Hybrid Approaches to Scheduling and Clustering

by

LATIFE GENÇ-KAYA

Submitted to the Tepper School of Business

in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

at the

CARNEGIE MELLON UNIVERSITY

November 2008

Advisor: John N. Hooker

Abstract

This dissertation consists of four self-contained chapters. The first two chapters concentrate on the circuit constraint. The circuit constraint requires that a sequence of n vertices in a directed graph describe a hamiltonian cycle. The constraint is useful for the succinct formulation of sequencing problems, such as the traveling salesman problem, which it formulates with only one constraint and n variables. In the first chapter, "The Circuit Polytope", we analyze the circuit polytope as an alternative to the traveling salesman polytope as a means of obtaining linear relaxations for sequencing problems. We provide a nearly complete characterization of the circuit polytope by showing how to generate, using a greedy algorithm, all facet-defining inequalities that contain at most $n-4$ terms. We suggest efficient separation heuristics. Finally, we show that proper choice of the numerical values that index the vertices can allow the resulting relaxation to exploit structure in the objective function.

In the second chapter, "A Filter for the Circuit Constraint", we present an incomplete filtering algorithm for the circuit constraint that removes redundant values by eliminating nonhamiltonian edges from the associated graph (i.e., edges that are part of no hamiltonian cycle). We identify nonhamiltonian edges by analyzing a smaller graph with labeled edges that is defined on a separator of the original graph. The complexity of the procedure is roughly the complexity of solving a max flow problem on a graph whose size is related to the size of the separator. We tested the procedure on a few thousand random instances of the circuit constraint having up to 15 variables. We found that it identified all infeasible instances and eliminated about one-third of the redundant domain elements in feasible instances.

In the third chapter, ”Optimal Crane Scheduling”, there is a list of jobs to be assigned and scheduled to two cranes that move on the same track and cannot bypass each other. We present a two-phase algorithm developed for ABB Corporate Research. A local search algorithm assigns jobs to cranes and sequences the jobs on each crane. Then, a specialized dynamic programming algorithm obtains optimal crane space-time trajectories for that assignment and sequencing. Theoretical results are proved to limit the number of crane trajectories that must be considered.

In the last chapter, ”The Minimum Product Cut Problem”, we consider minimum product cut problem that is to find an edge cut on an undirected graph with two distinct nonnegative edge weight functions where the product of cut values is minimum relative to two weight functions. We give a pseudo-polynomial 4-approximation algorithm for the minimum product cut problem that uses parametric search.

Table of Contents

1 The Circuit Polytope **1**

1.1 The Circuit Constraint . . . 1
1.2 The Circuit Polytope . . . 2
1.3 Arbitrary Domains . . . 4
1.4 Overview of the Results . . . 5
1.5 Dimension of the Polytope . . . 6
1.6 Facets of the Polytope . . . 9
1.7 Facet Generation . . . 14
1.8 Generation of Undominated Circuits . . . 16
1.9 Permutation and Two-term Facets . . . 21
1.10 Separation Heuristics . . . 24
1.11 Exploiting Cost Structure . . . 26
1.12 Conclusions and Future Research . . . 28

2 A Filter for the Circuit Constraint **31**

2.1 Motivation . . . 31
2.2 Basic Idea . . . 33
2.3 Previous Work . . . 35
2.4 Definitions . . . 36
2.5 Separator Graph . . . 37
2.6 Finding Separators . . . 39
2.7 A Cardinality Filter . . . 40
2.8 Additional Vertex Degree Filtering . . . 41
2.9 The Algorithm . . . 44
2.10 Computational Results . . . 45
2.11 Conclusions . . . 47

3 Crane Scheduling by Dynamic Programming **49**
3.1 Introduction 49
3.2 The Crane Scheduling Problem 51
3.3 Precedence Constraints 54
3.4 Simplifying the Optimal Control Problem 54
3.5 Dynamic Programming Recursion 60
3.6 Assignment and Sequencing by Local Search 61
3.7 Reduction of the State Space 64
3.8 Experimental Results 66

4 The Minimum Product Cut Problem **75**
4.1 Introduction 75
4.2 Related Work 76
4.2.1 $(1+\epsilon)$-Approximation Algorithm 77
4.3 Preliminaries 78
4.4 An Approximation Algorithm 79
4.4.1 4-Approximation Algorithm 79
4.4.2 A Simple Bound on Breakpoints 81
4.4.3 Binary Search on λ 83
4.5 Special Cases 85
4.5.1 Outerplanar Graphs 85

List of Figures

1.1 Greedy procedure for generating J-circuits that are undominated with respect to $J = J_+ \cup J_-$. 16
1.2 Separation heuristic for finding a set S of facet-defining inequalities with positive coefficients violated by a given point $\hat{x}$. 25
1.3 Separation heuristic for finding a set S of facet-defining inequalities with arbitrary coefficients violated by a given point $\hat{x}$. 25

2.1 *Graph G on vertices $\{1, \ldots, 6\}$ contains the solid edges, and the separator graph G_S on $S = \{1, 2, 3\}$ contains the solid (unlabeled) edges and dashed (labeled) edges within the larger circle. The small circles surround connected components of the separated graph.* 37
2.2 *Flow model for simultaneous gcc and out-degree filtering of nonhamiltonian edges. Heavy lines show the only feasible flow.* 44
2.3 *Filtering algorithm for the circuit constraint.* 45

3.1 Sample space-time trajectory for one task. The shaded vertical bars denote loading and unloading. 55
3.2 Canonical 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. 55
3.3 Minimal trajectory for the left crane (leftmost solid line). 56
3.4 Optimal solution for 10 jobs in the 60-job problem. 68
3.5 Optimal solution for 20 jobs in the 60-job problem. 69
3.6 Optimal solution for 30 jobs in the 60-job problem. 69
3.7 Optimal solution for 40 jobs in the 60-job problem. 70
3.8 Optimal solution for 50 jobs in the 60-job problem. 70
3.9 Optimal solution for 60 jobs in the 60-job problem. 71
3.10 State space size for 10 jobs in the 60-job problem, using 25-minute time windows. . 71
3.11 State space size for 20 jobs in the 60-job problem, using 35-minute time windows. . 72

3.12 State space size for 30 jobs in the 60-job problem, using 35-minute time windows. . 72
3.13 State space size for 40 jobs in the 60-job problem, using 40-minute time windows. . 73
3.14 State space size for 50 jobs in the 60-job problem, using 40-minute time windows. . 73
3.15 State space size for 60 jobs in the 60-job problem, using 55-minute time windows. . 74

4.1 *Algorithm $\mathcal{A}$ for Minimizing Product of Two Nonnegative Linear Costs* 77
4.2 4-*Approximation Algorithm for Minimum Product Cut* 81

List of Tables

1.1 Hyperplanes determined by undominated J-circuits for example (1.12). 16

1.2 (a) Cost data c_{ij}. (b) Values of $h(x_i, x_j)$ when $x_i \leq x_j$ and $h'(x_i, x_j)$ when $x_j \leq x_i$. 28

2.1 Performance of the filtering algorithm for circuit on random graphs that were hamiltonian. The filter successfully identified all random graphs that were nonhamiltonian. 46

3.1 Possible state transitions for crane c using an interval-valued state variable for processing time. 65

3.2 Computational results for the 60-job problem. 67

3.3 Effect of state space reduction on computation time for ten rounds. "Before" and "after" refer to results before and after state space reduction, respectively. 74

Chapter 1

The Circuit Polytope

1.1. The Circuit Constraint

The *circuit constraint* requires that a sequence of vertices in a directed graph define a hamiltonian circuit.

Let G be a directed graph on vertices $1, \ldots, n$, and let variable x_i denote the vertex that follows vertex i in the sequence. The *domain* D_i of each variable x_i (i.e, the set of values x_i can take) is the set of integers j for which (i, j) is an edge of G. The constraint

$$\text{circuit}(x_1, \ldots, x_n) \tag{1.1}$$

requires that $x = (x_1, \ldots, x_n)$ describe a hamiltonian circuit of G. For brevity, we will say that an x satisfying (1.1) is a *circuit*.

More precisely, x is a circuit if $\pi_1, \ldots, \pi_n$ is a permutation of $1, \ldots, n$, where $\pi_1 = 1$ and $\pi_{i+1} = x_{\pi_i}$ for $i = 1, \ldots, n-1$. Thus $\pi_1, \ldots, \pi_n$ indicates the order in which the vertices are visited. For example, if $\{1, 2, 3\}$ is the domain of each variable x_i, then $(x_1, x_2, x_3) = (3, 1, 2)$ is a circuit because $(\pi_1, \pi_2, \pi_3) = (1, 3, 2)$ is a permutation. The circuit goes from 1 to 3 to 2, and back to 1. However, $(x_1, x_2, x_3) = (1, 2, 3)$ is not a circuit, because $(\pi_1, \pi_2, \pi_3) = (1, 1, 1)$ is not a

permutation.

If x is a circuit, the sequence $x_1, \ldots, x_n$ is itself a permutation, but a given permutation x need not be a circuit. In fact, if the domain of each x_i is $\{1, \ldots, n\}$, then $n!$ values of x are permutations but only $(n-1)!$ of these are circuits. In the above example, there are six permutations but only two circuits, namely $(2, 3, 1)$ and $(3, 1, 2)$.

The circuit constraint is useful for formulating combinatorial problems that involve permutations or sequencing. One of the best known such problems is the traveling salesman problem, which may be very succinctly written

$$\begin{aligned} &\min \sum_{i=1}^{n} c_{ix_i} \\ &\text{circuit}(x_1, \ldots, x_n), \quad x_i \in D_i, \; i = 1, \ldots n \end{aligned} \tag{1.2}$$

where c_{ij} is the distance from city i to city j. The objective is to visit each city once, and return to the starting city, in such a way as to minimize the total travel distance.

Domain filtering methods for the circuit constraint appear in [4, 32] and [11] that is Chapter 2 of this dissertation. These can be useful for eliminating infeasible values from the variable domains. The object of this chapter is to study the circuit polytope, so as to obtain a relaxation for the circuit constraint that can be combined with filtering to accelerate solution further. The circuit polytope is an interesting object of study in its own right, one that to our knowledge has not been investigated.

1.2. The Circuit Polytope

The *circuit polytope* is the convex hull of the feasible solutions of (1.1) when G is a complete graph; that is, when each variable domain D_i is $\{1, \ldots, n\}$. To our knowledge, this polytope has not been studied. Rather, the circuit constraint is generally formulated by replacing the variables x_i with 0-1 variables y_{ij}, where $y_{ij} = 1$ if vertex j immediately follows vertex i in the hamiltonian circuit. The

traveling salesman problem (1.2), for example, is typically written

$$
\begin{aligned}
&\min \sum_{ij} c_{ij} y_{ij} \\
&\sum_j x_{ij} = \sum_j y_{ji} = 1, \quad i = 1, \ldots, n && (a) \\
&\sum_{\substack{i \in V \\ j \notin V}} y_{ij} \geq 1, \quad \text{all } V \subset \{1, \ldots, n\} \text{ with } 2 \leq |V| \leq n-2 && (b) \\
&y_{ij} \in \{0, 1\}, \quad \text{all } i, j && (c)
\end{aligned}
\tag{1.3}
$$

The polyhedral structure of problem (1.3) has been intensively analyzed, and surveys of this work may be found in [2, 20, 26].

Rather than introduce 0-1 variables, we analyze the circuit polytope directly. In particular, we provide an almost complete description of the polytope, in the sense that we show how to identify almost all facets of the polytope by identifying *undominated* circuits. A subset of these facet-defining inequalities can be assembled to obtain a tight continuous relaxation of the circuit constraint.

This approach has four possible advantages. (a) The facet-defining inequalities are expressed in terms of n variables, rather than n^2 variables as in the conventional approach. (b) The inequalities are quite different from the traditional traveling salesman cuts and may have complementary strengths. (c) Because the variables can take arbitrary values (not just $1, \ldots, n$), these values can be chosen to exploit structure in the objective function coefficients. (d) We can give a nearly complete description of the circuit polytope, which does not appear to be possible for the 0-1 traveling salesman polytope.

We have not demonstrated these advantages computationally. The goal of this chapter is to lay the theoretical groundwork by describing the circuit polytope, which is an interesting object of study in its own right.

1.3. Arbitrary Domains

A peculiar characteristic of the circuit constraint is that the values of its variables are indices of other variables. Because the vertex immediately after x_i is x_{x_i}, the value of x_i must index a variable. The numbers $1, \ldots, n$ are normally used as indices, but this is an arbitrary choice. One could just as well use any other set of distinct numbers, which would give rise to a different circuit polytope. Thus the circuit polytope cannot be fully understood unless it is characterized for general numerical domains, and not just for $1, \ldots, n$. This also provides more modeling flexibility that can be used to exploit problem structure (Section 1.11).

We therefore generalize the circuit constraint so that each domain D_i is drawn from an arbitrary set $\{v_0, \ldots, v_{n-1}\}$ of nonnegative real numbers. The constraint is written

$$\text{circuit}(x_{v_0}, \ldots, x_{v_{n-1}}) \tag{1.4}$$

It is convenient to assume $v_0 < \cdots < v_{n-1}$. Thus circuit$(x_0, x_{2.3}, x_{3.1})$ is a well-formed circuit constraint if the variable domains are subsets of $\{0, 2.3, 3.1\}$. The nonnegativity of the v_is does not sacrifice generality, since one can always translate the origin so that the feasible points lie in the nonnegative orthant.

To avoid an additional layer of subscripts, we will consistently abuse notation by writing x_{v_i} as x_i. We therefore write the constraint (1.4) as

$$\text{circuit}(x_0, \ldots, x_{n-1}) \tag{1.5}$$

Thus $x = (x_0, \ldots, x_{n-1})$ satisfies (1.5) if and only if $\pi_0, \ldots, \pi_{n-1}$ is a permutation of $0, \ldots, n-1$, where $\pi_0 = 0$ and $v_{\pi_i} = x_{\pi_{i-1}}$ for $i = 1, \ldots n-1$.

We define the circuit polytope $C_n(v)$ with respect to $v = (v_0, \ldots, v_{n-1})$ to be the convex hull of the feasible solutions of (1.5) for full domains; that is, each domain D_i is $\{v_0, \ldots, v_{n-1}\}$. All of the facet-defining inequalities we identify below for full domains are valid inequalities for smaller

domains, even if they may not define facets of the convex hull.

The circuit polytope has a different character than most polytopes studied in combinatorial optimization. Normally the shape of the polytope does not depend on particular numerical values, but only on the structure of the problem. Because the structure of the circuit polytope depends on the domain values, the polytope is partly a discrete and partly a continuous object. This will be reflected in combinatorial and numerical phases of the method for generating facets.

1.4. Overview of the Results

We first examine the dimensionality of the circuit polytope (Theorem 1). We then prove the basic result (Theorems 4 and 5), which is the following. Consider any subset of at most $n-4$ variables, and let a partial solution of the circuit constraint be one that assigns values to these variables only. Then the facet-defining inequalities containing these variables are precisely the valid inequalities defined by affinely independent sets of *undominated* partial solutions. Furthermore, these inequalities are valid if and only if they are satisfied by all undominated partial solutions.

We can therefore identify all facet-defining inequalities with at most $n - 4$ terms if we generate undominated partial solutions, which is a purely combinatorial problem that does not depend on the particular domain values $v_0, \ldots, v_{n-1}$. We solve this problem by describing a greedy algorithm that generates all undominated partial solutions for any given subset of variables (Theorems 6 and 7). We can now identify facet-defining inequalities by solving a continuous, numerical problem. We compute the inequalities defined by affinely independent sets of these partial solutions and check which ones are satisfied by the remaining partial solutions, given the particular numerical values of the domain elements. The inequalities that pass this test are facet-defining.

We next contrast the circuit polytope with the permutation polytope, which contains the circuit polytope, and whose facial structure is well known. We identify a large class of permutation facets that are also circuit facets (Corollary 8). The circuit polytope is more complicated than the permutation polytope, however, and unlike the permutation polytope, its structure depends on the domain

values. We also explicitly identify all two-term facets of the circuit polytope (Corollary 9).

We then address the separation problem, which is the problem of identifying facet-defining inequalities that are violated by a solution of the current relaxation of the problem. We describe two separation heuristics, one of which seeks separating inequalities with all positive coefficients, and one which seeks inequalities with arbitrary coefficients.

We conclude by showing how knowledge of the circuit polytope for arbitrary domains can allow one to exploit cost structure in the objective function of the problem.

1.5. Dimension of the Polytope

We begin by establishing the dimension of the circuit polytope.

Theorem 1. *The dimension of the circuit polytope $C_n(v)$ is $n-2$ for $n = 2, 3$ and $n-1$ for $n \geq 4$.*

Proof. The polytope $C_n(v)$ is a point (v_1, v_0) for $n = 2$ and the line segment from (v_1, v_2, v_0) to (v_2, v_0, v_1) for $n = 3$. In either case the dimension is $n - 2$.

To prove the theorem for $n \geq 4$, note first that all feasible points for (1.5) satisfy

$$\sum_{i=0}^{n-1} x_i = \sum_{i=0}^{n-1} v_i \tag{1.6}$$

(Recall that x_i is shorthand for x_{v_i}.) Thus, $C_n(v)$ has dimension at most $n - 1$. To show it has dimension exactly $n - 1$, it suffices to exhibit n affinely independent points in $C_n(v)$. Consider the following n permutations of $v_0, \ldots, v_{n-1}$, where the first $n-1$ permutations consist of v_0 followed by cyclic permutations of $v_1, \ldots, v_{n-1}$. The last permutation is obtained by swapping v_{n-2} and

v_{n-1} in the first permutation:

$$
\begin{array}{llllllll}
v_0, & v_1, & v_2, & \ldots, & v_{n-3}, & v_{n-2}, & v_{n-1} \\
v_0, & v_2, & v_3, & \ldots, & v_{n-2}, & v_{n-1}, & v_1 \\
v_0, & v_3, & v_4, & \ldots, & v_{n-1}, & v_1, & v_2 \\
 & & & \vdots & & & \\
v_0, & v_{n-2}, & v_{n-1}, & \ldots, & v_{n-5}, & v_{n-4}, & v_{n-3} \\
v_0, & v_{n-1}, & v_1, & \ldots, & v_{n-4}, & v_{n-3}, & v_{n-2} \\
v_0, & v_1, & v_2, & \ldots, & v_{n-3}, & v_{n-1}, & v_{n-2}
\end{array}
\tag{1.7}
$$

The rows of the following matrix correspond to circuit representations of the above permutations. Thus row i contains the values $x_0, \ldots, x_{n-1}$ for the ith permutation in (1.7).

$$
\begin{bmatrix}
v_1 & v_2 & v_3 & \cdots & v_{n-2} & v_{n-1} & v_0 \\
v_2 & v_0 & v_3 & \cdots & v_{n-2} & v_{n_1} & v_1 \\
v_3 & v_2 & v_0 & \cdots & v_{n-2} & v_{n-1} & v_1 \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\
v_{n-2} & v_2 & v_3 & \cdots & v_0 & v_{n-1} & v_1 \\
v_{n-1} & v_2 & v_3 & \cdots & v_{n-2} & v_0 & v_1 \\
v_1 & v_2 & v_3 & \cdots & v_{n-1} & v_0 & v_{n-2}
\end{bmatrix}
\tag{1.8}
$$

Since each row of (1.8) is a point in $C_n(v)$, it suffices to show that the rows are affinely independent.

Subtract $[v_{n-1} \;\; v_2 \;\; v_3 \;\; \cdots \;\; v_{n-2} \;\; v_{n-1} \;\; v_1]$ from every row of (1.8) to obtain

$$\begin{bmatrix} v_1 - v_{n-1} & 0 & 0 & \cdots & 0 & 0 & v_0 - v_1 \\ v_2 - v_{n-1} & v_0 - v_2 & 0 & \cdots & 0 & 0 & 0 \\ v_3 - v_{n-1} & 0 & v_0 - v_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ v_{n-2} - v_{n-1} & 0 & 0 & \cdots & v_0 - v_{n-2} & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & v_0 - v_{n-1} & 0 \\ v_1 - v_{n-1} & 0 & 0 & \cdots & v_{n-1} - v_{n-2} & v_0 - v_{n-1} & v_{n-2} - v_1 \end{bmatrix} \tag{1.9}$$

The rows of (1.8) are affinely independent if and only if the rows of (1.9) are. It now suffices to show that (1.9) is nonsingular, and we do so through a series of row operations. The first step is to subtract $(v_{n-2} - v_1)/(v_0 - v_1)$ times row 1, $(v_{n-1} - v_{n-2})/(v_0 - v_{n-2})$ times row $n-2$, and row $n-1$ from row n to obtain

$$\begin{bmatrix} v_1 - v_{n-1} & 0 & 0 & \cdots & 0 & 0 & v_0 - v_1 \\ v_2 - v_{n-1} & v_0 - v_2 & 0 & \cdots & 0 & 0 & 0 \\ v_3 - v_{n-1} & 0 & v_0 - v_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ v_{n-2} - v_{n-1} & 0 & 0 & \cdots & v_0 - v_{n-2} & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & v_0 - v_{n-1} & 0 \\ E_n & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix} \tag{1.10}$$

where

$$E_n = -\frac{v_{n-1} - v_{n-2}}{v_{n-2} - v_0}(v_{n-1} - v_{n-2}) - (v_{n-1} - v_1)$$

Interchange the first and last rows of (1.10) to obtain

$$\begin{bmatrix} E_n & 0 & 0 & \cdots & 0 & 0 & 0 \\ v_2 - v_{n-1} & v_0 - v_2 & 0 & \cdots & 0 & 0 & 0 \\ v_3 - v_{n-1} & 0 & v_0 - v_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ v_{n-2} - v_{n-1} & 0 & 0 & \cdots & v_0 - v_{n-2} & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & v_0 - v_{n-1} & 0 \\ v_1 - v_{n-1} & 0 & 0 & \cdots & 0 & 0 & v_0 - v_1 \end{bmatrix} \tag{1.11}$$

Note that $E_n < 0$ since $v_0 < \cdots < v_{n-1}$. Thus (1.11) is a lower triangular matrix with nonzero diagonal elements and is therefore nonsingular. □

As an example, consider

$$\text{circuit}(x_0, \ldots, x_6) \tag{1.12}$$

where each x_i has domain $\{v_0, \ldots, v_6\} = \{2, 5, 6, 7, 9, 10, 12\}$. The corresponding polytope $C_7(2, 5, 6, 7, 9, 10, 12)$ has dimension 6 and satisfies

$$x_0 + \cdots + x_6 = 51 \tag{1.13}$$

which describes its affine hull.

1.6. Facets of the Polytope

We now describe facets of the circuit polytope $C_n(v)$. The following lemma is key.

Lemma 2. *Suppose that the inequality*

$$\sum_{j \in J} a_j x_j \geq \alpha \tag{1.14}$$

is valid for circuit$(x_0, \ldots, x_{n-1})$ *and is satisfied as an equation by at least one circuit* x*. If* $|J| \leq n-4$ *and*

$$\sum_{j=0}^{n-1} d_j x_j = \delta \tag{1.15}$$

is satisfied by all circuits x *that satisfy (1.14) as an equation, then* $d_j = 0$ *for all* $j \notin J$*.*

Proof. It suffices to prove that $d_{j_0} = d_{j_1} = d_{j_3} = d_{j_4} = 0$ for any subset of four indices $j_0, \ldots, j_3 \notin J$. Note first that we can use (1.6) to eliminate any variable (say, x_{j_0}) from (1.15) and obtain an equation of the form (1.15) in which $d_{j_0} = 0$. We therefore assume without loss of generality that $d_{j_0} = 0$.

Now let x^0 be any circuit that satisfies (1.14) as an equation, and let the permutation described by x^0 be

$$v_0, \ldots, v_{j_0-1}, v_{j_0}, v_{j_0+1}, \ldots, v_{j_1-1}, v_{j_1}, v_{j_1+1}, \ldots, v_{j_2-1}, v_{j_2}, v_{j_2+1}, \ldots, v_{j_3-1}, v_{j_3}$$

Consider the circuits $x^1, \ldots, x^5$ that describe the following permutations, respectively:

$$v_0, \ldots, v_{j_0-1}, v_{j_0}, v_{j_2+1}, \ldots, v_{j_3-1}, v_{j_3}, v_{j_0+1}, \ldots, v_{j_1-1}, v_{j_1}, v_{j_1+1}, \ldots, v_{j_2-1}, v_{j_2}$$

$$v_0, \ldots, v_{j_0-1}, v_{j_0}, v_{j_1+1}, \ldots, v_{j_2-1}, v_{j_2}, v_{j_2+1}, \ldots, v_{j_3-1}, v_{j_3}, v_{j_0+1}, \ldots, v_{j_1-1}, v_{j_1}$$

$$v_0, \ldots, v_{j_0-1}, v_{j_0}, v_{j_1+1}, \ldots, v_{j_2-1}, v_{j_2}, v_{j_0+1}, \ldots, v_{j_1-1}, v_{j_1}, v_{j_2+1}, \ldots, v_{j_3-1}, v_{j_3}$$

$$v_0, \ldots, v_{j_0-1}, v_{j_0}, v_{j_0+1}, \ldots, v_{j_1-1}, v_{j_1}, v_{j_2+1}, \ldots, v_{j_3-1}, v_{j_3}, v_{j_1+1}, \ldots, v_{j_2-1}, v_{j_2}$$

$$v_0, \ldots, v_{j_0-1}, v_{j_0}, v_{j_2+1}, \ldots, v_{j_3-1}, v_{j_3}, v_{j_1+1}, \ldots, v_{j_2-1}, v_{j_2}, v_{j_0+1}, \ldots, v_{j_1-1}, v_{j_1}$$

We obtain $x^1, \ldots, x^5$ from x^0 by viewing the permutation represented by the latter as a concatenation of four subsequences, each ending in one of the values v_{j_i}. We fix the first subsequence and obtain x^1 and x^2 by cyclically permuting the remaining three subsequences. We obtain x^3, x^4 and x^5 by interchanging a pair of subsequences.

Note that variables $x_{j_0}, \ldots, x_{j_3}$ have the values shown below in each circuit x^i:

x_{j_0}	x_{j_1}	x_{j_2}	x_{j_3}	
v_{j_0+1}	v_{j_1+1}	v_{j_2+1}	v_0	(x^0)
v_{j_2+1}	v_{j_1+1}	v_0	v_{j_0+1}	(x^1)
v_{j_1+1}	v_0	v_{j_2+1}	v_{j_0+1}	(x^2)
v_{j_1+1}	v_{j_2+1}	v_{j_0+1}	v_0	(x^3)
v_{j_0+1}	v_{j_2+1}	v_0	v_{j_1+1}	(x^4)
v_{j_2+1}	v_0	v_{j_0+1}	v_{j_1+1}	(x^5)

and all other variables x_j have value x_j^0 in each circuit x^i. Thus all six circuits $x^0, \ldots, x^5$ satisfy (1.14) as an equation, so that $dx^i = \delta$ for $i = 0, \ldots, 5$. This implies

$$\frac{1}{2}\begin{bmatrix} (dx^0 + dx^1 + dx^5) - (dx^2 + dx^3 + dx^4) \\ (dx^0 + dx^2 + dx^5) - (dx^1 + dx^3 + dx^4) \\ (dx^0 + dx^3 + dx^5) - (dx^1 + dx^2 + dx^4) \\ (dx^0 + dx^2 + dx^4) - (dx^1 + dx^3 + dx^5) \\ (dx^0 + dx^4 + dx^5) - (dx^1 + dx^2 + dx^3) \\ (dx^0 + dx^1 + dx^3) - (dx^2 + dx^4 + dx^5) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Substituting the values of $x^0, \ldots, x^5$, we obtain

$$\begin{bmatrix} v_{j_2+1} - v_{j_1+1} & v_{j_1+1} - v_{j_2+1} & 0 & 0 \\ 0 & v_0 - v_{j_2+1} & v_{j_2+1} - v_0 & 0 \\ 0 & 0 & v_{j_0+1} - v_0 & v_0 - v_{j_0+1} \\ v_{j_0+1} - v_{j_2+1} & 0 & v_{j_2+1} - v_{j_0+1} & 0 \\ v_{j_0+1} - v_{j_1+1} & 0 & 0 & v_{j_1+1} - v_{j_0+1} \\ 0 & v_{j_1+1} - v_0 & 0 & v_0 - v_{j_1+1} \end{bmatrix} \begin{bmatrix} d_{j_0} \\ d_{j_1} \\ d_{j_2} \\ d_{j_3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

from which we can conclude that $d_{j_0} = d_{j_1} = d_{j_2} = d_{j_3}$. But since $d_{j_0} = 0$, this proves the lemma. □

It will be convenient denote by $x(J)$ the tuple $(x_{j_0}, \ldots, x_{j_m})$ when $J = \{j_0, \ldots, j_m\}$. We say that $x(J)$ is a *J-circuit* if it creates no cycles and is therefore a partial solution of the circuit constraint. That is, $x(J)$ is a J-circuit if $\pi_0, \ldots, \pi_m$ are all distinct, where $\pi_0 = j_0$ and $v_{\pi_i} = x_{\pi_{i-1}}$ for $i = 1, \ldots, m$. We will need the following lemma.

Lemma 3. *If $\bar{x}(J)$ is a J-circuit, then there is a circuit x such that $x(J) = \bar{x}(J)$.*

Proof. Let $J = \{j_0, \ldots, j_m\}$, and let $\{v_{i_0}, \ldots, v_{i_r}\}$ be the subset of domain values $v_0, \ldots, v_{n-1}$ that occur in neither $\{v_{j_0}, \ldots, v_{j_m}\}$ nor $\{\bar{x}_{j_0}, \ldots, \bar{x}_{j_m}\}$. Consider the directed graph $G_{\bar{x}(J)}$ that contains a vertex v_i for each $i \in \{0, \ldots, n-1\}$, a directed edge $(v_{j_k}, \bar{x}_{j_k})$ for $k = 0, \ldots, m$, and a directed edge $(v_{i_k}, v_{i_{k+1}})$ for each $k = 0, \ldots, r-1$. The maximal subchains of $G_{\bar{x}(J)}$ have the form

$$
\begin{array}{l}
v_{k_1} \to \cdots \to v_{k'_1} \to \bar{x}_{k'_1} \\
v_{k_2} \to \cdots \to v_{k'_2} \to \bar{x}_{k'_2} \\
\vdots \\
v_{k_p} \to \cdots \to v_{k'_p} \to \bar{x}_{k'_p} \\
v_{i_0} \to \cdots \to v_{i_r}
\end{array}
$$

where possibly $k_t = k'_t$ for some values of t. Because maximal subchains are disjoint, we can form a hamiltonian circuit in $G_{\bar{x}(J)}$ by linking the last element of each subchain to the first element of the next, and linking v_{i_r} to v_{k_1}. Let $v_{s_0}, \ldots, v_{s_{n-1}}$ be the resulting circuit. Then if x is given by $x_i = v_{s_{(i+1) \bmod n}}$ for $j = 0, \ldots, n-1$, then x is a circuit and $x(J) = \bar{x}(J)$. □

The idea of domination is central to characterizing facets of $C_n(v)$. Let $J = J_+ \cup J_-$ (with $J_+ \cap J_- = \emptyset$) be a subset of variable indices. For $i \in J$ we say that $x_i \preceq y_i$ if $i \in J_+$ and $x_i \leq y_i$, or $i \in J_-$ and $x_i \geq y_i$. Also $x_i \prec y_i$ if $x_i \preceq y_i$ and $x_i \neq y_i$. We say that $x'(J)$ *dominates* $x(J)$ with respect to $J = J_+ \cup J_-$ when $x'_j \preceq x_j$ for all $j \in J$. A J-circuit $x(J)$ is *undominated* if no other J-circuit dominates it.

The following theorem provides the basis for generating facets of $C_n(v)$ by generating undominated J-circuits.

Theorem 4. *Let S be the set of J-circuits that are undominated with respect to $J = J_+ \cup J_-$, where $1 \leq |J| \leq n-4$. Consider any subset of $|J|$ affinely independent J-circuits in S. If these J-circuits satisfy*

$$\sum_{j \in J} a_j x_j = \alpha, \quad \textit{where } a_j > 0 \; for \; j \in J_+ \; and \; a_j < 0 \; for \; j \in J_- \tag{1.16}$$

and the remaining J-circuits in S satisfy (1.14), then (1.14) defines a facet of $C_n(v)$.

Proof. Let $S = \{x^0(J), \ldots, x^m(J)\}$, and suppose $S' \subset S$ is a set of $|J|$ affinely independent circuits. We first show that (1.14) is valid; that is, satisfied by any circuit x. Because S contains all undominated J-circuits, $x(J)$ is dominated by some $x^i(J) \in S$ with respect to $J = J_+ \cup J_-$, which means that $a_j(x_j - x_j^i) \geq 0$ for all $j \in J$. Thus we have

$$\sum_{j \in J} a_j x_j \geq \sum_{j \in J} a_j x_j^i \geq \alpha$$

because $x^i(J)$ satisfies (1.14), and so x satisfies (1.14).

Now let (1.15) be any equation satisfied by all circuits x that satisfy (1.14) as an equation. Because $|J| \geq 1$ and S is therefore nonempty, at least one J-circuit $x^i(J) \in S$ satisfies (1.14) as an equation. Lemma 3 now implies that at least one circuit x^i satisfies (1.14) as an equation. Thus since $|J| \leq n-4$, we have from Lemma 2 that $d_j = 0$ for all $j \notin J$, so that

$$\sum_{j \in J} d_j x_j = \delta \tag{1.17}$$

Because the J-circuits in S' are affinely independent and satisfy (1.16) and (1.17), these two equations are the same up to a scalar multiple. Therefore, any equation satisfied by all circuits that satisfy (1.14) as an equation has the form (1.17). This means that (1.14) defines a facet of the circuit polytope. □

We show now that the previous theorem completely characterizes facet-defining inequalities having no more than $n-4$ terms.

Theorem 5. *Consider any inequality (1.14) that is facet-defining for a circuit polytope $C_n(v)$, and let $J_+ = \{j \mid a_j > 0\}$ and $J_- = \{j \mid a_j < 0\}$. Then there are affinely independent J-circuits $x^0(J), \ldots, x^{|J|-1}(J)$ that are undominated with respect to $J = J_+ \cup J_-$ and satisfy (1.16).*

Proof. Any facet-defining inequality (1.14) is satisfied as an equation by $n-1$ affinely independent circuits $\bar{x}^0, \ldots, \bar{x}^{n-1}$. Then $\{\bar{x}^0(J), \ldots, \bar{x}^{n-1}(J)\}$ has some subset $\{\bar{x}^{j_0}(J), \ldots, \bar{x}^{j_{|J|-1}}(J)\}$ of $|J|$ affinely independent J-circuits. These are undominated with respect to $J = J_+ \cup J_-$, because otherwise, some J-circuit $\hat{x}(J)$ strictly dominates some $\bar{x}^{j_i}(J)$ with respect to $J = J_+ \cup J_-$. Also by Lemma 3, $\hat{x}(J)$ is part of some circuit $\hat{x}$. This means

$$\sum_{j \in J} a_j \hat{x}_j < \sum_{j \in J} a_j \bar{x}_j^{j_i} = \alpha$$

and $\hat{x}$ violates (1.14). This implies that (1.14) is not valid and therefore is not facet-defining as assumed. □

1.7. Facet Generation

The results of the previous section indicate how to generate all facet-defining inequalities for $C_n(v)$ having at most $n-4$ terms. To generate all such facet-defining inequalities (1.14) in which $a_j > 0$ for $j \in J_+$ and $a_j < 0$ for $j \in J_-$, first generate the set S of all J-circuits that are undominated with respect to $J = J_+ \cup J_-$. Then consider all affinely independent subsets of $|J|$ J-circuits in S. Each subset uniquely defines an equation (1.16) up to scalar multiple. If the remaining J-circuits in S satisfy (1.14), then list (1.14) as a facet-defining inequality.

Note that we do not identify a facet by generating $n-1$ affinely independent circuits that define the facet, as this would be a difficult task in general. Rather, we generate $|J|$ affinely independent J-circuits that define the coefficients of an inequality containing $|J|$ terms. This inequality defines

a facet if it is valid, which we can easily check. In the next section we will show how to generate the undominated partial solutions efficiently with a greedy procedure.

As an example, we identify all facet-defining inequalities of the form

$$a_0x_0 + a_2x_2 + a_3x_3 \geq \alpha, \quad \text{with } a_0, a_2, a_3 > 0 \tag{1.18}$$

for example (1.12). Four J-circuits $\bar{x}^i(J)$ are undominated with respect to $J = J_+ = \{0, 2, 3\}$. They are independent of the particular domain values $v_0, \ldots, v_6$ and can therefore be written

$$\begin{aligned} \bar{x}^1(J) &= (v_1, v_0, v_2) \\ \bar{x}^2(J) &= (v_1, v_3, v_0) \\ \bar{x}^3(J) &= (v_2, v_1, v_0) \\ \bar{x}^4(J) &= (v_3, v_0, v_1) \end{aligned} \tag{1.19}$$

(We will show how to obtain these J-circuits using a greedy algorithm in the next section.) There are four subsets of three J-circuits ($|J| = 3$), shown in Table 1.1, and each uniquely defines a hyperplane and a corresponding inequality. The first inequality, defined by $\bar{x}^1(J)$, $\bar{x}^2(J)$, and $\bar{x}^3(J)$, is satisfied by the remaining J-circuit $\bar{x}^4(J)$, and similarly for the third inequality. The second and fourth inequalities, however, are violated by the remaining J-circuit and are not valid. This means there are exactly two facets defined by inequalities of the form (1.18), namely those defined by

$$\begin{aligned} 8x_0 + 4x_2 + 5x_3 &\geq 78 \\ 3x_0 + 7x_2 + 6x_3 &\geq 65 \end{aligned}$$

Now we find all facet-defining inequalities of the form (1.18) but with $a_0, a_2 > 0$ and $a_3 < 0$, so that $J_+ = \{0, 2\}$ and $J_- = \{3\}$. In this case, there is only one undominated circuit, $x(J) = (v_1, v_0, v_6)$. Because we do not have three undominated circuits to define a hyperplane, there are no facet-defining inequalities of this form.

Table 1.1: Hyperplanes determined by undominated J-circuits for example (1.12).

Defining J-circuits	Uniquely defined hyperplane $a(J)x(J) = \alpha$	Is $a(J)x(J) \geq \alpha$ valid?
$\bar{x}^1(J), \bar{x}^2(J), \bar{x}^3(J)$	$8x_0 + 4x_2 + 5x_3 = 78$	Yes, violated by $\bar{x}^4(J)$
$\bar{x}^1(J), \bar{x}^2(J), \bar{x}^4(J)$	$5x_0 + 8x_2 + 10x_3 = 101$	No, violated by $\bar{x}^3(J)$
$\bar{x}^1(J), \bar{x}^3(J), \bar{x}^4(J)$	$3x_0 + 7x_2 + 6x_3 = 65$	Yes, satisfied by $\bar{x}^2(J)$
$\bar{x}^2(J), \bar{x}^3(J), \bar{x}^4(J)$	$6x_0 + 3x_2 + x_3 = 53$	No, violated by $\bar{x}^1(J)$

For each ordering $j_0, \ldots, j_m$ of the elements of J:
  Let $\bar{J} = \{0, \ldots, n-1\}$ and $J' = \emptyset$.
  For $i = 0, \ldots, m$:
    Add j_i to J'.
    If $j_i \in J_+$ then let $\bar{x}_{j_i}$ be the minimum value v_k in $\{v_i \mid i \in \bar{J}\}$
      such that $\bar{x}(J')$ is a J'-circuit.
    Else let $\bar{x}_{j_i}$ be the maximum value v_k in $\{v_i \mid i \in \bar{J}\}$
      such that $\bar{x}(J')$ is a J'-circuit.
    Remove k from $\bar{J}$.
  Add $\bar{x}(J)$ to the list of undominated J-circuits.

Figure 1.1: Greedy procedure for generating J-circuits that are undominated with respect to $J = J_+ \cup J_-$.

1.8. Generation of Undominated Circuits

A simple greedy procedure can be used to generate all J-circuits $\bar{x}(J)$ that are undominated with respect to $J = J_+ \cup J_-$. It is applied for each ordering $j_0, \ldots, j_m$ of the elements of J. First, let $\bar{x}_{j_0}$ be the smallest domain value v_i if $j_0 \in J_+$, or the largest if $j_o \in J_-$. Then let $\bar{x}_{j_1}$ be the smallest (or largest) remaining domain value that does not create a cycle. Continue until all $\bar{x}_j$ for $j \in J$ are defined. The precise algorithm appears in Fig. 1.1.

Theorem 6. *The greedy procedure of Fig. 1.1 generates J-circuits that are undominated with respect to $J = J_+ \cup J_-$.*

Proof. Let $\bar{x}(J)$ be a J-circuit generated by the procedure for a given ordering $j_0, \ldots, j_m$. To see that $\bar{x}(J)$ is undominated with respect to $J = J_+ \cup J_-$, assume otherwise. Then there exists a J-circuit $\bar{y}(J)$ such that $\bar{x}(J) \succeq \bar{y}(J)$ and $\bar{x}_{j_t} \succ \bar{y}_{j_t}$ for some $t \in \{0, \ldots, m\}$. Let t be the smallest

such index, so that $\bar{x}_{j_k} = \bar{y}_{j_k}$ for $k = 0, \ldots, t-1$. This contradicts the greedy construction of $\bar{x}$, because $\bar{y}_{j_t}$ is available when $\bar{x}_{j_t}$ is assigned to x_{j_t}. $\square$

For example, the undominated circuits (1.19) for circuit constraint (1.12) can be generated by considering the six orderings of $J = J_+ = \{0, 2, 3\}$, listed on the left below. The resulting undominated J-circuits appear on the right.

$$
\begin{array}{ll}
0, 2, 3 & (v_1, v_0, v_2) = \bar{x}^1(J) \\
0, 3, 2 & (v_1, v_3, v_0) = \bar{x}^2(J) \\
2, 0, 3 & (v_1, v_0, v_2) = \bar{x}^1(J) \\
2, 3, 0 & (v_3, v_0, v_1) = \bar{x}^4(J) \\
3, 0, 2 & (v_1, v_3, v_0) = \bar{x}^2(J) \\
3, 2, 0 & (v_2, v_1, v_0) = \bar{x}^3(J)
\end{array}
$$

When $J_+ = \{0, 2\}$ and $J_- = \{3\}$, all six orderings result in the same J-circuit (v_1, v_0, v_6).

The greedy procedure not only generates undominated J-circuits, but generates all of them.

Theorem 7. *Any undominated circuit with respect to $J = J_+ \cup J_-$ can be generated in a greedy fashion for some ordering of the indices in J.*

Proof. Let $\bar{x}$ be a circuit that is undominated with respect to $J = J_+ \cup J_-$, where $|J| = m$, $J_+ = \{i_0, \ldots, i_p\}$ and $J_- = \{j_0, \ldots, j_q\}$. Suppose the variables are indexed so that $\bar{x}_{i_\ell} < \bar{x}_{i_{\ell'}}$ when $\ell < \ell'$ and $i_\ell, i_{\ell'} \in J_+$, and $\bar{x}_{j_\ell} > \bar{x}_{j_{\ell'}}$ when $\ell < \ell'$ and $j_\ell, j_{\ell'} \in J_-$.

Let $\bar{y}$ be a J-circuit that is generated in greedy fashion with respect to an ordering $k_0, \ldots, k_m$ determined in the following way. Let r and s index the elements of J_+ and J_-, respectively, with $r = 0$ and $s = 0$ initially. Also let $V = \{v_0, \ldots, v_{n-1}\}$ initially. At each step of the procedure, we assign the greedy value to x_j for the next $j \in J_+$ unless we can avoid deviating from $\bar{x}$ by assigning the greedy value to x_j for the next $j \in J_-$, or unless we have already assigned values to x_j for all $j \in J_+$. That is, for $\ell = 0, \ldots, m$, do the following. Let $v_{\min}$ be the smallest value in V such

that setting $x_{i_r} = v_{\min}$ does not create a cycle. Let $v_{\max}$ be the largest value in V such that setting $x_{j_s} = v_{\max}$ does not create a cycle. If $r \leq p$, and if $\bar{x}_{i_r} = v_{\min}$ or $\bar{x}_{j_s} < v_{\max}$ or $s > q$, then let $k_\ell = i_r$, let $\bar{y}_{i_r} = v_{\min}$, set $r = r + 1$, and remove $v_{\min}$ from V. Otherwise, let $k_\ell = j_s$, let $\bar{y}_{j_s} = v_{\max}$, set $s = s + 1$, and remove $v_{\max}$ from V. Then $(\bar{y}_0, \ldots, \bar{y}_m)$ is the greedy solution with respect to the ordering $k_0, \ldots, k_m$.

We claim that $\bar{x}_{j_\ell} = \bar{y}_{j_\ell}$ for $\ell = 0, \ldots, m$, which suffices to prove the theorem. Supposing to the contrary, let $\bar{\ell}$ be the smallest index for which $\bar{x}_{k_{\bar{\ell}}} \neq \bar{y}_{k_{\bar{\ell}}}$. Clearly $\bar{x}_{k_{\bar{\ell}}} \prec \bar{y}_{k_{\bar{\ell}}}$ is inconsistent with the greedy choice, because $\bar{x}_{k_{\bar{\ell}}}$ is available when $\bar{y}_{k_{\bar{\ell}}}$ is assigned to $x_{k_{\bar{\ell}}}$. Thus we have $\bar{x}_{k_{\bar{\ell}}} \succ \bar{y}_{k_{\bar{\ell}}}$

By hypothesis, $\bar{x}$ is undominated with respect to $J = J_+ \cup J_-$. We therefore have $\bar{x}_{k_\ell} \prec \bar{y}_{k_\ell}$ for some $\ell \in \{\bar{\ell}+1, \ldots, m\}$. Let $\hat{\ell}$ be the smallest such index. Then there are two cases: (1) $k_{\bar{\ell}}$ and $k_{\hat{\ell}}$ are both in J_+ or both in J_-, or (2) they are in different sets.

Case 1: $k_{\bar{\ell}}$ and $k_{\hat{\ell}}$ are both in J_+ or both in J_-. We will suppose that both are in J_+. The argument is symmetric if both are in J_-.

Let t be the index such that $i_t = k_{\bar{\ell}}$, and u the index such that $i_u = k_{\hat{\ell}}$. Then $\bar{x}_{j_t} > \bar{y}_{j_t}$ because $\bar{x}_{j_t} \succ \bar{y}_{j_t}$ and $j_t \in J_+$. Let t' be the largest index in $\{t, \ldots, u-1\}$ such that $\bar{x}_{i_{t'}} > \bar{y}_{i_{t'}}$. We know that t' exists because $\bar{x}_{i_t} > \bar{y}_{i_t}$. Thus we have two sequences of values related as follows:

$$\begin{array}{ccccccccccc}
\bar{x}_{i_0} & < \cdots < & \bar{x}_{i_{t-1}} & < & \bar{x}_{i_t} & < \cdots < & \bar{x}_{i_{t'-1}} & < & \bar{x}_{i_{t'}} & < \cdots < & \bar{x}_{i_{u-1}} & < & \bar{x}_{i_u} \\
= & & = & & > & & \geq & & > & & \geq & & < \\
\bar{y}_{i_0} & \cdots & \bar{y}_{i_{t-1}} & & \bar{y}_{i_t} & \cdots & \bar{y}_{i_{t'-1}} & & \bar{y}_{i_{t'}} & \cdots & \bar{y}_{i_{u-1}} & & \bar{y}_{i_u}
\end{array}$$

Let u' be the largest index for which $x_{j_{u'}}$ has been assigned a value at the time $\bar{y}_{i_u}$ is assigned to x_{i_u}. We have the two sequences of values

$$\begin{array}{ccccc}
\bar{x}_{j_0} & > \cdots > & \bar{x}_{j_{u'-1}} & > & \bar{x}_{j_{u'}} \\
\bar{y}_{j_0} & \cdots & \bar{y}_{j_{u'-1}} & & \bar{y}_{j_{u'}}
\end{array}$$

We first show that value $\bar{x}_{i_u}$ has not yet been assigned in the greedy algorithm when $\bar{y}_{i_u}$ is

assigned to x_{i_u}. That is, we show that $\bar{x}_{i_u} \notin \{\bar{y}_{i_0}, \ldots, \bar{y}_{i_{u-1}}\}$ and $\bar{x}_{i_u} \notin \{\bar{y}_{j_0}, \ldots, \bar{y}_{j_{u'}}\}$. To see that $\bar{x}_{i_u} \notin \{\bar{y}_{i_0}, \ldots, \bar{y}_{i_{u-1}}\}$, suppose to the contrary that $\bar{x}_{i_u} = \bar{y}_{i_w}$ for some $w \in \{0, \ldots, u-1\}$. This is impossible, because $\bar{x}_{i_u} > \bar{x}_{i_w} \geq \bar{y}_{i_w}$. Also $\bar{x}_{i_u} \notin \{\bar{y}_{j_0}, \ldots, \bar{y}_{j_{u'}}\}$, because assigning value $\bar{x}_{i_u}$ to x_{j_w} for some $w \in \{0, \ldots, u'\}$ contradicts the greedy construction of $\bar{y}$, due to the fact that value $\bar{y}_{i_u}$ was available at that time and is a superior choice.

We next show that value $\bar{x}_{i_{t'}}$ has not yet been assigned in the greedy algorithm when $\bar{y}_{i_u}$ is assigned to x_{i_u}. That is, we show that $\bar{x}_{i_{t'}} \notin \{\bar{y}_{i_0}, \ldots, \bar{y}_{i_{u-1}}\}$ and $\bar{x}_{i_{t'}} \notin \{\bar{y}_{j_0}, \ldots, \bar{y}_{j_{u'}}\}$. To begin with, we have that $\bar{x}_{i_{t'}} \notin \{\bar{y}_{i_0}, \ldots, \bar{y}_{i_{t'-1}}\}$, by virtue of the same reasoning just applied to $\bar{x}_{i_u}$. Also $\bar{x}_{i_{t'}} \neq \bar{y}_{i_{t'}}$, since by hypothesis $\bar{x}_{i_{t'}} > \bar{y}_{i_{t'}}$. To show that $\bar{x}_{i_{t'}} \notin \{\bar{y}_{i_{t'+1}}, \ldots, \bar{y}_{i_{u-1}}\}$, suppose to the contrary that $\bar{x}_{i_{t'}} = \bar{y}_{i_w}$ for some $w \in \{t'+1, \ldots, u-1\}$. Then since $\bar{x}_{i_{t'}} < \bar{x}_{i_w}$, we must have $\bar{x}_{i_w} > \bar{y}_{i_w}$. But this contradicts the definition of t' ($< w$) as the largest index in $\{0, \ldots, u-1\}$ such that $\bar{x}_{i_{t'}} > \bar{y}_{i_{t'}}$. Thus $\bar{x}_{i_{t'}} \neq \bar{y}_{i_w}$. Finally, $\bar{x}_{i_{t'}} \notin \{\bar{y}_{j_0}, \ldots, \bar{y}_{j_{u'}}\}$ because assigning value $\bar{x}_{i_{t'}}$ to x_{j_w} for some $w \in \{0, \ldots, u'\}$ contradicts the greedy construction of $\bar{y}$, due to the fact that $\bar{y}_{i_u}$ was available at the time and $\bar{y}_{i_u} > \bar{x}_{i_u} > \bar{x}_{i_{t'}}$.

Since $\bar{x}_{i_u} < \bar{y}_{i_u}$ and value $\bar{x}_{i_u}$ has not yet been assigned, setting $x_{i_u} = \bar{x}_{i_u}$ must create a cycle in $\bar{y}$, because otherwise $x_{i_u} = \bar{x}_{i_u}$ would have been the greedy choice. Also, setting $x_{i_u} = \bar{x}_{i_{t'}}$ was not the greedy choice because $\bar{y}_{i_u} > \bar{x}_{i_u} > \bar{x}_{i_{t'}}$. Thus setting $x_{i_u} = \bar{x}_{i_{t'}}$ must likewise create a cycle in $\bar{y}$, because $\bar{x}_{i_{t'}}$ has not yet been assigned. Now define $G_{\bar{y}(J)}$ as before and consider the maximal subchain in $G_{\bar{y}(J)}$ that contains $\bar{y}_{i_u}$. Let the segment of the subchain up to $\bar{y}_{i_u}$ be

$$v_{i_w} \to \cdots \to v_{i_u} \to \bar{y}_{i_u}$$

Because setting $x_{i_u} = \bar{x}_{i_u}$ creates a cycle in $\bar{y}$, we must have $\bar{x}_{i_u} = v_{i_w}$. Similarly, because setting $x_{i_u} = \bar{x}_{i_{t'}}$ creates a cycle in $\bar{y}$, we must have $\bar{x}_{i_{t'}} = v_{i_w}$. This implies $\bar{x}_{i_u} = \bar{x}_{i_{t'}}$, which is impossible because $\bar{x}_{i_u} > \bar{x}_{i_{t'}}$.

Case 2: $k_{\bar{\ell}} \in J_+$ and $k_{\hat{\ell}} \in J_-$, or $k_{\bar{\ell}} \in J_-$ and $k_{\hat{\ell}} \in J_+$. We can rule out the latter subcase immediately, because $k_{\bar{\ell}}$ can be in J_- only if $r > p$ when $\bar{y}_{k_{\bar{\ell}}}$ is assigned to $x_{k_{\bar{\ell}}}$. This means $k_{\hat{\ell}}$ must

be in J_- as well, since $x_{k_{\hat{\ell}}}$ is assigned after $x_{k_{\bar{\ell}}}$, and the situation reverts to Case 1. We therefore suppose $k_{\bar{\ell}} \in J_+$ and $k_{\hat{\ell}} \in J_-$.

Let t be the index such that $i_t = k_{\bar{\ell}}$, and u the index such that $j_u = k_{\hat{\ell}}$. Again $\bar{x}_{i_t} > \bar{y}_{i_t}$ because $\bar{x}_{i_t} \succ \bar{y}_{i_t}$ and $j_t \in J_+$. Thus, at the time value $\bar{y}_{i_t}$ was assigned to x_{i_t}, we had $\bar{x}_{j_s} < v_{\max}$ for the current value of s. So we have two sequences of values related as follows:

$$\begin{array}{ccccccccc} \bar{x}_{j_0} & > \cdots > & \bar{x}_{j_{s-1}} & > & \bar{x}_{j_s} & > \cdots & \bar{x}_{j_{u-1}} & > & \bar{x}_{j_u} \\ = & & = & & \leq & & \leq & & > \\ \bar{y}_{j_0} & \cdots & \bar{y}_{j_{s-1}} & & \bar{y}_{j_s} & \cdots & \bar{y}_{j_{u-1}} & & \bar{y}_{j_u} \end{array} \tag{1.20}$$

where $v_{\max} > \bar{x}_{j_s}$. Let t' be the largest index for which $x_{i_{t'}}$ has been assigned a value at the time $\bar{y}_{j_u}$ is assigned to x_{j_u}. We have two sequences of values related as follows:

$$\begin{array}{ccccccc} \bar{x}_{i_0} & < \cdots < & \bar{x}_{i_{t-1}} & < & \bar{x}_{i_t} & < \cdots < & \bar{x}_{i_{t'}} \\ = & & = & & > & & \\ \bar{y}_{i_0} & \cdots & \bar{y}_{i_{t-1}} & & \bar{y}_{i_t} & \cdots & \bar{y}_{i_{t'}} \end{array}$$

We first show that a cycle must be created if value $\bar{x}_{j_u}$ rather than $\bar{y}_{j_u}$ is assigned to x_{j_u}. Because $\bar{y}_{j_u} < \bar{x}_{j_u}$, it suffices to show that value $\bar{x}_{j_u}$ has not yet been assigned in the greedy algorithm when $\bar{y}_{j_u}$ is assigned to x_{j_u}. That is, we show that $\bar{x}_{j_u} \notin \{\bar{y}_{j_0}, \ldots, \bar{y}_{j_{u-1}}\}$ and $\bar{x}_{j_u} \notin \{\bar{y}_{i_0}, \ldots, \bar{y}_{i_{t'}}\}$. If $\bar{x}_{j_u} = \bar{y}_{j_w}$ for some $w \in \{0, \ldots, u-1\}$, then $\bar{x}_{j_u} < \bar{x}_{j_w} \leq \bar{y}_{j_w}$, which is impossible. Thus $\bar{x}_{j_u} \notin \{\bar{y}_{j_0}, \ldots, \bar{y}_{j_{u-1}}\}$. Also $\bar{x}_{j_u} \notin \{\bar{y}_{i_0}, \ldots, \bar{y}_{i_{t'}}\}$, because assigning value $\bar{x}_{j_u}$ to x_{i_w} for some $w \in \{0, \ldots, t'\}$ contradicts the greedy construction of $\bar{y}$, due to the fact that value $\bar{y}_{j_u}$ was available at that time and is a superior choice.

We next show that a cycle must be created if value $v_{\max}$ rather than $\bar{y}_{j_u}$ is assigned to x_{j_u}. Note that $v_{\max} \notin \{\bar{y}_{i_0}, \ldots, \bar{y}_{i_{t'}}\}$, because assigning value $v_{\max}$ to x_{i_w} for some $w \in \{0, \ldots, t'\}$ contradicts the greedy construction of $\bar{y}$, due to the fact that value $\bar{y}_{j_u}$ was available at that time and is a superior choice because $v_{\max} > \bar{x}_{j_s} > \bar{x}_{j_u}$. Now suppose, contrary to the claim, that assigning $v_{\max}$ to x_{j_u} does not create a cycle. Then since $v_{\max} > \bar{y}_{j_u}$, the value $v_{\max}$ must have

already been assigned in the greedy algorithm at the time $\bar{y}_{j_u}$ is assigned to x_{j_u}. This implies $v_{\max} \in \{\bar{y}_{j_s}, \ldots, \bar{y}_{j_{u-1}}\}$. But in this case we must have $\bar{y}_{j_s} = v_{\max}$, because assigning $v_{\max}$ to x_{j_s} does not create a cycle and, by definition, is the most attractive choice at the time. Thus (1.20) becomes

$$\begin{array}{ccccccccccc}
\bar{x}_{j_0} & > \cdots > & \bar{x}_{j_{s-1}} & > \bar{x}_{j_s} & > \cdots > & \bar{x}_{j_{s'-1}} & > \bar{x}_{j_{s'}} & > \cdots > & \bar{x}_{j_{u-1}} & > \bar{x}_{j_u} \\
= & & = & < & & \leq & < & & \geq & > \\
\bar{y}_{j_0} & \cdots & \bar{y}_{j_{s-1}} & \bar{y}_{j_s} & \cdots & \bar{y}_{j_{s'-1}} & \bar{y}_{j_{s'}} & \cdots & \bar{y}_{j_{u-1}} & \bar{y}_{j_u}
\end{array}$$

where $\bar{y}_{j_s} = v_{\max}$ and where s' is the largest index in $\{s, \ldots, u-1\}$ such that $\bar{y}_{j_{s'}} < \bar{x}_{j_{s'}}$. Now we can argue as in Case 1 that assigning $\bar{x}_{j_u}$ to x_{j_u} creates a cycle, and assigning $\bar{x}_{j_{s'}}$ to x_{j_u} creates a cycle, which implies $\bar{x}_{j_{s'}} = \bar{x}_{j_u}$, a contradiction because $\bar{x}_{j_{s'}} > \bar{x}_{j_u}$. We conclude that assigning $v_{\max}$ to x_{j_u} creates a cycle.

Having shown that assigning $\bar{x}_{j_u}$ to x_{j_u} creates a cycle, and assigning $v_{\max}$ to x_{j_u} creates a cycle, we derive as in Case 1 that $v_{\max} = \bar{x}_{j_u}$, a contradiction because $v_{\max} \geq \bar{x}_{j_s} > \bar{x}_{j_u}$. The theorem follows. $\square$.

1.9. Permutation and Two-term Facets

In this section we examine two special classes of facets of $C_n(v)$, permutation facets and two-term facets.

The *permutation polytope* $P_n(v)$ for a given domain $\{v_0, \ldots, v_{n-1}\}$ is the convex hull of all points whose coordinates are permutations of $v_0, \ldots, v_{n-1}$. The circuit polytope $C_n(v)$ is contained in $P_n(v)$ because every circuit $(x_0, \ldots, x_{n-1})$ is a permutation of $v_0, \ldots, v_{n-1}$. This means that every facet-defining inequality for $P_n(v)$ is valid for circuit but not necessarily facet defining. This raises the question as to which permutation facets are also circuit facets. We will identify a large family of permutation facets that can be immediately recognized as circuit facets.

The permutation polytope $P_n(v)$ has dimension $n-1$. The facets of $P_n(v)$ are identified in

[17, 36], and they are defined by

$$\sum_{j \in J} x_j \geq \sum_{j=0}^{|J|-1} v_j \tag{1.21}$$

for all $J \subset \{0, \ldots, n-1\}$ with $1 \leq |J| \leq n-1$. (Recall that $v_0 < \cdots < v_{n-1}$.) This result is generalized in [18] to domains with more than n elements.

For example, the permutation polytope $P_3(v)$ with $v = (2, 4, 5)$ is defined by

$$\begin{aligned}
&x_0 + x_1 + x_2 = 11 \\
&x_i \geq 2, \text{ for } i = 0, 1, 2 \\
&x_i + x_j \geq 6, \text{ for distinct } i, j \in \{0, 1, 2\}
\end{aligned}$$

We can see at this point that a facet-defining inequality for $P_n(v)$ need not be facet-defining for $C_n(v)$. The inequality $x_0 + x_1 \geq 6$ is facet-defining for $P_3(v)$ but not for $C_3(v)$, which is the line segment from $(4, 5, 2)$ to $(5, 2, 4)$.

Theorems 4, 6, and 7 allow us to identify a family of permutation facets that are also circuit facets.

Corollary 8. *The inequality (1.21) defines a facet of $C_n(v)$ if $1 \leq |J| \leq n-4$ and $j \geq |J|$ for all $j \in J$.*

Proof. Let $J = \{j_0, \ldots, j_m\}$. Due to Theorem 6 and the fact that $j \geq m$ for all $j \in J$, the following are undominated J-circuits with respect to $J = J_+$:

$$\text{all } \bar{x}(J) \text{ for which } \bar{x}_{j_0}, \ldots, \bar{x}_{j_m} \text{ is a permutation of } v_0, \ldots, v_m \tag{1.22}$$

Theorem 7 tells us that (1.22) is the complete set of J-circuits that are undominated with respect to

$J = J_+$. Consider the following J-circuits from (1.22):

$$\begin{aligned}
\bar{x}^0(J) &= (v_0, v_1, v_2, v_3, \ldots, v_{n-2}, v_{n-1}) \\
\bar{x}^1(J) &= (v_1, v_0, v_2, v_3, \ldots, v_{n-2}, v_{n-1}) \\
\bar{x}^2(J) &= (v_0, v_2, v_1, v_3, \ldots, v_{n-2}, v_{n-1}) \\
&\vdots \\
\bar{x}^m(J) &= (v_0, v_1, v_2, v_3, \ldots, v_{n-1}, v_{n-2})
\end{aligned} \tag{1.23}$$

where $\bar{x}^i(J)$ is obtained for $i > 0$ by swapping v_{i-1} and v_i in $\bar{x}^0(J)$. These circuits are affinely independent, as can be seen by subtracting $\bar{x}^0(J)$ from each. By construction, all the J-circuits (1.23) satisfy (1.21) as an equation. Thus the affinely independent J-circuits (1.22) satisfy (1.21) as an equation, and all the remaining J-circuits in (1.23) satisfy (1.21). So by Theorem 4, (1.21) is facet-defining. □

We can check on a case-by-case basis whether permutation facets other than those mentioned in Corollary 8 are circuit facets. For example, if $J = J_+ = \{2, 3, 4\}$, then application of the greedy procedure in Fig. 1.1 yields the undominated J-circuits

$$\begin{aligned}
\bar{x}^0 &= (v_0, v_1, v_2) & \bar{x}^3 &= (v_3, v_0, v_1) \\
\bar{x}^1 &= (v_0, v_2, v_1) & \bar{x}^4 &= (v_1, v_2, v_0) \\
\bar{x}^2 &= (v_1, v_0, v_2) & \bar{x}^5 &= (v_3, v_1, v_0)
\end{aligned}$$

Some subsets of three J-circuits, such as $\{\bar{x}^0, \bar{x}^1, \bar{x}^2\}$, satisfy (1.21) as an equation. Because the remaining J-circuits clearly satisfy (1.21), the permutation facet (1.21) is also a circuit facet.

Another special class of facet-defining inequalities are those containing two terms. Because a set of two undominated J-circuits (where $|J| = 2$) defines exactly one facet, the two-term facets can be exhaustively listed in closed form.

Corollary 9. *If $n \geq 6$, the two-term facets of $C_n(v)$ are precisely those defined by*

$$x_i + x_j \geq v_0 + v_1, \quad \textit{for distinct } i, j \in \{2, \ldots, n-1\}$$

$$(v_2 - v_0)x_0 + (v_2 - v_1)x_1 \geq v_2^2 - v_0 v_1$$

$$(v_1 - v_0)x_1 + (v_2 - v_0)x_i \geq v_1 v_2 - v_0^2, \quad \textit{for } i \in \{2, \ldots, n-1\}$$

$$x_i + x_j \leq v_{n-2} + v_{n-1}, \quad \textit{for distinct } i, j \in \{0, \ldots, n-3\}$$

$$(v_{n-2} - v_{n-3})x_{n-2} + (v_{n-1} - v_{n-3})x_{n-1} \leq v_{n-1}v_{n-2} - v_{n-3}^2$$

$$(v_{n-1} - v_{n-3})x_i + (v_{n-1} - v_{n-2})x_{n-2} \leq v_{n-1}^2 - v_{n-2}v_{n-3}, \quad \textit{for } i \in \{0, \ldots, n-3\}$$

The proof is straightforward.

1.10. Separation Heuristics

The greedy procedure described above for generating undominated J-circuits suggests some simple separation heuristics. Suppose we have a solution $\hat{x}$ of the current relaxation of the problem, and that $\hat{x}$ violates the circuit constraint. The *separation problem* is to find one or more facet-defining inequalities that separate $\hat{x}$ from the circuit polytope in the sense that $\hat{x}$ violates the inequalities. Separating inequalities can then be added to the relaxation to tighten it.

Suppose first that we seek separating inequalities with all positive coefficients, so that $J = J_+$. Given a point $\hat{x}$ to be separated, let $j_0, \ldots, j_{n-1}$ be an ordering of variable indices for which $\hat{x}_{j_0} \leq \cdots \leq \hat{x}_{j_{n-1}}$. We consider the sequence of subsets $J^0, J^1, \ldots, J^{n-1}$ where $J^i = \{j_0, \ldots, j_i\}$. Beginning with J^0, we try to generate facet-defining inequalities corresponding to each J^i, until we find a separating inequality. For each J^i we use the greedy procedure of Fig. 1.1 to generate all undominated J^i-circuits with respect to $J^i = J^i_+$ and use these J^i-circuits to generate facet-defining inequalities as described earlier. Any of the resulting inequalities violated by $\hat{x}$ are separating. If none are separating, we move to J^{i+1} and repeat. The precise algorithm appears in Fig. 1.2. A similar algorithm is shown in [18] to be a complete separation procedure for the permutation

Let $S = \emptyset$.
Order $j_0, \ldots, j_n$ so that $\hat{x}_{j_0} \leq \cdots \leq \hat{x}_{j_{n-1}}$.
For $k = 0, \ldots, k_{\max}$ while $S = \emptyset$:
  Let $J^k = \{j_0, \ldots, j_k\}$.
  Let $\bar{x}^0(J^k), \ldots, \bar{x}^m(J^k)$ be the undominated J^k-circuits generated
    by the greedy procedure of Fig. 1.1 with $J = J_+ = J^k$.
  For each $\{t_0, \ldots, t_k\} \subset \{0, \ldots, m\}$:
    Let $\sum_{i=0}^k a_{j_i} x_{j_i} = \alpha$ be an equation satisfied by $\bar{x}^{t_0}(J^k), \ldots, \bar{x}^{t_k}(J^k)$.
    If $\sum_{i=1}^k a_{j_i} \hat{x}_{j_i} < \alpha$ then add $\sum_{i=1}^k a_{j_i} x_{j_i} \geq \alpha$ to S.

Figure 1.2: Separation heuristic for finding a set S of facet-defining inequalities with positive coefficients violated by a given point $\hat{x}$.

Let $S = J_+ = J_- = \emptyset$.
Order $j_0, \ldots, j_n$ so that
  $\min\{\hat{x}_{j_0} - v_0, v_{n-1} - \hat{x}_{j_0}\} \leq \cdots \leq \min\{\hat{x}_{j_0} - v_0, v_{n-1} - \hat{x}_{j_0}\}$.
For $j = 0, \ldots, k_{\max}$:
  If $\hat{x}_{j_0} - v_0 \leq v_{n-1} - \hat{x}_{j_0}$ then add j to J_+.
  Else add j to J_-.
For $k = 0, \ldots, k_{\max}$ while $S = \emptyset$:
  Let $J^k = \{j_0, \ldots, j_k\}$, $J^k_+ = J^k \cap J_+$, $J^k_- = J^k \cap J_-$.
  Let $\bar{x}^0(J^k), \ldots, \bar{x}^m(J^k)$ be the undominated J^k-circuits generated
    by the greedy procedure of Fig. 1.1 with $J_+ = J^k_+$, $J_- = J^k_-$.
  For each $\{t_0, \ldots, t_k\} \subset \{0, \ldots, m\}$:
    Let $\sum_{i=0}^k a_{j_i} x_{j_i} = \alpha$ be an equation satisfied by $\bar{x}^{t_0}(J^k), \ldots, \bar{x}^{t_k}(J^k)$.
    If $\sum_{i=1}^k a_{j_i} \hat{x}_{j_i} < \alpha$ then add $\sum_{i=1}^k a_{j_i} x_{j_i} \geq \alpha$ to S.

Figure 1.3: Separation heuristic for finding a set S of facet-defining inequalities with arbitrary coefficients violated by a given point $\hat{x}$.

polytope.

In practice, the algorithm would not continue all the way to J^{n-1} when no separating inequalities are found, because it is impractical to generate all undominated J^k-circuits when k is large. Rather, the algorithm would stop at some predetermined maximum $k = k_{\max}$.

As an illustration, suppose that $(\hat{x}_0, \ldots, \hat{x}_6) = (6, 2, 5.5, 7, 5.7, 8, 9)$ in example (1.12). This is not a feasible solution, if only because it does not consist of values from the domain $\{2, 5, 6, 7, 9, 10, 12\}$. Here $(j_0, \ldots, j_6) = (1, 2, 4, 0, 3, 5, 6)$. For $J^0 = \{1\}$ we have the single facet-defining inequality $x_1 \geq 2$, but it does not separate $\hat{x}$. For $J^1 = \{1, 2\}$ we have the facet-defining inequality

$3x_1 + 4x_2 \geq 26$, which again does not separate $\hat{x}$. But for $J^2 = \{1, 2, 4\}$ we have three facet-defining inequalities

$$8x_1 + 5x_2 + 10x_4 \geq 101$$

$$12x_1 + 11x_2 + 15x_4 \geq 169$$

$$6x_1 + 3x_2 + 8x_4 \geq 73$$

Because the first and third are violated by $\hat{x}$, they are separating cuts.

The above heuristic can be modified slightly to generate separating inequalities with arbitrary signs. Rather than order the variables by nondecreasing size of $\hat{x}_j$, we can order them by nondecreasing size of $\min\{\hat{x}_j - v_0, v_{n-1} - \hat{x}_j\}$. Then we put $j \in J_+$ if $\hat{x}_j - v_0 \leq v_{n-1} - \hat{x}_j$ and $j \in J_-$ otherwise. The heuristic appears in Fig. 1.3.

1.11. Exploiting Cost Structure

One motivation for studying the circuit polytope for arbitrary domains is that it may allow us to exploit structure in a cost function that appears in the problem. A careful choice of the domain values can result in a tighter relaxation.

Suppose, for example, that the problem contains the cost function $\sum_i c_{ix_i}$ that appears in the traveling salesman problem (1.2). Associate each index i with a value v_i, and suppose that the costs c_{ij} have the property that, when the values v_i are properly chosen, $g(v_i, v_j) = c_{ij}$ is close to the value of an affine function $h(v_i, v_j)$ for $i < j$, and it is close to the value of an affine function $h'(v_i, v_j)$ when $j < i$. The v_is can be set to any nonnegative value, and the variables can be reordered if desired, to obtain a good affine fit. Then one can use computational geometry techniques to compute the convex hull of $S = \{(z, x_i, x_j) \mid z = g(x_i, x_j),\ x_i, x_j \in \{v_0, \ldots, v_{n-1}\}\}$. Consider all facets of the convex hull that are described by inequalities of the form

$$z \geq \beta_{0k} + \beta_{1k} x_i + \beta_{2k} x_j, \quad k \in K \tag{1.24}$$

Then all of the points of S are close to the facets described by (1.24).

Now let $Ax \geq b$ be a system of valid inequalities for the circuit polytope $C_n(v)$, where v is the vector of values just chosen. We can write a linear relaxation of the traveling salesman problem (1.2) that exploits the cost structure:

$$\begin{aligned} &\min \sum_{ij} z_{ij} \\ &z_{ij} \geq \beta_{0k} + \beta_{1k} x_i + \beta_{2k} x_j, \quad \text{for all } i, j \text{ and all } k \in K \\ &Ax \geq b \end{aligned} \tag{1.25}$$

For example, suppose the cost data c_{ij} are as in Table 1.2. If we let $(v_1, v_2, v_3) = (0, 1.5, 3.5)$, the values $g(v_i, v_j) = c_{ij}$ are close to the values of the affine function $h(v_i, v_j) = 4(v_i - v_j)$ for $i < j$ and close to $h'(v_i, v_j) = 4(v_j - v_i)$ for $j < i$. The convex hull of S has two facets of the form (1.24), namely

$$\begin{aligned} z &\geq \tfrac{26}{7} x_i - \tfrac{26}{7} x_j \\ z &\geq -\tfrac{26}{7} x_i + \tfrac{26}{7} x_j \end{aligned}$$

So if $Ax \geq b$ is a set of valid inequalities for $C_n(v)$, the relaxation (1.26) therefore becomes

$$\begin{aligned} &\min \sum_{i=0}^{2} \sum_{j=0}^{2} z_{ij} \\ &z_{ij} \geq \tfrac{26}{7} x_i - \tfrac{26}{7} x_j \quad \text{for all } i, j \in \{0, 1, 2\} \\ &z_{ij} \geq -\tfrac{26}{7} x_i + \tfrac{26}{7} x_j \quad \text{for all } i, j \in \{0, 1, 2\} \\ &Ax \geq b \end{aligned} \tag{1.26}$$

If c_{ij} is a distance, it may be possible to exploit the structure of the distance metric, particularly if it is rectilinear. Further details, along with an application to the quadratic assignment problem, may be found in [18].

Table 1.2: (a) Cost data c_{ij}. (b) Values of $h(x_i, x_j)$ when $x_i \leq x_j$ and $h'(x_i, x_j)$ when $x_j \leq x_i$.

(a)

		j		
		0	1	2
i	0	0	6	13
	1	6	0	9
	2	13	9	0

(b)

		x_j		
		0	1.5	3.5
x_i	0	0	6	14
	1.5	6	0	8
	3.5	14	8	0

1.12. Conclusions and Future Research

We provided a nearly complete characterization of the circuit polytope that identifies all facet-defining inequalities with at most $n - 4$ terms. In particular, we showed that the facet-defining inequalities with a specified sign pattern are precisely those valid inequalities that are defined by subsets of J-circuits that are undominated with respect to that sign pattern. Inequalities of this sort are valid when they are satisfied by all undominated J-circuits.

This allows us to identify all facet-defining inequalities with a two-phase procedure. A combinatorial phase generates all undominated J-circuits with respect to a desired sign pattern $J = J_+ \cup J_-$, using a greedy algorithm. A numerical phase then computes equations that are satisfied by affinely independent subsets of the undominated J-circuits and checks them for validity. The first phase is independent of the domain values $v_0, \ldots, v_{n-1}$, but the second is not. This two-phase procedure can be viewed as isolating the discrete and continuous aspects of the circuit polytope.

We also identified a family of permutation facets that are circuit facets and explicitly described all two-term circuit facets. We presented two separation heuristics based on the greedy procedure, and we showed how the circuit constraint with arbitrary variable domains can exploit cost structure in the objective function.

These results presented here lay the theoretical groundwork for the solution of sequencing problems with the help of linear relaxations comprised of circuit inequalities. Computational testing is the next step, together with investigation of how the separation heuristics can be tuned or altered to achieve best results. The cost matrices of typical problems can be examined to determine the extent to which cost can be approximated as an affine function or a rectilinear metric, to allow an effective

choice of domain values.

An interesting research question is whether circuit inequalities can be profitably converted to 0-1 inequalities and combined with known traveling salesman inequalities. For a given domain $\{v_0, \ldots, v_{n-1}\}$, the conversion could be based on the identity $x_i = \sum_j v_j y_{ij}$, where y_{ij} is the 0-1 variable that appears in the traveling salesman model (1.3).

Our primary goal, however, has been to explore the structure of the circuit polytope in the original space, as an alternative to the conventional 0-1 representation.

Chapter 2

A Filter for the Circuit Constraint

2.1. Motivation

The circuit constraint and the all-different (alldiff) constraint are closely related. Both can require that a set of variables describe a permutation, and both are used in models of sequencing, scheduling, and assignment problems. Yet while filtering for alldiff is well understood, much less is known about filtering for circuit. This is partly because achieving hyperarc consistency for alldiff is relatively straightforward and can be done in polynomial time, while it is an NP-hard problem for circuit.

We address the problem of filtering the circuit constraint for two reasons: (a) circuit is better suited to some modeling situations than alldiff; (b) a circuit formulation of a problem can often be added to an alldiff formulation, allowing a circuit filter to contribute to domain reduction.

The circuit constraint can be written

$$\text{circuit}(x_1, \ldots, x_n) \tag{2.1}$$

where the domain of each x_i is $D_i \subset \{1, \ldots, n\}$. The constraint requires that $y_1, \ldots, y_n$ be a cyclic

permutation of $1, \ldots, n$, where

$$\begin{aligned} y_{i+1} &= x_{y_i}, \quad i = 1, \ldots, n-1 \\ y_1 &= x_{y_n} \end{aligned} \tag{2.2}$$

Thus x_i is the item immediately following i in the permutation. The permutation is cyclic in the sense that no item is regarded as first. The filtering problem is to identify and remove values from D_i that x_i takes in no feasible solution of (2.1).

The all-different constraint

$$\text{alldiff}(y_1, \ldots, y_n) \tag{2.3}$$

requires that $y_1, \ldots, y_n$ take distinct values. The domain of y_i may be any finite set. If each domain is a subset of $\{1, \ldots, n\}$, the constraint requires that $y_1, \ldots, y_n$ be a permutation of $1, \ldots, n$, and y_i is the ith item in the permutation. Clearly a feasible solution $(x_1, \ldots, x_n)$ of the circuit constraint (2.1) must satisfy alldiff$(x_1, \ldots, x_n)$, but this is not sufficient to satisfy (2.1).

The circuit constraint is often interpreted as requiring that $x_1, \ldots, x_n$ define a hamiltonian cycle on a directed graph G. The vertices of G are $1, \ldots, n$, and the edges are all pairs (i, j) with $j \in D_i$. Then $(x_1, \ldots, x_n)$ satisfies (2.1) if and only if $(y_1, \ldots, y_n, y_1)$ describes a hamiltonian cycle, where $y_1, \ldots, y_n$ are defined by (2.2). Variable x_i can take the value j in a feasible solution of (2.1) when (i, j) is part of a hamiltonian cycle in G.

Thus filtering for the circuit constraint reduces to identifying and eliminating *nonhamiltonian* edges from G (i.e., edges that belong to no hamiltonian cycle). Since checking whether a graph is hamiltonian (i.e., contains a hamiltonian cycle) is NP-hard, the same is true of checking whether an edge is hamiltonian.

The circuit constraint is suited for problems in which one wishes to encode which task follows another, while the all-different constraint allows one to encode the position of a given task in the sequence. A typical application of the circuit constraint is to find a cyclic permutation of tasks that minimizes setup cost or time. Thus if s_{ij} is the setup cost of task j when it immediately follows task

i, one can minimize total setup cost by minimizing $\sum_i s_{ix_i}$ subject to (2.1). A frequent application of the alldiff constraint finds an assignment of tasks to workers that minimizes cost. If c_{ij} is the cost of assigning task j to worker i, then one can minimize total cost by minimizing $\sum_i c_{ix_i}$ subject to (2.3).

The setup cost problem can be reformulated using alldiff rather than circuit, but this in general requires two sets of variables in order to capture the domains. The circuit constraint (2.1) can be replaced by the alldiff constraint (2.3), and the objective function $\sum_i s_{ix_i}$ can be rewritten $\sum_i s_{y_i,y_{i+1}}$, where y_{n+1} is identified with y_1. But to restrict domains of the variables x_i, one must retain these variables and add the channeling constraints (2.2) to the model. Also much of the information encoded in the domains of the x_is is lost when one filters only the y_i domains.

The circuit constraint is therefore useful when one wishes to restrict which tasks may follow a given task. Even if both sets of variables are used in order to write the model with alldiff, it is advantageous to include the circuit constraint as well and apply a filtering algorithm to it.

The circuit constraint can also be added to models that already contain an alldiff constraint for which the variable domains are subsets of $\{1, \ldots, n\}$. Again the channeling constraints (2.2) are used. Application of filtering algorithms to the circuit as well as the alldiff may improve propagation and help solve the problem, particularly if information from the application allows one to restrict the domains of the x_is directly.

2.2. Basic Idea

We propose a filter that identifies some of the nonhamiltonian edges of a directed graph G by analyzing a smaller combinatorial object that partially captures the structure of G.

We first identify a separator of G, which is a subset S of vertices whose removal splits G into two or more connected components. We then define a "separator graph" on S that consists of the subgraph induced by S plus some additional "labeled" edges. The key result is that an unlabeled edge is hamiltonian in G only if it is a "permissible", which means that it belongs to a "permissible"

hamiltonian cycle in the separator graph. This allows one to identify nonhamiltonian edges in G by identifying nonpermissible edges in the separator graph.

The advantage of this approach is that it permits one to identify nonhamiltonian edges of a large graph by examining the hamiltonicity properties of a smaller, labeled graph. This assumes, of course, that reasonably small separators exist, which in turn presupposes that the original graph is not too dense. However, filtering is likely to be of little use when the graph is dense, since nearly all edges tend to be hamiltonian.

Since it can be expensive to identify nonpermissible edges even in a relatively small separator graph, we propose a secondary filter to detect some of them. We construct a flow graph in which certain edges have a maximum flow of zero only if a corresponding edge of the separator graph is nonpermissible. We check whether these edges have a maximum flow of zero by solving a single maximum flow problem on the flow graph and computing the strongly connected components of the corresponding residual graph. The complexity of this procedure is dominated by the complexity of solving the max flow problem, for which very fast algorithms are available.

The flow graph actually implements two kinds of secondary filtering. One is based on the fact that a permissible hamiltonian cycle must satisfy a certain generalized cardinality constraint. The other is based on the fact that the out(in)-degree of every vertex of a hamiltonian cycle is one. The flow graph encodes both filters by extending the well-known flow graph model for the cardinality constraint. A similar flow graph is constructed to combine the cardinality constraint with in-degree constraints.

In practice several separators can be identified, and the filtering algorithm applied for each. We use a simple breadth-first-search heuristic to find separators rapidly. The complexity of the overall algorithm is the complexity of solving a max flow problem for each separator identified. Each max flow problem is solved on a flow graph having $O(k^2)$ nodes and $O(k^3)$ edges, where k is the number of vertices in the separator.

2.3. Previous Work

One approach to filtering the circuit constraint is to make use of sufficient conditions for nonhamiltonicity of a directed graph G. If a vertex is inserted into an edge (i, j) of G to obtain a modified graph G_{ij}, (i, j) is hamiltonian if and only if G_{ij} is hamiltonian. Thus if some sufficient condition for the nonhamiltonicity of G_{ij} is satisfied, j can be removed from D_i.

Although the graph theoretic literature contains a number of sufficient conditions for hamiltonicity, much less is known about sufficient conditions for nonhamiltonicity (see [8] for a survey). Two obvious conditions are that a graph G is nonhamiltonian if (a) it has two or more strongly connected components, or (b) there is no way to match each vertex with a different successor, that is, alldiff$(x_1, \ldots, x_n)$ is infeasible. These conditions are easily checked but are quite strong.

Chvátal [6] analyzed three further conditions: an undirected graph G is nonhamiltonian if (c) some separator S separates G into more than $|S|$ connected components, or (d) there is no set of disjoint cycles (subtours) that collectively cover all the vertices of G, or (e) some subset of 3 vertices are contained in no cycle. One can verify that (c) is satisfied by exhibiting a separator S. Chvátal showed that one can verify that (d) is satisfied by exhibiting a certain kind of partition of the vertices, and that (e) is satisfied by exhibiting subgraphs having a certain structure. However, condition (c) is quite strong and is in fact a very restrictive special case of the condition we present below. Exhibiting certificates that establish condition (d) or (e) requires analysis of the entire graph and is likely to be too computationally expensive for purposes of filtering.

Chvátal also points out in [8] that a graph is nonhamiltonian if there is no feasible solution of the corresponding subtour elimination and comb inequalities. He shows that this condition generalizes (c), (d) and (e) above. The feasibility of these inequalities can be checked by linear programming, but this is impractical for filtering, since there are exponentially many inequalities.

Shufelt and Berliner directly address the problem of filtering the circuit constraint in [32]. They describe a set of patterns to identify hamiltonian and nonhamiltonian edges in an undirected graph. These patterns can be adapted for directed graphs and used to eliminate nonhamiltonian edges when

some part of the hamiltonian cycle has been constructed; the method is not intended for the general case in which arbitrary variable domains are given. The filter is essentially based on the strong conditions (a) and (b) mentioned above. Moreover, the analysis relies on the special structure of the problem for which it was developed, namely the construction of a knight's tour on a chessboard.

Caseau and Laburthe present a set of techniques in [4] to solve small(up to 30 nodes) TSP's with constraint propagation. They show that small TSP's can be efficiently solved with their proposed branch and bound strategies combined with a propagation scheme for the nocycle constraint. However, they also show that, these methods are not competitive to solve problems of larger sizes.

2.4. Definitions

Let $G = (V, E)$ be a directed graph. A pair of vertices $i, j \in V$ are *neighbors* if $(i, j) \in E$ or $(j, i) \in E$. A *directed path* P between $i_1, i_m \in V$ is a sequence of edges $(i_1, i_2), (i_2, i_3), \ldots, (i_{m-1}, i_m) \in E$. P is a *simple path* if $i_1, \ldots, i_{m-1}$ are distinct. The endpoints i_1, i_m are *connected* by P.

G is *connected* if any two vertices of G are connected by some directed path. For convenience we will say that an edge (i, j) connects two vertex sets V_1, V_2 if $i \in V_1$ and $j \in V_2$. An edge connects V_1 with subgraph (V_2, A_2) if it connects V_1 with V_2.

A nonempty vertex set $V' \subset V$ induces a *connected component* of G if V' induces a connected subgraph, and no edge of G connects V' with $V \setminus V'$. A set $S \subset V$ *separates* a connected graph G into connected components $C_1, \ldots, C_p$ if $V \setminus S$ induces a subgraph $\bar{G}_S$ of G with connected components $C_1, \ldots, C_p$. We say S is a *(vertex) separator* of G if it separates G into at least two connected components.[1]

A *directed cycle* of G is a directed path of which every vertex is an endpoint. A vertex set $V' \subset V$ induces a *strongly connected component* if every pair of vertices in V' belong to a directed cycle of G, but no vertex of V' belongs to the same directed cycle as a vertex in $V \setminus V'$. A *hamiltonian cycle* of G is a simple cycle whose vertices are precisely those in V. An edge (i, j) of G

[1] If S separates G into at least three components, S is a *shredder*.

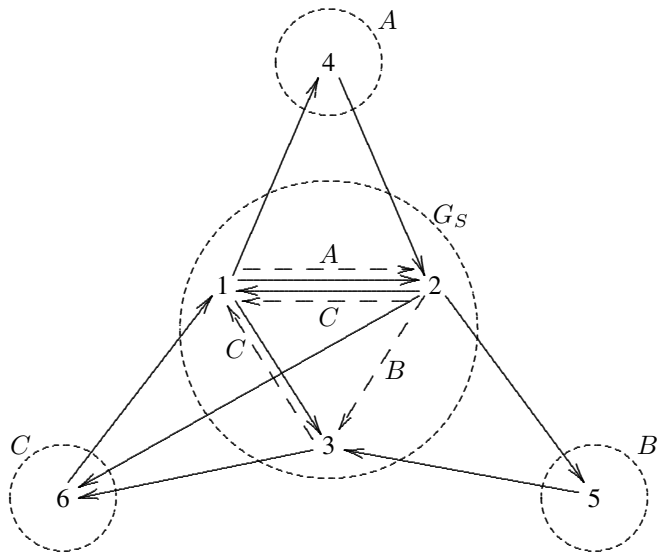

Figure 2.1: *Graph G on vertices $\{1, \ldots, 6\}$ contains the solid edges, and the separator graph G_S on $S = \{1, 2, 3\}$ contains the solid (unlabeled) edges and dashed (labeled) edges within the larger circle. The small circles surround connected components of the separated graph.*

is *hamiltonian* if (i, j) belongs to a hamiltonian cycle. (see [35] for more graph theory definitions.)

2.5. Separator Graph

The *separator graph* G_S for a separator S of $G = (V, E)$ consists of a directed graph with vertex set S and edge set E_S, along with a set L_S of *labels* corresponding to the connected components of $\bar{G}_S$. E_S contains (a) an *unlabeled* edge (i, j) for each $(i, j) \in E$, as well as (b) a *labeled* edge $(i, j)^C$ whenever $C \in L_S$ and $(i, c_1), (c_2, j) \in E$ for some pair of vertices c_1, c_2 in connected component C (possibly $c_1 = c_2$).

Consider for example the graph G of Fig. 2.1. Vertex set $S = \{1, 2, 3\}$ separates G into three connected components that may be labeled A, B and C, each of which contains only one vertex. Thus $L_S = \{A, B, C\}$, and the separator graph G_S contains the three edges that connect its vertices in G plus four labeled edges. For example, there is an edge $(1, 2)$ labeled A, which can be denoted $(1, 2)^A$, because there is a directed path from some vertex in component A through $(1, 2)$ and back to a vertex of component A.

A hamiltonian cycle of G_S is *permissible* if it contains at least one edge bearing each label in L_S. An edge of G_S is *permissible* if it is part of some permissible hamiltonian cycle of G_S. Thus the edges $(1,2)^A$, $(2,3)^B$ and $(3,1)^C$ form a permissible hamiltonian cycle in Fig. 2.1, and they are the only permissible edges.

Theorem 10. *If S is a separator of directed graph G, then G is hamiltonian only if G_S contains a permissible hamiltonian cycle. Furthermore, an edge of G connecting vertices in S is hamiltonian only if it is a permissible edge of G_S.*

Proof. Consider an arbitrary hamiltonian cycle H of G. We can construct a permissible hamiltonian cycle H_S for G_S as follows. Consider the sequence of vertices in H and remove those that are not in S; let $i_1, \ldots, i_m, i_1$ be the remaining sequence of vertices. H_S can be constructed on these vertices as follows. For any pair i_k, i_{k+1} (where i_{m+1} is identified with i_1), if they are adjacent in H then (i_k, i_{k+1}) is an unlabeled edge of G_S and connects i_k and i_{k+1} in H_S. If i_k, i_{k+1} are not adjacent in H then all vertices in H between i_k and i_{k+1} lie in the same connected component C of the subgraph of G induced by $V \setminus S$. This means (i_k, i_{k+1}) is an edge of G_S with label C, and $(i_k, i_{k+1})^C$ connects i_k and i_{k+1} in H_S. Since H passes through all connected components, every label must occur on some edge of H_S, and H_S is permissible.

We now show that if (i,j) with $i,j \in S$ is an edge of a hamiltonian cycle H of G, then (i,j) is an edge of a permissible hamiltonian cycle of G_S. But in this case (i,j) is an unlabeled edge of G_S, and by the above construction (i,j) is part of H_S. □

Corollary 11. *If $|L_S| > |S|$ for some separator S, then G is nonhamiltonian.*

Proof. The separator graph G_S has $|S|$ vertices and therefore cannot have a hamiltonian cycle with more than $|S|$ edges. □

Since the labels in L_S correspond to connected components of $\bar{G}_S$, $|L_S| > |S|$ if and only if S separates G into more than $|S|$ connected components. The above corollary therefore restates (for

directed graphs) Chvátal's condition, mentioned earlier, that an undirected graph G is nonhamiltonian if some separator S separates G into more than $|S|$ connected components.

Chvátal defines an undirected graph G to be *1-tough* when no S separates G into more than $|S|$ components [7]. Since the same concept can be defined for directed graphs, Corollary 11 says that a graph is hamiltonian only if it is 1-tough. The existence of a permissible hamiltonian cycle in every separator graph can therefore be viewed as a generalization of 1-toughness that leads to a much weaker sufficient condition for nonhamiltonicity.

Corollary 12. *If $|L_S| = |S|$ for some separator S, then no edge connecting vertices of S is hamiltonian.*

Proof. An edge e that connects vertices in S is unlabeled in G_S. If e is hamiltonian, some hamiltonian cycle in G_S that contains e must have at least $|S|$ labeled edges. But since the cycle must have exactly $|S|$ edges, all the edges must be labeled and none can be identical to e. □

2.6. Finding Separators

We use a straightforward breadth-first-search heuristic to find separators of G. We arrange the vertices of G in levels as follows. Arbitrarily select a vertex i of G as a *seed* and let level 0 contain i alone. Let level 1 contain all neighbors of i in G. Let level k (for $k \geq 2$) contain all vertices j of G such that (a) j is a neighbor of some vertex on level $k-1$, and (b) j does not occur in levels 0 through $k-1$. If $m \geq 2$, the vertices on any given level k ($0 < k < m$) form a separator of G. Thus the heuristic yields $m-1$ separators.

The heuristic can be run several times as desired, each time beginning with a different vertex on level 0.

2.7. A Cardinality Filter

The next step of the algorithm is to identify nonpermissible edges of G_S for each separator S. This can in principle be done by an exact algorithm that discovers all such edges. We present here, however, a relaxation of the permissibility condition that can be filtered quickly.

The relaxation is based partly on the global cardinality constraint, which is written

$$\text{gcc}(X, V, \ell, u) \tag{2.4}$$

where X is a finite set of variables, $V = (v_1, \ldots, v_n)$, and ℓ, u are n-tuples of nonnegative integers. The domain of each $x_j \in X$ is a subset of $\{v_1, \ldots, v_n\}$. The constraint requires that at least ℓ_i and at most u_i variables in X take the value v_i, for each $i = 1, \ldots, n$. Hyperarc consistency can be achieved for gcc in $O(|X|^2|V|)$ time using a well-known max flow model [30].

Let X contain a variable x_{ij} (with $i < j$) for each unordered pair i, j of neighboring vertices of G_S. (Vertices are neighbors when they are connected by a labeled or unlabeled edge.) A given permissible hamiltonian cycle H_S of G_S assigns values to the variables as follows: $x_{ij} = U$ if unlabeled edge (i, j) or (j, i) is part of H_S; $x_{ij} = C$ if labeled edge $(i, j)^C$ or $(j, i)^C$ is part of H_S; and $x_{ij} = D$ (for dummy) otherwise. Then since each of the labels $C_1, \ldots, C_p$ in G_S must be assigned to some edge, a permissible hamiltonian cycle satisfies

$$\text{gcc}(X, (C_1, \ldots, C_p, U, D),), (1, \ldots, 1, 0, |X| - |S|), (\infty, \ldots, \infty, |X| - |S|)) \tag{2.5}$$

Since values other than D are assigned to exactly $|S|$ edges, the remaining $|X| - |S|$ variables must receive the value D.

An unlabeled edge (i, j) of G_S is permissible only if $x_{ij} = U$ in some feasible solution of (2.5), and similarly for (j, i). Thus

Theorem 13. *An edge (i, j) of G is hamiltonian, and (j, i) is hamiltonian, only if $x_{ij} = U$ in some feasible solution of (2.5).*

If a gcc filter removes U from the domain of x_{ij}, then neither (i, j) nor (j, i) is hamiltonian. In the example of Fig. 2.1, (2.5) is

$$\text{gcc}(X, (A, B, C, U, D), (1, 1, 1, 0, 0), (\infty, \infty, \infty, \infty, 0))$$

Since $X = \{x_{12}, x_{13}, x_{23}\}$ contains only three variables, no x_{ij} can take the value U, which means that none of the unlabeled edges $(1, 2), (2, 1), (1, 3)$ is permissible. Thus none of these edges is hamiltonian in the original graph.

2.8. Additional Vertex Degree Filtering

Vertex degree filtering is based on the fact that the in-degree and out-degree of every vertex in a hamiltonian cycle is one. We can perform out-degree and gcc filtering simultaneously (or in-degree and gcc filtering simultaneously) by modifying the gcc constraint (2.5) and enlarging the capacitated flow graph that models the constraint.

Rather than introducing a variable x_{ij} for every unordered pair of vertices in G_S, introduce a variable y_{ij} for each *ordered* pair of vertices i, j. The domain of y_{ij} contains label C for each edge $(i, j)^C$ in G_S and the element U if unlabeled edge (i, j) is in G_S. (Due to the construction of the flow network, there will be no need for a dummy element D.) A permissible hamiltonian cycle must satisfy the constraint

$$\text{gcc}(Y, (C_1, \ldots, C_p, U), (1, \ldots, 1, 0), (\infty, \ldots, \infty)) \tag{2.6}$$

where Y is the set of variables y_{ij}. Constraint (2.6) can be combined with out-degree filtering by constructing a capacitated flow graph.

We first recall some basic properties of flow graphs. A capacitated flow graph G is a directed graph with a capacity range $[\ell_{ij}, u_{ij}]$ for each edge (i, j). A *flow* f on G assigns a flow volume on every edge so that the total flow entering each vertex equals the total flow leaving the vertex. The

flow f is feasible if $\ell_{ij} \leq f_{ij} \leq u_{ij}$ for each edge (i, j). The *residual graph* $R(f)$ for a given feasible f is the graph on the same vertices as G that contains an edge (i, j) with capacity range $[0, u_{ij} - f_{ij}]$ whenever $f_{ij} < u_{ij}$ and an edge (j, i) with capacity $[0, f_{ij} - \ell_{ij}]$ whenever $f_{ij} > \ell_{ij}$. An *augmenting path* from j to i is a path in $R(f)$ that does not include edge (i, j). The following is a standard result of flow theory.

Theorem 14. *A given feasible flow f on graph G maximizes the flow on edge (i, j) if and only if there is no augmenting path from j to i.*

Since (i, j) is an edge of $R(f)$, there is an augmenting path from j to i if and only if (i, j) is contained in a directed cycle of $R(f)$. Such a cycle exists if and only if i and j belong to the same strongly connected component of $R(f)$. Thus one can check which edges have a maximum flow of zero by computing the strongly connected components of $R(f)$.

We now construct a capacitated flow graph G_S^{out} with the following vertices

source s and sink t

U and $C_1, \ldots, C_p$

a vertex for each $y_{ij} \in Y$

a vertex for every vertex of G_S

and the following directed edges

(s, C_i) with capacity range $[1, \infty)$ for $i = 1, \ldots, p$

(s, U) with capacity range $[0, \infty)$

(C, y_{ij}) with capacity range $[0, 1]$ for every edge $(i, j)^C \in E_S$

(U, y_{ij}) with capacity range $[0, 1]$ for every unlabeled edge $(i, j) \in E_S$

(y_{ij}, i) with capacity range $[0, 1]$ for every ordered pair (i, j) such that

(i, j) or $(i, j)^C$ belongs to E_S

(i, t) with capapcity range $[0, 1]$ for every vertex i of G_S

return edge (t, s) with capacity range $[|S|, |S|]$

The gcc and out-degree constraints are simultaneously satisfiable if and only if G_S^{out} has a feasible flow. The same is true of the graph G_S^{in} constructed in an analogous way to enforce in-degree constraints. Thus

Theorem 15. *An edge (i, j) of G is nonhamiltonian if there is a separator S of G for which the maximum flow on arc (U, y_{ij}) of either G_S^{out} or G_S^{in} is zero.*

This can be checked by first computing a feasible flow f on G_S^{out} and on G_S^{in}. The maximum flow on (U, y_{ij}) is zero if (a) f places zero flow on (U, y_{ij}), and (b) a flow of zero on this arc satisfies the optimality condition of Theorem 14.

For example, the network G_S^{out} for the graph G and separator S of Fig. 2.1 is shown in Fig. 2.2. Since the flow of zero on edges (U, y_{12}), (U, y_{13}) and (U, y_{21}) is maximum in each case, the three edges $(1, 2)$, $(1, 3)$, and $(2, 1)$ are nonhamiltonian.

It is unclear how to combine gcc with both out-degree and in-degree filtering in the same flow model.

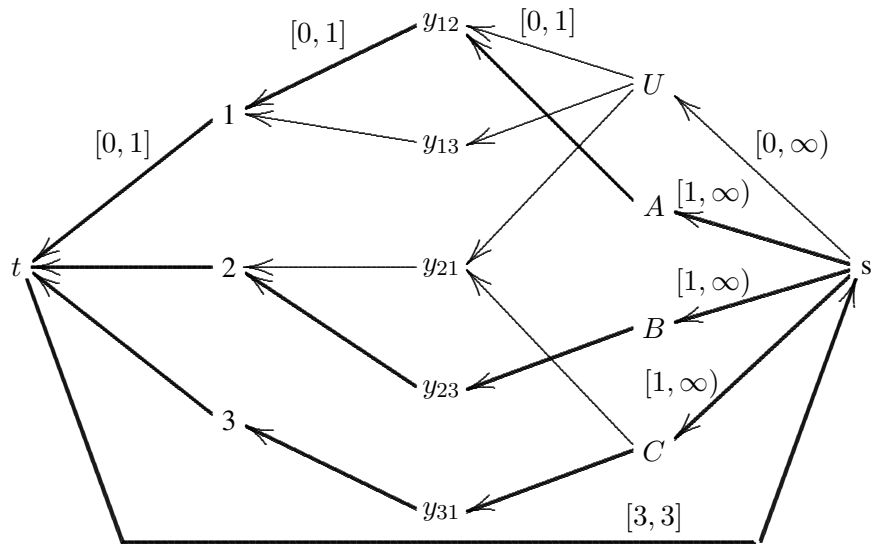

Figure 2.2: *Flow model for simultaneous gcc and out-degree filtering of nonhamiltonian edges. Heavy lines show the only feasible flow.*

2.9. The Algorithm

The filtering algorithm is summed up in Fig. 2.3. If the given graph is $G = (V, E)$, the complexity of finding separators is $O(|E|)$ for each seed vertex.

A feasible flow exists for G_S^{out} (or G_S^{in}) if and only if the maximum flow on (t, s) is $|S|$. There are numerous max flow algorithms with various complexities [1, 12, 28], the best ones having complexity close to $O(mn)$, where n is the number of vertices and m the number of edges. For example, the algorithm of [12] has complexity $O(mn \log(n^2/m))$.

Once a feasible flow is found for G_S^{out} and for G_S^{in}, one can check which edges have a maximum flow of zero by computing the strongly connected components of G_S^{out} and G_S^{in}. Since the classical algorithm for finding strongly connected components has complexity $O(m)$ [33], the filtering complexity for a given S is dominated by the complexity $O(mn)$ of the max flow algorithm, where n is the maximum of the number of vertices in G_S^{out} and G_S^{in}, and m is the maximum of the number of edges in G_S^{out} and G_S^{in}.

We can assume that $|L_S| \leq |S|$, since otherwise G is immediately recognized as nonhamilto-

Let G be the directed graph associated with circuit$(x_1, \ldots, x_n)$.
Let D_i be the current domain of x_i for $i = 1, \ldots, n$.
Let s be a limit on the size of separators considered.
For one or more vertices i of G:
  Use the breadth-first-search heuristic to create a collection $\mathcal{S}$ of separators,
    with i as the seed.
  For each $S \in \mathcal{S}$ with $|S| \leq s$:
    For $G'_S = G_S^{out}, G_S^{in}$:
      If G'_S has a feasible flow f then:
        For each edge (U, y_{ij}) of G'_S on which f places zero flow:
          If (U, y_{ij}) satisfies the condition of Theorem 14 then delete j from D_i.
      Else stop; circuit$(x_1, \ldots, x_n)$ is infeasible.

Figure 2.3: *Filtering algorithm for the circuit constraint.*

nian. Thus

$$n = O(|S| + |Y| + |L_S|) \leq O(|Y|) \leq O(|S|^2)$$

$$m = O(|Y||L_S|) \leq O(|Y||S|) \leq O(|S|^3)$$

So the filtering complexity for each separator S is approximately $O(|S|^5)$. In practice $|S|$ can be bounded by considering only small separators. Also the best implementations of max flow algorithms are extremely fast. The PRF algorithm of [5], for instance, reportedly solves problems on 250,000 vertices in under two minutes. Other implementations are described in [19].

2.10. Computational Results

We implemented the algorithm of Fig. 2.3 in order to investigate what fraction of nonhamiltonian edges it detects. We randomly generated directed graphs in which each possible directed edge occurs with probability p, using various values of p. We discarded all disconnected graphs. We identified all nonhamiltonian edges using an exhaustive search algorithm, for purposes of comparison with the filter. Due to the intensive computation required to perform exhaustive search for a large number of instances, we carried out the tests on relatively small graphs (up to 15 vertices).

We generated multiple separators using the breadth-first-search heuristic described above. Each vertex was used as a seed, and the vertices on each intermediate level were used as a separator.

Table 2.1: Performance of the filtering algorithm for circuit on random graphs that were hamiltonian. The filter successfully identified all random graphs that were nonhamiltonian.

No. vertices	No. instances	Avg. no. edges	Avg. density	Nonham. edges	Avg. no. detected	Avg. % detected
6	885	17.9	0.43	5.0	2.3	37
7	584	20.1	0.36	7.2	3.5	45
8	440	24.0	0.33	8.8	4.1	43
9	573	28.8	0.32	8.9	3.6	39
10	376	31.5	0.29	11.1	4.1	37
11	167	32.7	0.25	13.6	4.5	34
12	129	36.2	0.23	14.5	4.8	32
13	135	40.6	0.22	15.5	4.7	34
14	88	43.0	0.20	18.4	5.1	28
15	156	48.4	0.20	17.0	5.9	34

Since the original graphs never have more than 15 vertices, the separators were all small, and there was no need to limit the size of separators used.

We found that the filter detected *all* nonhamiltonian graphs. The results for hamiltonian graphs appear in Table 2.1. The density of each graph is computed as $m/(n(n-1))$, where m is the number of edges. The average number of edges and average density are shown. The average number of nonhamiltonian edges is indicated, as is the average number of nonhamiltonian edges detected, and the average fraction of nonhamiltonian edges detected.

The filter detects about one-third of nonhamiltonian edges, somewhat more for smaller graphs. Although one might expect a higher fraction to be detected in sparser graphs, the data do not confirm this at least for the range of densities used here. The densities vary from about 60 to 150% of the average for the smaller graphs and 75 to 130% of the average for the larger graphs, and scatterplots reveal no relationship between the density and effectiveness of the filter.

In view of the filter's effectiveness at detecting nonhamiltonian graphs, one could in principle test each edge (i, j) for hamiltonicity by testing the modified graph G_{ij} for hamiltonicity, as described earlier. We did not pursue this idea, due to the computational cost of applying the filter to each G_{ij}.

2.11. Conclusions

We presented a filter for the circuit constraint that identifies some of the nonhamiltonian edges in the associated graph. It does so by analyzing a combinatorial object (a separator graph) that is generally much smaller than the original graph but captures much of its structure. "Nonpermissible" edges in the separator graph are nonhamiltonian in the original graph, and the corresponding values can be filtered from variable domains.

Nonpermissible edges are identified by a secondary filter that solves a max flow problem on an associated flow graph and identifies edges on which the max flow is zero. The complexity of the filtering algorithm is essentially that of solving a max flow problem on the flow graph corresponding to each separator used.

We tested the effectiveness of this filter on a few thousand random graphs with up to 15 vertices, corresponding to instances of the circuit constraint with up to 15 variables. Although the filter is incomplete, it identified all nonhamiltonian graphs, which means that it recognized all infeasible instances of circuit. On hamiltonian graphs, it identified about one-third of the nonhamiltonian edges, which means that it eliminated about one-third of the redundant values in the variable domains.

These results are preliminary but suggest that a filter based on a fast max flow algorithm can accomplish a significant amount of domain reduction for the circuit constraint, although it falls far short of achieving hyperarc consistency.

Several research issues remain. One is how circuit filtering interacts with alldiff filtering, and how much can be accomplished by using both filters simultaneously. Another issue is how much one can restrict the size of separators of large graphs while still retaining an effective filter. It is unclear how effectiveness would be measured in this context, since it is computationally impractical to identify all nonhamiltonian edges for comparison purposes. A third issue, of course, is whether the circuit filter described here can be useful across a variety of practical applications.

Chapter 3

Crane Scheduling by Dynamic Programming

3.1. 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 two or more hoists to move along a single horizontal track attached to 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 crane scheduling problem may appear straightforward at first, but it is highly combinatorial in nature. The combinatorics are threefold: each task must be assigned to a crane, the tasks assigned

to each crane must be sequenced, and the space-time trajectory of each crane must be calculated to carry out the tasks in the right sequence and at the right times. In fact, it is not unusual for managers to put together a production 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 address this problem by developing an algorithm for the crane scheduling problem. It assumes that a production schedule is given in the form of time windows within which each pickup and delivery must occur. It further assumes that there are only two cranes, leaving the multi-crane problem to future research. The algorithm attempts to schedule each task as soon after its earliest start time as possible.

Due to the difficulty of the problem, the algorithm for assigning and sequencing is not exact. The task assignment and sequencing are determined by a local search method. However, once the assignment and sequencing are fixed, a strictly optimal space-time trajectory is sought with a specialized dynamic programming algorithm that exploits the structure of the problem.

We use an exact algorithm to compute the optimal trajectory because it is important to know how much delay in the production schedule is really necessary to accommodate the cranes. A trajectory obtained heuristically typically incurs significant delays that reduce the productivity of expensive equipment. In such cases it is useful to learn whether a better solution could be found with an exact algorithm, or simply does not exist.

Although the assignment and sequencing algorithm is inexact, a local search heuristic is likely to be more effective for this portion of the problem. The sequencing of tasks in a satisfactory solution is likely to follow the sequencing of the production schedule fairly closely, since otherwise the delays could be substantial. There are also many constraints on sequencing that tend to reduce the search space. Similarly, assignments of tasks to cranes are somewhat limited in number by the fact that only a fairly balanced allocation is likely to yield a good solution. The optimal interleaving of tasks in a space-time trajectory, however, is difficult to find heuristically because one must plot a trajectory to a high degree of resolution to determine its feasibility. We therefore use an exact

algorithm for this portion of the problem.

3.2. The Crane Scheduling Problem

In practice, a crane scheduling problem typically consists of a number of *jobs*, each of which requires that the crane make several *stops*. The crane performs some kind of *processing* at each stop, during which the crane must be stationary. Processing may include picking up a ladle, loading in-process material, unloading material, cleaning a production unit, or some other operation. It is convenient to refer to each processing stop as a *task*, which means that a job may consist of several tasks.

The location and processing time for each task are given, as are time windows for the job as a whole. The release time of a job becomes the release time of the first task of that job, and the deadline for a job becomes the deadline of the last task of the job. Constraints are imposed to require that the tasks corresponding to a given job be performed consecutively by the same crane in the proper order. There may be additional precedence and assignment constraints as well.

The problem data are therefore:

$R_j =$ release time of task j

$D_j =$ deadline of task j

$L_j =$ stop location for task j

$P_j =$ processing time for task j

Constraints on crane assigments and task sequencing

If task j is part of a job that consists of tasks $i, i+1, \ldots, k$ (for $k > i$), then the task release time is the earliest possible start time for that task, given the job release time:

$$R_j = R_i + \sum_{\ell=i}^{j-1} \left(P_\ell + \frac{|L_{\ell+1} - L_\ell|}{v} \right)$$

where v is the maximum possible distance traveled by a crane during one time period. Similarly the task deadline is the latest possible finish time, given the job deadline:

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

Section 3.3 discusses the assignment and sequencing constraints in detail. We suppose for generality that there are cranes $1, \ldots, m$, where crane 1 is the *leftmost crane* and crane m the *rightmost crane*, although we solve the problem only for $m = 2$. $T_{\max}$ is the length of the time horizon. Also

$$\bar{L}_0, \bar{L}_1 = \text{leftmost and rightmost crane locations}$$

$$\Delta = \text{minimum crane separation}$$

Thus the rightmost position of the left crane is $\bar{L}_1 - \Delta$, and analogously for the right crane.

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

k_j = crane assigned to task j

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

$$
\begin{array}{lll}
\min\ f(x,y,u,k) & & \\
\left.\begin{array}{l}
\bar{L}_0 \le x_{ct} \le \bar{L}_1 \\
x_{ct} - v \le x_{c,t+1} \le x_{ct} + v \\
y_{ct} > 0 \Rightarrow x_{ct} = L_{y_{ct}}
\end{array}\right\} \text{all } c,t & \begin{array}{l}(a)\\(b)\\(c)\end{array} & \\
x_{ct} \le x_{c+1,t} - \Delta,\ c = 1,\ldots,m-1,\ \text{all } t & (d) & \\
\left.\begin{array}{l}
R_j \le \tau_j \le D_j - P_j,\ \text{all } j \\
y_{k_j t} = j,\ t = \tau_j,\ldots,\tau_j + P_j - 1
\end{array}\right\} \text{all } j & \begin{array}{l}(e)\\(f)\end{array} & \\
y_{ct} \in \{0,\ldots,n\},\ \text{all } c,t & & \\
k_j \in \{1,\ldots,m\},\ \text{all } j & &
\end{array}
\tag{3.1}
$$

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 the required amount of time once it starts.

The objective function $f(k,x,y,a)$ may be defined in various ways. We define it to be a weighted sum of processing delay over the tasks, with an individual weight α_j for each task. The processing delay is defined as a convex combination of (a) the time lapse between the release time and the start of processing and (b) the time lapse between the earliest finish time and the finish of processing. The earliest finish time of task j is $R_j + P_j$. Thus

$$
f(x,y,u,k) = \beta \sum_j \alpha_j(\tau_j - R_j) + (1-\beta) \sum_j \alpha_j(d_j - R_j - P_j) \tag{3.2}
$$

where $0 \le \beta \le 1$, and where d_j is the finish time and is defined by the additional constraints

$$
d_{y_{ct}} \ge t,\ \text{all } c,t
$$

A local search heuristic will be used to assign jobs to cranes and to determine their sequence. Once this is done, the problem that remains is an optimal control problem, which will be solved by dynamic programming.

3.3. Precedence Constraints

In practice, precedence constraints are imposed on groups of jobs as well as individual jobs. We therefore assume that the jobs are partitioned into groups J_i. A precedence relation $J_i < J_k$ means that (a) all the jobs in J_i must be assigned to the same crane, and similarly for all the jobs in J_k, and (b) if the jobs in J_i are assigned to the same crane as the jobs in J_k, then all the jobs in group J_i must precede all the jobs in J_k. That is, every task associated with a job in J_i must finish before any task associated with a job in J_k starts. No particular order is imposed on jobs within a group J_i, and the jobs in J_i need not be consecutive. Even when $J_i < J_k$, the jobs in J_i need not be processed immediately before the jobs in J_k. That is, some task associated with a job in neither J_i nor J_k may be processed after all tasks in J_i are processed and before any task in J_k is processed. When J_i and J_k are singletons, the relation becomes a conventional precedence constraint between two individual jobs.

3.4. Simplifying the Optimal Control Problem

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 *minimal* trajectories. We will show that when there are two cranes, then some pair of minimal trajectories is optimal—provided the objective function depends only on when processing occurs.

Let a *processing schedule* for a given crane consist of the times at which processing starts for each task j. Thus the objective (3.2) is a function of the processing schedule. We define the *canonical* trajectory for the left crane, with respect to a given processing schedule, to be one that observes the processing schedule and that, between loading and unloading, follows the leftmost

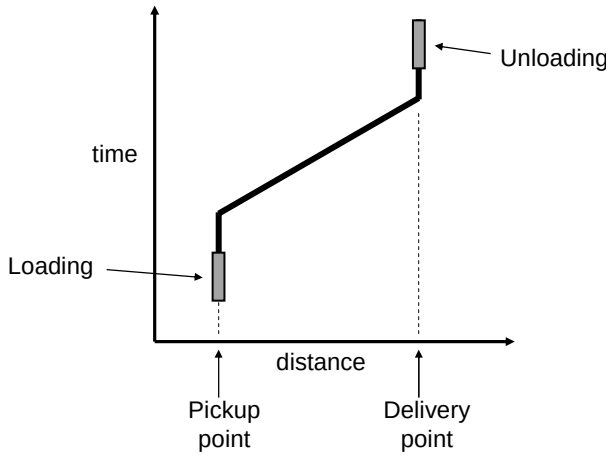

Figure 3.1: Sample space-time trajectory for one task. The shaded vertical bars denote loading and unloading.

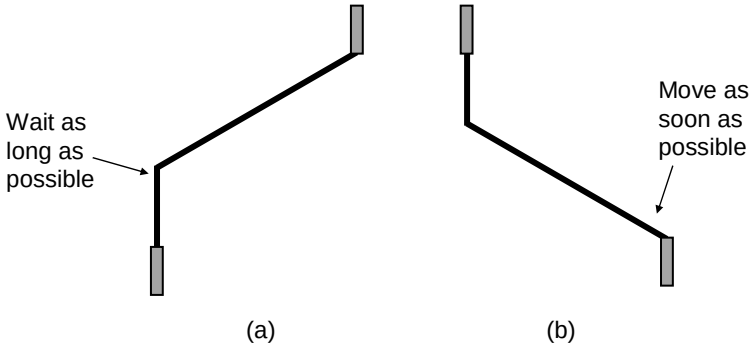

Figure 3.2: Canonical trajectory for the left crane (a) when the destination is to the right of the origin, and (b) when the destination is to the left of the origin.

trajectory that never moves in the direction away from its destination. For example, the trajectory in Figure 3.1 is not canonical, but the trajectories of Figure 3.2 are canonical with respect to the processing schedule shown by the thick vertical bars.

More precisely, if the destination is to the right of the origin, then the left crane follows the canonical trajectory if it leaves a loading (unloading) position as late as possible so as to arrive at the destination just as unloading (loading) starts (Fig. 3.2a). If the destination is to the left of the original, the crane leaves the origin as early as possible (Fig. 3.2b). Thus at any time the crane is either stationary or moving at maximum speed. The canonical trajectory for the right crane follows

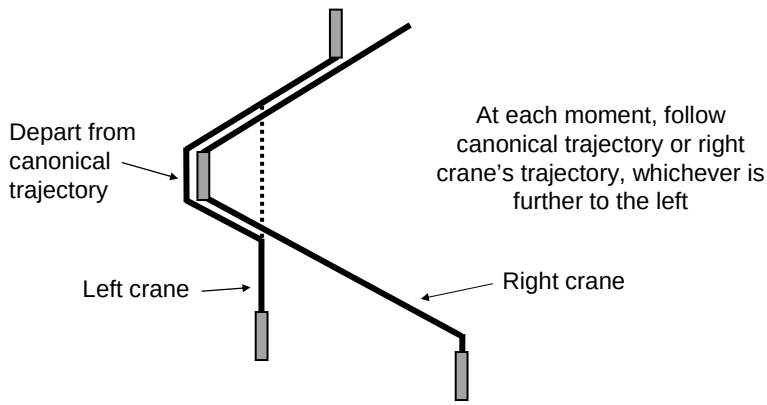

Figure 3.3: Minimal trajectory for the left crane (leftmost solid line).

the rightmost trajectory: it leaves the origin 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 *minimal* with respect to the right crane if at each moment it is the rightmost of (a) the canonical trajectory for the left crane and (b) the trajectory that runs parallel to and just to the left of the right crane's trajectory (Fig. 3.3). More precisely, trajectory x_1' is minimal for the left crane, with respect to trajectory x_2 for the right crane, if the canonical 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 minimal with respect to the left crane if it is the leftmost of the canonical trajectory for the right crane and the left crane's trajectory. That is, $x_2'(t)$ is minimal if $x_2'(t) = \max\{\bar{x}_2(t), x_1(t) + \Delta\}$, where $\bar{x}_2(t)$ is the canonical trajectory.

Theorem 16. *Suppose the objective function of (3.1) depends only on the processing schedule. If (3.1) has an optimal solution, then some optimal pair of trajectories are minimal with respect to each other.*

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

function value, the minimal 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 canonical trajectories for the left and right cranes with respect to the processing schedules in the optimal trajectories.

Consider the minimal 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 minimal 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 minimal 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 minimal with respect to x_2', to prove the theorem it suffices to show that x_2' is minimal 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

$$(x_1''(t), x_2'(t)) = (\bar{x}_1(t), \bar{x}_2(t)) \tag{3.3}$$

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 (3.3) 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 subcases 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 minimal trajectories allow us to consider a very restricted subset of trajectories when computing the optimum.

Corollary 17. *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 processing location 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 processing location 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 processing location, 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 minimal 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 $t+1$, so that $x_1(t+1) < \bar{x}_1(t+1)$. But since

$$x_1(t+1) = \min\{\bar{x}_1(t+1), x_2(t+1) - \Delta\}$$

this means $x_1(t+1) = x_2(t+1) - \Delta$, and crane 1 is adjacent to the other crane. Since crane 1 is moving left between t and $t+1$, 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. □

3.5. 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 four factors: (a) each crane must make certain pickups and deliveries in a certain order; (b) each pickup and delivery must occur at a certain location; (c) each crane must remain in each pickup and delivery location a certain amount of time; and (d) the cranes must not interfere with each other.

This calls for three state variables for each crane. Two state variables are x_{ct} and y_{ct} as defined in model (3.1). The third state variable 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:

$$g_{t+1}(z_{t+1}) = \min_{z_t \in S^{-1}(z_{t+1})} \{h(t, y_t, u_t) + g_t(z_t)\} \tag{3.4}$$

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 (3.2), the cost $h(t, y_t, u_t)$ is $\sum_c h_c(t, y_t, u_t)$, where

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

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 (3.4) computes the minimum $g_{t+1}(z_{t+1})$ and the state $z_t = s_{t+1}^{-1}(z_{t+1})$ that achieves the minimum. Thus $s_{t+1}^{-1}(z_{t+1})$ points to the state that would precede z_{t+1} in the optimal trajectory if z_{t+1} were in the optimal trajectory. The cost table $g_{t+1}(\cdot)$ is stored in memory until $g_{t+2}(\cdot)$ is computed, and then released. 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 $t = N-1, N-2, \ldots, 0$.

3.6. Assignment and Sequencing by Local Search

The assignment of jobs to cranes and the sequencing of jobs on each crane is determined by a simple local search heuristic. An initial *pattern* (i.e., an initial assignment and sequencing) is created using either a balanced or a greedy heuristic. Then a local search consisting of several *rounds* is conducted in order to find feasible crane trajectories. In each round, a certain number of patterns in a neighborhood of the current pattern are examined until a feasible one is found. The number of

patterns (size of neighborhood) to explore is determined by a parameter which can be used to adjust the behavior of the algorithm for various datasets (i.e. when perhaps an initial feasible pattern is hard to find). Feasibility (in terms of trajectory conflicts) is checked by sending the pattern to the dynamic programming algorithm. Since this is an expensive operation, the local search move generator makes sure to never send patterns to the dynamic programming module if they violate assignment or precedence constraints. After a few rounds, the best solution found by the dynamic programming module is selected.

One of the following heuristics is used to obtain an initial pattern:

1. *Balanced allocation.* Assign jobs to cranes in the order of their release times. Assign jobs with stop points closer to the left side of the track to the left crane, and others to the right crane, subject to precedence and crane assignment constraints. Maintain a realistic load balancing between the two cranes by requiring that the ratio of the number of jobs assigned to the two cranes is less than a given user specified parameter. Through experimentation, we determined that (for the available dataset) it was most effective to require the crane with fewer jobs to have at least one-fourth as many jobs as the other crane.

2. *Greedy allocation.* Assign jobs to cranes in the order of their release times. Each job is assigned to the nearest crane, subject to precedence and assignment constraints, assuming that each crane is located at the last stop point of the job most recently assigned to it. Initially the two cranes are assumed to be at their docking positions.

In each round, a (restricted) neighborhood of the current pattern is explored to find a feasible pattern. The neighborhood contains patterns that

1. satisfy precedence and assignment constraints,

2. pass a bounding test, and

3. can be reached by one of the following operators:

SWAP1(c, j, k)	swap the positions of jobs j and k on crane c
SWAP2(c, j, k)	move job j from crane c to k's position on the other crane and k from the other crane to j's position on crane c.
MOVE(c, j, k)	move job j from crane c to a position immediately after k on the other crane.

The jobs assigned to each crane are stored in doubly linked lists, so that the operators can be applied in constant time.

The bounding test is performed on a given pattern by computing a lower bound $v_{\min}$ on the objective function value that results from an optimal solution having that pattern. The pattern is excluded if $v_{\min}$ is greater than or equal to the value of the best solution found in previous rounds. The bound $v_{\min}$ is computed by supposing that each crane can process the jobs assigned to it by moving directly from one stop to another without delay and without yielding to the other crane. The start time τ_j of each task is assumed to be the start time that results from this assumption, and the cost is computed accordingly.

To generate neighboring patterns, operators are selected randomly at first, and later biased towards more "successful" operators. Success is measured by the frequency with which the operator led to a feasible solution in previous rounds. Each pattern that passes tests 1 and 2 is sent to the dynamic programming algorithm, until a feasible solution is found or a maximum number of neighbors (20 was enough in our tests) is reached. The dynamic programming algorithm computes an optimal solution (pair of crane trajectories) having the specified pattern, or else determines that there is no feasible solution with that pattern.

In each round, the neighborhood is centered around the first feasible solution found in the previous round. In the first round, the neighborhood is centered around the initial solution generated as described above, and this initial solution is the first one sent to the dynamic programming algorithm.

3.7. 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

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

The possible state transitions for either crane c are shown in Table 3.1. 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.

Table 3.1: 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 $u_1 > 0$.

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 $h_{t+1}(x_t, y_t, (i,j))$ 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

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

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

$$h_{t+1}((L_y, L_{y'}), (y, y'), (i, j)) \tag{3.5}$$

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 minus 1). In each period we insert the cost (3.5) 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 (3.5) is stored as a table entry $h_{t+1}(x_t, y_t, (i, j))$ 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.

3.8. Experimental Results

Computational tests were performed on a problem that reflect a realistic scheduling problem. It contains 60 jobs, each with a nominal time window of 40 minutes. These time windows were adjusted at run time for the experiments described here. The computation times reported are on a desktop PC running Windows XP with a Pentium D processor 820 (2.8 GHz).

We found that the balanced allocation heuristic is more effective than the greedy allocation

Table 3.2: Computational results for the 60-job problem.

Jobs	Time window (mins)	Computation time for one DP (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

heuristic, although results could be different with other problem sets. The greedy allocation resulted in assignments that were difficult to correct. Often, 50-60 applications of the local operator were necessary to find a feasible solution. We attempted several variations on both these heuristics, but the code complexity of those variations was significant, and there did not seem to be a noticeable improvement in solution quality.

We found that the dynamic programming (DP) algorithm is fast enough to solve the 60-job in reasonable time, although not quickly. The total computation time depends on how many rounds of the assignment and sequencing heuristic are used, and thus on how many times the DP is solved. Table 3.2 shows computation times for one DP solution on the first 10 jobs of the problem, the first 20 jobs, and so forth up to all 60 jobs. We found a feasible solution for all of these problems, except the last, using ten rounds of the heuristic algorithm. To obtain a solution of the full 60-job problem we enlarged all time windows to 95 minutes by postponing the deadlines.

Normally, one DP is solved in each round of the heuristic algorithm, although more may be solved if there is difficulty in finding a feasible solution. The DP solution requires significantly more time in the last round, because the optimal trajectory is retraced only in the last round, and this requires substantial swapping of data in and out of memory. Table 3.2 includes the computation time for recovering the optimal trajectory. Thus most runs of the DP algorithm (all but the last) require less time than indicated in the table.

The optimal solutions appear in Figs. 3.4–3.9. The horizontal axis represents distance along

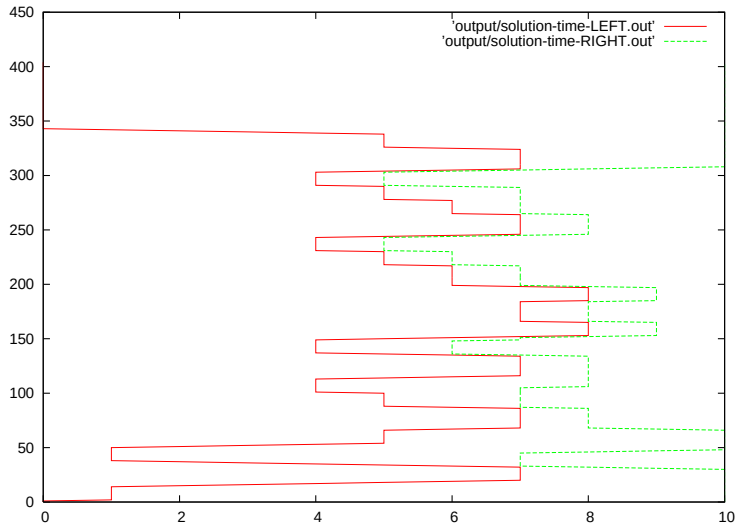

Figure 3.4: Optimal solution for 10 jobs in the 60-job problem.

the track in 10.85-meter increments, so that the entire track is 108.5 meters long. The vertical axis represents time in 6-second increments. Thus the schedule for the 60-job problem spans about 2300 time increments, 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. Note that the cranes are at rest most of the time. The trajectories are minimal trajectories as defined above, which ensures a certain consistency in the way the two cranes interact.

Figures 3.10–3.15 track the evolution of state space size over time. The horizontal axis corresponds to time states of the DP algorithm, which are separated by 6 seconds. The vertical axis is the number of states created in each time period. The state space size remains quite reasonable, never exceeding 2000 states, even though the theoretical maximum is astronomical.

Computation time is sensitive to the width of the time windows. Typically, only a few time windows must be wide to allow a feasible solution, but it is difficult to predict which are the critical windows. 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 3.7.

Table 3.3 reveals the critical importance of these techniques. Without them, we were unable to

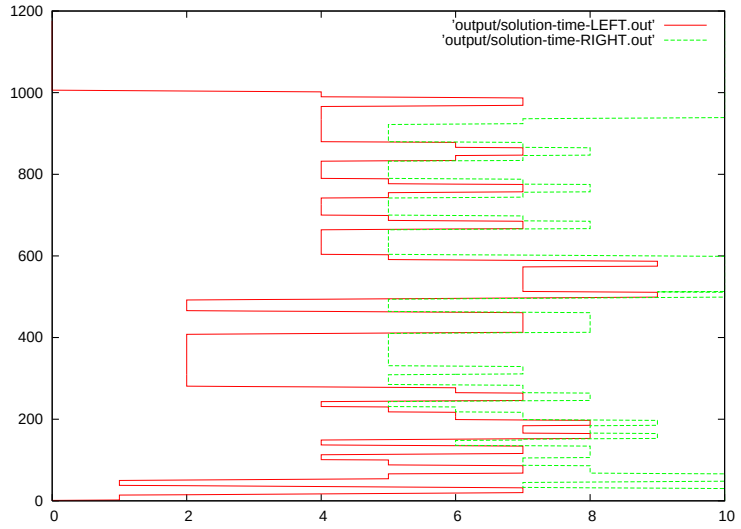

Figure 3.5: Optimal solution for 20 jobs in the 60-job problem.

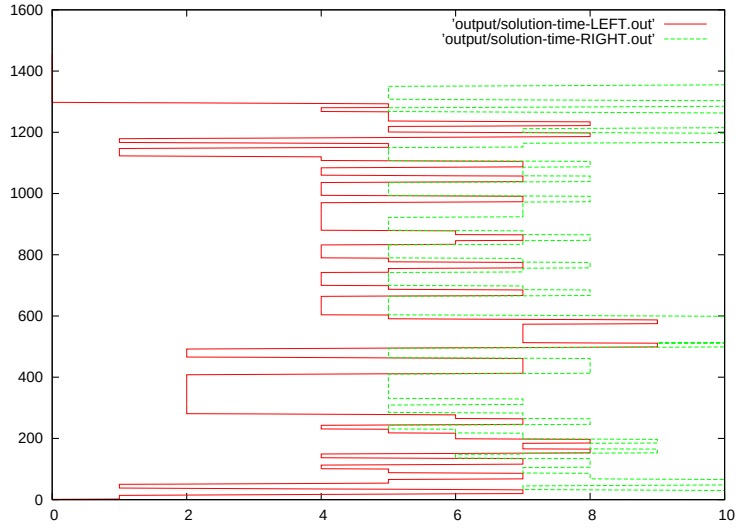

Figure 3.6: Optimal solution for 30 jobs in the 60-job problem.

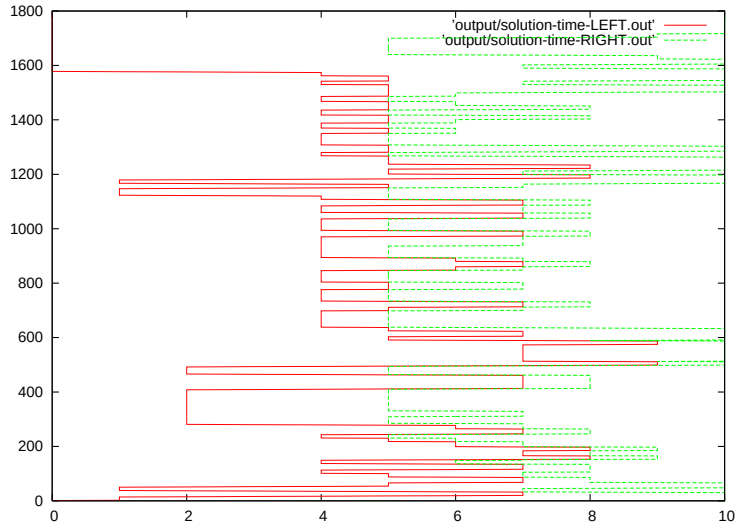

Figure 3.7: Optimal solution for 40 jobs in the 60-job problem.

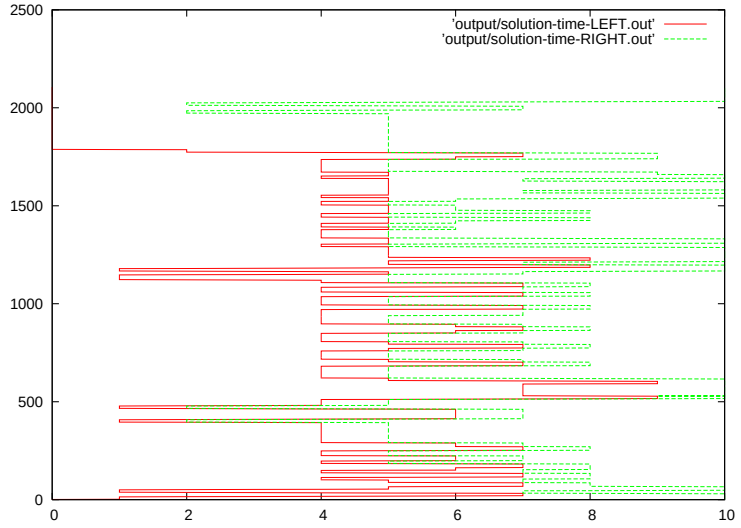

Figure 3.8: Optimal solution for 50 jobs in the 60-job problem.

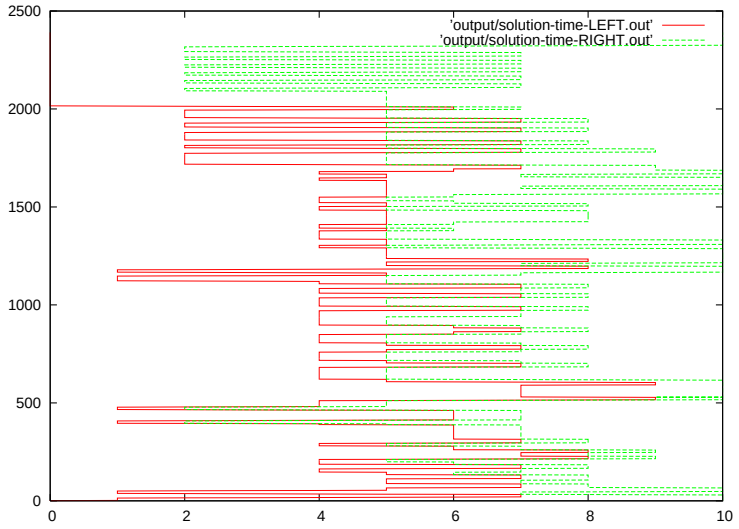

Figure 3.9: Optimal solution for 60 jobs in the 60-job problem.

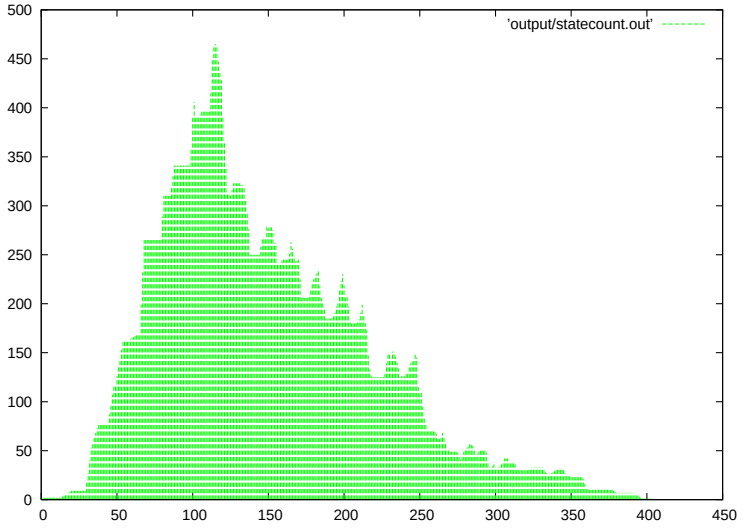

Figure 3.10: State space size for 10 jobs in the 60-job problem, using 25-minute time windows.

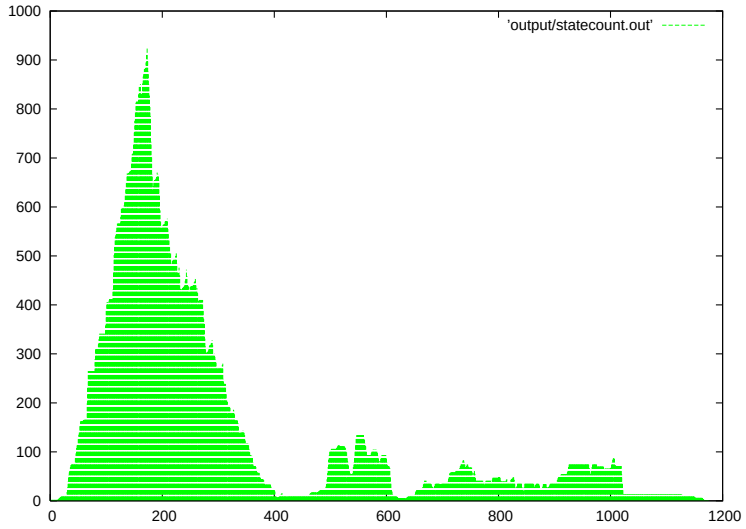

Figure 3.11: State space size for 20 jobs in the 60-job problem, using 35-minute time windows.

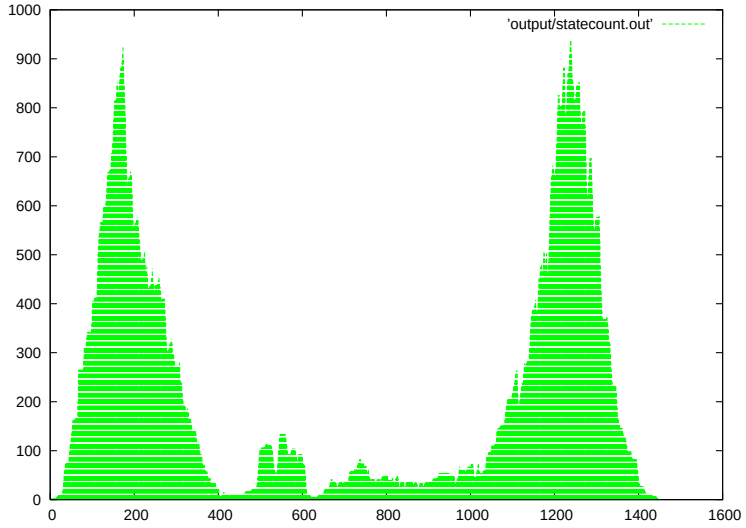

Figure 3.12: State space size for 30 jobs in the 60-job problem, using 35-minute time windows.

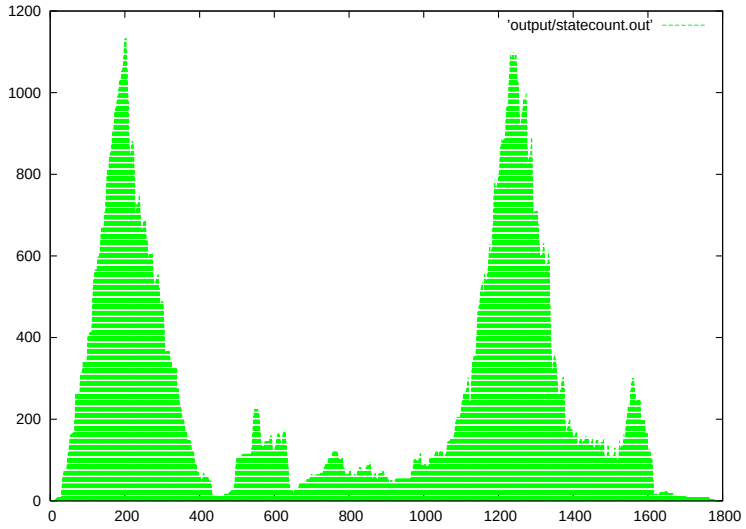

Figure 3.13: State space size for 40 jobs in the 60-job problem, using 40-minute time windows.

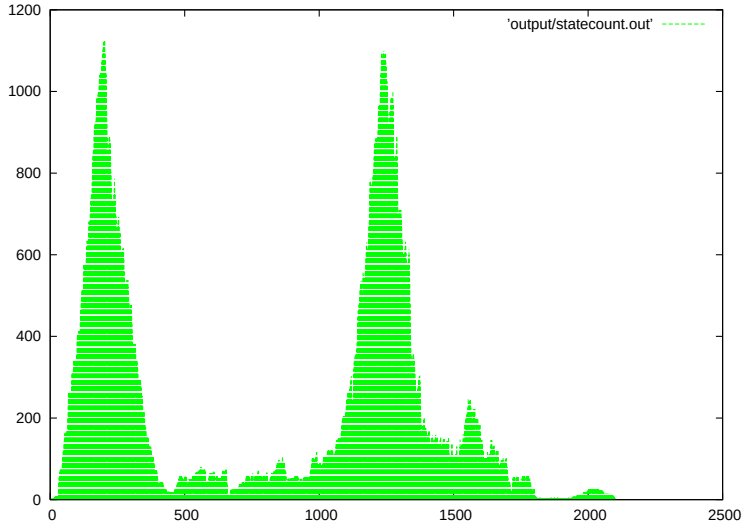

Figure 3.14: State space size for 50 jobs in the 60-job problem, using 40-minute time windows.

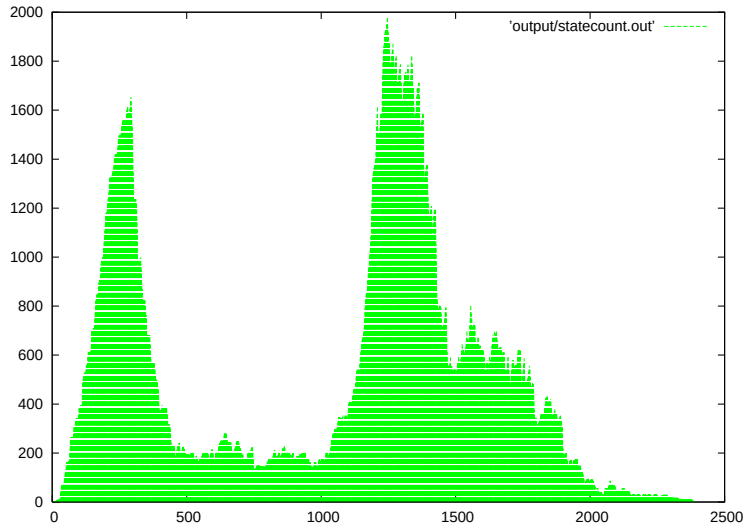

Figure 3.15: State space size for 60 jobs in the 60-job problem, using 55-minute time windows.

Table 3.3: Effect of state space reduction on computation time for ten rounds. "Before" and "after" refer to results before and after state space reduction, respectively.

Jobs	Time window (mins)	Avg # states (optimal DP)		Peak # states (optimal DP)		Time for 10 rounds (secs)*	
		Before	After	Before	After	Before	After
10	25	3224	139	9477	465	158	20
20	35	3200	144	22,204	927	826	86
30	35	3204	216	22,204	940	1438	150

* Multiple DPs solved in some rounds.

solve the 60-job instance with more than 30 jobs. 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, as shown in Table 3.3, the state space reduction techniques reduce the number of states by a factor of about 20, and the computation time by a factor of 10. It is this state space reduction that makes the full 60-job problem tractable.

Chapter 4

The Minimum Product Cut Problem

4.1. Introduction

In this chapter we consider the minimum product cut problem on an undirected graph. Given two nonnegative linear cost functions on edges, the minimum product cut problem is to find an edge cut on the graph at which the product of cut values relative to weight functions is minimized.

Let $G = (V, E)$ be a connected undirected graph with two nonnegative linear edge weight functions $c_1 : E \rightarrow \mathbb{R}_+$ and $c_2 : E \rightarrow \mathbb{R}_+$. The minimum cut problem relative to c_i is to find a bipartition (cut) of the nodes minimizing the weight of the edges going between the parts. The problem can be efficiently solved by any max-flow algorithm and the max-flow min-cut theorem [1]. The problem of finding minimum ratio cut, where the ratio of the cut values $\frac{c_1(\delta(S))}{c_2(\delta(S))}$ [1] is minimized, can be solved in polynomial time by applying Megiddo's method [25]. We consider an extension of the minimum cut problem where the goal is to find a bipartition $(S, \bar{S})$ of the node set V minimizing the product of the sum of the weights of the edges crossing the bipartition.

$$min_{S \subset V}[c_1(\delta(S))c_2(\delta(S))] \tag{4.1}$$

[1] $\delta(S)$ denotes the edges that cross the cut S and $c_i(\delta(S))$ is the sum of the weights of the edges -relative to c_i weight function- that cross the cut S

Example. When both weight functions are identical and constant, the product value of a cut is the square of the number of edges crossing the cut times the square of the constant weight. If all weights are unit then minimum product cut value is simply the square of the cardinality of the cut. In such cases, a minimum cut is still optimal for the minimum product cut problem.

An application of this problem arises in clustering problems. When c_1 and c_2 are two distinct nonnegative similarity weights, the minimum product cut is a good measure of separation relative to c_1 and c_2 at the same time.

4.2. Related Work

Minimum product cut problem is also a special case of the problem of minimizing the product of two linear cost functions over a polytope as described by Goyal in [14]. General problem Π is formulated as

$$\begin{aligned} \min\ (c_1^T x) \cdot (c_2^T x) \\ x \in P \end{aligned} \tag{4.2}$$

where $c_1, c_2 \in \mathbb{R}^n$, $P = \{x \in \mathbb{R}^n | Ax \geq b\}$ and $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$.

This is itself a special case of general quadratic programming (QP) problem where the objective function is $f(x) = (a^T x + x^T C x)$ with $a \in \mathbb{R}^n, C \in \mathbb{R}^{n \times n}$. If $f(x)$ is nonconvex then the problem is in general NP-Hard. Nonconvex QP is extensively studied in the literature and it has a wide range of applications from portfolio analysis to VLSI design, optimal power flow and economic dispatch [13].

The objective function in 4.2 is in general nonconvex and the problem is NP-Hard [24], but the complexity of the special cases of 0-1 problems such as minimum product cut is still open. An FPTAS for 4.2 is given in [21]. Goyal [14] describes a different and faster $(1+\epsilon)$-approximation algorithm briefly explained in Section 4.2. Independent from this work, we present a 4-approximation algorithm for the minimum product cut problem in Section 4.4.

4.2.1 $(1+\epsilon)$-Approximation Algorithm

It is already known due to Kern and Woeginger [21] that the objective function $(c_1^T x) \cdot (c_2^T x)$ attains its minimum at an extreme point of polytope P. Therefore, Goyal [14] considers the following parametric problem $\Pi(B)$ for a given parameter $B > 0$

$$\begin{aligned} \min\ (c_1^T x) \\ c_2^T x \leq B \\ x \in P \end{aligned} \tag{4.3}$$

and shows that a basic optimal solution to this problem can be written as a convex combination of at most two extreme points of polytope P. Moreover, at one of these extreme points of P, objective function value of the main problem 4.2 is at most parameter B times the optimal objective value of 4.3.

He presents the following parametric $(1+\epsilon)$-approximation algorithm for minimizing the product of two nonnegative linear functions that finds an extreme point of P that is a $(1+\epsilon)$-approximation for the problem formulated as 4.2.

Given $c_1, c_2 \in \mathbb{R}_+^n$, polytope P and $\epsilon > 0$.
Initialize $M \leftarrow \frac{\max_{x \in P} c_2^T x}{\min_{x \in P} c_2^T x}$, $N_M = \lceil \log_{1+\epsilon} M \rceil$ and $c_s \leftarrow \infty$.

1. For $j = 1, \ldots, N_M$,
   (a) Let $B = (1+\epsilon)^j$ and let $\tilde{x}(B)$ be a basic optimal solution for $\Pi(B)$.
   (b) Find $\hat{x}(B) \in extr(P)$ such that
   $$(c_1^T \hat{x}(B)) \cdot (c_2^T \hat{x}(B)) \leq (c_1^T \tilde{x}(B)) \cdot B.$$
   (c) If $c_s > (c_1^T \hat{x}(B)) \cdot (c_2^T \hat{x}(B))$, then
   $$\begin{aligned} x_s &\leftarrow \hat{x}(B) \\ c_s &\leftarrow (c_1^T \hat{x}(B)) \cdot (c_2^T \hat{x}(B)) \end{aligned}$$
2. Return the solution x_s.

Figure 4.1: *Algorithm $\mathcal{A}$ for Minimizing Product of Two Nonnegative Linear Costs*

Algorithm A returns an approximate solution that is an extreme point of the polytope. Therefore

it can be applied to 0-1 problems when either the convex hull of feasible integer solutions or the convex hull of the dominant [2] of feasible integer solutions is known. His algorithm has applications to problems such as Minimum Product Spanning Tree, Minimum Product Matching, Minimum Product Submodular Flows, Minimum Product s,t-Min-Cut, Minimum Product s,t-Path. Although polyhedral description of minimum s,t-cuts, $s, t \in V$, is not available, convex hull of dominant of 0-1 incidence vectors of minimum s,t-cuts is known [10].

4.3. Preliminaries

A *c-approximation algorithm* for a minimization problem runs in polynomial time and for all instances of the problem it produces a solution value at most c times the optimal. The ratio c is the performance ratio of the algorithm, that is the maximum ratio by which the result of a c-approximation algorithm may depart from the optimal solution. (For more details see [16] and [34])

A *bond* is a minimal nonempty edge cut in a connected graph. Observe that if G is a connected graph, then an edge cut $E' \subset E$ is a bond if and only if $G - E'$ has exactly two connected components.

Lemma 18. *Let $G(V, E)$ be an undirected connected graph with nonnegative linear edge weight functions c_1 and c_2. Then the optimal solution to the minimum product cut problem is a bond.*

Proof. A bond is a minimal nonempty edge cut in a graph. Therefore any edge cut in a graph contains a bond. Since weight functions are nonnegative, cut values relative to c_1 and c_2 are at least as much as the cut values of any bond contained in that cut. Therefore in any cut, product of the cut values relative to c_1 and c_2 is at least the product value of any bond in the cut. Thus, the minimum is reached at a bond. □

Define function $P(\lambda)$, $\lambda \in \mathbb{R}_+$ as $P(\bar{\lambda}) = min_{S \subset V}[(c_1 + \bar{\lambda} c_2)(\delta(S))]$. i.e., $P(\lambda)$ is the minimum $c_1 + \bar{\lambda} c_2$-cut value. $P(\lambda)$, $\lambda \geq 0$, is well-defined and is a piece-wise linear concave function of λ. We will use the term "$\lambda - mincut$" to denote minimum $c_\lambda = c_1 + \lambda c_2$-cut. Points at

[2] Dominant of K is defined as $Dom(K) = \{x : x \geq y \; for \; some \; y \in K\}$.

which the slope of $P(\lambda)$ changes are called *breakpoints*. If $\lambda^1 < \lambda^2$ are two consecutive breakpoints of $P(\lambda)$ and C^* is a minimum λ^*-cut where $\lambda^1 < \lambda^* < \lambda^2$, then C^* is optimal over the entire closed interval $[\lambda^1, \lambda^2]$ and nowhere else. At each breakpoint, the minimum cut changes but it stays the same until the next breakpoint. However, the minimum cut value increases as the parameter λ increases.

We define $Next(\lambda)$ such that for a fixed value λ^* of λ, $Next(\lambda^*)$ is the first breakpoint of $P(\lambda)$ that is strictly larger than λ^*, if there is any. If there is no such breakpoint, then $Next(\lambda^*) = \infty$.

Let M be a minimum cut algorithm and consider it as a binary decision tree T. Without loss of generality assume that each branch point in T is a comparison. One branch is for " $\geq$ " and the other is for " $<$ " relation. Given a value of λ, $\bar{\lambda}$, the evaluation of $P(\bar{\lambda})$ defines a path Q through T. Gusfield's algorithm [15], developed from Megiddo's method [25], takes a value of λ and an optimal solution at that λ value, i.e., a $\bar{\lambda}$-min cut as input, and outputs the smallest breakpoint greater than $\bar{\lambda}$ and an optimal solution (minimum cut) at that breakpoint. In other words, given $\bar{\lambda}$, it computes $Next(\bar{\lambda})$. It simulates algorithm M symbolically and forces M to follow path Q even though $\bar{\lambda}$ is not known until the end of the computation. Its running time is at most twice the running time of algorithm M, that is running time of min cut algorithm in our case. (For more detail see [15]).

4.4. An Approximation Algorithm

4.4.1 4-Approximation Algorithm

In this section we will describe a 4-approximation algorithm for the minimum product cut problem. The algorithm uses parametric search over the parameter λ to find $\bar{\lambda}$ such that *product* value of the cut is minimized at $\bar{\lambda}$-min cut. The idea is similar to the one described in [29].

The algorithm searches over all λ-min cuts, $\lambda \geq 0$, to find a cut that has minimum product cut value i.e., it solves the problem

$$min_{\lambda \geq 0}[c_1(\delta(S_\lambda))c_2(\delta(S_\lambda))]$$

where $c_\lambda = c_1 + \lambda c_2$, and S_λ is a minimum c_λ-cut.

Theorem 19. *There exists a value of λ, λ^*, such that S_{λ^*} is a 4-approximation for the minimum product cut problem.*

Proof. Let S^* be an optimal solution to the minimum product cut problem and $\lambda^* = \alpha \frac{c_1(\delta(S^*))}{c_2(\delta(S^*))}$, $\alpha > 0$.

Consider minimum S_{λ^*} cut. Since λ and c_2 are nonnegative and S_{λ^*} is a minimum λ^*-cut, we can write

$$\begin{aligned} c_1(\delta(S_{\lambda^*})) &\leq c_1(\delta(S_{\lambda^*})) + \lambda^* c_2(\delta(S_{\lambda^*})) \\ &\leq c_1(\delta(S^*)) + \lambda^* c_2(\delta(S^*)) \\ &= (1+\alpha) c_1(\delta(S^*)) \end{aligned}$$

where the second inequality follows from the optimality of S_{λ^*}.

We can also write

$$\begin{aligned} c_2(\delta(S_{\lambda^*})) &\leq \frac{1}{\lambda^*}(c_1(\delta(S_{\lambda^*})) + \lambda^* c_2(\delta(S_{\lambda^*}))) \\ &\leq \frac{1}{\lambda^*}(1+\alpha) c_1(\delta(S^*)) \\ &= \left(\frac{1+\alpha}{\alpha}\right) c_2(\delta(S^*)) \end{aligned}$$

since c_1 and λ are nonnegative.

Then

$$c_1(\delta(S_{\lambda^*})) c_2(\delta(S_{\lambda^*})) \leq \frac{(1+\alpha)^2}{\alpha} c_1(\delta(S^*) c_2(\delta(S^*)).$$

However, for $\alpha > 0$, $\frac{(1+\alpha)^2}{\alpha}$ attains its minimum at $\alpha = 1$. Therefore, for $\alpha = 1$ we get

$$c_1(\delta(S_{\lambda^*})) c_2(\delta(S_{\lambda^*})) \leq 4 c_1(\delta(S^*) c_2(\delta(S^*)). \ \square$$

This result suggests that if all breakpoints of $P(\lambda)$ and parametric minimum cuts at those breakpoints are considered, a cut at which product cut value is at most four times of the optimal one can be found. If the number of breakpoints of $P(\lambda)$ is super-polynomial than we may not find the exact optimal solution by this parametric algorithm. Carstensen [3] showed by an example that parametric minimum cut problem can have exponential number of breakpoints. The graph in Carstensen's example includes directed edges and negative edge weights. However, we conjecture that the number of breakpoints can also be super-polynomial when the graph is undirected and the weight functions are nonnegative.

Given $G = (V, E)$, $c_1, c_2 \in \mathbb{R}^n_+$, Minimum Cut Algorithm M.
Initialize $\lambda \leftarrow 0$. Use M to find λ-min cut.
Set $S \leftarrow \lambda - min\ cut$ and $Prod \leftarrow c_1(\delta(S))c_2(\delta(S))$.

1. While $\lambda < \infty$:
   (a) Use Gusfield's [15] algorithm to find $Next(\lambda)$.
   (b) $\lambda \leftarrow Next(\lambda)$.
   (c) If $c_1(\delta(S_{Next(\lambda)}))c_2(\delta(S_{Next(\lambda)})) < Prod$ then

$$\begin{array}{ll} S & \leftarrow S_{Next(\lambda)} \\ Prod & \leftarrow c_1(\delta(S_{Next(\lambda)}))c_2(\delta(S_{Next(\lambda)})) \end{array}$$

2. Return solution S and $Prod$.

Figure 4.2: 4-*Approximation Algorithm for Minimum Product Cut*

4.4.2 A Simple Bound on Breakpoints

Theorem 20. *The number of breakpoints of $P(\lambda)$ is o(min{the number of distinct c_1-cut values, the number of distinct c_2-cut values}).*

Proof. We know that $P(\lambda)$ is a piecewise linear concave function. If $\lambda^1 < \lambda^2$ are two adjacent breakpoints then c_1 value of λ^1-min cut is less than c_1 value of λ^2-min cut and c_2 value of λ^1-min cut is greater than c_2 value of λ^2-min cut. Therefore at each breakpoint both c_1 and c_2 values change. Thus, $P(\lambda)$ can have at most o(min{the number of distinct c_1-cut values, the number of distinct c_2-cut values}) breakpoints. □

Corollary 21. *If at least one of the weight functions is uniform (all weights are either* 0 *or some constant* c*) then the number of breakpoints is polynomial and algorithm has a faster (polynomial) running time.*

Proof. Without loss of generality, assume that c_1 is uniform, then c_1 assigns either 0 or some constant c to each edge. If the graph has n nodes, then there are at most $\frac{(n^2-n)}{2}$ edges. Hence, there are at most that many distinct cut values relative to c_1 weight function. Among the parametric cuts having the same c_1-cut value, only one with the smallest c_2-cut value can contribute to a breakpoint. Hence we have at most $\frac{(n^2-n)}{2}$ breakpoints in such cases. □

We do parametric search for non-negative values of λ. At $\lambda = 0$, minimum c_λ-cut is a minimum c_1-cut, therefore we have a value to start with for the parameter λ and a solution to parametric min-cut problem at that λ value. Gusfield's algorithm is applied with input $\lambda = 0$ and S =minimum c_1-cut to find the next breakpoint and a parametric minimum-cut at that point. Product values of these two cuts are compared and the best one is chosen. Successive iteration of Gusfield's algorithm over all breakpoints provides product values of all optimum parametric cuts.

Many polynomial minimum-cut algorithms are known [1]. Therefore, once a breakpoint and an optimal parametric cut with respect to that point are known, by Gusfield's algorithm, finding the next breakpoint and an optimal parametric cut at that breakpoint takes polynomial time. Therefore, if the number of breakpoints is polynomial then overall running time of the algorithm is strongly polynomial.

Corollary 22. *Let* G *is defined on* n *nodes and* c_1 *and* c_2 *are integer functions. If* $c_1 < C_1$ *and* $c_2 < C_2$ *for some integers* C_1 *and* C_2*, then the number of breakpoints is at most* $min\left\{\frac{(n^2-n)}{2}C_1, \frac{(n^2-n)}{2}C_2\right\}$.

Proof. There are at most $\frac{(n^2-n)}{2}C_1$ distinct c_1-cut values and at most $\frac{(n^2-n)}{2}C_2$ distinct c_2-cut values since the graph has at most $\frac{(n^2-n)}{2}$ edges. Then, the claim follows from Theorem 20. However, in such cases, the running time of the algorithm is pseudo-polynomial. □

4.4.3 Binary Search on λ

An alternate implementation of this algorithm uses binary search on λ. We know that the product value of λ^*-min cut is at most 4 times the optimum. Therefore we perform binary search for scaled values for the numerator and denominator of λ.

All c_i-cuts range from a lower bound L_i to an upper bound U_i for $i = 1, 2$. A lower bound for a c_i-cut, L_i, is the minimum c_i-cut value tat can be found easily by applying any min-cut algorithm. An upper bound for c_i cut, U_i, is the sum of all edge weights in the graph relative to c_i weight function. Since product value of $\lambda^* = \frac{c_1(\delta(S^*))}{c_2(\delta(S^*))}$-minimum cut provides a 4-approximation for the minimum product cut problem, we try each λ having numerator

$$L_1, L_1(1+\epsilon), L_1(1+\epsilon)^2, \ldots, L_1(1+\epsilon)^{log(U_1/L_1)}$$

and denominator

$$L_2, L_2(1+\epsilon), L_2(1+\epsilon)^2, \ldots, L_2(1+\epsilon)^{log(U_2/L_2)}$$

for some $\epsilon > 0$, where the log is taken to the base $(1+\epsilon)$.

The number of parameters that must be searched is

$$O(\log_{1+\epsilon}(U_1/L_1)\log_{1+\epsilon}(U_2/L_2)).$$

Theorem 23. *There exists a* $2\left[\frac{1}{(1+\epsilon)} + (1+\epsilon)\right]^2$*-approximation algorithm for the minimum product cut problem, for any* $\epsilon > 0$.

The running time is $O((\log_{(1+\epsilon)}(U_1/L_1)\log_{(1+\epsilon)}(U_2/L_2)T(min-cut))$ *where* U_i *and* L_i *are upper and lower bounds for the value of any* c_i*-cut (for* $i = 1, 2$*) and* $T(min-cut)$ *is the running time of any min-cut algorithm.*

Proof. Let S^* be a minimum product cut and $\lambda^* = \frac{c_1(\delta(S^*))}{c_2(\delta(S^*))}$. In a parametric search, some numerator smaller than or equal to $(1+\epsilon)c_1(\delta(S^*))$ and some denominator greater than or equal to $\frac{c_2(\delta(S^*))}{(1+\epsilon)}$ are considered for λ. Since both numerators and denominators are increased by a factor of

$(1+\epsilon)$, at least one parameter value is $\tilde{\lambda}$ such that

$$\lambda^* \leq \tilde{\lambda} \leq (1+\epsilon)^2 \lambda^*.$$

Now, since minimum parametric cut value gets larger as λ increases, we can write

$$\begin{aligned} c_1(\delta(S_{\tilde{\lambda}})) &\leq c_1(\delta(S_{\tilde{\lambda}})) + \tilde{\lambda} c_2(\delta(S_{\tilde{\lambda}})) \\ &\leq c_1(\delta(S_{(1+\epsilon)^2\lambda^*})) + (1+\epsilon)^2 \lambda^* c_2(\delta(S_{(1+\epsilon)^2\lambda^*})) \\ &\leq c_1(\delta(S^*)) + (1+\epsilon)^2 \lambda^* c_2(\delta(S^*)) \\ &\leq (1+(1+\epsilon)^2) c_1(\delta(S^*)). \end{aligned}$$

The third inequality follows since $S_{(1+\epsilon)^2\lambda^*}$ is a minimum c_λ-cut at $\lambda = (1+\epsilon)^2 \lambda^*$.

Since S_{λ^*} is a λ^*-min cut and $S_{\tilde{\lambda}}$ is a $\tilde{\lambda}$-min cut, we can write

$$c_1(\delta(S_{\lambda^*})) + \lambda^* c_2(\delta(S_{\lambda^*})) \leq c_1(\delta(S_{\tilde{\lambda}})) + \lambda^* c_2(\delta(S_{\tilde{\lambda}}))$$

and

$$c_1(\delta(S_{\tilde{\lambda}})) + \tilde{\lambda} c_2(\delta(S_{\tilde{\lambda}})) \leq c_1(\delta(S_{\lambda^*})) + \tilde{\lambda} c_2(\delta(S_{\lambda^*})).$$

Therefore, the two lines $c_1(\delta(S_{\lambda^*})) + \lambda c_2(\delta(S_{\lambda^*}))$ and $c_1(\delta(S_{\tilde{\lambda}})) + \lambda c_2(\delta(S_{\tilde{\lambda}}))$ intersects at some λ such that $\lambda^* \leq \lambda \leq \tilde{\lambda}$. Then, since $\lambda^* \leq \tilde{\lambda} \leq (1+\epsilon)^2 \lambda^*$, and

$$c_1(\delta(S_{\tilde{\lambda}})) + \tilde{\lambda} c_2(\delta(S_{\tilde{\lambda}})) \leq c_1(\delta(S_{\lambda^*})) + \tilde{\lambda} c_2(\delta(S_{\lambda^*}))$$

we can write

$$c_1(\delta(S_{\tilde{\lambda}})) + (1+\epsilon)^2 \lambda^* c_2(\delta(S_{\tilde{\lambda}})) \leq c_1(\delta(S_{\lambda^*})) + (1+\epsilon)^2 \lambda^* c_2(\delta(S_{\lambda^*})).$$

Therefore,

$$\begin{aligned} c_2(\delta(S_{\tilde{\lambda}})) &\leq \frac{c_1(\delta(S_{\tilde{\lambda}})) + (1+\epsilon)^2\lambda^* c_2(\delta(S_{\tilde{\lambda}}))}{(1+\epsilon)^2\lambda^*} \\ &\leq \frac{c_1(\delta(S_{\lambda^*})) + (1+\epsilon)^2\lambda^* c_2(\delta(S_{\lambda^*}))}{(1+\epsilon)^2\lambda^*} \\ &\leq \frac{2c_1(\delta(S^*)) + (1+\epsilon)^2\lambda^* 2c_2(\delta(S^*))}{(1+\epsilon)^2\lambda^*} \\ &= 2\left(\frac{1+(1+\epsilon)^2}{(1+\epsilon)^2}\right) c_2(\delta(S^*). \end{aligned}$$

where the third inequality follows from the proof of Theorem 19.

Thus we get,

$$c_1(\delta(S_{\tilde{\lambda}}))c_2(\delta(S_{\tilde{\lambda}})) \leq 2\left[(1+\epsilon) + \frac{1}{1+\epsilon}\right]^2 c_1(\delta(S^*))c_2(\delta(S^*)). \square$$

4.5. Special Cases

4.5.1 Outerplanar Graphs

An *outerplanar* graph is a planar graph that has an embedding on the plane with all vertices appearing on the outerface. The number of bonds in outerplanar graphs is polynomially bounded. Moreover, all of these bonds can be efficiently found in polynomial time, hence the exact solution.

Lemma 24. *The number of bonds in an n-node outerplanar graph is $\binom{n}{2}$.*

Proof. Any bond in an outerplanar graph is a partition of the outer cycle into two pieces that we get when it is cut by a line. Suppose that this is not true. Then there exists a bond that has at least 4 pieces of the outer cycle such that neighbor pieces belong to different sides of the bond. Take any two pieces belonging to the same side of the bond. Since both sides are connected components in a bond, these two pieces have to be connected by a chord. This is also true for the other pair of pieces. However, then these two chords cross. This contradicts to the assumption that G is planar. Hence,

any bond in an outerplanar graph looks like two pieces of a cycle cut by a line. □

A cycle can be cut by a line by choosing two points on the cycle and drawing a line separating them. Therefore, there are at most $\binom{n}{2} = \frac{n^2-n}{2}$ bonds in an outerplanar graph. By Lemma 18 and Lemma 24, the optimal solution to the minimum product cut problem can be found by finding all bonds and minimum product cut among these bonds in polynomial time. The number of the edges in an outerplanar graph is $O(n)$. All bonds and hence, the minimum product cut can be found in $O(n^3)$ time in an outerplanar graph given its embedding.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Linear Programming and Network Flows, 3rd ed.* Prentice-Hall, Upper Saddle River, NJ, 1993.

[2] E. Balas and M. Fischetti. Polyhedral theory for the asymmetric traveling salesman problem. In G. Gutin and A. P. Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 117–168. Kluwer, Dordrecht, 2002.

[3] P. Carstensen. Complexity of some parametric integer and network programming problems. *Mathematical Programming*, 26:64–75, 1983.

[4] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In L. Naish, editor, *Proceedings, Fourteenth International Conference on Logic Programming (ICLP 1997)*, volume 2833, pages 316–330. The MIT Press, 1997.

[5] B. Cherkassky and A. Goldberg. On implementing push-relabel method for the maximum flow problem. Technical report, Department of Computer Science, Stanford University, 1994.

[6] V. Chvátal. Edmonds polytopes and weakly hamiltonian graphs. *Mathematical Programming*, 5:29–40, 1973.

[7] V. Chvátal. Tough graphs and hamiltonian circuits. *Discrete Mathematics*, 5:215–228, 1973.

[8] V. Chvátal. Hamiltonian cycles. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinooy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, pages 403–430. Wiley, New York, 1985.

[9] C. F. Daganzo. The crane scheduling problem. *Transportation Research*, 23B(3):159–175, 1989.

[10] N. Garg and V. V. Vazirani. A polyhedron with all s-t cuts as vertices, and adjacency of cuts. *Mathematical Programming*, 70(1):17–25, 1995.

[11] L. Genc-Kaya and J. N. Hooker. A filter for the circuit constraint. In F. Benhamou, editor, *Principles and Practice of Constraint Programming (CP 2006)*, volume 4204 of *Lecture Notes in Computer Science*, pages 706–710. Springer, 2006.

[12] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[13] N.I.M. Gould and P. L. Toint. A quadratic programming bibliography. *Numerical Analysis Group Internal Report*, 1, 2000.

[14] V. Goyal, L. Genc-Kaya, and R. Ravi. An FPTAS for minimizing the product of two non-negative linear costs. WP 2008-E16.

[15] D. Gusfield. Parametric combinatorial computing and a problem of program module distribution. *J. Assoc. Comput. Mach.*, 30(3):551–563, 1983.

[16] D. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.

[17] J. N. Hooker. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. Wiley, New York, 2000.

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

[19] D. Johnson and C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, Providence, RI, 1993. American Mathematics Society.

[20] M. Jünger, G. Reinelt, and G. Rinaldi. The traveling salesman problem. In M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, editors, *Network Models*, Handbooks in Operations Research and Management Science, pages 225–330. Elsevier, Amsterdam, 1995.

[21] W. Kern and G. Woeginger. Quadratic programming and combinatorial minimum weight product problems. *Mathematical Programming*, 110(3):641–649, 2007.

[22] A. Lim, B. Rodrigues, Fei Xiao, and Yi Zhu. Crane scheduling using tabu search. In *International Conference on Tools with Artificial Intelligence, (ICTAI'02)*, pages 146–153, 2002.

[23] A. Lim, B. Rodrigues, and Zhou Xu. Solving the crane scheduling problem using intelligent search schemes. In *Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 747–751. Springer, 2004.

[24] T. Matsui. NP-hardness of linear multiplicative programming and related problems. *Journal of Global Optimization*, 9(2):113–119, 1996.

[25] N. Megiddo. Combinatorial optimization with rational objective functions. *Math. Oper. Res.*, 4:414–424, 1979.

[26] D. Naddef. Polyhedral theory and branch-and-cut algorithms for the symmetric TSP. In G. Gutin and A. P. Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 29–116. Kluwer, Dordrecht, 2002.

[27] R. I. Peterkofsky and C. F. Daganzo. A branch and bound solution method for the crane scheduling problem. *Transportation Research*, 24B(3):159–172, 1990.

[28] V. Ramachandran. The complexity of minimum cut and maximum flow problems in an acyclic network. *Networks*, 17:387–392, 1987.

[29] R. Ravi, M. V. Marathe, R. Sundaram, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt. Bicriteria network design problems. *Journal of Algorithms*, 28(1):142–171, 1998.

[30] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings, 13th National Conference on Artificial Intelligence (AAAI 1996), Part 1*, pages 209–215. AAAI, 1996.

[31] R. Rodosek and M. Wallace. A generic model and hybrid algorithm for hoist scheduling problems. In *Proc. 4th Int. Conf. on Principles and Practice of Constraint Programming (CP98)*, pages 385–399. Springer, 1998.

[32] J. Shufelt and H. Berliner. Generating hamiltonian circuits without backtracking from errors. *Theoretical Computer Science*, 132:347–375, 1994.

[33] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[34] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[35] D. B. West. *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, NJ.

[36] H. P. Williams and H. Yan. Representations of the all_different predicate of constraint satisfaction in integer programming. *INFORMS Journal on Computing*, 13:96–103, 2001.

[37] Y. Zhu and A. Lim. Crane scheduling with spatial constraints. *Naval Research Logistics*, 51:386–406, 2004.