f_{L(r)}(e) + \mathrm{LP}(s, 1) < \ell, \text{ or}$$
$$\mathrm{SP}(T, r) + f_{L(r)}(e) + \mathrm{SP}(s, 1) > u$$

These conditions are are sufficient but not necessary for determining the redundancy of an edge. In Section 5.1 we provide a small example demonstrating how this condition fails to remove a redundant edge.

The shortest and longest path information can also be used to help us refine nodes in the MDD. For example, we may regard two edges $e_1 \in E(r_1, s)$ and $e_2 \in E(r_2, s)$ as approximately equivalent for the `among` constraint if

$$\mathrm{LP}(T, r_1) + f_{L(r_1)}(e_1) = \mathrm{LP}(T, r_2) + f_{L(r_2)}(e_2) \text{ or},$$
$$\mathrm{SP}(T, r_1) + f_{L(r_1)}(e_1) = \mathrm{SP}(T, r_2) + f_{L(r_2)}(e_2).$$

Another approximation into equivalence classes is by considering the impact of an edge on the 'tightness' of an `among` constraint. That is, for each inequality defining the `among` constraint, we consider the 'slack' of an edge $e \in E(r,s)$ to be $\ell - (\mathrm{SP}(T,r) + f_{L(r)}(e) + \mathrm{SP}(s,1))$, respectively $u - (\mathrm{LP}(T,r) + f_{L(r)}(e) + \mathrm{LP}(s,1))$. The slack reflects the number of variables that can still be assigned to a value in S without violating the respective inequality. We say that an edge is 'tight' if its slack is at most a given threshold τ , and 'loose' otherwise. The equivalence classes (with respect to each inequality) then belong to all tight edges and all loose edges entering a node in the MDD.

3.4.9 Propagating the `sequence` Constraint

The sequence constraint is a generalization of the `among` constraint that states that at least ℓ and at most u values in S are assigned to *every* subsequence of q consecutive variables. Formally, let $X = (x_1, \ldots, x_n)$ be a sequence of variables, S a set of domain values, $0 \leq \ell \leq u \leq q \leq n$ constants then the constraint

$$\texttt{sequence}(X,S,q,\ell,u) = \bigwedge_{i=1}^{n-q+1} \texttt{among}(\{x_i,\ldots,x_{i+q-1}\},S,\ell,u).$$

So we can reduce propagating a `sequence` constraint to propagating its consituent `among` constraints. There is a loss of strength by using this reduction (cf. [58]) already when using domain propagation.

We can speed things up when propagating a `sequence` constraint using this reduction provided we are aware of the global structure. Instead of doing one top-down and bottom-up pass for each of the constituent `among` constraints we will do a single top-down and bottom-up pass for the `sequence` constraint essentially by 'gluing' together the shortest and longest path length information for each of the `among`s.

Since each variable $x_i \in X$ appears in at most q `among` constraints, each node in the MDD only needs to store the shortest and longest path information for the q `among` constraints that it is involved in. This drastically reduces the number of times we retrieve the same MDD information for processing (compared to serially processing each of the `among` constraints). Second, we are now in a position to use the information at each node to make more global refinement decisions. For example, we consider an edge 'tight' for the `sequence` constraint only if some prespecified number of constituent `among` constraints are 'tight' (in the sense of the previous subsection).

3.4.10 Propagating the Generalized Cardinality Constraint

The *generalized cardinality constraint* (`GCC`) is an extension of the `alldiff` constraint that counts how many variables take each of a given set of values. As with the `alldiff` constraint, this is an extremely useful constraint that appears in many constraint programming models. Formally, if $X = \{x_1, \ldots, x_n\}$ is a set of variables, $v = (v_1, \ldots, v_m)$ a vector of values, $\ell = (\ell_1, \ldots, \ell_m)$ a vector of lower bounds, and $u = (u_1, \ldots, u_m)$ a vector of upper bounds then the constraint

$$\begin{aligned}\texttt{gcc}(X, v, \ell, u) = \{(d_1, \ldots, d_n) \colon d_i \in D(x_i)\, \forall i \in \{1, \ldots, n\},\\ \ell_j \leq |\{i \in \{1, \ldots, n\} \colon d_i = v_j\}| \leq u_j\, \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}\}\,.\end{aligned}$$

Just as with the `sequence` constraint we can express the generalized cardinality constraint as several `among` constraints of a special type:

$$\texttt{gcc}(X, v, \ell, u) = \bigwedge_{i=1}^{m} \texttt{among}(X, \{v_i\}, \ell_i, u_i).$$

As with with `sequence` constraint we can leverage the global aspect of the constraint in the refining step. In fact we can do better than this by reusing the filtering technique based on network flows for `gcc` (cf. [26]). We will revisit this point in Section 3.5.

3.4.11 Propagating the `Unary Resource` Constraint

Consider the following setup: there are N activities to be scheduled on a single machine (resource). Each activity a_i has an earliest possible start time est_i, a latest possible completion time lct_i, and a processing time p_i. We will model the problem using N variables $X = (x_1, \ldots, x_n)$ where $x_i = a_j$ implies that activity j is the i^{th} activity to consume the resource.

Each node u in the MDD has three pieces of information: $est(u)$, $lct(u)$, and ImpliedUp(u). Given a node s an in outgoing edge e we define

$$est(s) \otimes e = \max\{est(s) + p_e, est_e\},$$

and $est^a \oplus est^b = \min\{est^a, est^b\}$. Similarly,

$$lct(s) \otimes e = \min\{lct(s) + p_e, lct_e\},$$

and $lct^a \oplus lct^b = \max\{lct^a, lct^b\}$. The ImpliedUp information is defined as for the `alldiff` constraint. We delete an edge $e \in E(r, s)$ if

$$[est_e, lct_e] \not\subseteq [est(r), lct(r)] \text{ or } e \in \text{ImpliedUp}(r).$$

3.4.12 Propagating the `element` Constraint

We will look at the simplest form of the `element` constraint: `element` $(x_i, (a_1, \dots, a_m), x_j)$ where the a_k are constants. This means that the variable x_j must take the x_i^{th} value in the list $(a_1, \dots, a_m)$, that is, $x_j = a_{x_i}$.

We will focus on the top-down pass since the bottom-up pass will look almost exactly the same. Without loss of generality assume that $i < j$. We define

$$I(s) \otimes e = \begin{cases} \emptyset, & L(s) < i \vee L(s) \geq j \\ e, & L(s) = i \\ I(s), & i < L(s) < j \end{cases}$$

and $I^a \oplus I^b = I^a \cup I^b$. We delete an edge $e \in E(s,t)$ where $L(s) = j$ if $e \notin I(s)$. It is easy to see that a single top-down and bottom-up pass will achieve MDD consistency. Moreover, when we use this filtering scheme for an MDD of width one we recover the filtering algorithm for domain stores that achieves domain consistency.

The information we store at each node can be used to refine the MDD much in the same way the information for $x_i = x_j$ is used to refine the MDD.

3.5 Reusing Domain Propagators

3.5.1 Motivation

There has been a lot of research dedicated to finding efficient domain store propagators. So one possible intermediate step to finding efficient MDD propagators may be to find an effective way to reuse domain propagators. We will present one such method based on the framework presented in the previous section.

We start by reviewing a method to reuse domain propagators presented in Section 5 of [3]. In this scheme the authors consider the family of MDDs $\{M_e\}$ for each edge e of the MDD M obtained by removing all paths in M not containing that edge. From each MDD M_e they compute its induced domain relaxation $D^{\times}(M_e)$ whose k^{th} component is defined by

$$D_k^{\times}(M_e) = \bigcup_{v \in \text{ layer } k} E^{out}(v).$$

For each domain relaxation $D^{\times}(M_e)$ the algorithm computes the simultaneous fixed-point D^{dom} of the domain propagators they wish to reuse. Then for each assignment $x_k = \alpha$ consistent with $D^{\times}(M_e)$ but not with D^{dom} the scheme places a no-good $x_k \neq \alpha$ on the edge e and deletes the edge e if $\mathtt{tail}(e) = k$ and $e = \alpha$. Otherwise, the scheme 'moves' the no-goods towards the layer in the MDD which corresponds to the variable they restrict, and are only allowed to move past a node if all paths through that node agree on the no-good. This ensures that no feasible solutions are removed. The authors note that this type of filtering will reach a fixed-point after a polynomial number of passes through the MDD, and thus applies each domain propagator a polynomial number of times.

The scheme we will be presenting is closely related to the scheme presented in [3]. However, using our framework we will see that all the algorithms in the previous section can be understood as special cases of this particular setup, including the 'specialized filtering' algorithms for the inequality propagator and `alldiff` presented in [3]. It is important to note that the specialized algorithms presented earlier are more efficient since the information they store at each node is smaller and faster to compute than the information used by the algorithm that follows.

3.5.2 Using Domain Information

Instead of using information tailored to specific constraints through the MDD using $\otimes$ and $\oplus$ (e.g., shortest paths, time intervals, etc.), we will explicitly use domain information. For each node s let M_s be the MDD obtained by removing all paths in M not containing s. We will be interested in incrementally constructing (via the top-down and bottom-up passes) an induced domain relaxation $D^{\times}(M_s)$ whose k^{th} component is given by

$$D_k^{\times}(s) = \bigcup_{v \in \text{ layer } k} E^{out}(v).$$

We note that the 'node' induced domain relaxation $D_{\times}(M_s)$ is the union the 'edge' domain relaxations $D^{\times}(M_e)$ of [3] for all edges leaving s.

Consider a node s on layer k of the MDD. The top-down pass will compute the 'prefix' information for $D^{\times}(M_s)$, that is, it will compute $D_i^{\times}(M_s)$ for $i < k$. The bottom-up pass will compute the 'suffix' information of $D^{\times}(M_s)$, that is, $D_i^{\times}(M_s)$ for $i > k$.

Let us examine the top-down pass and define $\otimes$ and $\oplus$ for a node s on layer k. The information $I(s)$ stored at each node consists of n components, where each component (a set) corresponds to the domain of the induced domain relaxation for the node. Then for an edge $e \in E^{out}(s)$

$$I(s) \otimes e = (I(s), e, I(\text{head}(e)))$$

and $\oplus$ is the componentwise union, that is,

$$I^a \oplus I^b = (I^a_1 \cup I^b_1, \ldots, I^a_n \cup I^b_n).$$

The operations for the bottom-up pass are defined similarly. Now we have the option of reusing domain propagators in the same manner as [3]. For each induced domain relaxation $D^\times(M_s)$ the algorithm computes the simultaneous fixed-point D^{dom} of the domain propagators that we wish to reuse. Then for each assignment $x_k = \alpha$ consistent with $D^\times(M_s)$ but not with D^{dom} we place a no-good $x_k \neq \alpha$ on the node s and delete an edge α if $L(s) = k$ and $\alpha \in E^{out}(s)$. Otherwise, we 'move' the no-goods towards the layer in the MDD which corresponds to the variable they restrict, and are only allowed to move past a node if all paths through that node agree on the no-good. This ensures that no feasible solutions are removed.

3.5.3 A Faster Framework for Reusing Domain Propagators

Recall that our specialized algorithms tend to filter edges after computing information from a single top-down and bottom-up pass. How can we reconcile this with passing no-goods carefully to ensure that feasible solutions are not discarded?

A domain propagator p can be viewed as a function that maps domains to domains. In practice, domain propagators are restricted to being *monotone* ($D_1 \subseteq D_2 \Rightarrow p(D_1) \subseteq p(D_2)$) and *decreasing* ($p(D) \subseteq D$) in order to make constraint propagation well-behaved. Finally, propagators must implement relaxations of the constraints they are modeling, that is, they may not remove any assignment that is supported by a feasible solution. Such propagators are called *correct*.

By computing a single bottom-up and top-down pass we 'cache' the induced domain relaxations for each node. As soon as we filter a single edge, the induced domains relaxations for all paths involving that edge becomes 'stale'. However, provided that the domain propagator is well-behaved in the manner described earlier, we can still delete edges based on the stale information without worrying about removing feasible solutions. The price we pay for using stale information for domain propagators that are correct, monotone and decreasing is weakened filtering, that is, by using stale information we may not filter edges that would otherwise be filtered with updated domain information.

In fact, the passing of no-goods in the previous schemes is nothing else but an efficient way of updating the stale domain information in order to make stronger inferences. Observe that updating the no-goods consistently (that is, only moving a no-good past a node if all paths through that node agree on the no-good) is just implementing the $\oplus$ operator of the last section in disguise. Recall that the $\oplus$ operator implements the componentwise union of domains. Since passing a no-good is essentially passing 'complemented' domain information, the equivalent $\oplus$ operation becomes the componentwise intersection of (complemented) domains.

3.5.4 The Relationship with 'Specialized' Propagators

How can we relate reusing domain propagators to all the previous 'specialized' schemes for MDD filtering? In each of the specialized algorithms the information stored at a node is a constraint specific 'summary' of the domain relaxation induced by that node.

To make this concrete, let us consider the example of propagating the `among` constraint: `among`$(X, \{1\}, \ell, u)$, where $X = (x_1, \dots, x_n)$ and $D(x_i) \subseteq \{0, 1\}$ for all $x_i \in X$. Domain propagation for this constraint is easy. Let f_0 and f_1 denote the number of variables whose domains are fixed to zero and one respectively. Then

- If $f_1 > u$ or $\ell < n - f_0$ then the constraint is inconsistent.
- If $f_1 = u$ then remove 1 from the domains of all unfixed variables.
- If $f_0 = n - \ell$ then remove 0 from the domains of all unfixed variables.
- Otherwise the constraint is domain consistent.

Recall that for each node in the MDD our propagation scheme for `among` computed the shortest and longest path from the root to the sink involving that node. In an MDD of width 1 there is but a single path from the root to the sink. Thus, every node on the path will have the same information: the shortest and longest path from the root to the sink. But the shortest path and longest paths from the root to the sink are f_1 and $n - f_0$ respectively. Moreover, the MDD edge filtering conditions for an edge e reduce to testing whether $f_0 + e < n - \ell$ or $f_1 + e > u$. And so, the specialized MDD edge filtering scheme is using the same inference technique as the domain relaxation filtering scheme for `among` but with much less overhead (computing and storing the shortest and longest path lengths instead of the domain relaxation information).

Every specialized MDD filtering scheme presented in Section 3.4 can be interpreted this way. The information stored at each node summarizes domain relaxation information in a way that is sufficient for filtering edge domains. This can be much more efficient in terms of computation time and memory. However, a key advantage of storing domain information as opposed to constraint specific information is that the domain information can be computed once and used as input to *several* existing domain store filtering algorithms for various constraints.

3.5.5 A Scheme for the Partial Updating of Node Information

As the authors of [3] point out, propagating filtering schemes based on domain propagators to achieve a simultaneous fixed-point in the MDD will very costly. Indeed, in our own experiments we have noticed that there is definitely a cost-benefit tradeoff with applying more 'agressive' propagation. This idea is already present and addressed in modern constraint programming systems based on domain stores. Constraint programming solvers typically employ nontrivial scheduling systems for domain propagation (cf. [56]).

Our current approach for MDD propagation involves two passes of the MDD. The first pass is a bottom-up pass that caches suffix information. We perform no refinement or filtering during the bottom-up pass. During the top-down pass after we compute the top-down information for a node we can filter the nodes outgoing edges and split the node if we choose to. [1]

In order to limit the amount of information updating (equivalent to passing no-goods) we restrict ourselves to filtering only the outgoing edges of a node we are processing even if our filtering algorithm indicates that we are able to reduce the domains of other variables in induced domain relaxation of that node. This ensures that we only have to update the top-down (prefix) information of the node we are currently processing.

Let us be more explicit: suppose we are processing a node s during the top-down pass and we run our filtering algorithm on the induced domain relaxation of s, $D^{\times}(s)$. Suppose we want to filter a value from the domain of a variable x_k whose index differs from $L(s)$. Then we would need to know exactly which nodes on layer k are connected to s and update only those nodes, otherwise we may remove feasible solutions. Even if we cache predecessor/successor information along with the domain information, we will eventually need to update this connectivity information which involves work similar to propagating a no-good from s towards layer k.

[1] We can just as easily start with a top-down pass and filter/refine during a bottom-up pass.

So, by allowing ourselves to filter only outgoing edges of s we are trading the ability to make stronger inferences at a much higher cost for weakened filtering with no additional updating of information. Notice that while suffix information may become very stale during the top-down pass, the prefix information is more current since a node's prefix information is computed after all its predeceeding nodes have filtered their outgoing edges.

Refining is also simple in this setup. Suppose we want to split the node t that we are currently processing (during the top-down pass) into the nodes t_1 and t_2. Both t_1 and t_2 will have t's suffix information and can compute their prefix information from t's prefix information and $E^{in}(t)$.

Remark 3.5.1. If we want to reuse domain propagators in this scheme we may want to modify them slightly. When processing a node, the algorithm outlined above needs to know whether a particular domain value (edge) in the induced domain relaxation (of the current node being processed) is supported by a feasible solution. Algorithms that answer this query may be computationally more efficient that traditional domain store filtering algorithms that attempt to filter all variable domains simultaneously.

Remark 3.5.2. There are situations in which we may want to relax the restriction of only filtering a node's outgoing edges as described above. For example, if our filtering algorithm indicates that we can reduce the domain of a variables whose layer is close to the current layer being processed we may want to filter the corresponding edges, since the work required to do this correctly is a function of the product of the distance between layers and the maximum width of the MDD.

3.6 Achieving MDD consistency

In this section we restrict ourselves to filtering a given MDD without refining. Clearly achieving MDD consistency is at least as hard as achieving domain consistency. In [3] the authors demonstrate that although the `alldifferent` constraint has a polynomial-time algorithm that achieves domain consistency it is NP-hard to achieve MDD consistency on an MDD of polynomial size.

Let us consider what happens when we iterate our general scheme above for a single constraint until we reach a fixed point. Assume that testing an edge requires time and space that is bounded by a polynomial in the size of the MDD. In particular, $\otimes$ and $\oplus$ require polynomial time and space (which is the case for computing induced domain relaxation information). Then each top-down and bottom-up pass will require work that is polynomial in the space and size of the original MDD. Iterating the algorithm to a fixed-point requires that each iteration delete at least one edge. Thus the number of (top-down and bottom-up) iterations required to reach a fixed-point is bounded by the number of edges in the MDD. It follows that the time and work required to propagate a constraint in our framework to a fixed-point is a polynomial function of the size of the MDD.

This has an important consequence. Since achieving MDD consistency for a constraint may be NP-hard, it follows that achieving a fixed point in the MDD using our framework may not result in MDD consistency for a constraint (provided P $\neq$ NP). This is true even when we reuse domain propagators that achieve domain consistency.

However, there are cases in which propagating to a fixed-point is sufficient to achieve MDD consistency. This idea is related to the standard 'shaving' technique in the constraint programming literature. If our filtering scheme for a constraint is strong enough to delete an edge if and only if the edge is not supported by any feasible solution then propagating this constraint to a fixed-point will achieve MDD consistency. In general, this is a much stronger requirement than deleting an edge if and only if it is not witnessed by a feasible solution of the *MDD relaxation induced by the edge*. Observe that reusing domain information essentially discards order dependedent (path dependent) information. This loss is expected in a sense when designing efficient algorithms since a given MDD may have exponentially many paths (relative to its size).

For example, let us revist the filtering scheme presented in Section 3.4.7 for propagating two-sided inequalities to a fixed-point, that is, the constraint

$$lb \leq \sum_{j \in J} f_j(x_j) \leq ub.$$

First we consider the case where all variable domains are binary and the codomains of all the separable functions f_j are restricted to belong to a bounded set (e.g., $f_j(x_j) \in \{0, 1\}$ for all $j \in J$). An edge is deleted if and only if it is not witnessed by any feasible solution. Computing $\otimes$ and $\oplus$ requires polynomial time and space relative to the size of the MDD. Testing an edge, however, is equivalent to solving a subset-sum problem. So when the domains of the variables are all binary, this test can be done in polynomial time. When the domains are not binary then testing an edge can be done in pseudo-polynomial time. In the former case by iterating the filtering algorithm to a fixed-point we achieve a polynomial time filtering algorithm.

3.7 Primal Heuristics and Branching Strategies

So far we have concentrated on ‘dual’ side of using MDDs, that is, using the MDD as a mechanism to prove feasibility or infeasibility of a problem. Since the MDD is a relaxation of the feasible set, we can explore the uses of an MDD from a primal perspective.

For each constraint we define a *violation* function that measures how much a potential solution violates the constraint. This measure should be normalized across constraints. We can now employ any of the plethora of local search techniques to explore the MDD to find a feasible solution.

In cases where the local search procedure fails to find a feasible solution we can still leverage the information gained to guide branching. We simply branch in the search tree in a manner that moves us towards the best solution found by the local search procedure.

3.7.1 MDD-Based Constraint Optimization and Strong Branching

A framework for MDD-based constraint optimization is presented in [3]. A separable objective function $\sum_i f_i(x_i)$ can be minimized over an MDD using a single shortest-path computation (in the same spirit as propagating an inequality constraint). Since, an MDD is a relaxation of the space of optimal feasible solutions the shortest-path calculation provides a lower bound on the optimal value. Thus we can use MDD relaxation in a branch-and-bound scheme to solve optimization problem.

We can adapt strong branching in this setting. Once we have finished processing a search tree node we create several temporary copies of the the node’s MDD. Now we ‘explore’ a branching choice on each copy of the MDD but only by doing some minimal amount of propagation (this step needs to be fast). We can quickly compute a bound on the objective function for each partially propagated branching choice. We then branch according to the choice that shows the most promise.

3.8 Conclusion

We have presented a general framework for propagating constraints in an MDD. We have described specialized filtering procedures for several important classes of global constraints. We also provide a systematic way of reusing domain store propagators within this framework and provide several alternatives for embedding this technique within a constraint solving system. An interesting corollary is that *all* the specialized algorithms presented, both old and new, can be understood as more efficient implementations of the technique of reusing currently existing domain store propagators.

Next, we presented a short note on the complexity of our framework. Iterating our scheme to a fixed-point requires a number of iterations that is bounded by the number of edges in the MDD. By providing sufficient conditions on the strength of filtering we show that certain domain propagation techniques in our framework will achieve MDD consistency in polynomial time.

Finally, we provide some thoughts on how to use MDDs to augment branching strategies and how to incorporate MDDs in primal heuristics for solving both constraint satsification and optimization problems.

Chapter 4

An MDD-based Constraint Programming System

4.1 Introduction

Our goal is to design a general purpose finite-domain MDD-based constraint programming system. This chapter outlines some of the basic design choices made for our system as well as the motivations for these choices.

4.2 Working with Finite-Domains

First, let us fix some terminology. A *domain* is a finite set of integers. A domain is *failed* if it is empty. A domain is *fixed* if it is a singleton. The intersection (union) of two domains is simply their set-theoretic intersection (resp. union). Finally, domain D_1 is *stronger* than D_2 if $D_1 \subseteq D_2$. We will use interval (or range) notation $[\ell, u]$ to represent the set of consecutive integers $\{x \in \mathbb{Z} \colon \ell \leq x \leq u\}$.

Recall our setup for MDDs: each edge leaving layer i is labeled with an element of the domain $D(x_i)$ of x_i, and no label occurs more than once on the edges leaving any given node. The set $E(p, q)$ of edges from node p to node q may contain multiple edges, and we denote each with its label.

An edge with label v leaving a node in layer i represents an assignment $x_i = v$. Each path in the MDD from **T** to **0** or **1** can be denoted by the edge labels $v_1, \ldots, v_n$ on the path and is identified with the assignment $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$.

In our implementation we chose to implement MDDs as simple directed graphs, that is, we disallow multiple edges between nodes. Instead we identify the multiple edges and aggregate the labels into what we call *edge-domains*. In an MDD of width one, the edge-domains are the variable domains.

There are several popular representations of domains; the two most common are ranges and bit-vectors. A *range sequence* for a finite set of integers I is the shortest sequence of disjoint intervals

$$s = ([b_1, e_1], \ldots, [b_k, e_k]),$$

with $b_i < b_{i+1}$ such that the intervals cover I (that is, $I = \cup_{i=1}^{k} [b_i, e_i]$). It follows that a range sequence is unique, consists of non-empty intervals and that $e_i + 1 < b_{i+1}$ for $1 \leq i < k$. A bit vector for a finite set of integers is simply a string of bits in which the i^{th} bit is set to 1 if and only if $i \in I$.

There are definitely space and time tradeoffs between bit vectors and range sequences and the operations that are needed to be performed on them. Range sequences are typically implemented using linked-lists (although balanced binary tree structures do exist but are typically deemed as ‘too heavy’) whereas bit vectors are usually implemented as consecutive words in memory with additional data to store the minimum and maximum values. In practice, range sequences are typically used for general purpose applications as they scale better.

We have decided to implement range sequences to represent edge-domains. The basic building block is the `INTERVAL` class which provides the operations: `min()`, `max()` and `contains(v)` which return the minimum value of the interval, the maximum value of the interval and true if `min`() $\leq v \leq$ `max`() and false otherwise.

The `DOMAIN` class implements a range sequence as an ordered linked-list of `INTERVALS`. We have provided fast implementations for common operations required of domains:

- `set_to_empty()`: set the domain to the empty set;
- `empty()`: is the set is empty?
- `size()`: returns the number of elements in the set;
- `contains(`v`)`: does the set contain v?
- `add()`: add an element/list of values/interval to the set;
- `remove()`: remove an element/list of values/interval from the set;

- `union()`: form the union with another set;
- `intersection()`: form the intersection with another set.

Since it is quite common during constraint processing to iterate over all possible values in a variable's domain, the `DOMAIN` class provides an `enumerator` object to facilitate such operation. The following methods describe the functionality of the `DOMAIN::enumerator` class:

- `empty()`: is the variable domain empty?
- `reset()`: reset the enumerator to the first element (if any);
- `at_end()`: is the enumerator at the last element?
- `value()`: returns the current value;
- `move_to_next()`: move to the next value in the domain.

4.3 The MDD Implementation

We decided to implement an MDD as a layered simple directed graph with adjacent nodes belonging to adjacent layers. Every edge connects two adjacent layers and has an edge domain. This differs from the 'theoretical' presentation in Chapter 3 where we allowed multiple edges between nodes and each edge was labeled with a distinct value.

The `NODE` Class

The `NODE` class has the following data:

- `index_`: a unique identifier that allows for efficient representations of functions from the set of nodes to pointers of data using arrays of small size;
- `layer_`: indicates the layer of the MDD that the node resides;
- `in_`: an array of pointers to the incoming edges;
- `out_`: an array of pointers to the outgoing edges.

It is important that each node quickly have access to both its incoming and outgoing edges for the various operations required by the MDD. There are a few methods in the class but they are used mainly for testing the correctness of the code.

The `EDGE` class

The `EDGE` class has the following data:

- `index_`: a unique identifier that allows for efficient representations of functions from the set of edges to pointers of data using arrays of small size;
- `tail_`: a pointer to the tail node of the edge;
- `head_`: a pointer to the head node of the edge;
- `domain_`: a pointer to the set of values associated with the edge.

The `POOL` class

The `POOL` class is a template class that helps implement efficient maps from pool objects to pointers of data. For example, if we have n nodes then we want to assign each node an unique index between $0 \leq \text{index} < n$. If we had a static set of nodes then this could be accomplished using a counter. Implementing a function from nodes to pointers of data can be implemented as an array of the pointers to the desired data.

We would like to keep this simple representation of functions but for a dynamic set of objects such as the nodes and edges in the MDD. Our simple approach is to keep a list of 'free' indices, that is, indices that were assigned to a node (or edge) but later become free to be re-assigned when their node (or edge) was deleted.

The `POOL` class has the following data:

- `data_`: an array of the templated type (for our purposes, pointers to `NODE` or `EDGE` objects);
- `free_index_store_`: a list of 'free' indices to be re-used in `data_`.

The `POOL` class provides the following interface:

- `insert(T)`: insert T (an object of the templated type) into the pool;

- `remove(index)`: mark index as free to be re-used. Note: this method does *not* free/delete the object stored at index;
- `size()`: return how many elements are in the pool.

Currently, the `POOL` class does not provide 'garbage collection' since synchronizing various maps/functions that use the indices provided by the pool would have to be notified and updated to be consistent with the result of the 'garbage collection'.

The `MDD` class

The `MDD` class has the following data:

- `num_vars_`: the number of variables (layers minus one);
- `max_width_`: the maximum number of nodes on any layer in the MDD;
- `nodes_`: a doubly-indexed array of pointers to nodes where the first index indicates the layer of the nodes. This allows us to quickly iterate over the nodes in a layer;
- `edges_`: a pool of pointers to `EDGE` objects;
- `domains_`: a cache to store the domain relaxation of the MDD;
- `node_pool_`: a pool of pointers to `NODE` objects.

Next we provide a description of some members of the `MDD` interface:

- `add_node(layer)`: add a node to layer (if possible);
- `add_edge(tail, head, domain)`: add an edge to the MDD;
- `delete_node(v)`: delete a node in the MDD;
- `delete_edge(e)`: delete an edge in the MDD;
- `terminal(v)`: is a node terminal in the MDD?
- `intersect_domain(v, domain)`: intersect the domains of the outgoing edges of v with `domain`. This is useful when implementing variable partition branching schemes;

- `get_domain_relaxation()`: compute and cache the domain relaxation of the current MDD;
- `create_domain_relaxation(domains)`: create the domain store relaxation given the domains of the variables;
- `cleanup_dangling_nodes()`: delete nodes and edges that do not belong to any path in the MDD from the root to the sink;
- `enumerate()`: enumerate all solutions (feasible and infeasible) encoded by the MDD.

Adding and deleting a node on a layer of the MDD always occurs at the end of the array. In particular, deleting a node from a layer may require swapping pointers so that the pointer to the node being deleted is at the end of the array. We use the same procedure for modifying a node's `in_` and `out_` arrays when deleting an edge. Thus, it is crucial to iterate over nodes in a layer carefully when performing operations that may modify the layer (similarly, a nodes incoming or outgoing edges). When performing batch operations that may delete nodes on a layer (e.g., filtering) you should typically iterate in reverse through the nodes in the layer. Similarly, when performing batch operations that may add nodes on a layer (e.g., refining) you should typically iterate in the forward direction through the nodes in the layer.

4.4 Specifying a Problem

In order to specify a constraint satisfaction (or constraint optimization) problem we have provided a simple interface for users. A user is able to specify his own variables, and constraints and are provided with a mechanism to add his own constraints. A user is able to specify the `GOAL` of the problem, which may be to find a single solution, find all solutions or find an optimal solution. We will provide a quick overview of this interface.

The `VARIABLE` class

This is a very simple class that encapsulates data associated with a CSP variable in our setup. The data in `VARIABLE` class includes:

- `label_`: every variable must be assigned a unique label (a string) that allows us to refer to that variable;

- `domain_`: the domain of the variable in the domain relaxation of the MDD;
- `mdd_index_`: the layer in the MDD that corresponds to the variable.

The `CONSTRAINT` class

This is a core class for the solver and will be discussed in detail in Section 4.5.

The `PROBLEM` class

The `PROBLEM` class has the following data:

- `status_`: is the problem feasible, infeasible or is its status currently unknown?
- `label_`: a string that serves as an identifier for the problem;
- `goal_`: indicates whether the solution procedure should look for a single feasible solution, look for all feasible solutions or solve an optimization problem;
- `solutions_`: a list of all feasible solutions found by the solver;
- `variables_`: a list of variables (pointers to objects of the `VARIABLE` class) specified by the problem;
- `constraints_`: a collection of constraints (pointers to objects of the `CONSTRAINT` class) specified by the problem;

The `PROBLEM` interface consists of the following methods:

- `add_variable(var)`: add the variable `var` to the problem;
- `add_constraint(con)`: add the constraint `con` to the problem;

Currently, a variable is assigned to the layer in the MDD in the order the variable is added to the problem. We may want to consider generalizing the interface to allow the user to specify a variable's layer in the MDD or provide some type of priority scheme to help guide the layer assignment.

4.5 The `CONSTRAINT` class

A constraint C on variables $\{x_1, \ldots, x_n\}$ with domains $\{D(x_i)\}_{i=1}^n$ is a subset of $D(x_1) \times \cdots \times D(x_n)$. An assignment of values $x_i = v_i \ni D(x_i)$ is *feasible* for C if $(v_1, \ldots, v_n) \in C$, otherwise we say that the assignment is *infeasible*. This is essentially an *extensional* definition of constraints which is a popular view of constraints in the constraint programming community. The set of variables $\{x_1, \ldots, x_n\}$ involved in C is known as the scope of the constraint C, denoted $\mathrm{scope}(C)$.

If an assignment $x_i = v$ can be extended to a feasible solution s of C we say that v is *supported* or *witnessed* by s (for C). The process of removing infeasible values from domains (values not supported by C) is known as *domain reduction* or *domain filtering*.

Clearly, in most cases it is impossible to work with a constraint presented extensionally. Instead, most constraints have some underlying structure that allows us to work with a constraint in a more tractable manner. Constraint solvers typically implement a constraint C using a collection of *domain filtering* algorithms. In MDDs constraints are implemented using a collection of *edge filtering* algorithms as well as *node-splitting* algorithms. [1]

Abstractly, we can view a filtering algorithm ϕ_c for a constraint C as a function that maps domains to domains. More precisely, given a constraint C with scope $\{x_1, \ldots, x_n\}$ and $D = \prod_{i=1}^n D(x_i)$ a filtering algorithm ϕ_c is a function $\phi_c \colon D \to D$. In order for our constraint propagation algorithms to be correct we require that the edge filtering algorithms are

- *monotone*: $D_1 \subseteq D_2 \Rightarrow \phi_c(D_1) \subseteq \phi_c(D_2)$, and
- *decreasing*: $\phi_c(D) \subseteq D$.

Finally, edge filtering algorithms must implement relaxations of the constraints they are modeling, that is, they may not remove any assignment that is supported by a feasible solution. Such filtering algorithms are called *correct*.

One important (and sometimes overlooked) function of constraints is that they can help guide our outer search process (see Section 4.6). Consider, for example, the most popular branching scheme: branching on an unfixed variable x_j in which the domain of x_j is partitioned into two or more subsets and the subproblems are created by restricting x_j to each of these subsets. We can think of a variable x_j belonging to a domain

[1] For MDDs we use the word *propagate* to mean both filtering and refining.

as a constraint on x_j and that branching on x_j is one 'function' of this 'domain constraint'. In general, we allow a user the capability for a constraint to create subproblems (restrictions) based on the MDD of the current subproblem (that is, branching on a constraint).

The `CONSTRAINT` base class

The `CONSTRAINT` base class is a virtual class (it cannot be instantiated) which is used as a guide for designing constraints in our solver. When a user creates a new constraint they need to augment the enumerated type `CONSTRAINT::TYPE` to include their constraint class. Currently, we have implemented the following constraints

- `DOMAIN_CONSTRAINT`: requires that a variable belong to its domain;
- `AT_MOST`: models the constraint $\sum_{i \in I} a_i x_i \leq b$;
- `AMONG_AT_MOST`: models the constraint $\sum_{i \in I} \delta_S(x_i) \leq b$, where $\delta_S(x_i) = 1$ if $D(x_i) \cap S \neq \emptyset$ and is equal to zero otherwise;
- `CARD_AT_MOST`: models the constraint $\sum_{i \in I} \delta_v(x_i) \leq b$, where $\delta_v = 1$ if $v \in D(x_i)$ and is equal to zero otherwise.

For (basic) scheduling purposes we have included a second enumerated type `CONSTRAINT::STATUS`. We list the possible `STATUS` values and their purpose:

- `READY`: the constraint is scheduled to be propagated;
- `SUSPENDED`: the constraint is not currently schedule to be propagated;
- `REDUNDANT`: the constraint may be safely removed from the problem without affecting feasibility.

The data for the `CONSTRAINT` class include:

- `label_`: a string (identifier) associated with the constraint;
- `variables_`: a list of pointers the the variables in the constriants scope;
- `status_`: what is the scheduling status for the constraint;
- `type_`: what is the constraint type;

- `min_support_index_`: the first layer in the MDD that the constraint is involved in;
- `max_support_index_`: the last layer in the MDD that the constraint is involved in.

The data `min_support_index_` and `max_support_index_` are useful to speed up propagation of a constraint during the top-down and bottom-up passes. For specialized propagators the filtering information required by a constraint will typically not need to be passed between layers outside the range [`min_support_index_`, `max_support_index_`].

The `CONSTRAINT` interface consists of the following methods:

- `feasible(s)`: is `s` a feasible solution to the constraint?
- `able_to_branch()`: does the constraint provide an implementation of `branch()`?
- `branch()`: returns a list of constraints that will provide the restrictions that define each branching subproblem along with an estimated score (similar to pseudo-costs) to help rank each subproblem during the outer search;
- `initialize_info()`: initialize the information that the constraint associates with each node of the MDD;
- `compute_incoming_info(M, v)`: compute the incoming information required by the constraint for the node `v` in the MDD `M` (this information is computed during the top-down pass);
- `compute_outgoing_info(M, v)`: compute the outgoing information required by the constraint for the node `v` in the MDD `M` (this information is computed during the bottom-up pass);
- `compute_refining_score(M, v)`: calculate a score that indicates how ‘valuable’ it is for this constraint to refine the node `v` in the MDD `M`;
- `refine_incoming(M, v)`: refine the edges incident to `v` in the MDD `M`;
- `filter_outgoing(M, v)`: filter the domains of the edges leaving `v` in the MDD `M`;
- `post_process(M)`: perform any post-processing required once the constraint has been propagated.

The DOMAIN_CONSTRAINT class

This class implements the (trivial) constraint $x_i \in D(x_i)$. This constraint class exists solely because of the generic way in which we branch (we only branch on constraints). There are several common branching strategies for variables which are captured by the enumerated type `DOMAIN_CONSTRAINT::STRATEGY`:

- `SPLIT_MIN`: create two branches, the first branch restricts the variable to the minimum value in the current domain and the second branch restricts the variable to the remaining values;
- `SPLIT_MAX`: create two branches, the first branch restricts the variable to the maximum value in the current domain and the second branch restricts the variable to the remaining values;
- `SPLIT_MID`: create two branches splitting the domain of the variable at its midpoint (one branch is restricted to the smallest values while the other branch is restricted to the largest values);
- `SPLIT_ALL`: create one branch for each value where each branch restricts the variable to exactly one of the possible values in its domain.

The key data required by the `DOMAIN_CONSTRAINT` class not inherited from the base class is `strategy_` which indicates the branching strategy. The interface implements the following methods:

- `able_to_branch()`: returns true if the variables current domain is larger than one and false otherwise;
- `branch()`: returns the constraints required to enforce the given branching strategy. The scores (pseudo-costs) returned are based on the reduction in the size of the variables domain for each restriction;
- `filter_outgoing()`: simply intersects each edge domain with its corresponding variable domain.

The AT_MOST class

This class is used to propagate the constraint

$$\sum_{i \in I} a_i x_i \leq b.$$

Recall that the information stored at each node s of the MDD for an inequality constraint consists of:

- $d_T(s)$: the length of the shortest path from the root **T** to the node, and
- $d_1(s)$: the length of the shortest path from the node to the sink **1**,

where the length of an edge-value pair (e, v), $e = (s, t)$ is given by $a_{L(s)}v$. In terms of the general framework (see Section 3.4.1) we compute the shortest-path from the root **T** during a top-down pass through the MDD using $\otimes$ and $\oplus$ defined as follows for an edge-value pair (e, v) where $e = (s, t)$

- $d_T(s) \otimes v = d_T(s) + a_{L(s)}v$, and
- $I^a \oplus I^b = \min\{I^a, I^b\}$.

Similarly, we compute the shortest-path to the sink **1** during a bottom-up pass through the MDD. We remove a value v from the domain $D(e)$ of an edge $e = (s, t)$ when every path through the edge-value (e, v) has a length greater than b, that is, when

$$d_T(s) + a_{L(s)}v + d_1(t) > b.$$

We also use the information d_T and d_1 to refine a node by considering the impact of an edge-value pair on the ‘tightness’ of the inequality constraint. That is, we consider the *slack* of an edge-value pair (e, v) where $e = (s, t)$ to be

$$b - (d_T(s) + a_{L(s)}v + d_1(t)).$$

An edge is *tight* if its slack is at most a given threshold and *loose* otherwise. The equivalence classes (with respect to each inequality) then belong to all tight edge-value pairs and all loose edge-value pairs entering a node in the MDD.

The data for this class includes:

- `coefficients_`: the coefficients a_i in the inequality;
- `rhs_`: the right-hand side of the inequality;
- `SP_from_root_`: an array that encodes the function $s \mapsto d_T(s)$;
- `SP_from_sink_`: an array that encodes the function $s \mapsto d_1(s)$;
- `edge_value_pairs_data`: stores the slack of all incoming edge-value pairs for a single node in the MDD (this is a temporary variable that is used every time a node in the MDD is processed by the constraint);

- `refining_score_`: indicates how valuable it is for the solver to let the constraint refine a node in the MDD (this is a temporary variable that is used every time a node in the MDD is processed by the constraint).

The interface includes all the methods from the base class `CONSTRAINT` and adds a single new method `f()` that will be overloaded by the subclasses `AMONG_AT_MOST` and `CARD_AT_MOST`, since these constraints can be described as

$$\sum_{i \in I} f(a_i x_i) \leq b.$$

for a suitable definition of f. In the `AT_MOST` class the function f is just the identify, that is, $f(x) = x$ for all x.

The `AMONG_AT_MOST` class

The `AMONG_AT_MOST` class is a subclass of the `AT_MOST` class in which the function `f` has been redefined. Given a set S the defining inequality for this constraint is

$$\sum_{i \in I} f_s(x_i) \leq b,$$

where $f_s(x_i) = 1$ if $x_i \in S$ and is equal to zero otherwise. The data for this class that augments the `AT_MOST` class is:

- `domain_`: the set S used to define f_s above.

The method `f` implements the function f_s.

The `CARD_AT_MOST` class

The `CARD_AT_MOST` class is a subclass of the `AT_MOST` class in which the function `f` has been redefined. Given a value v the defining inequality for this constraint is

$$\sum_{i \in I} f_v(x_i) \leq b,$$

where $f_v(x_i) = 1$ if $x_i = v$ and is equal to zero otherwise. The data for this class that augments the `AT_MOST` class is:

- `value_`: the value used to define f_v above.

The method `f` implements the function f_v.

4.6 Constraint-Based Search

Our outer search algorithm uses the traditional recursive divide-and-conquer strategy of branching search. Given a problem P that is too difficult to solve as given (typically after constraint propagation) the branching algorithm creates a series of *restrictions* (or *subproblems*) $P_1, \ldots, P_k$ whose union contains P (the restrictions should be *exhaustive*). Ideally, the restrictions are disjoint but we do allow them to overlap. In this case, we say that we have *branched* on P. Next, the search algorithm attempts to solve each restriction. If some branch P_i is solved then

- the solution process is terminated with a feasible solution if our goal is to find a feasible solution, or
- the solution is added to the list of feasible solutions for P if our goal is to find all feasible solutions, or
- the solution becomes the incumbent if it is better than the previous incumbent solution.

If a restriction is too difficult to solve then the search procedure branches further on the restriction. This solution process continues recursively. To ensure that this procedure terminates, the branching mechanism must be designed in such a manner so that the problems become more tractable as the number of restrictions increase.

The `SEARCH_NODE` class

The `SEARCH_NODE` class stores data required to describe a restriction of a problem after some number of branching steps. This class includes an enumerated type `SEARCH_NODE::STATUS` that indicates that the current search node (subproblem) is:

- `FEASIBLE`: the subproblem has only feasible solutions, or
- `INFEASIBLE`: the surproblem has only infeasible solutions, or
- `BOTH`: the subproblem has both feasible and infeasible solutions, or
- `UNKNOWN`: the solutions defined by the subproblem have not been evaluated.

The data for this class includes:

- `M_`: a pointer to the MDD used by the search node (subproblem);
- `variables_`: a container with pointers to the variables in the subproblem;
- `constraints_`: a container with pointers to the constraints defining the subproblem;
- `branching_constraint_`: a pointer to the constraint used to restrict the parent and form the subproblem;
- `processed_`: indicates whether the search node has been processed;
- `status_`: the status of the solutions encoded by the subproblem;
- `score_`: the score used by the node selection procedure of the outer search algorithm.

The interface for the `SEARCH_NODE` class includes the following methods:

- `update_variable_domains`: update each variable's domain to reflect the current state of the MDD (e.g., post-filtering);
- `terminal(level)`: returns true if there are at most `level` unfixed variables in the MDD or if the subproblem is infeasible and false otherwise;
- `calculate_score()`: compute the actual score for this node (should be invoked post-processing) to provide more accurate scoring information for subproblems derived from this node;
- `score_first_fail()`: computes the sum of the cardinalities of all variable domains. This scoring strategy results in a first-fail node selection strategy for the outer search procedure;
- `count_unfixed()`: counts the number of unfixed variables.

The `BRANCHING_STRATEGY` class

The `BRANCHING_STRATEGY` base class is a virtual class (it cannot be instantiated) which is used as a guide for designing branching strategies in our solver. When a user develops a new branching strategy he needs to augment the enumerated type `BRANCHING_STRATEGY::STRATEGY` to include his strategy. Currently, we have implemented the following common branching strategy:

- `VD_FIRST_FAIL`: branch on the variable with the smallest unfixed domain.

The interface consists of a single method:

- `get_branching_constraint(search_node N)`: given the search tree node `N` find the constraint that will be used to create the restrictions. Recall that each constraint 'knows how to branch on itself' (see Section 4.5) for details.

The `VD_FIRST_FAIL` class

The `VD_FIRST_FAIL` class is the subclass of `BRANCHING_STRATEGY` that implements variable domain first fail branching heuristics. It chooses to branch on the variable with the smallest domain. The class provides an enumerated type `VD_FIRST_FAIL::BREAK_TIES` that the user can set to indicate how to break ties. There are currently two tie breaking rules, although a user can easily augment the list of rules:

- `LEX_FIRST`: choose the variable with the smallest index;
- `RANDOM`: choose a random variable.

The data for the class consists of the sole member:

- `rule_`: indicates the tie-breaking rule.

The interface consists of the method:

- `get_branching_constraint(search_node N)`: given the search node `N` returns the `VARIABLE_DOMAIN` constraint to use for branching.

4.6.1 The `SOLVER` class

The data for the `SOLVER` class includes:

- `problem_`: the problem to be solved;
- `initial_mdd_`: a pointer to the initial MDD, typically this will be an MDD of width 1;
- `branching_strategy_`: a pointer to the branching strategy to use during the search procedure;
- `termination_level_`: the number of unfixed variables to declare a search node terminal and trigger an enumeration of the MDD;

- `number_feasible_solutions_`: the number of feasible solutions found by the solver;
- `total_nodes_created_`: the total number of search tree nodes created by the solver;
- `total_nodes_processed_`: the total number of search tree nodes that were processed by the solver;
- `number_choice_points_`: the number of non-terminal nodes in the search;
- `number_infeasible_nodes_`: the number of terminal nodes in the search that did not contain a feasible solution;
- `mdd_failures_`: the total number of infeasible solutions enumerated at terminal nodes where all variables have nonempty domains;
- `solution_time_`: the total time required by the solver;
- `Q_`: a priority queue containing the unprocessed search nodes ranked by their score.

The interface for the `SOLVER` class includes:

- `initialize()`: initializes the data used by the solver;
- `setup_mdd()`: creates an MDD of width 1 (the domain relaxation) given the variable domains;
- `enumerate_node()`: enumerates all solutions in the MDD checking each for feasibility;
- `process_node_serial()`: process a search node serially, that is, process the MDD for each constraint one at a time and pass the resulting MDD to the next constraint (see Algorithm 2);
- `process_choice_point()`: update the search node's score, create subproblems based on the branching constraint and initialize the scheduling information for propagating the constraints in each subproblem (see Algorithm 4);
- `process_branching_constraint()`: process the branching constraint (this method is is almost exactly like `process_constraint()`;
- `process_constraint()`: process a constraint using our general framework, that is, perform a bottom-up pass to compute 'suffix' information and then filter and refine during the top-down pass (see Algorithm 3);

- `solve()`: run the outer search procedure according to the parameters that the user has set (see Algorithm 1).

Algorithm 1: SOLVER::solve()

```
repeat
    $N \leftarrow Q$.pop()
    process_node_serial (N)
    $D \leftarrow D^{\times}(M)$ ;                    // find the domain store relaxation of $M$
    if $\nexists i$ such that $D_i = \emptyset$ then
        $BC \leftarrow$ branching_strategy.get_branching_constraint($N$)
        if at a choice point then process_choice_point (N, BC)
        else enumerate_node (N)
    if goal is to find one solution and we have found one then break;
until $Q$ is empty
if found a feasible solution then return true
else return false
```

Algorithm 2: SOLVER::process_node_serial(N, BC)

```
Data: search node $N$ and branching constraint $BC$
$not\_infeasible \leftarrow$ true
if $BC$ is nonempty then $not\_infeasible \leftarrow$ process_branching_constraint($N$)
if $not\_infeasible$ then
    initialize_active_constraints ()
    foreach active constraint $C$ do
        if process_constraint ($N, C$)==false then
            return false
return true
```

4.7 Conclusions and Future Work

Our goal was to design a basic constraint programming system in which the domain store has been replaced by an MDD store. We feel that we have achieved this goal. The system is fast and general enough to allow a user to quickly add filtering and refining methods for constraints, modify the branching strategy and overall search procedure and to easily evaluate their ideas.

Algorithm 3: SOLVER::process_constraint(N, C)

```
Data: search node $N$, constraint $C$
$M \leftarrow$ the MDD in $N$
$C$.initialize_info ($M$)
foreach layer $L$ do
    $M$.cleanup_dangling_nodes ($L$)
    foreach vertex $v \in L$ do
        $C$.compute_outgoing_info ($M, v$) ;                // compute suffix information
    if $|L| == 0$ then return false
foreach layer $L$ do
    $M$.cleanup_dangling_nodes ($L$)
    foreach vertex $v \in L$ do
        $C$.compute_incoming_info ($M, v$) ;                // compute prefix information
        if $L$ has space and should refine $v$ then
            $C$.refine_incoming ($M, v$)
            $C$.compute_incoming_info ($M, v$)
    foreach vertex $v \in L$ do
        $C$.filter_outgoing ($M, v$)
    if $|L| == 0$ then return false
return true
```

Algorithm 4: SOLVER::process_choice_point(N, BC)

```
Data: search node $N$, a constraint to branch on $C$
N.calculate_score () ;    // compute actual score to give children accurate baseline for their est. score
(branching_constraints, est. scores) $\leftarrow$ C.branch ()
foreach branching constraint $BC$ do
    create a search node $N_{BC}$ for $BC$
    add $BC$ to $N_{BC}$'s list of constraints
    update scheduling information for constraints in $N_{BC}$
    $Q$.push($N_{BC}$)
```

There are several avenues to speed up the solution process. First, we can add primal heuristics to help find feasible solutions early in the search. This will be extremely important in order to be competetive with state-of-the-art constraint programming systems for solving general CSPs.

Second, we can add a scheduling system that decides when to propagate constraints. This is an important step in developing a full-fledged constraint programming system since propagating every constraint at each search tree node may not be worthwhile.

A third avenue is to modify the solver to have a queue of ‘choices’ instead of fully instantiated search nodes. A choice will consist of a pointer to a parent node and a constraint branch. Each choice point registers with the parent node (to do reference counting). Then when we pop the head of the queue we fully instantiate the child search node and unregister the choice with the parent (the parent can delete itself when it no longer has any choices registered with it). The space required by this can be much less than instantiating search nodes every time we branch. Instantiating search nodes prior to processing is not much of a problem for certain search strategies (such as depth-first search) but for other search patterns one starts paying a price in terms of memory usage. Finally, when creating a search node we perform ‘deep-copies’ of the variables, constraints and MDDs. We can instead record ‘deltas’ for these objects, that is, how they differ from the original definitions of the constraints, variables and predecessors’ MDDs.

Each of these approaches requires a substantial amount of work both in terms of developing a theory and an effective implementation and are open research problems for MDD-based constraint solving systems.

Chapter 5

Propagating Among Constraints

In this chapter we study MDD-based propagation for `among` constraints, which are of central importance in employee scheduling and production sequencing problems. Recall that the `among` constraint can be written as

$$\texttt{among}(X, S, \ell, u) \tag{5.1}$$

where I is an index set, X is a set of variables $\{x_i \mid i \in I\}$, S is a set of domain elements, and $0 \leq \ell \leq u \leq |X|$. The constraint requires that at least ℓ and at most u of the variables in X take a value in S. Thus if we let $\delta(v)$ be 1 when $v \in S$ and 0 otherwise, the `among` constraint requires that

$$\ell \leq \sum_{i \in I} \delta(x_i) \leq u \tag{5.2}$$

We experimentally demonstrate that search tree reduction and computation time, as compared to the traditional domain store, can be dramatically reduced already for MDDs of relatively small width.

Interestingly, huge savings in computation time are possible particularly for the more difficult problem instances that we considered. For example, to solve one specifically hard instance, the domain store needed 1,012,562 backtracks and 1684.7 seconds of computation time, while our MDD store with maximum width of just 4 reduced this to 2 backtracks and 0.04 seconds. This clearly shows the benefit and potential of MDD-based propagation for `among` constraints.

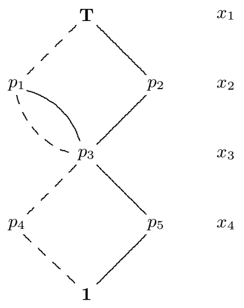


Figure 5.1: An MDD in which the solid edge from p_1 to p_3 (representing $x_2 = 1$) is redundant for the constraint $\texttt{among}((x_1, x_2, x_3, x_4), \{1\}, 2, 2)$.

5.1 MDD Filtering Heuristics for Among

Although we can filter a MDD for `among` in polynomial time (see Section 3.4.8), the computational effort may not be justified. It is faster to apply a simple sufficient condition for removing an infeasible edge. For an `among` constraint defined on a set of variables X, let SP(r, s) be the length of a shortest path from r to s, and LP(r, s) the length of a longest path, where the length of an edge is given by its labels (either 0 or 1). Then if node r is in a layer corresponding to a variable in X, we filter an edge in $e \in E(r, s)$ if

$$\begin{gathered} \text{LP}(T, r) + \delta(e) + \text{LP}(s, 1) < \ell, \text{ or} \\ \text{SP}(T, r) + \delta(e) + \text{SP}(s, 1) > u \end{gathered} \tag{5.3}$$

We update LP and SP after each edge is deleted.

A small example shows that (5.3) is not a necessary condition for redundancy of an edge. The solid edge from p_1 to p_3 in Fig. 5.1 is redundant for

$$\texttt{among}((x_1, x_2, x_3, x_4), \{1\}, 2, 2) \tag{5.4}$$

but fails to satisfy (5.3).

A still faster heuristic postpones updating LP and SP until all edges are tested. It uses a variation of (5.3):

$$\begin{gathered} LP_+(\mathbf{T}, r) + \delta(e) + LP_+(s, \mathbf{1}) < \ell \quad \text{or} \\ SP_-(\mathbf{T}, r) + \delta(e) + SP_-(s, \mathbf{1}) > u \end{gathered} \tag{5.5}$$

in which LP_+ is an upper bound on LP, and SP_- a lower bound on SP. Initially we compute tight bounds LP_+ and SP_- and use (5.5) to test all the edges for redundancy. Here, 'tight bounds' refers to the bounds LP and SP used in (5.3). Because deleting an edge never increases the longest path length nor decreases the shortest path length between two nodes, these values remain valid bounds as we delete redundant edges. If any edges are deleted, we have the option of recomputing LP_+ and SP_- and repeating the process.

We note that one round of either of these heuristic filtering algorithms achieves consistency on an MDD of width 1, and therefore achieves traditional domain consistency for `among`. Also, an important aspect of the above heuristic filtering algorithms is that they can be applied independent of the variable ordering of the MDD.

5.2 Refining the MDD

The current MDD can be refined to reflect more accurately a given `among` constraint. Consider for example the MDD of Fig. 5.2(a). No filtering is possible for the `among` constraint (5.4). However, we can refine the MDD by *splitting* node p_3. We observe that the edges coming into p_3 from above are not *equivalent*, in the sense that the paths from **T** to p_3 containing one edge do not have the same set of feasible completions as the paths from **T** to p_3 containing the other edge. We therefore split p_3 into p_3' and p_3'' as in Fig. 5.2(b). We can now filter edges (p_3', p_4) and (p_3'', p_5) using (5.3), resulting in Fig. 5.2(c).

Splitting results in a tighter relaxation, because the filtered MDD after splitting allows only two solutions $(x_1, x_2, x_3, x_4) = (0, 0, 1, 1), (1, 1, 0, 0)$, whereas the filtered MDD before splitting admitted four solutions, $(0, 0, 0, 0)$, $(0, 0, 1, 1)$, $(1, 1, 0, 0)$, $(1, 1, 1, 1)$. In fact, the MDD after splitting excludes all solutions that violate the `among` constraint.

In general, edges entering a given node are partitioned into equivalence classes, and ideally the node is split into one copy for each equivalence class. However, this may enlarge the width of the MDD beyond the limit, in which case some of the equivalence classes must be merged. Also, edge equivalence may be costly to compute in practice, in which case an approximation of equivalence is used.

The shortest and longest path information can also be used to help us refine nodes in the MDD. For example, we may regard two edges $e_1 \in E(r_1, s)$ and $e_2 \in E(r_2, s)$ as approximately equivalent for the `among`

$$\mathrm{LP}(T, r_1) + \delta(e_1) = \mathrm{LP}(T, r_2) + \delta(e_2) \text{ or,}$$

$$\mathrm{SP}(T, r_1) + \delta(e_1) = \mathrm{SP}(T, r_2) + \delta(e_2).$$

Another approximation into equivalence classes is by considering the impact of an edge on the ‘tightness’ of an `among` constraint. That is, for each inequality defining the `among` constraint, we consider the ‘slack’ of

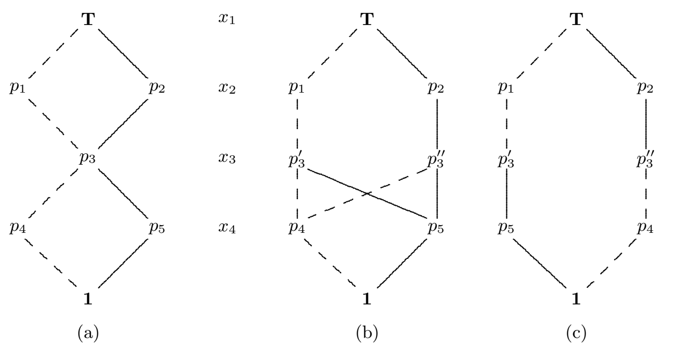


Figure 5.2: Refining MDD (a) by splitting node p_3 yields (b), which after filtering for the constraint `among`$((x_1, x_2, x_3, x_4), \{1\}, 2, 2)$ yields (c).

an edge $e \in E(r,s)$ to be $\ell - (\mathrm{SP}(T,r) + \delta(e) + \mathrm{SP}(s,1))$, respectively $u - (\mathrm{LP}(T,r) + \delta(e) + \mathrm{LP}(s,1))$. The slack reflects the number of variables that can still be assigned to a value in S without violating the respective inequality. We say that an edge is ‘tight’ if its slack is at most a given threshold τ , and ‘loose’ otherwise. The equivalence classes (with respect to each inequality) then belong to all tight edges and all loose edges entering a node in the MDD. For the random instances considered in the experimental section, we set $\tau = 1$, while for the nurse rostering instances we set $\tau = 3$.

After a round of node splitting on each layer, we run the filtering heuristic. If there are multiple `among` constraints, we test for equivalence with respect to all the constraints and refine for each one individually.

5.3 Experimental Results

We have implemented the algorithms presented in the previous section to evaluate the performance of MDD filtering of `among` constraints. That is, we have built from scratch a constraint programming solver that applies a fixed-width MDD store instead of a domain store (see Chapter 4). All the experiments are performed using a 2.33GHz Intel Xeon machine with 8GB memory.

The main goal of our experiments is to empirically assess the impact of the width of the MDD on the resulting search tree size and computation time. We performed experiments on randomly generated problem instances, and on structured ‘nurse rostering’ problem instances.

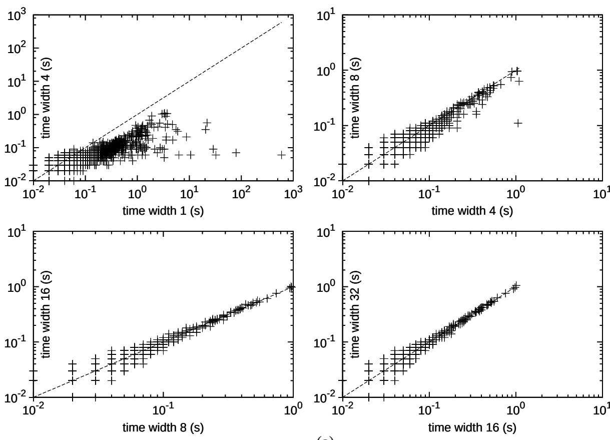
5.3.1 Random Instances

The first set of experiments is conducted on randomly generated instances. The main parameters that define these instances are the number of (binary) variables n, the number of `among` constraints, the number of variables in each `among` constraint, and for each `among`, a lower and upper bound on the number of variables taking value 1. In addition, the variable indices in each `among` are sampled from a normal distribution (modulo n), where the mean is chosen uniformly at random from $[1..n]$, while the standard deviation is a parameter to be arbitrarily chosen. We note that for many practical problems, the variable indices in an `among` constraint are nearly consecutive, see for example the nurse rostering instances in the next section. This would correspond to random instances in which the normal distribution from which the variable indices are sampled has a low standard deviation.

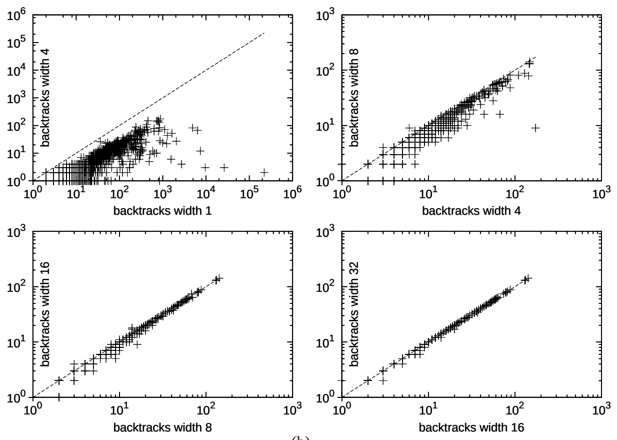
We have experimented with several parameter combinations, but we will only report results for specific parameter settings that capture the general qualitative behavior over the parameter space. These parameters are as follows. For all random instances, the number of variables is 50, while each `among` constraint consists of 5 variables chosen at random with a fixed lower bound of 2 and upper bound of 3. The variable indices are chosen from the normal distribution described above with standard deviations $\sigma = 1$, $\sigma = 2.5$, $\sigma = 5$ and $\sigma = 7.5$.

In our random experiments we vary the number of `among` constraints (from 5 to 200, by steps of 5) in each instance, and we generate 100 instances for each number. Each instance is solved by our MDD solver using varying widths. Note that width 1 corresponds to the traditional domain store. In Figures 5.3–5.6, we provide scatter plots of the running times and number of backtracks for all instances. The subplots are arranged to indicate the 'marginal' change in solution time or backtracks due to width, that is, we compare the results for width 1 vs. width 4, width 4 vs. width 8, width 8 vs. width 16 and width 16 vs. width 32. Note that these are all log-log plots. Points on the diagonal represent instances for which the measured quantity are equal (that is, computation time or the number of backtracks). Points below the diagonal imply that the MDD with higher width had a measured quantity less than the the MDD of lower width, while the opposite holds true for points above the diagonal.

Immediately we notice that increasing the maximum width of the MDD almost always results in fewer backtracks. The number of backtracks required by an MDD of width 4 never exceeds the number of backtracks required by an MDD of width 1. Moreover, for several of the harder instances there is already an



(a)



(b)

Figure 5.3: ($\sigma = 1$) Comparing the effect of MDD width in terms of backtracks (a) and time (b).

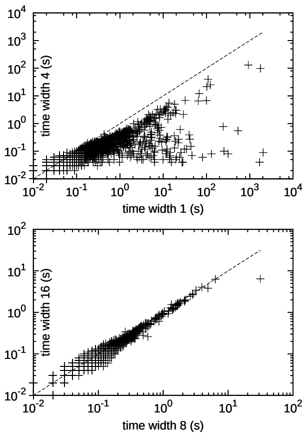
enormous reduction in the number of backtracks using an MDD of width 4 (in some cases more than 5 orders of magnitude). There are a few difficult instances for which using an MDD of width 8 results in at least one order of magnitude fewer backtracks than those required by an MDD of width 4.

The results for computation time are a little more varied. For the problems with lower standard deviation on the variable indices ($\sigma = 1$ and $\sigma = 2.5$) it always pays off to use an MDD of maximum width up to 8 except for the very simplest of problems (that is, those problems that can be solved in under half a second). There are several cases where the solution time decreases by over three orders of magnitude.

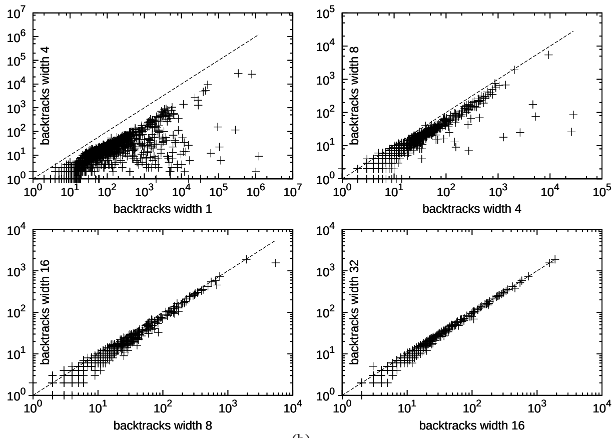
For the problems with a higher standard deviation on the variable indices ($\sigma = 5$ and $\sigma = 7.5$) we observe that there is not always an absolute decrease in solution time when we use an MDD of width 4 (or width 8) instead of an MDD of width 1. However, we would argue that when an MDD of low width (width 4 or width 8) performs worse than the traditional domain store (an MDD of width 1) it does so by a small margin. On the other hand, there are many difficult instances for which the computation time decreases by at least one order of magnitude. In terms of reducing computation time the wider MDDs (width 16 and width 32) don't seem to help or hurt much when compared to narrower MDDs (width 4 and 8).

5.3.2 Nurse Rostering Instances

We next conduct experiments on a set of instances inspired by nurse rostering problems, taken from [58]. The instances are of three different classes, and combine constraints on the minimum and maximum number of working days for sequences of consecutive days of given lengths. For example, class *C-I* demands to work at most 6 out of each 8 consecutive days (max6/8) and at least 22 out of every 30 consecutive days

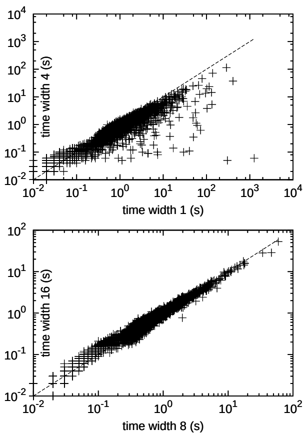


(a)

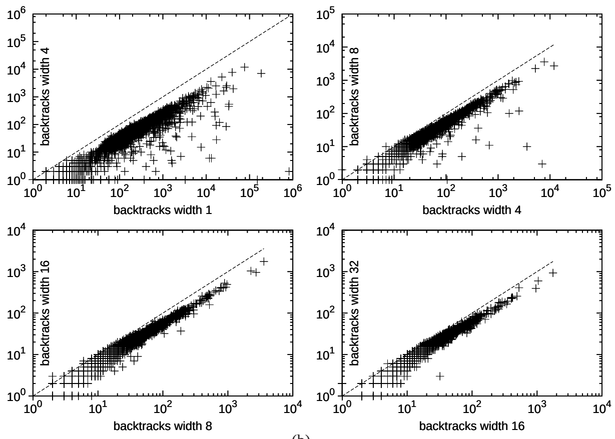


(b)

Figure 5.4: ($\sigma = 2.5$) Comparing the effect of MDD width in terms of backtracks (a) and time (b).

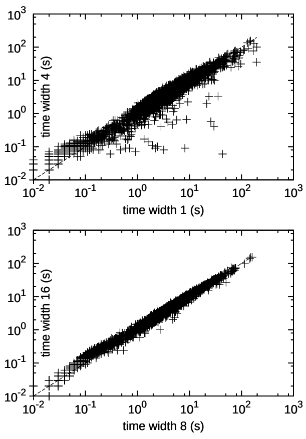


(a)

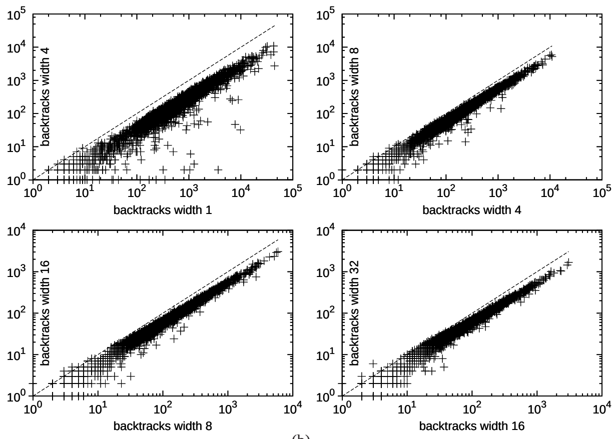


(b)

Figure 5.5: ($\sigma = 5$) Comparing the effect of MDD width in terms of backtracks (a) and time (b).



(a)



(b)

Figure 5.6: ($\sigma = 7.5$) Comparing the effect of MDD width in terms of backtracks (a) and time (b).

instance		**Width 1**		**Width 2**		**Width 4**		**Width 8**		**Width 16**		**Width 32**		**Width 64**	
	size	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
C-I	40	61225	55.63	22443	28.67	8138	12.64	1596	3.84	6	0.07	3	0.09	2	0.10
	50	62700	88.42	20992	48.82	3271	12.04	345	2.76	4	0.08	3	0.13	3	0.16
	60	111024	196.94	38512	117.66	3621	19.92	610	6.89	12	0.24	8	0.29	5	0.34
	70	174417	375.70	64410	243.75	5182	37.05	889	12.44	43	0.80	13	0.59	14	0.90
	80	175175	442.29	64969	298.74	5025	44.63	893	15.70	46	1.17	11	0.72	12	1.01
C-II	40	179743	173.45	60121	79.44	17923	32.59	3287	7.27	4	0.07	4	0.07	5	0.11
	50	179743	253.55	73942	166.99	9663	38.25	2556	18.72	4	0.09	3	0.12	3	0.18
	60	179743	329.72	74332	223.13	8761	49.66	1572	16.82	3	0.13	3	0.18	2	0.24
	70	179743	391.29	74332	279.63	8746	64.80	1569	22.35	4	0.18	2	0.24	2	0.34
	80	179743	459.01	74331	339.57	8747	80.62	1577	28.13	3	0.24	2	0.32	2	0.45
C-III	40	91141	84.43	29781	38.41	5148	9.11	4491	9.26	680	1.23	7	0.18	6	0.13
	50	95484	136.36	32471	75.59	2260	9.51	452	3.86	19	0.43	7	0.24	3	0.20
	60	95509	173.08	32963	102.30	2226	13.32	467	5.47	16	0.50	6	0.28	3	0.24
	70	856470	1986.15	420296	1382.86	37564	186.94	5978	58.12	1826	20.00	87	3.12	38	2.29
	80	882640	2391.01	423053	1752.07	33379	235.17	4236	65.05	680	14.97	55	3.27	32	2.77

Table 5.1: Nurse rostering instances. The effect of MDD width when finding one feasible solution.

(min22/30). For class *C-II* these numbers are max6/9 and min20/30, and for class *C-III* these numbers are max7/9 and min22/30. In addition, all classes require to work between 4 and 5 days per calendar week. The planning horizon ranges from 40 to 80 days.

The results are presented in Tables 5.1–5.3. We report the total number of backtracks upon failure (BT) and computation time in seconds (CPU) for our MDD solver using width 1, 8, and 32. Again, the MDD of width 1 corresponds to a domain store.

In Table 5.1 we report the results for finding a first feasible solution. For all problem classes we observe a nearly monotonically decreasing sequence of backtracks and solution time as we increase the width up to 32. The rate of decrease of the solution metrics seems to be exponential in many cases. A typical result (the instance (*C-III* on 60 days)) shows that where an MDD of width 1 requires 95,509 backtracks and 173.08 seconds of computation time, an MDD of width 32 only requires 6 backtracks and 0.28 seconds of computation time to find a first feasible solution.

instance		**Width 1**		**Width 2**		**Width 4**		**Width 8**		**Width 16**		**Width 32**		**Width 64**	
	size	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
C-I	40	230550	212.01	77941	106.28	17175	30.79	6741	17.09	106	3.11	99	3.10	90	3.15
	50	238192	339.47	87345	208.68	9362	39.54	2273	18.26	937	10.27	3378	14.18	669	10.37
	60	247500	458.38	93321	292.75	8022	57.35	2068	30.57	394	18.58	62	17.32	41	17.35
	70	260647	579.09	104411	401.44	9979	75.17	2044	29.12	412	12.05	734	12.59	259	12.18
	80	273187	699.72	111769	501.30	9887	80.71	1621	26.15	133	5.41	33	4.72	28	5.24
C-II	40	518489	469.32	182106	247.56	40279	75.36	8933	22.87	37	0.24	40	0.32	35	0.41
	50	518499	721.02	219610	500.22	26443	105.36	4598	34.10	32	0.33	30	0.43	28	0.57
	60	518509	914.14	219839	660.32	25138	142.46	3470	36.99	29	0.44	29	0.66	30	0.87
	70	518519	1158.65	219845	830.58	25057	186.03	3580	51.09	30	0.60	28	0.92	28	1.32
	80	518529	1312.85	219855	1023.10	25057	230.46	3580	63.89	31	0.78	28	1.18	27	1.78
C-III	40	455495	563.99	157984	363.10	25071	199.13	24319	206.70	2039	161.81	454	159.34	74	163.14
	50	1006980	2064.11	575231	1706.58	198368	1035.08	99114	878.23	49671	794.05	124141	900.59	3764	716.54
	60	1969337	5284.32	815078	3706.08	250889	2466.10	37172	1885.00	71790	1947.87	331	1785.60	287	1808.53
	70	3559033	9374.91	2042509	6751.75	266207	1519.52	83826	797.72	24195	483.56	1616	341.55	61464	710.98
	80	4201778	12042.30	2191133	8574.52	185755	1228.34	22488	302.61	1835	115.38	834	94.62	81	95.88

Table 5.2: Nurse rostering instances. The effect of MDD width when finding all feasible solutions.

In order to make a comparison to [58], we also report the results for computing all feasible solutions. In Table 5.2 we notice that the results for the reduction in the number of backtracks is very similar to that for finding one feasible solution although the sequences are not strictly decreasing. For example, the instance (*C-I* on 80 days) is solved by the domain store using 273,187 backtracks while the MDD store of width 32 needs only 33 backtracks. This reduction is reflected in the CPU time as well, which is reduced from around 699.72 seconds to around 4.72 seconds for this instance. The results in terms of computation time for those instances in class (*C-III*) are not as drammatic as those for the other classes. This is because most of the time is spent enumerating feasible solutions which tends to 'smoothen' the total computation time.

In Table 5.3 we compare our results to those presented in [58] (which were run on a 2.8GHz Intel Xeon machine), to provide a comparison with more advanced filtering algorithms based on global constraints for the domain store. In the column '`gcc+seq`', advanced filtering algorithms for `gcc` and `sequence` constraints are applied, while the results for column '`genseq`' are obtained by applying one single `gen-sequence` constraint. We remark that these instances were specifically designed to be modeled (perfectly) with a single `gen-sequence` constraint, which explains the zero backtracks for

instance		**MDD width 1**		**MDD width 8**		**MDD width 64**		**gcc+seq [58]**		**genseq [58]**	
	size	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
C-I	40	231k	212.01	6,741	17.09	90	3.15	185k	216.49	0	0.77
	50	238k	339.47	2,273	18.26	669	10.37	186k	369.12	0	2.09
	60	248k	458.38	2,068	30.57	41	17.35	188k	621.99	0	3.60
	70	261k	579.09	2,044	29.12	259	12.18	196k	840.52	0	1.88
	80	273k	699.72	1,621	26.15	28	5.24	198k	1,061.62	0	0.61
C-II	40	518k	469.32	8,933	22.87	35	0.41	394k	390.93	0	0.01
	50	518k	721.02	4,598	34.10	28	0.57	394k	660.74	0	0.02
	60	519k	914.14	3,470	36.99	30	0.87	394k	1,074.26	0	0.03
	70	519k	1,158.65	3,580	51.09	28	1.32	394k	1,432.20	0	0.04
	80	519k	1,312.85	3,580	63.89	27	1.78	394k	1,786.62	0	0.05
C-III	40	455k	563.99	24,319	206.70	74	163.14	328k	417.63	0	34.43
	50	1,007k	2,064.11	99,114	878.23	3,764	716.54	457k	1,061.24	0	150.87
	60	1,969k	5,284.32	37,172	1,885.00	287	1,808.53	730k	2,822.09	0	339.89
	70	3,559k	9,374.91	83,826	797.72	61,464	710.98	1,744k	5,048.84	0	60.82
	80	4,202k	12,042.30	22,488	302.61	81	95.88	1,847k	7,457.36	0	15.41

Table 5.3: Nurse rostering instances: MDD filtering compared to state-of-the-art domain filtering.

'`genseq`'. Clearly, the global constraints allow to reduce further the search space of the domain store (compare 'width 1' with '`gcc+ seq`' and '`genseq`'), but it is interesting that our MDD store performs much better than '`gcc+seq`'. Namely, these global constraints group together multiple `among` constraints, whereas our MDD-store only applies (heuristic) filtering on individual `among` constraints. This clearly shows the power of propagating structural information through an MDD store rather than a domain store.

5.4 Conclusion

We studied MDD-based propagation for `among` constraints as a more refined alternative to traditional domain store filtering algorithms. We presented efficient heuristic MDD filtering algorithms that can be applied to any variable ordering of the MDD. We have also shown how these algorithms can be complemented with MDD refinement procedures based on the `among` constraints. Computational results have shown that MDD-based propagation can dramatically reduce the search space and computation time as compared to a domain store. This provides evidence that domain stores might be profitably replaced (or complemented) by MDD stores in CP solvers.

Chapter 6

Optimal Movement of Factory Cranes

6.1 Introduction

Manufacturing facilities frequently rely on track-mounted cranes to move in-process materials or equipment from one location to another. A typical arrangement, and the type studied here, allows one or more hoists to move along a single horizontal track that is normally mounted on the ceiling. Each hoist may be mounted on a crossbar that permits lateral movement as the crossbar itself moves longitudinally along the track. A cable suspended from the crossbar raises and lowers a lifting hook or other device.

When a production schedule for the plant is drawn up, cranes must be available to move materials from one processing unit to another at the desired times. The cranes may also transport cleaning or maintenance equipment. Since the cranes operate on a single track, they must be carefully scheduled so as not to interfere with each other. One crane may be required to yield (move out of the way) to permit another crane to pick up or deliver its load.

The problem is combinatorial in nature because one must not only compute a space-time trajectory for each crane, but must decide which crane yields to another and when. A decision made at one point may create a bottleneck that has unforeseen repercussions much later in the schedule. It is not unusual for production planners to put together a schedule that seems to allow ample time for crane movements, only to find that the crane operators cannot keep up with the schedule. As the cranes lag further and further behind, the production schedule must be adjusted in an ad hoc manner to allow them to catch up.

In this chapter we analyze the problem of scheduling two cranes and describe an exact algorithm, based on dynamic programming, to solve it. The problem data consist of time windows, crane assignments, and job sequencing. That is, the problem specifies a release time and deadline for each job, an assignment of each job to a crane, and the order in which the jobs assigned to each crane are to be carried out. Several objectives are possible, but in our experience the primary goal has been to follow the production schedule as closely as possible.

This research is part of a larger project in which both heuristic and exact algorithms have been developed for use in crane scheduling software. The heuristic method makes crane assignment and sequencing decisions as well as computing space-time trajectories, and it is fast enough to accommodate large problems involving several cranes. However, once the assignments and sequencing are given, the heuristic method may fail to find feasible trajectories when they exist and reject good solutions as a result. We therefore found it important to solve the trajectory problem exactly for a given assignment and sequencing, in at least some of the smaller problem instances, as a check on the heuristic method. The exact algorithm has practical value in its own right, because two-crane problems are common in industry, and the algorithm solves instances of respectable size within a minute or so. Nonetheless, we see it as having an equally important role in the creation of benchmarks against which heuristic methods can be tested and tuned for best performance.

We begin by deriving structural results for the two-crane problem that restrict the trajectories that must be considered to certain *canonical* trajectories. This not only makes the problem tractable for dynamic programming by reducing the state space, but it also accelerates the heuristic solution of larger two-crane problems by dramatically reducing the possibilities that must be enumerated. Moreover, the canonical trajectories simplify the operation of the cranes, and enhance safety, by restricting the crane movements to certain predictable patterns. For example, cranes always move at the same speed, never stand at rest except at a pickup or delivery point, and never yield to another crane except when moving alongside that crane (at a safe distance). In addition, the left crane keeps to the left as much as possible, and the right crane to the right.

We then describe a dynamic programming algorithm for the optimal trajectory problem. The state space is large, due to the fine space-time granularity with which the problem must be solved, as well as the necessity of keeping up with which task a crane is performing and how long it has been processing that task. To deal with these complications we introduce a novel state space description that represents many states implicitly as a cartesian product of intervals. The state space is efficiently stored and updated in a data structure that uses an array of two-dimensional circular queues. These enhancements accelerate solution by at least an order of magnitude and allow us to solve problems of realistic size within a reasonable time. The paper concludes with computational results and directions for further research.

6.2 Previous Work

To our knowledge, no previous work computes space-time trajectories that allow cranes to yield, and none obtains structural results that restrict the types of trajectories that must be considered. The literature on crane scheduling tends to cluster around two types of problems: movement of materials from one vat to another in an electroplating or similar process (typically referred to as *hoist scheduling* problems), and loading and unloading of container ships in a port.

A classification scheme for hoist scheduling problems appears in [41]. It is assumed in these problems that each item visits the same vats in the same order, in most cases consecutively. The objective is to minimize cycle time, which is the time lapse between the entry of two consecutive items into the system. Much research in this area deals with the single-hoist cyclic scheduling problem [53, 5, 6, 35, 48, 50, 12, 40]. Because there is only one hoist, the space-time trajectory of the hoist is not an issue, so long as it picks up and delivers items at the right time. Even this restricted problem is NP-complete [34].

Several papers deal with cyclic two-hoist and multi-hoist problems. One approach partitions the vats into contiguous subsets, assigns a hoist to each subset, and schedules each hoist within its partition [62, 63, 65]. A better solution can generally be obtained, however, by allowing a vat to be served by more than one hoist. This has been accomplished by careful scheduling of the hoists to avoid collisions, based on a case-by-case analysis of the various ways that they can approach each other [33, 59, 54, 36, 11, 37, 39]. None of these studies compute space-time trajectories of the hoists or allow one hoist to yield to another. They avoid collisions by setting departure and arrival times so that no interference is possible when hoists go directly from one vat to the next.

Although we do not address the assignment of tasks to cranes in the present paper, our problem is otherwise more general than hoist scheduling problems in several respects: (a) rather than requiring that every item visit the same sequence of stations, we allow each job to specify an arbitrary subset of tasks in any order; (b) we solve for an optimal space-time trajectory of each crane that allows it to make additional movements in order to yield to the other crane; (c) we accommodate release times and deadlines for the jobs; and (d) we allow for a variety of objective functions.

Port cranes are generally classified as quay cranes and yard cranes. Quay cranes may be mounted on a single track, as are factory cranes, but the scheduling problem differs in several respects. The cranes load (or unload) containers into ships rather than transferring items from one location on the track to another. A given crane can reach several ships, or several holds in a single ship, either by rotating its arm or perhaps by moving laterally along the track. The problem is to assign cranes to loading (unloading) tasks, and schedule the tasks, so that the cranes do not interfere with each other [13, 52, 42, 29, 38, 66].

Yard cranes are typically mounted on wheels and can follow certain paths in the dockyard to move containers from one location to another. Existing solution approaches schedule departure and arrival times for the cranes so that they do not interfere with each other, but the actual space-time trajectories are not examined [64, 49].

6.3 The Optimal Trajectory Problem

In practice, a crane scheduling problem typically consists of a number of *jobs*, each of which specifies several *tasks* to be performed consecutively. For example, a job may require that a crane pick up a ladle at one location, fill the ladle with molten metal at a second location, deliver the metal to a third location, and then return the ladle. Tasks may also involve maintenance and cleaning activities. The same crane must perform all the tasks in a job and must remain stationary at the appropriate location while processing each task.

The location and processing time for each task are given, as are release times and deadlines. We also suppose that each job has been pre-assigned to a certain crane, and the jobs assigned to a crane must be performed in a fixed order. Each job assigned to a given crane must finish before the next job assigned to that crane begins.

In this study we explicitly account only for the longitudinal movements of the crane along the track. We assume that the crane has time to make the necessary lateral and vertical movements as it moves from one task location to another. This results in little loss of generality, because any additional time necessary for lateral or vertical motion can be built into the processing time for the task.

The problem data are:

R_j = release time of task j

D_j = deadline for task j

L_j = processing location (stop) for task j

P_j = processing time for task j

$c(j)$ = crane assigned to task j

v = maximum crane speed

$0, L_{\max}$ = leftmost and rightmost crane locations

Δ = minimum crane separation

Δt = time increment

Note that we refer to the processing location of a task as a *stop*.

If release times $\bar{R}_i$ and deadlines $\bar{D}_i$ are given for each job i rather than each task j, then the task release time R_j is the earliest possible start time for that task:

$$R_j = \bar{R}_i + \sum_{\ell=k}^{j-1} \left(P_\ell + \frac{|L_{\ell+1} - L_\ell|}{v} \right)$$

where k is the first task in job i. Similarly, the task deadline is the latest possible finish time, given the job deadline:

$$D_j = \bar{D}_i - \sum_{\ell=j+1}^{k'} \left(\frac{|L_\ell - L_{\ell-1}|}{v} + P_\ell \right)$$

where k' is the last task in job i.

We suppose for generality that there are cranes $1, \ldots, m$, where crane 1 is the *left crane* and crane m the *right crane*, although we solve the problem only for $m = 2$. $T_{\max}$ is the length of the time horizon. The problem variables are:

x_{ct} = position of crane c at time t

y_{ct} = task being processed by crane c at time t (0 if none)

τ_j = time at which task j starts processing

Task j therefore finishes processing at time $\tau_j + P_j$. We assume that the tasks are indexed so that tasks assigned to a given crane are processed in order of increasing indices.

The problem with n tasks and m cranes may now be stated

$$
\begin{array}{lll}
\min\ f(\tau) & & \\
0 \le x_{ct} \le L_{\max} & \multirow{3}{*}{} & (a) \\
x_{ct} - v\Delta t \le x_{c,t+\Delta t} \le x_{ct} + v\Delta t & \left.\right\} \text{all } c,t & (b) \\
y_{ct} > 0 \Rightarrow x_{ct} = L_{y_{ct}} & & (c) \\
x_{ct} \le x_{c+1,t} - \Delta,\ c = 1,\ldots,m-1,\ \text{all } t & & (d) \\
R_j \le \tau_j \le D_j - P_j,\ \text{all } j & & (e) \\
y_{c(j)t} = j,\ t = \tau_j,\ldots,\tau_j + P_j - \Delta t & \left.\right\} \text{all } j & (f) \\
\{c(j) = c(j'),\ j < j'\} \Rightarrow \tau_j < \tau_{j'},\ \text{all } j,j' & & (g) \\
y_{ct} \in \{0,\ldots,n\},\ \text{all } c,t & &
\end{array}
\tag{6.1}
$$

Constraint (a) requires that the cranes stay on the track, and (b) that their speed be within the maximum. Constraint (c) implies that a crane must be at the right location when it is processing a task. Constraint (d) makes sure the cranes do not interfere with each other. Constraint (e) enforces the time windows, and (f) ensures that processing continues for the required amount of time once it starts. Constraint (g) requires that the tasks assigned to a crane be processed in the right order.

We assume that the objective $f(\tau)$ is a function only of the task start times, because this is sufficient for practical application and allows us to prove the structural results below. Generally one is interested in conforming to the production schedule as closely as possible. For instance, one might minimize the lapse between the release time R_k and the start time τ_k of the first task k in a job, or the lapse between the earliest finish time $R_{k'} + P_{k'}$ and the completion time $\tau_{k'} + P_{k'}$ of the last task k' in a job, or some combination of these. We used the more general objective

$$
f(\tau) = \sum_j \alpha_j\,(\tau_j - R_j) \tag{6.2}
$$

but normally set α_j to a positive value only when task j is the first or last task of a job. One might also be concerned that the cranes make no unnecessary movements. We can incorporate this into the objective function only if it has the form $f(x,\tau)$, but there is no need to do so. By restricting the cranes to the canonical trajectories defined by the structural results below, we avoid unnecessary movements.

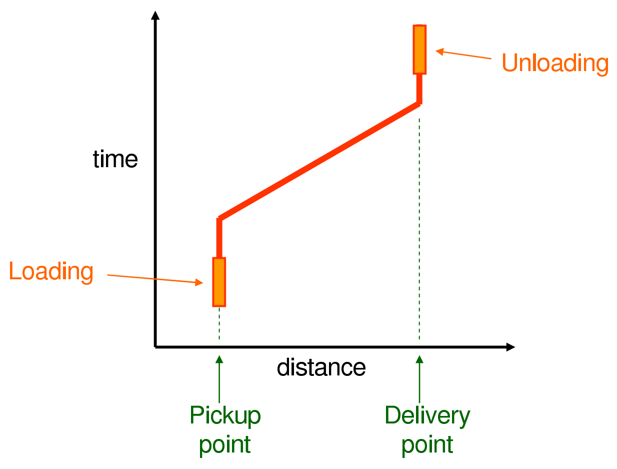


Figure 6.1: Sample space-time trajectory for one task. The shaded vertical bars denote processing, which in this case consists of loading and unloading.

6.4 Canonical Trajectories

Optimal control of the cranes is much easier to calculate when it is recognized that only certain trajectories need be considered, namely those we call canonical trajectories. We will show that when there are two cranes, some pair of canonical trajectories is optimal.

Let a *processing schedule* for a given crane consist of the vector τ of task start times. We define the *extremal* trajectory for the left crane, with respect to a given processing schedule, to be one that observes the processing schedule and that, while not processing a task, always follows the leftmost trajectory that never moves in the direction away from the next stop. For example, the trajectory in Figure 6.1 is not extremal because the crane moves to the right sooner than necessary.

More precisely, if the next stop (processing location) is to the right of the current stop, then the left crane follows the canonical trajectory if it leaves the current stop as late as possible so as to arrive at next stop just as processing starts (Fig. 6.2a). If the next stop is to the left of the current one, the crane leaves the current stop as early as possible (Fig. 6.2b). Thus at any time the crane is either stationary or moving at maximum speed. The extremal trajectory for the right crane follows the rightmost trajectory: it leaves the current stop as late as possible if moving to the left, and as early as possible if moving to the right.

A trajectory for the left crane is *canonical* with respect to the right crane if at each moment it is the rightmost of (a) the extremal trajectory for the left crane and (b) the trajectory that runs parallel to and just

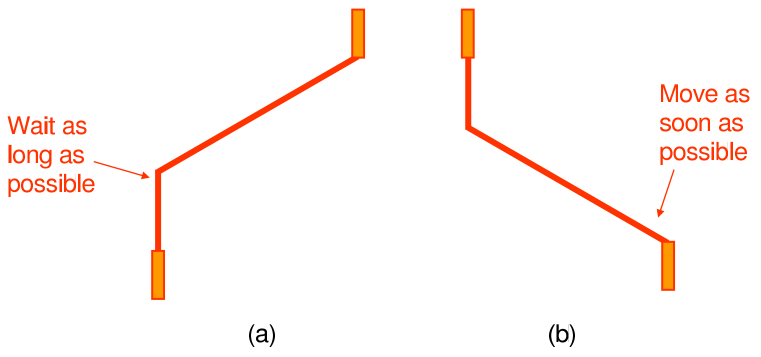


Figure 6.2: Extremal trajectory for the left crane (a) when the destination is to the right of the origin, and (b) when the destination is to the left of the origin.

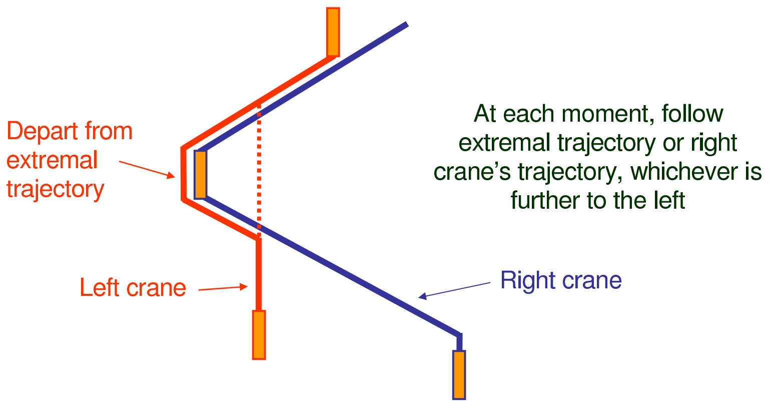

Depart from extremal trajectory
At each moment, follow extremal trajectory or right crane's trajectory, whichever is further to the left
Left crane
Right crane

Figure 6.3: Canonical trajectory for the left crane (leftmost solid line).

to the left of the right crane's trajectory (Fig. 6.3). More precisely, trajectory x_1' is canonical for the left crane, with respect to trajectory x_2 for the right crane, if the extremal trajectory $\bar{x}_1$ for the left crane satisfies $x_1'(t) = \min\{\bar{x}_1(t), x_2(t) - \Delta\}$ at each time t. A trajectory for the right crane is canonical with respect to the left crane if it is the leftmost of the extremal trajectory for the right crane and the left crane's trajectory. That is, $x_2'(t)$ is canonical if $x_2'(t) = \max\{\bar{x}_2(t), x_1(t)+\Delta\}$, where $\bar{x}_2(t)$ is the extremal trajectory. Finally, a pair of trajectories is canonical if the trajectories are canonical with respect to each other.

Theorem 6.4.1. *If the two-crane problem (6.1) has an optimal solution, then some optimal pair of trajectories is canonical.*

Proof. The idea of the proof is to replace the left crane's optimal trajectory with a canonical trajectory with respect to the right crane's optimal trajectory. Then assign the right crane a canonical trajectory with respect to the left crane's new trajectory, and finally assign the left crane a canonical trajectory with respect to the right crane's new trajectory. At this point it is shown that the trajectories are canonical with respect to each other. Since these replacements never change the objective function value, the canonical trajectories are optimal, and the theorem follows.

Thus let $x^* = (x_1^*, x_2^*)$ be a pair of optimal trajectories for a two-crane problem. Let $\bar{x}_1, \bar{x}_2$ be extremal trajectories for the left and right cranes with respect to the processing schedules in the optimal trajectories.

Consider the canonical trajectory x_1' for the left crane with respect to x_2^*, which is given by $x_1'(t) = \min\{\bar{x}_1(t), x_2^*(t) - \Delta\}$. We claim that (x_1', x_2^*) is optimal. First note that it has the same objective function value as x^*, since x_1' has the same processing schedule as x_1^*. Furthermore, it is feasible because the cranes do not interfere with each other, and the speed of the left crane is never greater than v. The cranes do not interfere with each other because $x_1'(t) \leq x_2^*(t) - \Delta$ for all t, due to $x_1'(t) \leq x_1^*(t)$ and $x_1^*(t) \leq x_2^*(t) - \Delta$. To show that the speed of the left crane is never more than v it suffices to show that the average speed in the left-to-right direction between any pair of time points t_1, t_2 is never more than v, and similarly for the average speed in the right-to-left direction. The former is

$$\begin{aligned}\frac{x_1'(t_2) - x_1'(t_1)}{t_2 - t_1} &= \frac{\min\{\bar{x}_1(t_2), x_2^*(t_2) - \Delta\} - \min\{\bar{x}_1(t_1), x_1^*(t_1) - \Delta\}}{t_2 - t_1} \\ &\leq \max\left\{\frac{\bar{x}_1(t_2) - \bar{x}_1(t_1)}{t_2 - t_1}, \frac{x_2^*(t_2) - x_2^*(t_1)}{t_2 - t_1}\right\} \leq v\end{aligned}$$

where the first inequality is due to the fact that

$$\min\{a, b\} - \min\{c, d\} \leq \max\{a - c, b - d\}$$

for any a, b, c, d, and the second inequality due to the fact that $\bar{x}_1$ and x_2^* are feasible trajectories. The speed in the right-to-left direction is similarly bounded.

Now consider the canonical trajectory x_2' for the right crane with respect to x_1', given by $x_2'(t) = \max\{\bar{x}_2(t), x_1'(t) + \Delta\}$. It can be shown as above that (x_1', x_2') is optimal.

Finally, let x_1'' be the canonical trajectory for the left crane with respect to x_2', given by $x_1''(t) = \min\{\bar{x}_1(t), x_2'(t) - \Delta\}$. Again (x_1'', x_2') is optimal. Since x_1'' is canonical with respect to x_2', to prove the theorem it suffices to show that x_2' is canonical with respect to x_1''; that is, $\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = x_2'(t)$ for all t. To show this we consider four cases for each time t.

Case 1: $\bar{x}_1(t) + \Delta \leq \bar{x}_2(t)$. We first show that

$$(x_1''(t), x_2'(t)) = (\bar{x}_1(t), \bar{x}_2(t)) \tag{6.3}$$

by considering the subcases (a) $x_2^*(t) \leq \bar{x}_1(t)$ and (b) $\bar{x}_1(t) < x_2^*(t)$. In subcase (a),

$$x_1'(t) = \min\{\bar{x}_1(t), x_2^*(t) - \Delta\} = x_2^*(t) - \Delta$$

which implies

$$x_2'(t) = \max\{\bar{x}_2(t), x_1'(t) + \Delta\} = \max\{\bar{x}_2(t), x_2^*(t)\} = \bar{x}_2(t)$$

and

$$x_1''(t) = \min\{\bar{x}_1, x_2'(t) - \Delta\} = \min\{\bar{x}_1, \bar{x}_2(t) - \Delta\} = \bar{x}_1(t)$$

In subcase (b), $x_1'(t) = \bar{x}_1(t)$, which implies $x_2'(t) = \max\{\bar{x}_2(t), \bar{x}_1(t)+\Delta\} = \bar{x}_2(t)$ and again $x_1''(t) = \bar{x}_1$. Now from (6.3) we have

$$\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = \max\{\bar{x}_2(t), \bar{x}_1(t) + \Delta\} = \bar{x}_2(t) = x_2'(t)$$

as claimed.

The remaining cases suppose $\bar{x}_2(t) < \bar{x}_1(t) + \Delta$ and consider the situations in which $x_2^*(t)$ is less than or equal to $\bar{x}_2(t)$, between $\bar{x}_2(t)$ and $\bar{x}_1(t) + \Delta$, and greater than $\bar{x}_1(t) + \Delta$.

Case 2: $x_2^*(t) \leq \bar{x}_2(t) < \bar{x}_1(t) + \Delta$. It can be checked that $(x_1''(t), x_2'(t)) = (\bar{x}_2(t) - \Delta, \bar{x}_2(t))$ and $\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = \max\{\bar{x}_2(t), \bar{x}_2(t)\} = \bar{x}_2(t) = x_2'(t)$, as claimed.

Case 3: $\bar{x}_2(t) < x_2^*(t) \leq \bar{x}_1(t) + \Delta$. Here $(x_1''(t), x_2'(t)) = (x_2^*(t) - \Delta, x_2^*(t))$ and $\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = \max\{\bar{x}_2(t), x_2^*(t)\} = x_2^*(t) = x_2'(t)$.

Case 4: $\bar{x}_2(t) < \bar{x}_1(t) + \Delta < x_1^*(t)$. Here $(x_1''(t), x_2'(t)) = (\bar{x}_1(t), \bar{x}_1(t) + \Delta)$ and $\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = \max\{\bar{x}_2(t), \bar{x}_1(t) + \Delta\} = \bar{x}_1(t) + \Delta = x_2'(t)$. This completes the proof.

The properties of canonical trajectories allow us to consider a very restricted subset of trajectories when computing the optimum.

Corollary 6.4.2. *If the two-crane problem has an optimal solution, then there is an optimal solution with the following characteristics:*

(a) While not processing a task, the left (right) crane is never to the right (left) of both the previous and the next stop.

(b) While not processing a task, the left (right) crane is moving in a direction toward its next stop if it is to the right (left) of the previous or next stop.

(c) A crane never moves in the direction away from its next stop unless it is adjacent to the other crane at all times during such motion.

(d) While not processing a task, the left (right) crane can be stationary only if it is (i) at the previous or the next stop, whichever is further to the left (right), or (ii) adjacent to the other crane.

Proof.

(a) If crane 1 (the left crane) is to the right of both its previous and next stop at some time t, then $x_1(t) > \bar{x}_1(t)$. This is impossible in a canonical trajectory, in which $x_1(t) = \min\{\bar{x}_1(t), x_2(t) - \Delta\}$. The argument is similar for crane 2.

(b) Suppose crane 1 is to the right of its previous stop. Due to (a), it is not to the right of its next stop, which must therefore be to the right of the previous stop. We cannot have $x_1(t) > \bar{x}_1(t)$ as in (a), and we cannot have $x_1(t) < \bar{x}_1(t)$, since this means the crane cannot reach its next stop in time. So crane 1 is on its canonical trajectory, which means that it is moving toward its next stop. The argument is similar if crane is to the right of the next stop.

(c) From (a) and (b), at a given time t crane 1 can be moving in the direction opposite its next stop only if it is at or to the left of both the previous and next stops. This means that it will be to the left of both at time $t + \Delta t$, so that $x_1(t + \Delta t) < \bar{x}_1(t + \Delta t)$. But since

$$x_1(t + \Delta t) = \min\{\bar{x}_1(t + \Delta t), x_2(t + \Delta t) - \Delta\}$$

this means $x_1(t+\Delta t) = x_2(t+\Delta t) - \Delta$, and crane 1 is adjacent to the other crane. Since crane 1 is moving left between t and $t + \Delta t$, it must be adjacent to the other crane at time t as well.

(d) From (a) and (b), a stationary crane 1 must be at or to the left both the previous and the next stop. If it is at one of them, then (i) applies. If it is to the left of both, then $x_1(t) < \bar{x}_1(t)$, which again implies that $x_1(t) = x_2(t) - \Delta$, and (ii) holds.

6.5 Dynamic Programming Recursion

The optimal control problem for the cranes is not simply a matter of computing an optimal space-time trajectory. It is complicated by three factors: (a) each crane must perform tasks in a certain order; (b) each task must be performed at a certain location for a certain amount of time; and (c) the cranes must not interfere with each other. We chose to solve the problem with dynamic programming because it has the flexibility to deal with these additional constraints while preserving optimality (up to the precision allowed by the space and time granularity). The drawback is a potentially exploding state space, but we will show how to keep it under control for problems of reasonable size. To simplify notation, we assume from here out that $\Delta t = 1$.

There are three state variables for each crane. Two of them are x_{ct} and y_{ct} as defined in model (6.1), and the third is

$$u_{ct} = \begin{cases} \text{amount of time crane } c \text{ will have been processing at time } t+1 \\ \text{(0 if the crane is neither processing nor starts processing at time } t\text{)} \end{cases}$$

In principle the recursion is straightforward, although a practical implementation requires careful management of state transitions and data structures. Let $x_t = (x_{1t}, x_{2t})$, and similarly for y_t and u_t. Also let $z_t = (x_t, y_t, u_t)$. It is convenient to use a forward recursion:

$$g_{t+1}(z_{t+1}) = \min_{z_t \in S^{-1}(z_{t+1})} \{h(t, y_t, u_t) + g_t(z_t)\} \tag{6.4}$$

where $g_t(z_t)$ is the cost of an optimal trajectory between the initial state and state z_t at time t, $h(t, y_t, u_t)$ is the cost incurred at time t, and $S^{-1}(z_{t+1})$ is the set of states at time t from which the system can move to state z_{t+1} at time $t+1$. Given the cost function (6.2), the cost $h(t, y_t, u_t)$ is $\sum_c h_c(t, y_t, u_t)$, where

$$h_c(t, y_t, u_t) = \begin{cases} \alpha_{y_{ct}}(t - R_{y_{ct}}) & \text{if } u_{ct} = 1 \\ 0 & \text{otherwise} \end{cases}$$

The boundary condition is

$$g_0(z_0) = 0$$

when z_0 is the initial state. The optimal cost is $g_{T_{\max}}(z_{T_{\max}})$, where $z_{T_{\max}}$ is the desired terminal state.

For each state z_{t+1} the recursion (6.4) computes the minimum $g_{t+1}(z_{t+1})$ and the state $z_t = s^{-1}_{t+1}(z_{t+1})$ that achieves the minimum. Thus $s^{-1}_{t+1}(z_{t+1})$ points to the state that would precede z_{t+1} in the optimal trajectory if z_{t+1} were in the optimal trajectory. For a basic recursion, the cost table $g_{t+1}(\cdot)$ is stored in memory until $g_{t+2}(\cdot)$ is computed, and then released (this is modified in the next section). Thus only two consecutive cost tables need be stored in memory at any one time. The table $s_{t+1}^{-1}(\cdot)$ of pointers is stored offline. Then if z_T is the final state, we can retrace the optimal solution in reverse order by reading the tables $s_{t+1}^{-1}(\cdot)$ into memory one at a time and setting $z_t = s_{t+1}^{-1}(z_{t+1})$ for $t = N-1, N-2, \ldots, 0$.

6.6 Reduction of the State Space

We can substantially reduce the size of the state space if we observe that in practical problems, the cranes spend much more time processing than moving. The typical processing time for a state ranges from two to five minutes (sometimes much longer), while the typical transit time to the next location is well under a minute. Furthermore, the state variables representing location and task assignment (x_{ct} and y_{ct}) cannot change while the crane is processing.

These facts suggests that the processing time state variable u_{ct} should be replaced by an *interval* $U_{ct} = [u_{ct}^{lo}, u_{ct}^{hi}] = \{u_{ct}^{lo}, u_{ct}^{lo}+1, \ldots, u_{ct}^{hi}\}$ of consecutive processing times. A single "state" $(x_t, u_t, U_{ct}) = (x_t, u_t, (U_{1t}, U_{2t}))$ now represents a set of states, namely the Cartesian product

$$\{(x_t, y_t, (i,j)) \,|\, i \in U_{1t},\ j \in U_{2t}\}$$

The possible state transitions for either crane c are shown in Table 6.1. The transitions in the table are feasible only if they satisfy other constraints in the problem, including those based on time windows, the physical length of the track, and interactions with the other crane. The transitions can be explained, line by line, as follows:

1. Because the processing time interval is the singleton $[0,0]$, the crane can be in motion and can in particular move to either adjacent location. When it arrives at the next location, the currently assigned task can start processing if the crane is in the correct position, in which case the state interval is $U_{ct} = [0,1]$ to represent two possible states: one in which the task does not start processing at time $t+1$, and one in which it does (the interval is $[1,1]$ if the deadline forces the task to start processing at $t+1$). If the crane is in the wrong location for the task, the state remains $[0,0]$.

2. None of the states in the interval $[0, u_2]$ allow processing to finish at time $t+1$. So all of the processing time states advance by one—except possibly the zero state, in which processing has not yet started and can be delayed yet again if the deadline permits it.

3. The last state in the interval $[0, P_{y_{ct}}]$ allows processing to finish at time $t+1$. This state splits off from the interval and assumes one of the processing state intervals in line 1. The other states evolve as in line 2.

4. Because the task is underway in all states, all processing times advance by one.

5. This is similar to line 3 except that there is no zero state.

There is no need to store a pointer $s^{-1}_{t+1}(x_t, y_t, (i,j))$ for every state $(x_t, y_t, (i,j))$ in (x_t, y_t, U_t). This is because when $u_{ct} \geq 2$, the state of crane c preceding (x_{ct}, y_{ct}, u_{ct}) must be $(x_{ct}, y_{ct}, u_{ct}-1)$. Thus we store $s^{-1}_{t+1}(x_t, y_t, (i,j))$ only when $i \leq 1$ or $j \leq 1$.

However, we must store the cost $g_{t+1}(x_t, y_t, (i,j))$ for every (i,j), because it is potentially different for every (i,j). Fortunately, it is not necessary to update this entire table at each time period, because most of the costs evolve in a predictable fashion. If $i, j \geq 2$, then

$$g_{t+1}(x_y, y_t, (i,j)) = g_t(x_t, y_t, (i-1, j-1))$$

So for each pair of tasks (y, y') we maintain a two-dimensional circular queue $Q_{yy'}(\cdot,\cdot)$ in which the cost

$$g_{t+1}((L_y, L_{y'}), (y, y'), (i,j)) \tag{6.5}$$

for $i, j \geq 2$ is stored at location

$$Q_{yy'}((t+i-2) \bmod M, (t+j-2) \bmod M)$$

where M is the size of the array $Q_{yy'}(\cdot,\cdot)$ (i.e., the longest possible processing time). In each period we insert the cost (6.5) into Q only for pairs (i,j) in which $i=2$ or $j=2$; the costs for other pairs with $i, j \geq 2$ were computed in previous periods. Thus only one row and one column of the Q array are altered in each time period, which substantially reduces computation time. When $i \leq 1$ or $j \leq 1$, the cost (6.5) is stored as a table entry $g_{t+1}(x_t, y_t, (i,j))$ that is updated at every time period, as with pointers.

The array $Q_{yy'}(\cdot,\cdot)$ is created when the state $((L_y, L_{y'}), (y, y'), (i,j))$ is first encountered with $i, j \geq 2$. The array is kept in memory over multiple periods until it is no longer updated, at which time it is deleted.

6.7 Experimental results

We report computational tests on a representative problem that is based on an actual industry scheduling situation. There are 60 jobs, each of which contains from two to eight tasks. We obtain smaller instances by scheduling only some of the jobs, namely the first ten (in order of release time), the first twenty, and so forth. Results on other problems we have examined are similar.

Release times were obtained from the production schedule, but no deadlines were given. We initially set the deadline of each job to be 40 minutes after each release time, with the expectation that these may have to be relaxed to obtain a feasible solution.

We divided the 108.5-meter track into ten equal segments, so that each distance unit represents 10.85 meters. Each crane can traverse the length of the track in about one minute. Because we want the crane to move one distance unit for each time unit, we set the time unit at six seconds. The 60-job schedule requires about four hours to complete, which means that the dynamic programming procedure has about $T_{\max} = 2400$ time stages.

Table 6.2 shows computation times obtained on a desktop PC running Windows XP with a Pentium D processor 820 (2.8 GHz). The assignment and sequencing of jobs used in each instance is the best one that was obtained by a heuristic procedure. Feasible solutions were found for all the instances except the full 60-job problem. To obtain a feasible solution of this problem, we were obliged to enlarge the time windows from 40 to 95 minutes by postponing the deadlines. This illustrates the combinatorial nature of the problem, because the addition of only ten jobs created new bottlenecks that delayed at least one job nearly 95 minutes beyond its release time. Wider time windows result in a larger state space and thus greater computation time. Nonetheless, the 60-job problem with 95-minute windows was solved in well under a minute.

The optimal trajectories for selected instances appear in Figs. 6.4–6.6. The horizontal axis represents distance along the track in 10.85-meter units. The vertical axis represents time in 6-second units. Thus the schedule for the 60-job problem spans about 2300 time units, or 230 minutes. The space-time trajectory of the left crane appears as a solid line, and as a dashed line for the right crane. The left crane begins and ends at the leftmost position, and analogously for the right crane. Note that the cranes are at rest most of the time. The trajectories are canonical trajectories as defined above, which ensures a certain consistency in the way the two cranes interact. In the 60-job instance, the left crane finishes before the right crane, which may indicate a poor allocation of jobs to cranes.

Figures 6.7–6.9 track the evolution of state space size over time. The horizontal axis corresponds to time stages, which again are separated by six seconds. The number of time stages exceeds the duration of the optimal trajectory, because trajectories with longer durations are considered in the solution process. The vertical axis is the number of states at each time stage. The state space size remains quite reasonable, never

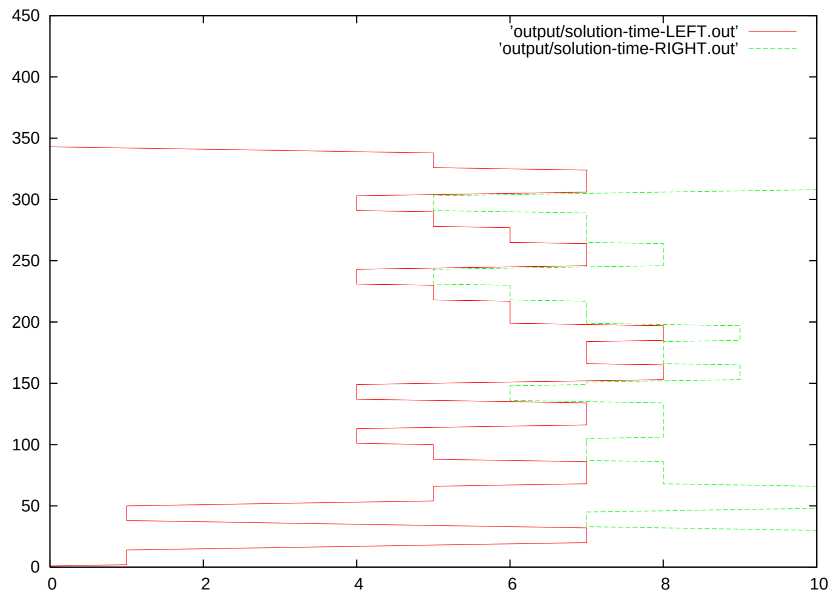


Figure 6.4: Optimal solution of the 10-job instance.

exceeding 2000 states, even though the theoretical maximum is astronomical.

We found computation time to be sensitive to the width of the time windows. Typically, only a few time windows must be wide to allow a feasible solution, because only a few jobs must be delayed so that others may be completed on time. Yet it is difficult or impossible to predict which are the critical jobs. It is therefore necessary to be able to solve problems in which all of the time windows are wide, perhaps on the order of 90 minutes as in the 60-job instance. It was to accommodate wide time windows that we developed the state space reduction techniques of Section 6.6.

Table 6.3 reveals the critical importance of these techniques. For each of the three problem instances, the table shows the average time and state space size required to compute the optimal trajectories for ten different job assignments and sequencings. The assignments and sequencings were those obtained in ten iterations of a heuristic method. Without the state space reduction technique, the dynamic programming algorithm could scale up to only 30 jobs, and even then only for narrow time windows. The width of the time windows is reduced in these experiments to make the problem easier to solve, while still maintaining feasibility. The 30-job instance has a feasible solution with 35-minute time windows, but larger instances require wider time windows to achieve feasibility, and this causes the state space to explode. However, the table shows that the state space reduction technique reduces the number of states by a factor of about 20, and the computation time by a factor of ten. It is this state space reduction that makes the full 60-job problem

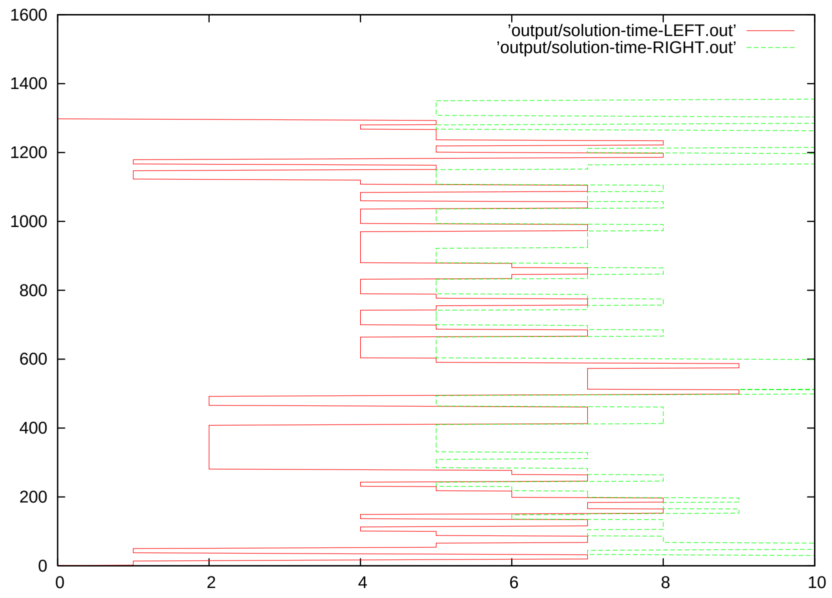


Figure 6.5: Optimal solution of the 30-job instance.

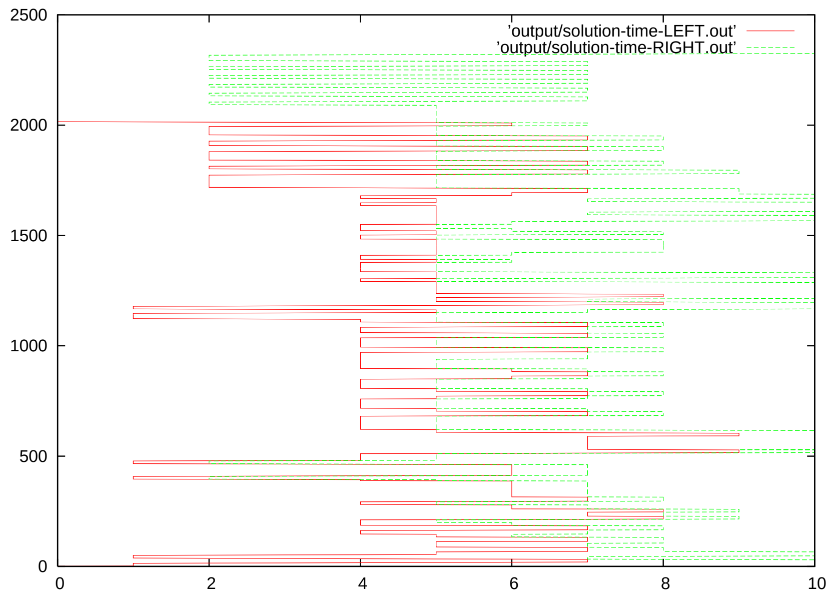


Figure 6.6: Optimal solution of the 60-job instance.

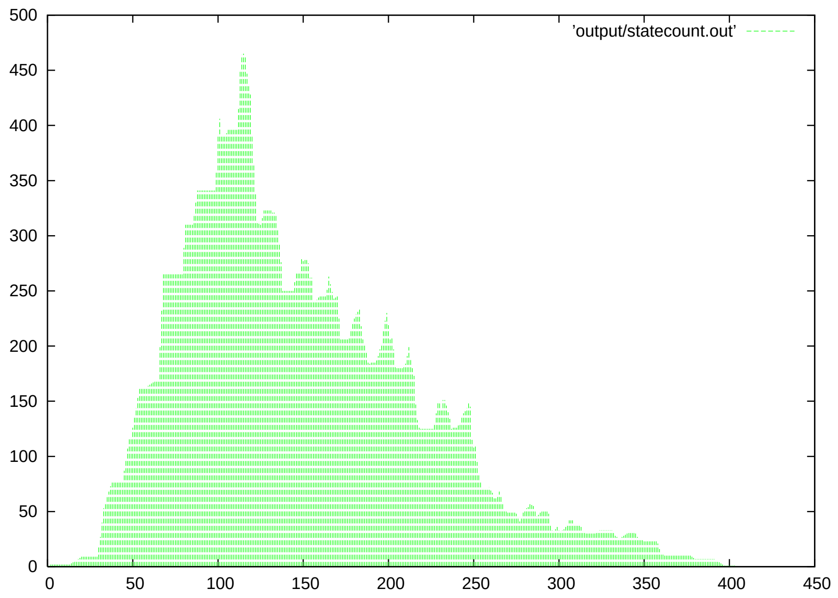


Figure 6.7: Evolution of the state space size for the 10-job instance. The horizontal axis is the time stage, and the vertical axis the number of states.

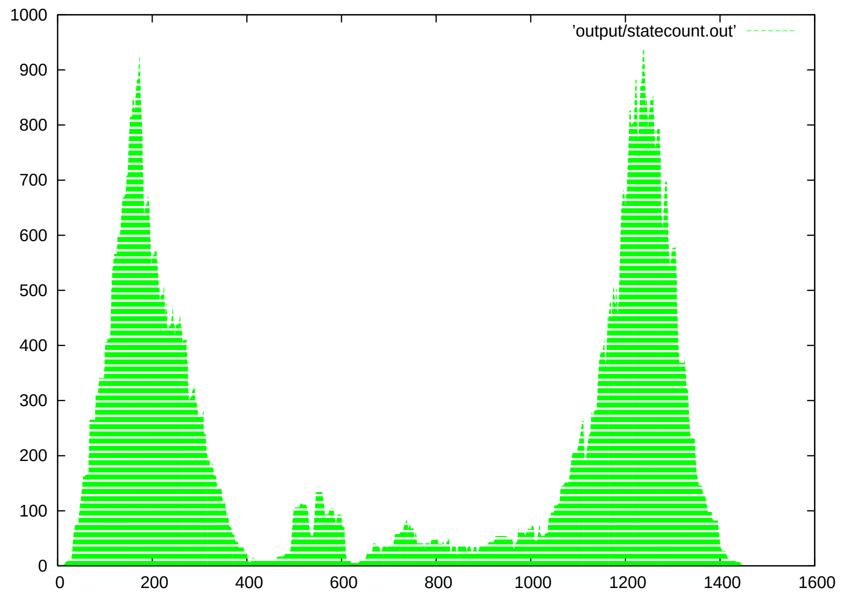


Figure 6.8: Evolution of the state space size for the 30-job instance.

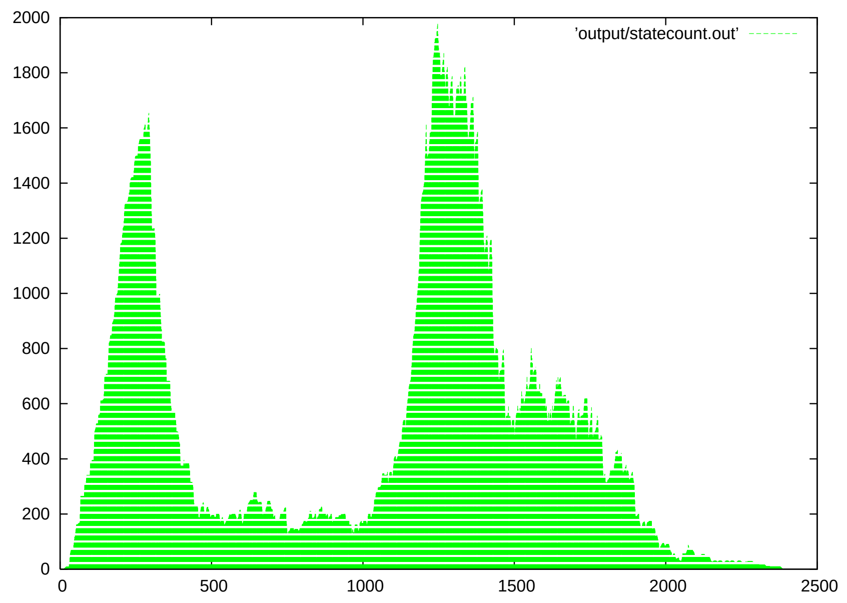


Figure 6.9: Evolution of the state space size for the 60-job instance.

tractable.

6.8 Conclusions and Future Research

We presented a specialized dynamic programming algorithm that computes optimal space-time trajectories for two interacting factory cranes. The state space is economically represented in such a way that medium-sized problems can be solved to optimality. The technique is useful both for solving a significant number of practical problems and as a benchmarking and calibration tool for heuristic methods that solve larger problems. Unlike other methods, it specifies precisely how cranes can yield to one another to minimize delay in carrying out a production schedule.

We also proved structural theorems to show that only certain types of trajectories need be considered to obtain an optimal solution. This not only accelerates solution of the problem, but it permits easier and safer operation of the cranes.

An obvious direction for future research is to attempt to generalize the structural results to three or more cranes. This would allow heuristic methods that are capable of solving large, multi-crane problems to examine fewer trajectories. Another useful research program would be a systematic empirical comparison of heuristic methods with the exact algorithm described here to determine how best to design and tune a heuristic algorithm.

Table 6.1: Possible state transitions for crane c using an interval-valued state variable for processing time.

State at time t	*State at time $t+1$*
1. $(x_{ct}, y_{ct}, [0,0])$	$(x', y_{ct}, [0,0])^1$ or $(x', y_{ct}, [0,1])^{1,2}$ or $(x', y_{ct}, [1,1])^{1,2,3}$
2. $(x_{ct}, y_{ct}, [0,u_2])^4$	$(x_{ct}, y_{ct}, [0,u_2+1])$ or $(x_{ct}, y_{ct}, [1,u_2+1])^{2,4}$
3. $(x_{ct}, y_{ct}, [0,P_{y_{ct}}])$	$(x_{ct}, y_{ct}, [0,P_{y_{ct}}])$ or $(x_{ct}, y_{ct}, [1,P_{y_{ct}}])^3$ or $(x_{ct}, y', [0,0])^5$ or $(x_{ct}, y', [0,1])^{2,5}$ or $(x_{ct}, y', [1,1])^{2,3,5}$
4. $(x_{ct}, y_{ct}, [u_1,u_2])^{4,6}$	$(x_{ct}, y_{ct}, [u_1+1,u_2+1])$
5. $(x_{ct}, y_{ct}, [u_1,P_{y_{ct}}])^6$	$(x_{ct}, y_{ct}, [u_1+1,P_{y_{ct}}])$ or $(x_{ct}, y', [0,0])^5$ or $(x_{ct}, y', [0,1])^{2,5}$ or $(x_{ct}, y', [1,1])^{2,3,5}$

[1]The next location x' is $x_{ct}-1$, x_{ct}, or $x_{ct}+1$.

[2]This transition is possible only if task y_{ct} processes at location x'.

[3]This transition is possible only if task y_{ct} can start no later than time $t+1$.

[4]Here $0 < u_2 < P_{y_{ct}}$.

[5]Task y' is the task that follows task y_{ct} on crane c.

[6]Here $u_1 > 0$.

Table 6.2: Computational results for subsets of the 60-job problem.

Jobs	Time window (mins)	Computation time (sec)
10	40	6.8
20	40	7.6
30	40	15.8
40	40	16.7
50	40	18.8
60	95	48.1

Table 6.3: Effect of state space reduction on state space size and computation time. Each instance is solved for 10 different jobs assignments and sequencings. “Before” and “after” refer to results before and after state space reduction, respectively.

Jobs	Time window	Avg number of states		Peak number of states		Average time (sec)	
	(min)	Before	After	Before	After	Before	After
10	25	3224	139	9477	465	15.8	2.0
20	35	3200	144	22,204	927	82.6	8.6
30	35	3204	216	22,204	940	143.8	15.0

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.

[2] H. R. Andersen. An introduction to binary decision diagrams. Lecture notes, available online, IT University of Copenhagen, 1997.

[3] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In C. Bessiere, editor, *Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2007.

[4] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[5] R. Armstrong, L. Lei, and S. Gu. A bounding scheme for deriving the minimal cycle time of a single-transporter N-stage process with time-window constraints. *European Journal of Operational Research*, 78:130–140, 1994.

[6] P. Baptiste, B. Legeard, M.-A. Manier, and C. Varnier. A scheduling problem optimisation solved with constraint logic programming. In *Second International Conference on the Practical Application of Prolog*, pages 47–66, London, 1994.

[7] B. Becker, M. Behle, F. Eisenbrand, and R. Wimmer. BDDs in a branch and cut framework. In S. Nikoletseas, editor, *Experimental and Efficient Algorithms, Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA 05)*, volume 3503 of *Lecture Notes in Computer Science*, pages 452–463. Springer, 2005.

[8] D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, 2003.

[9] D. Billings, L. Peña, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, January 2002. Special Issue on Games, Computers and Artificial Intelligence.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[11] A. Che and C. Chu. Single-track multi-hoist scheduling problem: A collision-free resolution based on a branch-and-bound approach. *International Journal of Production Research*, 42:2435–2456, 2004.

[12] H. Chen, C. Chu, and J.-M. Proth. Cyclic scheduling of a hoist with time window constraints. *IEEE Transactions on Robotics and Automation*, 14:144–152, 1998.

[13] C. F. Daganzo. The crane scheduling problem. *Transportation Research Part B*, 23:159–175, 1989.

[14] Rina Dechter. *Constraint Processing*. Morgan Kauffman, 2003.

[15] A. Gilpin. *Algorithms for abstracting and solving imperfect information games*. PhD thesis, Carnegie Mellon University, Computer Science Department, 2009.

[16] A. Gilpin and T. Sandholm. A competitive Texas Hold'em poker player via automated abstraction and real-time equilibrium computation. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Boston, MA, 2006.

[17] A. Gilpin and T. Sandholm. Lossless abstraction method for sequential games of imperfect information. *Journal of the ACM*, 54(5), 2007.

[18] A. Gilpin, T. Sandholm, and T. B. Sørensen. Potential-aware automated abstraction of sequential games, and holistic equilibrium analysis of Texas Hold'em poker. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Vancouver, Canada, 2007.

[19] A. Gilpin, T. Sandholm, and T. B. Sørensen. A heads-up no-limit texas hold'em poker player: Discretized betting models and automatically generated equilibrium-finding programs. In *International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Estoril, Portugal, 2008.

[20] J.-L. Goffin. On the convergence rate of subgradient optimization methods. *Mathematical Programming*, 13:329–347, 1977.

[21] T. Hadzic and J. N. Hooker. Postoptimality analysis for integer programming using binary decision diagrams, presented at GICOLAG workshop (Global Optimization: Integrating Convexity, Optimization, Logic Programming, and Computational Algebraic Geometry), Vienna. Technical report, Carnegie Mellon University, 2006.

[22] T. Hadzic and J. N. Hooker. Cost-bounded binary decision diagrams for 0-1 programming. Technical report, Carnegie Mellon University, 2007.

[23] T. Hadzic, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In P. J. Stuckey, editor, *Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *Lecture Notes in Computer Science*, pages 448–462. Springer, 2008.

[24] T. Hadzic, J. N. Hooker, and P. Tiedemann. Propagating separable equalities in an MDD store. In L. Perron and M. A. Trick, editors, *Proceedings of the International Workshop on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for Combintaorial Optimization Problems (CPAIOR 2008)*, volume 5015 of *Lecture Notes in Computer Science*, pages 318–322. Springer, 2008.

[25] J. Hirriart-Urruty and C. Lemaréchal. *Fundamentals of Convex Analysis*. Springer-Verlag, Berlin, 2001.

[26] J. N. Hooker. *Integrated Methods for Optimization*. Springer, 2007.

[27] A. Juditsky, G. Lan, A. Nemirovski, and A. Shapiro. Stochastic approximation approach to stochastic programming, 2007. Working paper.

[28] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *International Journal on Multiple-Valued Logic*, 4:9–62, 1998.

[29] K. H. Kim and Y.-M. Park. A crane scheduling method for port container terminals. *European Journal of Operational Research*, 156:752–768, 2004.

[30] D. Koller, N. Megiddo, and B. von Stengel. Efficient computation of equilibria for extensive two-person games. *Games and Economic Behavior*, 14(2):247–259, 1996.

[31] G. Lan, Z. Lu, and R. Monteiro. Primal-dual first-order methods with $O(1/\epsilon)$ iteration-complexity for cone programming, 2009. To appear in *Math. Prog.*

[32] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959.

[33] L. Lei, R. Armstrong, and S. Gu. Minimizing the fleet size with dependent time-window and single-track constraints. *Operations Research Letters*, 14:91–98, 1993.

[34] L. Lei and T. J. Wang. A proof: The cyclic hoist scheduling problem is NP-complete. Working paper, Rutgers University, 1989.

[35] L. Lei and T. J. Wang. Determining optimal cyclic hoist schedules in a single-hoist electroplating line. *IIE Transactions*, 26:25–33, 1994.

[36] J. Leung and G. Zhang. Optimal cyclic scheduling for printed circuit board production lines with multiple hoists and general processing sequence. *IEEE Transactions on Robotics and Automation*, 19:480–484, 2003.

[37] J. M. Y. Leung, G. Zhang, X. Yang, R. Mak, and K. Lam. Optimal cyclic multi-hoist scheduling: A mixed integer programming approach. *Operations Research*, 52:965–976, 2004.

[38] A. Lim, B. Rodrigues, F. Xiao, and Y. Zhu. Crane scheduling with spatial constraints. *Naval Research Logistics*, 51:386–406, 2004.

[39] J. Liu and Y. Jiang. An efficient optimal solution to the two-hoist no-wait cyclic scheduling problem. *Operations Research*, 53:313–327, 2005.

[40] J. Liu, Y. Jiang, and Z. Zhou. Cyclic scheduling of a single hoist in extended electroplating lines: A comprehensive integer programming solution. *IIE Transactions*, 34:905–914, 2002.

[41] M.-A. Manier and C. Bloch. A classification for hoist schedling problems. *International Journal of Flexible Manufacturing Systems*, 15:37–55, 2003.

[42] L. Mocchia, J.-F. Cordeau, M. Gaudioso, and G. Laporte. A branch-and-cut algorithm for the quay crane scheduling problem in a container terminal. *Naval Research Logistics*, 53:45–59, 2005.

[43] A. Nemirovski. Prox-method with rate of convergence $O(1/t)$ for variational inequalities with Lipschitz continuous monotone operators and smooth convex-concave saddle point problems. *SIAM Journal on Optimization*, 15(1):229–251, 2004.

[44] Y. Nesterov. A method for unconstrained convex minimization problem with rate of convergence $O(1/k^2)$. *Doklady AN SSSR*, 269:543–547, 1983. Translated to English as *Soviet Math. Docl.*

[45] Y. Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*. Applied Optimization. Kluwer Academic Publishers, 2004.

[46] Y. Nesterov. Excessive gap technique in nonsmooth convex minimization. *SIAM Journal on Optimization*, 16(1):235–249, 2005.

[47] Y. Nesterov. Smooth minimization of non-smooth functions. *Math. Program.*, 103(1):127–152, 2005.

[48] W. C. Ng. A branch and bound algorithm for hoist scheduling of a circuit board production line. *International Journal of Flexible Manufacturing Systems*, 8:45–65, 1996.

[49] W. C. Ng. Crane scheduling in container yards with inter-crane interference. *European Journal of Operational Research*, 164:64–78, 2005.

[50] W. C. Ng and J. Leung. Determining the optimal move times for a given cyclic schedule of a material handling hoist. *Computers and Industrial Engineering*, 32:595–606, 1997.

[51] M. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, Cambridge, MA, 1994.

[52] R. I. Peterkofsky and C. F. Daganzo. A branch and bound solution method for the crane scheduling problem. *Transportation Research Part B*, 24:159–172, 1990.

[53] L. W. Phillips and P. S. Unger. Mathematical programming solution of a hoist scheduling problem. *AIIE Transactions*, 8:219–321, 1976.

[54] R. Rodosek and M. Wallace. A generic model and hybrid algorithm for hoist scheduling problems. In M. Maher and J.-F. Puget, editors, *Principle and Practice of Constraint Programming (CP 1998)*, volume 1520, Pisa, 1998. Springer.

[55] I. Romanovskii. Reduction of a game with complete memory to a matrix game. *Soviet Mathematics*, 3:678–681, 1962.

[56] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[57] J. Shi and M. Littman. Abstraction methods for game theoretic poker. In *Computers and Games*, pages 333–345. Springer-Verlag, 2001.

[58] W.-J. van Hoeve, G. Pesant, L.-M. Rousseau, and A. Sabharwal. New Filtering Algorithms for Combinations of Among Constraints. *Constraints*, 14:273–292, 2009.

[59] C. Varnier, A. Bachelu, and P. Baptiste. Resolution of the cyclic multi-hoists scheduling problem with overlapping partitions. *INFOR*, 35:309–324, 1997.

[60] B. von Stengel. Efficient computation of behavior strategies. *Games and Economic Behavior*, 14:220–246, 1996.

[61] B. von Stengel. Equilibrium computation for games in strategic and extensive form. In Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani, editors, *Algorithmic Game Theory*. Cambridge University Press, 2007.

[62] L. Wei and T. J. Wang. The minimum common-cycle algorithm for cycle scheduling of two material handling hoists with time window constraints. *Management Science*, 37:1629–1639, 1991.

[63] G. Yang, D. P. Ju, W. M. Zheng, and K. Lam. Solving multiple hoist scheduling problems by use of simulated annealing. *Transportation Research Part B*, 36:537–555, 2001.

[64] C. Zhang, Y.-W. Wan, J. Liu, and R. J. Linn. Dynamic crane deployment in container storage yards. *Ruan Jian Xue Bao (Journal of Software)*, 12:11–17, 2002.

[65] Z. Zhou and L. Li. A solution for cyclic scheduling of multi-hoists without overlapping. *Annals of Operations Research*, (online), 2008.

[66] Y. Zhu and A. Lim. Crane scheduling with non-crossing constraint. *Journal of the Operational Research Society*, 57:1464–1471, 2006.