June 1991 Report No. STAN-CS-91-1366

Thesis

1

Combinatorial Algorithms for Optimization Problems

by

Edith Cohen

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
June 5, 1991

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
Combinatorial Algorithms For Optimization Problems

5. FUNDING NUMBERS

6. AUTHOR(S)
Edith Cohen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Dept. of Computer Science
Stanford University
STANFORD, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ONR
ONR

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N00014-88-K-0166
N00014-91-C-0026

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
UNLIMITED

12b. DISTRIBUTION CODE

13.

Abstract. Linear programming is a very general and widely used framework. In this thesis we consider several combinatorial optimisation problems that can be viewed as classes of linear programming problems with special structure. It is known that polynomial time algorithms exist for the general linear programming problem. It is not known, however, whether any of them are *strongly polynomial*. In addition, it seems that the general problem is inherently sequential. For problems with special structure, our goals are to develop sequential and parallel algorithms that are faster than those known for general linear programming and to determine whether strongly polynomial algorithms exist. (i) We develop a technique that extends the classes of problems known to have strongly polynomial algorithms, or known to be quickly solvable in parallel. This technique is used to obtain a fast parallel algorithm and a strongly polynomial algorithm for detecting cycles in periodic graphs of fixed dimension. We mention additional applications to parametric extensions of problems where the number of parameters is fixed. (ii) We introduce algorithms for solving linear systems where each inequality involves at most two variables. These algorithms improve over the sequential and parallel running times of previous algorithms. These results are combined with additional ideas to yield faster algorithms for some generalised network flow problems.

14. SUBJECT TERMS

15. NUMBER OF PAGES
168

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

COMBINATORIAL ALGORITHMS FOR OPTIMIZATION PROBLEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Edith Cohen
June 1991

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Andrew V. Goldberg
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Nimrod Megiddo

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Serge A. Plotkin

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

Linear programming is a very general and widely used framework. In this thesis we consider several combinatorial optimization problems that can be viewed as classes of linear programming problems with special structure. It is known that polynomial time algorithms exist for the general linear programming problem. It is not known, however, whether any of them are *strongly polynomial* (informally, a polynomial time algorithm is strongly polynomial if the number of arithmetic operations performed is bounded by a polynomial function of the number of variables and inequalities, i.e., is independent of the size of the numbers). In addition, it seems that the general problem is inherently sequential (fast parallel algorithms cannot be obtained). For problems with special structure, our goals are to develop sequential and parallel algorithms that are faster than those known for general linear programming and to determine whether strongly polynomial algorithms exist.

We develop a technique that extends the classes of problems known to have strongly polynomial algorithms, or known to be quickly solvable in parallel. This technique is used to obtain a fast parallel algorithm and a strongly polynomial algorithm for detecting cycles in periodic graphs of fixed dimension. We mention additional applications to parametric extensions of problems where the number of parameters is fixed.

We introduce algorithms for solving linear systems where each inequality involves at most two variables. These algorithms improve over the sequential and parallel running times of previous algorithms. These results are combined with additional ideas to yield faster algorithms for some generalized network flow problems.

To my parents *Amy and Eliahu*	להורי **אמי ואליהו**
my sister *Mirit*	לאחותי **מירית**
and my brother *Menashe*	ולאחי **מנשה**
with all my love	באהבה

Acknowledgements

I would like to thank the three members of my committee: my advisor Andrew Goldberg for suggesting research problems and offering advice, Nimrod Megiddo for closely working with me and guiding me to become an independent researcher, and Serge Plotkin for carefully reading this thesis and sharing numerous interesting ideas.
I would also like to thank Christos Papadimitriou, my advisor during my first year, and Ernst Mayr for early guidance; Leo Guibas and Rajeev Motwani for interesting classes; Don Knuth and Alex Schäffer for their advice; the people of IBM Almaden for giving me the opportunity to spend there fruitful and enjoyable time throughout my PhD studies; and Adam Grove, Tomás Feder, Vwani Roychowdhury, Éva Tardos, Steve Vavasis, and Pete Veinott for beneficial discussions related to my thesis work.
I would like to mention, in random order, Alex Wang, Tom Henzinger, Greg Plaxton, Tomek Radzik, Shoshi Wolf, Inderpal Mumick, Yossi Friedman, Gidi Avrahami, Daphne Koller, Alon Levy, Efi Fogel, Eran Yehudai, Sherry Listgarten, Tim Fernando, Misha Kharitonov, David Cyrluk, Yossi Azar, Moni Naor, Seffi Naor, Rajeev Alur, Evan Cohn, Anil Gangolli, Aaron Goldberg, Ramsey Haddad, Robert Kennedy, Dror Maydan, Steven Phillips, Martin Rinard, Ken Ross, Shaibal Roy, David Salesin, Ashok Subramanian, and Orli Waarts.
For the fantastic period I had at Stanford I am most grateful to my best friend Ling, to the people mentioned above, and to the Sierras, the Pacific, and the California sun.

Last, I am obliged to my parents for encouraging my curiosity, revealing to me the joy and satisfaction of learning and understanding, and giving me the confidence and stubbornness needed to pursue my ideas and goals.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Linear programming (LP) is a very widely studied and commonly used class of optimization problems that encompasses many combinatorial optimization problems. A linear programming problem of n variables and m inequalities consists of a matrix $A \in R^{m \times n}$ and two vectors $b \in R^m$, $c \in R^n$. The goal is to find a vector $x \in R^n$ (an assignment of values to the variables) such that $Ax \leq b$ (x satisfies the inequalities) and $c^T x$ (the value of the *objective function*) is maximized. The algorithms that are most commonly used in practice are variants of the simplex method, due to Dantzig [16]. None of these variants, however, is known to run in polynomial time on all instances. The existence of a polynomial-time algorithm for the general LP problem was in doubt until Khachiyan [40] obtained such an algorithm by modifying the "ellipsoid method," a tool used in nonlinear optimization. A few years later, Karmarkar [37] introduced "interior point methods," which hold the promise of yielding LP algorithms that are both provably polynomial and efficient in practice. Many others followed (see e.g. [36, 54, 60]). Currently, the best worst-case upper bound for the problem is due to Vaidya [60].

Strongly polynomial algorithms: Vaidya's bound, as well as all known bounds on the running times of general LP algorithms, depends not only on n and m but also on the size of the entries. An algorithm for a subset of LP problems is *strongly polynomial* (SP) if (i) the number of arithmetic operations is bounded by a polynomial in m and n and (ii) the algorithm does not generate numbers whose size (binary representation) is larger than some polynomial function of the input size. An important open problem is whether general LP has an SP algorithm.

Asymptotic notation: We use the standard asymptotic notation to measure and compare the resource use (running time, space, processors) of algorithms. Suppose g is a monotone increasing function into the positive reals and f is a positive function defined on the same domain as g. $f = O(g)$ (resp., $f = o(g)$, $f = \Omega(g)$, $f = \omega(g)$, $f = \theta(g)$) if $\exists c > 0$ such that $\overline{\lim} f/g \leq c$ (resp., $\overline{\lim} f/g = 0$, $f/g > c$ infinitely often, $\overline{\lim} f/g = \infty$, $f = O(g)$ and $f = \Omega(g)$). We also use the "soft bound" notation $\tilde{O}(f) \equiv O(f \text{ polylog } f)$. The common notation for the class of problems solvable in polynomial time is $\mathcal{P}$.

Randomized Algorithms: *Randomized* algorithms make decisions according to the outcomes of flipping fair coins. Two types of randomized algorithms are mentioned in the literature, *Monte Carlo* algorithms and *Las Vegas* algorithms. An asymptotic running time of $O(f)$ is interpreted as follows: (i) Deterministic algorithms are guaranteed to terminate in $O(f)$ time. (ii) Monte Carlo algorithms are guaranteed to terminate in $O(f)$ time and give an answer that is correct with some constant probability $p > 2/3$. (iii) Las Vegas algorithms terminate in $O(f)$ expected time and are guaranteed to return a correct answer. The randomized algorithms discussed in this thesis are Las Vegas type.

Parallel Algorithms: Within the class $\mathcal{P}$, we want to identify problems for which we can benefit by using a parallel computer, where many processors are available and able to work concurrently. The PRAM (parallel random access machine) [24] is the abstract parallel machine model that takes the role played by the Turing machine or RAM in sequential computation. The PRAM is a shared memory multiprocessor machine. Several types of PRAM'S are considered in the literature, according to whether they allow concurrent read and write operations from or to the same memory location. The CREW model, for example, allows concurrent reads but only exclusive writes. The CRCW model allows both concurrent reads and concurrent writes. A parallel implementation of a particular algorithm has *optimal speedup* if the product of processors and time is of the order of the sequential running time. A complexity class that somewhat captures the notion of efficient parallel computation is $\mathcal{NC}$. The class $\mathcal{NC}$ was introduced by Pippenger [53] and is robust in the sense that it applies to many parallel machine models. A problem is in $\mathcal{NC}$ if it can be solved in polylogarithmic time using a polynomial number of processors. Obviously, $\mathcal{NC} \subset \mathcal{P}$. A fundamental open problem is whether $\mathcal{P} = \mathcal{NC}$. A problem is P-complete if the existence of an $\mathcal{NC}$ algorithm for it implies $\mathcal{P} = \mathcal{NC}$, or equivalently, if every problem in $\mathcal{P}$ can be reduced to it using logarithmic space. $\mathcal{P}$-complete problems are the "hardest" problems in $\mathcal{P}$. General LP was shown to be $\mathcal{P}$-complete by Dobkin, Lipton, and Rice [18]. The common belief is that $\mathcal{P} \neq \mathcal{NC}$. Hence, $\mathcal{P}$-complete problems in general, and LP in particular, are viewed to be inherently sequential.

Examples of classes of LP problems with special structure: When a class of LP problems with special structure is considered, one might try to find either an SP (resp., $\mathcal{NC}$) algorithm for this class or an SP (resp., logspace) reduction of general LP problems to problems of this class (see [10]). Such a reduction asserts that proving that this class is SP (resp., in $\mathcal{NC}$) is as hard as proving the same for general LP. In the thesis we consider LP problems of special structure. We are concerned both with the quantitative problem of improving the parallel and sequential time bounds and with the qualitative question of strong polynomiality. We give examples of classes of LP problems that are "easier" than general LP:

- The maximum flow problem has an SP algorithm due to Edmonds and Karp [21] and Dinic [17] (better bounds are given in [28, 52]).

- The more general minimum-cost circulation problem was shown by Tardos [58] to have an SP algorithm (see, e.g., [2, 27] for better bounds).
- Tardos [59] generalized her min-cost circulation SP algorithm to LP instances where the entries in the matrix A are bounded by a polynomial in $m + n$ (b and c can still be general).
- Megiddo [47] gave a linear time algorithm for LP problems with a fixed number of variables.
- Megiddo [46] gave an SP algorithm for solving linear systems with at most two variables per inequality. Faster sequential and parallel algorithms are presented in Chapter 5.

Generalized circulation (see [42]) is an interesting network flow problem. It is not known to have an SP algorithm and there is no known SP reduction of general LP problems to it. We present some partial results in Chapter 6.

1.2 Outline of the thesis

The results included in the thesis are joint work with Nimrod Megiddo. The thesis contains two disjoint sets of results. The first set consists of Chapters 2, 3, and 4 (an extended abstract appeared in [6], see also [7, 9]). The second set consists of Chapters 5 and 6 (an extended abstract appeared in [11]). Chapter 7 contains a conclusion. Each chapter is more or less independent of the other chapters.

In Chapter 2 we present an algorithm to detect the existence of directed cycles in periodic graphs. A d-dimensional periodic graph is an infinite digraph, where isomorphic finite sets of vertices are associated with the points of the d-dimensional grid. Periodic graphs have a very regular structure; the edges are such that the periodic graph "looks the same" from any grid point. A periodic graph can be represented by a finite directed graph where d-dimensional integer vectors are associated with the edges. When resolving graph properties of a periodic graph, we would like to find algorithms whose running time is polynomial in the size of the directed graph representing it. Periodic graphs in general, and the cycle detection problem in particular, were studied in previous papers [34, 39, 41, 51]. Previous algorithms, however, solved the problem by reducing it to polynomially many LP problems (see [39, 41]). These results left open the existence of strongly polynomial or $\mathcal{NC}$ algorithms for the problem. The algorithm presented in Chapter 2 is a strongly polynomial and $\mathcal{NC}$ algorithm, when the dimension d is fixed. To complement the result we also show that when d is part of the input, the existence of a strongly polynomial time or $\mathcal{NC}$ algorithm for the problem implies the existence of such an algorithm for general LP problems. We also show how the same algorithm can be applied to compute strongly connected components of periodic graphs and to schedule the computation of systems of uniform recurrences.

The cycle detection algorithm is based on reducing the problem to solving instances of the parametric minimum cycle problem with d parameters. The latter problem, which is interesting in its own right, can be solved by a sequence of LP problems. In Chapter 3 we present a method that allows us to perform the computation in SP time and in $\mathcal{NC}$. The parametric minimum cycle problem is defined as follows. Consider a digraph where weights are associated with the edges. The weights are linear functions of d variables ("parameters"). Each set of values for the parameters corresponds to a set of scalar weights associated with the edges of the graph. The goal is, roughly, to find the set of values for which the weight of the minimum-weight cycle is maximized.

The purpose of Chapter 4 is to present the algorithm of Chapter 3 as a general method to achieve strongly polynomial bounds. The scheme used to maximize the function that maps sets of parameter values to the value of the minimum-weight cycle can actually be applied to minimize (resp., maximize) large family of convex (resp., concave) functions. The minimization requires a number of operations that is polynomial in the number of operations needed to evaluate the function (when the dimension of the domain is fixed). In Chapter 4 we omit many of the details specific to the parametric minimum cycle problem. To allow for independent reading many definitions and statements are repeated. For some of the proofs, however, the reader is referred to the appropriate place in Chapter 3.

In Chapter 5 we present faster algorithms to solve linear systems of inequalities where at most two variables appear in each inequality (TVPI systems). We give a deterministic $\tilde{O}(mn^2)$ time algorithm and a randomized $\tilde{O}(n^3 + mn)$ expected time algorithm, where m is the number of inequalities and n is the number of variables. In parallel, these algorithms run in $\tilde{O}(n)$ time with optimal speedup. The previously best known algorithm due to Megiddo runs in $O(mn^3 \log m)$ time sequentially, and $O(n^3 \log m)$ time with optimal speedup in parallel [46].

Chapter 6 is concerned with generalized network flow problems. In a generalized network, each edge $e = (u, v)$ has a positive "flow multiplier" a_e associated with it. The interpretation is that if a flow of x_e enters the edge at node u, then a flow of $a_e x_e$ exits the edge at v. We present algorithms for generalized network flow problems that utilize the results of Chapter 5.

The uncapacitated generalized transshipment problem (UGT) is defined on a generalized network where demands and supplies (real numbers) are associated with the vertices and costs (real numbers) are associated with the edges. The goal is to find a flow such that the excess or deficit at each vertex equals the desired value of the supply or demand, and the sum over the edges of the product of the cost and the flow is minimized. Adler and Cosares [1] reduced the restricted uncapacitated generalized transshipment problem, where only demand nodes are present, to solving a single TVPI system. Therefore, the algorithms of Chapter 5.1.1 result in a faster algorithm for restricted UGT.

Generalized circulation is defined on a generalized network with demands at the nodes and capacity constraints on the edges (i.e., upper bounds on the amount of flow). The goal is to find a flow such that the flow excesses at the nodes are proportional to the demands and maximized. We present a new algorithm that solves the capacitated generalized flow problem by iteratively solving instances of UGT. The algorithm can be used to find an optimal flow or an approximation. When used to find a constant factor approximation, the algorithm yields a bound that is not only more efficient than previous algorithms but also SP. This is the first SP approximation algorithm for generalized circulation; the existence of this approximation algorithm is particularly interesting since it is not known whether the problem has an SP algorithm.

1.3 Notation

We use boldface notation for vectors and matrices. Suppose $\boldsymbol{U}$ is a matrix and $\boldsymbol{d}$ is a column vector, denote by:

d_i	- the i'th entry in the vector $\boldsymbol{d}$,
U_{ij}	- the ij entry in the matrix $\boldsymbol{U}$,
$\boldsymbol{U}_{i\bullet}$	- the i'th row of the matrix $\boldsymbol{U}$,
$\boldsymbol{U}_{\bullet j}$	- the j'th column of the matrix $\boldsymbol{U}$,
$\boldsymbol{U}^T$	- the transposed matrix $(\forall i, j, U_{ij} = U^T_{ji})$, and
$\boldsymbol{d}^T$	- the corresponding row vector.

We use the following special vectors:

$\mathbf{e}$	is the vector with all entries 1,
$\mathbf{0}$	is the vector will all entries 0, and
$\boldsymbol{e}^i$	is such that $e^i_i = 1$, and for $j \neq i$, $e^i_j = 0$.

Suppose A_i $(1 \leq i \leq k)$ are sets of elements. Denote by:

$\lvert A_i \rvert$	the number of elements in the set A_i.
$A_i \times A_j$	the cross product of A_i and A_j, that is, the set of ordered pairs (a_i, a_j) where $a_i \in A_i$, $a_j \in A_j$.
$\times_{1 \leq i \leq k} A_i$	the cross product of the sets $A_1, \ldots, A_k$, that is, the set of k-tuples $(a_1, \ldots, a_k)$ where $a_i \in A_i$ $(1 \leq i \leq k)$.
A_i^ℓ	the set of all ℓ-tuples of elements of A_i.
$A_i^{m \times n}$	the set of all $m \times n$ matrices whose entries are elements of A_i.

We use the notation R, R_+, Q, Z, and N, for the sets of real numbers, positive real numbers, rational numbers, integers, and natural numbers, respectively. Hence,

R^k	denotes the k-dimensional Euclidean space,
Q^k	denotes the k-dimensional linear space over the rationals, and
Z^k	denotes the k-dimensional integer grid.

Consider two sets $F \subset R^n$ and $F' \subset R^n$.
$F + F' = \{y \in R^n | y = x + x',$ where $x \in F, x' \in F'\}$ is the *sum* of F and F'.
When every $y \in F + F'$ can be uniquely represented as $y = x + x'$ where $x \in F$, $x' \in F'$, we refer to $F \oplus F' \equiv F + F'$ as the *direct sum* of F and F'.
Denote the projection of F on the coordinates $J \subset \{1, \ldots, n\}$ by $F_J \subset R^{|J|}$. Also,

- F is *convex* iff for every $x \in F$, $y \in F$ and $0 \leq \alpha \leq 1$, we have $\alpha x + (1-\alpha)y \in F$.
- F is a *linear subspace* iff $F = \{x \in R^n | Ax = 0\}$ for some matrix $A \in R^{m \times n}$.
- For a linear subspace F, denote the orthogonal complement of F by $F^{\perp} = \{x \in R^n | \forall y \in F, x^T y = 0\}$.
- F is a *flat* iff $F = \{x \in R^n | Ax = b\}$ for some matrix $A \in R^{m \times n}$ and vector $b \in R^m$. The subspace parallel to the flat F is $\{x | Ax = 0\}$.
- F is a *cone* iff for all $x, y \in F$ and $\alpha \geq 0$, $\beta \geq 0$, $\alpha x + \beta y \in F$. A cone is *pointed* iff it does not contain a linear subspace.
- The *lineality space* of a cone F is the largest subspace L such that for all $x \in F$, $x + L \subset F$.

We use the notation:

aff F	for the affine hull of F, that is, the smallest flat which contains F,
lin F	for the subspace spanned by F,
interior F	for the interior of F, and
rel int F	for the relative interior (interior relative to aff F).

We denote by $g : A \rightarrow B$ a function from a domain A into B. For a subset $H \subset A$ of the domain, denote by $g|H$ the restriction of g to H, that is, the function $g|H : H \rightarrow B$ such that $\forall a \in H, g(a) = g|H(a)$.

A function $g : R^n \rightarrow R$ is *convex* (resp., *concave*) if for all $x \in R^n$, $y \in R^n$, and $0 \leq \alpha \leq 1$, $g(\alpha x + (1-\alpha)y) \leq \alpha g(x) + (1-\alpha)g(y)$ (resp., $g(\alpha x + (1-\alpha)y) \geq \alpha g(x) + (1-\alpha)g(y)$).

By $G = (V, E)$ we denote a directed graph, where V is the set of vertices and E is the set of edges. We use the convention $|V| = n$, $|E| = m$. By $G = (V, E, \boldsymbol{f})$ we denote a directed graph with a "weight" function $\boldsymbol{f} : E$ which associates weights with the edges. For a vertex $v \in V$, let $\mathrm{in}(v) \subset V$ and $\mathrm{out}(v) \subset V$, respectively, denote the sets of edges entering and leaving v.

Suppose $\boldsymbol{x}^T = (x_1, \ldots, x_n)$ are indeterminate.

- $g(x_1, \ldots, x_n)$ is a *linear function* if there exist real numbers $c_0, \ldots, c_n$ such that $g(x_1, \ldots, x_n) = c_0 + \sum_{i=1}^{n} c_i x_i$.
- $\sum_{i=1}^{n} c_i x_i = c_0$ is a *linear equation.*
- $\sum_{i=1}^{n} c_i x_i \geq c_0$ and $\sum_{i=1}^{n} c_i x_i \leq c_0$ are *linear inequalities.*
- Both linear equations and linear inequalities are referred to as *linear constraints.*
- A *linear system* is a set of linear constraints. Set of m inequalities is represented by a matrix $\boldsymbol{A} \in R^{m \times n}$ and a vector $\boldsymbol{b} \in R^m$, as $\boldsymbol{Ax} \leq \boldsymbol{b}$. A vector is *feasible* if it satisfies all the constraints.
- A *linear programming problem* consists of a linear system and a linear *objective function* given by a vector $\boldsymbol{c} \in R^n$. The goal is to maximize (or minimize) $\boldsymbol{c}^T\boldsymbol{x}$ subject to the inequalities $\boldsymbol{Ax} \leq \boldsymbol{b}$.

Chapter 2

Detecting cycles in periodic graphs

This chapter is concerned with the problem of recognizing, in a graph with rational vector-weights associated with the edges, the existence of a cycle whose total weight is the zero vector. This problem is known to be equivalent to the problem of recognizing the existence of cycles in periodic (dynamic) graphs and to the validity of systems of recursive formulas. It was previously conjectured that combinatorial algorithms exist for the cases of two- and three-dimensional vector-weights. This chapter gives strongly polynomial algorithms for any fixed dimension. Moreover, these algorithms also establish membership in the class $\mathcal{NC}$. On the other hand, it is shown that when the dimension of the weights is not fixed, the problem is equivalent to the general linear programming problem under strongly polynomial and logspace reductions. The algorithms discussed here are based on reducing the problem to solving instances of the parametric minimum cycle problem. When the dimension of the vector-weights is fixed, the problem can be solved within the same time bound of solving an instance of the parametric minimum cycle problem on the same graph. The latter problem is defined in Chapter 3, where we present $\mathcal{NC}$ and strongly polynomial algorithm for it when the number of parameters is fixed. Earlier versions of the results presented in this chapter appeared in [6, 9].

2.1 Introduction

This chapter is concerned with the following problem:

Problem 2.1.1 Given is a digraph $G = (V, E, f)$ where $f : E \rightarrow R^d$ associates with each edge of G a d-vector of rational numbers. Determine whether G contains a zero-cycle, i.e., a cycle whose edge vectors sum to the zero vector.

Problem 2.1.1 is important since it has been shown by Iwano and Steiglitz [34] that an infinite "periodic graph" has a cycle if and only if the "dependence graph" which generates it has a zero-cycle. The periodic graph is well-defined when the weights are integral. Given a graph $G = (V, E)$ with integral vector-weights $c_{ij} \in R^d$ (called the *dependence graph*), the corresponding *periodic graph* is generated as follows. Place a copy of the vertex set of G at each point of the integral lattice in R^d. For every lattice point z and for every edge $(i, j) \in E$, connect the copy of vertex i that is located at z with the copy of vertex j that is located at $z + c_{ij}$. The problem on this infinite periodic graph is to identify a cycle or conclude that none exists. See figures 2.1 and 2.2 for examples of dependence graphs with corresponding periodic graphs.

The problem was first introduced in a paper by Karp, Miller and Winograd [39], where it was raised in the context of recursive definitions. They gave an algorithm which amounts to solving polynomially many linear programs. They stated the problem of validity of recursive definitions as detecting a cycle in a periodic graph. Consider n functions $F_1, \ldots, F_n$ on the d-dimensional integral lattice defined by

$$F_i(z) = \psi_i(F_1(z - c_{i1}), \ldots, F_n(z - c_{in})) ,$$

(the c_{ij}'s are integral vectors) where the ψ_i's are specified (assume for simplicity the boundary conditions $F(z) = 0$ for $z \notin R^n_+$). The corresponding dependence graph has n vertices $v_1, \ldots, v_n$, and edges (v_i, v_j) of weight c_{ij} if $F_i(z)$ depends on $F_j(z - c_{ij})$. In order for the functions to be well defined it is necessary and sufficient that there will be no cycle whose total vector weight is nonnegative. The problem of detecting a

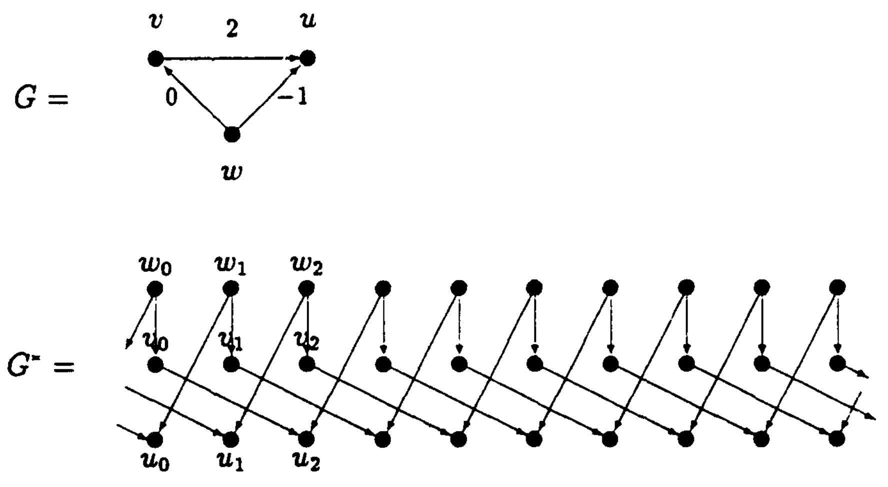
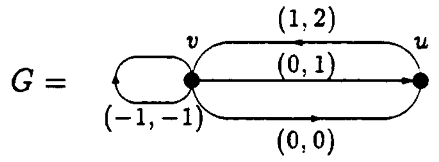


Figure 2.1: A one-dimensional dependence graph G and the periodic graph G^{∞}

nonnegative cycle can be reduced to the problem of detecting a zero cycle by adding at each vertex i, d loops with weights $(-1,0,\ldots,0)$, $(0,-1,\ldots,0)$, $\ldots$, $(0,\ldots,0,-1)$. As an example, consider the recurrence relation $F(z) = F(z-1) + F(z-2)$. The corresponding dependence graph consists of a single vertex and two loops of weights -1 and -2.

The problem was re-introduced by Iwano and Steiglitz [34] following Orlin [51]. Orlin studied properties of one-dimensional periodic graphs which included computing strongly connected components. Kosaraju and Sullivan [41] presented a polynomial-time algorithm. The time complexity of the latter is $O(Zn\log n)$, where Z is the complexity of a certain linear programming problem with $2m$ variables and $m+n+k$ constraints ($n = |V|$, $m = |E|$). They also presented $O(Z)$ algorithms for the cases $d = 2,3$.

It was conjectured in [41] that combinatorial algorithms exist for the cases $d = 2,3$. Although the notion of a combinatorial algorithm is not well-defined, we believe we have confirmed this conjecture and have in fact proven much more than what was expected. We show that not only for $d = 2,3$, but also for any fixed d, the problem can be solved in strongly polynomial time, and indeed by "combinatorial"



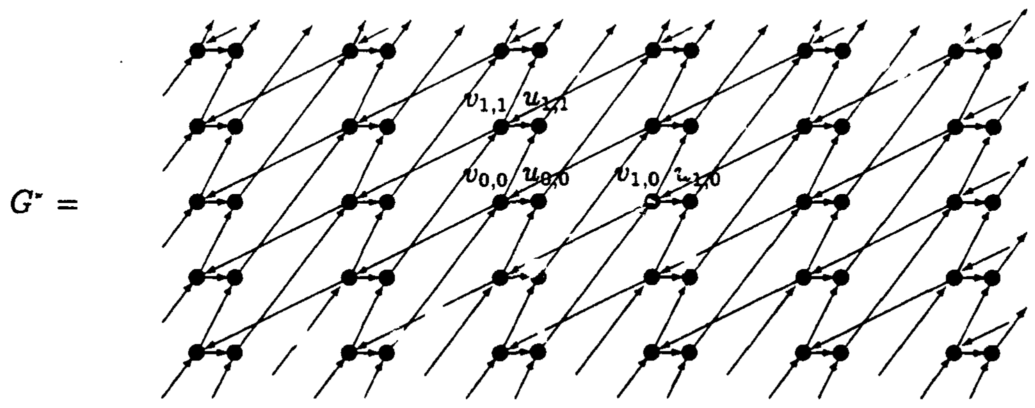


Figure 2.2: A two-dimensional dependence graph G and the periodic graph G^∞

algorithms. Furthermore, our algorithms can be implemented on a parallel machine so that membership in the class $\mathcal{NC}$ is established for any fixed d. To complement these results, we also show that the general problem (where d is considered part of the input) is as hard as the general linear programming problem in a sense as follows. We show that any linear programming problem can be reduced in strongly polynomial time and logspace to our problem with general d.

Consider an instance of Problem 2.1.1 where $m = |E|$ is the number of edges and $n = |v|$ is the number of vertices. We show that when d is fixed the problem can be solved within the following bounds:

i. $O(\log^{2d} n + \log^{d} m)$ parallel time on $O(n^3 + m)$ processors.

ii. $O(m(\log^{2d} n + \log^{d} m))$ sequential time, when $m = \Omega(n^3 \log n)$.

iii. $O((n^3 + m)\log^{2d} n)$ sequential time, when $m = O(n^3 \log n)$ and $m = \Omega(n^2)$.

iv. $O(n^3 \log^{2(d-2)} n + nm \log^{2(d-1)} n)$ sequential time, when $m = O(n^2)$.

The constant factors hidden in the above bounds are of the order of $O(3^{d^2})$, and arise from the multi-dimensional search algorithm [5, 19, 47]. The dominating factor in the time complexity arises from solving $O(d)$ instances of the parametric minimum cycle problem with $d-1$ parameters, m edges and n nodes (see Chapter 3).

It is worth mentioning that other properties of periodic graphs were considered in the literature. The computation of strongly connected components [51, 9] (see Section 2.6.2), scheduling the computation of systems of recursive definitions [9, 39, 56] (see Section 2.6.1), planarity testing [33], computing connected components, recognizing bipartiteness [12, 35, 51], and computing a minimum average cost spanning tree [12, 51].

In Section 2.2 we give some necessary definitions. In Section 2.3 we give an overview of the basic ideas underlying our algorithms. In Section 2.4 we describe a strongly polynomial algorithm for detecting zero-cycles. This algorithm is stated in terms of solving instances of a parametric version of the minimum cycle problem. The latter problem and a strongly polynomial algorithm for it are introduced in Chapter 3. Section 2.5 contains some necessary geometric lemmas. Section 2.6 discusses two other problems on periodic graphs for which the cycle detection algorithm is applicable. One problem is computing the strongly connected components, the other is scheduling the computation of systems of uniform recurrences which are modeled by periodic graphs. Concluding remarks are given in Section 2.7.

2.2 Preliminaries

Definition 2.2.1 Given a graph $G = (V, E)$, a *circulation* $\boldsymbol{x} = (x_{ij})$ $((i,j) \in E)$ is a solution of the system:

$$\sum_j (x_{ij} - x_{ji}) = 0 \qquad (i = 1, \ldots, n)$$

$$\boldsymbol{x} \geq \mathbf{0}\ .$$

Let $E(x)$ denote the set of *active* edges, i.e., edges (i, j) with $x_{ij} > 0$. Vertices incident on active edges are said to be *active in x*. If the active edges form a connected subgraph of G, then we say that the circulation x is *connected.* If these edges form a simple cycle, then we say that x is a *simple cycle*, and with no ambiguity we continue to talk about the set of active vertices as a *simple cycle.*

Remark 2.2.2 Every circulation x is a sum of connected circulations, corresponding to the decomposition of $E(x)$ into strongly connected components. Moreover, it is also well known (and easy to see) that every circulation can be represented as a sum of simple cycles. If a connected circulation $x = (x_{ij})$ consists of rational numbers, then it is proportional to an integral circulation. A connected integral circulation can be represented by a cycle $(u_0, u_1, \ldots, u_q = u_0)$ $((u_{i-1}, u_i) \in E)$, not necessarily simple, where x_{ij} is interpreted as the number of times the edge (i, j) is traversed throughout the cycle. It is easy to construct irrational circulations that cannot be interpreted this way.

Definition 2.2.3 Given vector weights $c_{ij} = (c^1_{ij}, \ldots, c^k_{ij})^T$ $((i, j) \in E)$ (i.e., using the notation of Problem 2.1.1, $c_{ij} = f(e)$ where $e = (i, j)$), a circulation $x = (x_{ij})$ is called a *zero-circulation* if it satisfies the vector equation $\sum_{i,j} c_{ij} x_{ij} = \mathbf{0}$. An integral connected nontrivial zero-circulation is called a *zero-cycle.*

2.3 An overview

We first present an informal overview of the basic ideas involved in the zero-cycle detection algorithm.

Suppose $G = (V, E, f)$ contains a vector zero-cycle C, i.e., the sum of the vector weights c_{ij} around C is equal to the zero vector. Obviously, for any $\lambda \in R^d$, the sum of the scalar weights $\lambda^T c_{ij}$ around C is zero. It follows that for every $\lambda \in R^d$, the weight of the minimum cycle relative to the scalars $\lambda^T c_{ij}$ is nonpositive. In other words, if there exists a $\lambda \in R^d$ such that all the cycles are positive relative to $\lambda^T c_{ij}$, then this $\boldsymbol{\lambda}$ certifies that there are no zero-cycles. On the other hand, it can be shown that if for every $\boldsymbol{\lambda} \neq \mathbf{0}$ there exists a negative cycle, then there exists a vector zero-cycle.

The observation of the preceding paragraph suggests that one might first attempt to find a $\boldsymbol{\lambda}$ for which all the cycles are positive relative to the weights $\boldsymbol{\lambda}^T \boldsymbol{c}_{ij}$. In other words, we wish to maximize over $\boldsymbol{\lambda}$ the weight of the minimum cycle relative to the scalar weights $\boldsymbol{\lambda}^T \boldsymbol{c}_{ij}$.

This task can be viewed as a parameterized extension of the well-known problem of detecting the existence of a negative cycle or finding a minimum weight cycle in a graph with scalar weights. The latter scalar weights problem can be viewed as asking for an evaluation of a function at a given $\boldsymbol{\lambda}$, which can be solved by running an all pairs shortest paths algorithm. The search for $\boldsymbol{\lambda}$ as above can be formulated as an optimization problem over the $\boldsymbol{\lambda}$-space, where one seeks to maximize the function of the minimum weight of any cycle relative to the $\boldsymbol{\lambda}^T \boldsymbol{c}_{ij}$'s. However, there is a certain difficulty with this approach since the minimum is not well-defined when there are negative cycles. Note that we do not require the cycle to be simple, since the problem of finding a minimum simple cycle is $\mathcal{NP}$-hard. However, we can instead consider one of the following quantities: (i) the minimum cycle-mean, i.e., the minimum of the average weight per edge of the cycle, or (ii) the minimum of the total weight of cycles (not necessarily simple) consisting of at most n edges. It is easy to see that the sign of the minimum cycle-mean (which is the same as the sign of the quantity defined in (ii)) distinguishes the following three cases: I. there exists a negative cycle, II. there exists a zero cycle but no negative cycles, and III. all the cycles are positive.

If an algorithm for either (i) or (ii) of the preceding paragraph is given, which uses only additions, comparisons and multiplications by constants, then such an algorithm can be "lifted" to solve the optimization problem. Very roughly, the basic idea (which is explained in [44, 45, 46]) is to run the given algorithm simultaneously on a continuum of values of $\boldsymbol{\lambda}$, while repeatedly restricting the set of these values, until the optimum is found. Another interpretation of the lifted algorithm is that it operates on linear forms rather than constants. When the lifted algorithm needs to compare two linear forms, it first computes a hyperplane which cuts the space into two halfspaces, such that the outcome of the comparison is uniform throughout each of them. The algorithm then consults an "oracle" (whose details are given later) for selecting the correct halfspace, and moves on. The lifted algorithm maintains a polyhedron P which is the intersection of the correct halfspaces.

As noted above, if a vector $\boldsymbol{\lambda}$ is found such that all the cycles are positive then we are done. Otherwise, the lifted algorithm concludes that $\boldsymbol{\lambda} = \mathbf{0}$ is an optimal solution, i.e., for every $\boldsymbol{\lambda}$ there exists a nonpositive cycle, so the choice of $\boldsymbol{\lambda} = \mathbf{0}$ maximizes the weight of a minimum cycle. However, the zero vector itself does not convey enough information. Nonetheless, the algorithm actually computes a vector $\bar{\boldsymbol{\lambda}} \neq \mathbf{0}$ (called a separating vector) in the relative interior of the set of optima[1], along with a "certificate" of optimality. The certificate consists of vector circulation values $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_r$. These values span in nonnegative linear combinations a suitable linear space, proving that there is no direction to move so that the minimum cycle becomes positive. This "certificate" is used to actually find a zero-cycle when the algorithm decides that one exists. The scalar weights $\bar{\boldsymbol{\lambda}}^T \boldsymbol{c}_{ij}$ then induce a decomposition of the graph, where two vertices are in the same component if they belong to the same scalar zero-cycle. It is then shown that a vector zero-cycle exists in the given graph if and only if such a cycle exists in one of the components. Also, if there is only one component (and the graph has more than one vertex) then there exists a zero-cycle. These observations suggest an algorithm which iteratively computes a separating vector, decomposes the graph accordingly, and works on the components independently. The depth of the decomposition tree is bounded by the dimension of the weights.

The part we have so far left open is the "oracle" which recognizes the correct halfspace. It turns out that, as in [47], the oracle can be implemented by recursive calls to the same algorithm in a lower dimension. This will be explained later in the chapter.

We have outlined the general framework for establishing the qualitative result of

[1]There is also the possibility that the zero vector is the only optimal solution, so there is no separating vector. However, in this case, assuming strong connectivity of the graph, it can be shown that a zero-cycle exists.

strongly polynomial time bounds for any fixed dimension. However, to get more efficient algorithms and to establish membership in $\mathcal{NC}$, we perform multi-dimensional searches as in [5, 19, 47]. By doing so we reduce the number of calls to an "oracle" algorithm which actually need to be performed, to a polylog in the number of decisions. The design can be viewed as an integration of the techniques of [45] and [47] (and the further improvements of [5, 19]).

2.4 Detecting zero-cycles

In this section we develop an algorithm which decides the existence of a zero-cycle in the vector-weighted graph $G = (V, E, \boldsymbol{f})$, $\boldsymbol{f} : E \rightarrow Z^d$. If a zero-cycle exists in G, we find an explicit one. The algorithm introduced in this section uses as a subroutine the parametric minimum cycle algorithm of Chapter 3.

Proposition 2.4.1 *A graph $G = (V, E, \boldsymbol{f})$ with vector weights (see Problem 2.1.1) has a zero-cycle (see Definition 2.2.3) if and only if it has a connected zero-circulation.*

Proof: Note that if there exists a connected zero-circulation then there exists a *rational* connected one. Hence, there exists an integral connected zero-circulation which is equivalent to a zero-cycle (see Remark 2.2.2). ∎

Definition 2.4.2 Given a vector-weighted graph $G = (V, E, \boldsymbol{f})$, we use the following definitions and notation:

i. Let $\mathcal{K}$ denote the cone of vectors $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_d)^T$ for which the scalar-weighted graph $(V, E, \boldsymbol{f}^T\boldsymbol{\lambda})$ has no negative cycles.

ii. A nonzero vector $\boldsymbol{\lambda} \in \operatorname{rel\,int} \mathcal{K}$ (the relative interior of $\mathcal{K}$) is called a *separating vector* for G.

iii. A separating vector $\boldsymbol{\lambda}$ for which the scalar-weighted graph $(V, E, \boldsymbol{f}^T\boldsymbol{\lambda})$ has only positive cycles is called a *witness* for G.

A witness proves the nonexistence of nontrivial zero-circulations. Although for this purpose the vector does not have to be in relint $\mathcal{K}$, we add this as a requirement which is helpful in the recursion.

Remark 2.4.3 The cone $\mathcal{K}$ can be described as the projection on the $\boldsymbol{\lambda}$-space (R^d) of a cone in R^{n+d} (the space of $(\pi_1, \ldots, \pi_n, \boldsymbol{\lambda})$) which is characterized by the inequalities:

$$\pi_i - \pi_j + \boldsymbol{\lambda}^T \mathbf{c}_{ij} \geq 0 \quad ((i,j) \in E) .$$

Note that the system of inequalities above is the linear programming dual of the zero-circulation problem.

Definition 2.4.4

i. Given $G = (V, E, \boldsymbol{f})$, denote by CIRC($G$) the set of all circulation values $\mathbf{c} = \sum_{i,j} \mathbf{c}_{ij} x_{ij}$ (where $x = (x_{ij})$ is a circulation in G).

ii. Given a separating vector $\boldsymbol{\lambda} \neq \mathbf{0}$ (i.e., $\boldsymbol{\lambda} \in$ relint $\mathcal{K}$), denote by ORTH($G, \boldsymbol{\lambda}$) the set of vectors $\mathbf{c} \in$ CIRC(G) which are orthogonal to $\boldsymbol{\lambda}$.

Note that CIRC(G) is a convex polyhedral cone.

Proposition 2.4.5 *The set $\mathcal{K}$ is precisely the set of vectors $\boldsymbol{\lambda}$ such that $\boldsymbol{\lambda}^T \mathbf{c} \geq 0$ for all $\mathbf{c} \in$ CIRC(G).*

Proof: For any circulation x and any set of scalars π_i,

$$\sum_{i,j} (\pi_i - \pi_j) x_{ij} = 0 .$$

If the (vector) value of x is $\mathbf{c}$, then

$$\boldsymbol{\lambda}^T \mathbf{c} = \sum_{i,j} (\boldsymbol{\lambda}^T \mathbf{c}_{ij}) x_{ij} ,$$

By Remark 2.4.3, if $\boldsymbol{\lambda} \in \mathcal{K}$ then $\boldsymbol{\lambda}^T \mathbf{c} \geq 0$. Conversely, if $\boldsymbol{\lambda}^T \mathbf{c} \geq 0$ for all $\mathbf{c} \in$ CIRC(G), then obviously there are no negative cycles in $(V, E, \boldsymbol{f}^T \boldsymbol{\lambda})$, so $\boldsymbol{\lambda} \in \mathcal{K}$. ∎

Theorem 2.4.6

i. ORTH(G,λ) *is independent of* λ, *and hence will be denoted by* ORTH(G). *In fact,* ORTH(G) *is the lineality space of* CIRC(G). *(In case* $\mathcal{K} = \{0\}$, *define* ORTH(G) *to be the entire* R^d.*)*

ii. ORTH$(G) = (\text{lin}\,\mathcal{K})^{\perp}$, *that is, the orthogonal complement of the linear subspace spanned by* $\mathcal{K}$ *(hence it is a linear subspace).*

Proof: The proof is based on a geometric analysis which is given in Section 2.5. ∎

The zero-cycle detection algorithm partitions the graph recursively into node disjoint subgraphs. The tree structure defined by this partitioning process, with subgraphs as nodes, is referred to as the ***decomposition tree*** of the graph G. In this partition, the subgraphs are the connected components of a "maximal" (in the sense of the number of active edges) zero-circulation. This definition implies that a zero-cycle exists in G if and only if a zero-cycle exists at least in one of the subgraphs which G is partitioned into. If a subgraph is not partitioned any further, it is a "leaf" of the decomposition tree, and for this subgraph the algorithm determines the existence of a zero-cycle directly. In [39] and [41] this partition is computed by solving a set of linear programming problems in order to decide for each edge whether or not it is active in any zero-circulation in G. The subgraphs are the connected components induced by the active edges. In this chapter, the computation of the partition is done differently by an algorithm that gives strongly polynomial time bounds.

For a given graph $G = (V, E, \boldsymbol{f})$ the algorithm first tries to find a witness (if a witness is found a zero-cycle does not exist and we stop). In case a witness does not exist, a separating vector is computed. The computation of a witness or a separating vector is done by using the parametric minimum cycle algorithm developed in Chapter 3. The algorithm then proceeds to compute a partition of G using the separating vector found in the previous step. If the partition has only one subgraph, it is shown that a zero-cycle exists in G; otherwise, the algorithm proceeds recursively on the subgraphs. Note that a witness or a separating vector can be computed by solving linear programming problems. The difficulty is to find a strongly polynomial time solution.

In the rest of this section we first discuss the two subroutines used by the algorithm and then proceed to the algorithm itself (Subsection 2.4.3). The first subroutine is the computation of a witness or a separating vector (Subsection 2.4.1). The second (Subsection 2.4.2) is the partitioning of the graph when a separating vector is given.

2.4.1 Computing a witness or a separating vector

Problem 2.4.7 Given is a graph $G = (V, E, \boldsymbol{f})$. Find a witness for G (see Definition 2.4.2) if one exists; otherwise, find a separating vector $\boldsymbol{\lambda}$ or conclude that no such vector exists,[2] and provide a collection C of circulations with vector-values $\boldsymbol{c}^1, \ldots, \boldsymbol{c}^r$ along with a set of positive numbers $\alpha_1, \ldots, \alpha_r$ such that $r = O(d)$, $\text{cone}\{\boldsymbol{c}^1, \ldots, \boldsymbol{c}^r\} \supseteq \text{ORTH}(G)$, and $\sum_{i=1}^{r} \alpha_i \boldsymbol{c}^i = \boldsymbol{0}$.

Remark 2.4.8 The collection C is used to compute an explicit zero-cycle if one exists. It enables us to construct a circulation of any given value $\boldsymbol{c}' \in \text{ORTH}(G)$. The decision problem (existence of a zero-cycle) can be solved even if C is not given.

Proposition 2.4.9 *Problem 2.4.7 can be solved using three applications of the parametric minimum cycle algorithm on G with $d-1$ parameters.*

Proof: Deferred to Chapter 3. ∎

The following proposition is used for the proof of Proposition 2.4.9.

Proposition 2.4.10 *Given vectors $\boldsymbol{c}^1, \ldots, \boldsymbol{c}^r \subset R^d$, and a subspace $S \subset R^d$, the following two conditions are equivalent:*

i. For every $\boldsymbol{\lambda} \notin S$,

$$\min\{\boldsymbol{\lambda}^T \boldsymbol{c}^1, \ldots, \boldsymbol{\lambda}^T \boldsymbol{c}^r\} < 0 .$$

[2]Note that $\mathcal{K} \neq \emptyset$ since $\boldsymbol{0} \in \mathcal{K}$; a separating vector exists if and only if $\mathcal{K} \neq \{\boldsymbol{0}\}$.

ii. $\text{cone}\{c^1, \ldots, c^r\} \supseteq S^\perp$.

Proof: The equivalence follows from Farkas' Lemma (see Proposition 2.4.12). First we assume (i) and show that (ii) is implied. Consider $z \in S^\perp$. If a vector $y \in R^d$ is such that $y^T z < 0$, then obviously $y \notin S$. The latter, combined with (i) gives the left hand side condition on Farkas' Lemma. Therefore, from the right hand side we have $z \in \text{cone}\{c_1, \ldots, c_r\}$.

We show that (ii) implies (i). Assume that $z \in S^\perp \Rightarrow z \in \text{cone}\{c_1, \ldots, c_r\}$. It follows from Farkas' Lemma that for all $z \in S^\perp$, we have $(\forall y \in R^d) y^T z < 0 \Rightarrow \min\{y^T c_i\} < 0$. Consider a vector $\lambda \notin S$. There must exist $z \in S^\perp$ such that $z^T \lambda < 0$ (otherwise, $\forall z \in S^\perp, z^T y = 0$ in contradiction to $\lambda \notin S$). We have $z^T \lambda < 0$. Therefore it follows from the left hand side of Farkas' Lemma that $\min\{\lambda^T c_i\} < 0$. ∎

Corollary 2.4.11 *Let the vectors* $c^1, \ldots, c^r$ *be circulation values. If for every vector* $\lambda \notin \text{lin}\,\mathcal{K}$, $\min\{\lambda^T c^1, \ldots, \lambda^T c^r\} < 0$, *then* $\text{cone}\{c^1, \ldots, c^r\} \supseteq \text{ORTH}(G)$.

Proof: Take $S = \text{lin}\,\mathcal{K}$, and recall from Theorem 2.4.6 part (ii) that $\text{ORTH}(G) = (\text{lin}\,\mathcal{K})^\perp$. ∎

Proposition 2.4.12 *[Farkas' Lemma [22]] For any vectors* $z, c_i \in R^d$, $i = 1, \ldots, r$,

$$(\forall y \in R^d)(y^T z < 0 \Rightarrow \min_i\{y^T c_i\} < 0) \Leftrightarrow z \in \text{cone}(c_i) \ .$$

2.4.2 Computing the partition

After computing a separating vector, the zero-cycle detection algorithm proceeds to compute a partition of the graph. In this subsection we define this partition, and discuss some of its properties. We also present the algorithm that computes the partition when the separating vector is given.

The essence of the following proposition is mentioned in [41].

Proposition 2.4.13 *Let $G = (V, E, w)$ be a scalar-weighted graph with no negative cycles. Using one application of an all-pairs shortest path algorithm we can find vertex disjoint subgraphs $G_1, \ldots, G_q$ of G with the following properties. Edges or vertices that are not active in any zero-cycle of G are not contained in any of the G_i's. Two vertices u and v are in the same G_i if and only if there exists a (scalar) zero-cycle of G in which both u and v are active.*

Proof: Apply an all-pairs shortest path algorithm to compute the distance d_{uv} between all pairs of vertices $u, v \in V$. Two vertices u, v are in the same subgraph G_i if and only if $d_{uv} + d_{vu} = 0$. If $d_{vv} > 0$, then v is not a part of a zero-cycle and does not belong to any G_i.

In order to identify all the edges that participate in some zero-cycle, do the following. Select arbitrarily some vertex w and use a single-source shortest path algorithm to compute the distances π_v ($v \in V$) from the vertex w to all other vertices. For every edge (u, v) define,

$$\delta_{uv} \equiv \pi_u - \pi_v + d_{uv} \geq 0 \ .$$

Determine that (u, v) is an active edge if and only if $\delta_{uv} = 0$. ∎

Remark 2.4.14 Each component of the partition of Proposition 2.4.13 contains a zero-cycle where all the vertices of the component are active. This zero-cycle can be constructed easily from the shortest paths.

Proposition 2.4.15 *Suppose $\boldsymbol{\lambda}$ is a separating vector of $G = (V, E, \boldsymbol{f})$. Consider the scalar weights $w = \boldsymbol{f}^T \boldsymbol{\lambda}$ on the edges of G. Observe that by the definition of a separating vector, there are no negative cycles in the scalar weighted (V, E, w). Let $G_1, \ldots, G_q$ be the partition of G into subgraphs as defined in Proposition 2.4.13, relative to the scalar weights w. Under these conditions, a (vector) zero-cycle exists in G if and only if a (vector) zero-cycle exists in one of the G_i's.*

Proof: The 'if' part is trivial. For the 'only if' part, suppose $\boldsymbol{x}$ is a (vector) zero-cycle of $G = (V, E, \boldsymbol{f})$. Then $\boldsymbol{x}$ is a scalar zero-cycle of $(V, E, \boldsymbol{f}^T \boldsymbol{\lambda})$. By the definition in Proposition 2.4.13, all the vertices active in $\boldsymbol{x}$ are in the same component G_i and hence $\boldsymbol{x}$ is a vector zero-cycle of G_i. ▮

Proposition 2.4.16 *If* CIRC(G) *contains a nontrivial linear subspace then a nontrivial (vector) zero-circulation exists in G.*

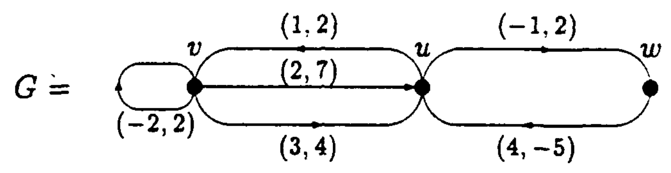
Proof: The proof is immediate. ▮

Proposition 2.4.17 *If $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_r\} \subset R^\ell$ and $\alpha_i > 0$ $(i = 1, \ldots, r)$ are such that $\sum_{i=1}^r \alpha_i \mathbf{c}_i = \mathbf{0}$, then for any $\boldsymbol{v} \in R^\ell$, it takes $O(\ell^2 r)$ time either to find nonnegative rational constants $\beta_1, \ldots, \beta_r$ such that $\boldsymbol{v} = \sum \beta_i \mathbf{c}_i$, or to recognize that no such constants exist.*

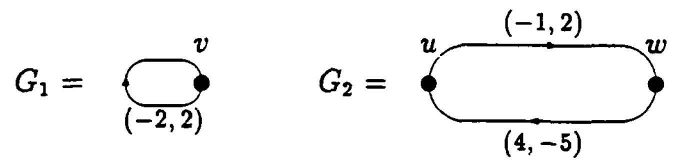
Proof: Express $\boldsymbol{v}$ as a linear combination of the vectors in C by solving the linear system of equations $\boldsymbol{v} = \sum \gamma_i \mathbf{c}_i$. This system has ℓ equations and r variables, and thus can be solved by Gaussian eliminations using $O(\ell^2 r)$ operations. If γ_i are nonnegative take $\beta_i = \gamma_i$; otherwise, denote $\alpha = \min_{1 \le i \le r} \alpha_i$, $\gamma = \min_{1 \le i \le r} \gamma_i$ and let $\beta_i = \gamma_i - (\gamma/\alpha)\alpha_i$. It is easy to verify that β_i $(1 \le i \le r)$ are nonnegative and $\sum_{i=1}^r \beta_i \mathbf{c}_i = \mathbf{0}$. ▮

Proposition 2.4.18 *Let $\boldsymbol{\lambda}$ be a separating vector of $G = (V, E, \boldsymbol{f})$. Let $G_1, \ldots, G_q$ be the partition of G into subgraphs (as defined in Proposition 2.4.13), relative to the scalar weights $\boldsymbol{f}^T\boldsymbol{\lambda}$. If the partition constitutes a single subgraph (i.e, $q = 1$), then G has a (vector) zero-cycle.*

Proof: If G has a single component relative to $\boldsymbol{f}^T\boldsymbol{\lambda}$, then all active vertices and edges are contained in G_1. Observe that all cycles with vector value in ORTH(G) are scalar zero-cycles relative to $\boldsymbol{f}^T\boldsymbol{\lambda}$. There exists a scalar zero-cycle in $(V, E, \boldsymbol{f}^T\boldsymbol{\lambda})$ in which all the vertices of G_1 are active. Thus, there exists a value $\mathbf{c} \in$ ORTH(G) which is attained at a circulation where all the vertices in G_1 are active, so this circulation is connected. By Theorem 2.4.6 and Proposition 2.4.16, there exists a circulation, not necessarily connected, whose value is $-\mathbf{c}$. The active vertices in



Separating Vector: $\boldsymbol{\lambda} = (1, 1)$



Witness Vector: $\boldsymbol{\lambda} = (-1, 1)$ Witness Vector: $\boldsymbol{\lambda} = (1, -1)$

Figure 2.3: Example of the decomposition of a graph G

this circulation must be contained in G_1. By combining the connected circulation supporting $\boldsymbol{c}$ with the one supporting $-\boldsymbol{c}$, we obtain a connected (nontrivial) zero-circulation, that is, a zero-cycle of G. ∎

Remark 2.4.19 Suppose we have a set C of (vector) cycle values such that cone $C \supseteq$ ORTH(G). The vector zero-cycle of Proposition 2.4.18 can be explicitly constructed as follows. We compute a connected zero-circulation relative to the scalar weights $\boldsymbol{f}^T\boldsymbol{\lambda}$, in which all the vertices are active. The vector value of this circulation is $\boldsymbol{c} \in$ ORTH(G) (see Remark 2.4.14). It follows from Proposition 2.4.17 that we can construct a circulation with value $-\boldsymbol{c}$. The combination of the two circulations is a connected zero-circulation.

Remark 2.4.20 Assume the graph G does not have a separating vector (that is, ORTH(G) $= R^d$). If we are given a set C of cycle values whose conic hull equals R^d, then a zero-circulation can be constructed as follows. Find a cycle in which all the vertices are active (G is strongly connected). Denote the value of this cycle by

c. It follows from Remark 2.4.14 that we can find a circulation with value $-c$. The combination of the two circulations is a (nontrivial) connected zero-circulation.

Remark 2.4.21 Remarks 2.4.19 and 2.4.20 discuss the construction of an explicit zero-cycle. Observe that if C is of size $O(d)$, then the time complexity of constructing a zero-cycle is $O(d^3)$ (see Proposition 2.4.17).

Proposition 2.4.22 *A witness for G exists if and only if G does not have a nontrivial zero-circulation.*

Proof: The proof is immediate. ▮

2.4.3 The algorithm

Algorithm 2.4.23 [zero-cycle detection]

i. Run an algorithm for Problem 2.4.7 on G (see Proposition 2.4.9). If a witness for G is found then stop. Otherwise, find a collection C of circulation values such that cone $C \supseteq \mathrm{ORTH}(G)$, and either find a separating vector λ or conclude that none exists. In the latter case, conclude that a connected zero-circulation, and hence a zero-cycle, exist in G (see Remark 2.4.20 for an explicit construction of the zero-cycle). Otherwise,

ii. Construct the partition of G which is defined in Propositions 2.4.13 and 2.4.15. If the partition is empty then G does not have a zero-cycle. Otherwise,

iii. If there is only one component (i.e., $q = 1$), then by Proposition 2.4.18, $G = (V, E, f)$ has a zero-cycle (see Remark 2.4.19 for how to find the zero-cycle explicitly).

iv. Run the zero-cycle detection algorithm on $G_1, \ldots, G_q$ (recursively). By Proposition 2.4.15, G has a zero-cycle if and only if at least one of $G_1, \ldots, G_q$ has one.

In the rest of the present section we prove the correctness and analyze the complexity of Algorithm 2.4.23.

Proposition 2.4.24 *If G is partitioned into $G_1, \ldots, G_q$ (see Proposition 2.4.15) and for some G_i, $\dim(\mathrm{ORTH}(G_i)) = \dim(\mathrm{ORTH}(G))$, then G_i will not be partitioned any further by the algorithm.*

Proof: Since $\mathrm{ORTH}(G_i) \subseteq \mathrm{ORTH}(G)$, equality of dimension implies equality of the sets, so a separating vector for G is a separating vector for G_i. ∎

Corollary 2.4.25 *Algorithm 2.4.23 terminates after at most $d - 1$ phases of partitioning.*

Proposition 2.4.26 *The time complexity of the zero-cycle detection algorithm for a graph $G = (V, E, f)$ (where f is d-dimensional) is dominated by the complexity of 3d applications of solving Problem 2.4.7 on G.*

Proof: First, observe that the complexity of explicitly constructing a zero-cycle (see Remark 2.4.21) is dominated by the complexity of the rest of the algorithm. Consider the recursion tree of Algorithm 2.4.23. The recursion tree corresponds to the decomposition tree of the graph G. By Corollary 2.4.25 this tree has d levels. Each level is a phase of partitioning a collection of subgraphs $G_1, \ldots, G_q$, with total number of $n = |V|$ vertices. The total computation done at such a phase is solving Problem 2.4.7 for each subgraph G_i, and then, if needed, partitioning it as described in Proposition 2.4.13. Observe that the time and processor complexities of solving Problem 2.4.7 and partitioning all the subgraphs at a certain phase, are dominated by the complexities of the same computation done on the graph G. Recall (see Proposition 2.4.13) that a partitioning operation amounts to an all-pairs shortest path computation. Therefore, the complexity of computing the partition is dominated by the complexity of solving Problem 2.4.7. It follows that at each level of the tree, the total complexity of the computation is dominated by the complexity of solving Problem 2.4.7 on G. ∎

Theorem 2.4.27 *The complexity of the zero-cycle detection algorithm for a graph $G = (V, E, f)$ (where f is d-dimensional) is essentially dominated by $3d$ applications of the parametric minimum cycle algorithm of Chapter 3, applied to instances with $d - 1$ parameters which involve the graph G.*

Proof: The proof follows from Propositions 2.4.9 and 2.4.26. ∎

2.5 Geometric lemmas

In this section we give the necessary lemmas which establish the proof of Theorem 2.4.6. The reader is referred to [31] for background.

For any subset C of R^d, denote

$$C^+ = \{\boldsymbol{v} : (\forall \boldsymbol{u} \in C)(\boldsymbol{v}^T \boldsymbol{u} \geq 0)\} .$$

Recall that a cone which does not contain a nontrivial linear space is said to be *pointed.*

The following proposition states well known facts about cones [30].

Proposition 2.5.1

i. ***Every cone C is a direct sum, $C = L \oplus C_p$, of a linear subspace L (the lineality space of C) and a pointed cone C_p.***

ii. ***The cone C_p is contained in the orthogonal complement of L in*** $\operatorname{lin} C$.

iii. $\dim(C_p) = \dim(C) - \dim(L)$.

Proposition 2.5.2

i. ***If $L \subset R^d$ is a linear subspace, then $L^+ = L^\perp$.***

ii. ***For every cone C we have $C^+ = C_p^+ \cap L^\perp$, where $C = L \oplus C_p$ as above.***

Proof: The proof of part i follows from the fact that if L is a linear subspace and $\boldsymbol{y} \in L^+$, then $\boldsymbol{y}^T\boldsymbol{d} = 0$ for all $\boldsymbol{d} \in L$. Part ii follows from the equality $C^+ = C_p^+ \cap L^+$ and from part i. ▮

Proposition 2.5.3 *If C is a pointed cone, C^+ is of full dimension.*

Proof: The following claim is a consequence of the duality theorem of linear programming. For any finite set of vectors $\boldsymbol{u}^1, \ldots, \boldsymbol{u}^r$, if there does not exist a vector $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_r)^T \geq \mathbf{0}$, $\boldsymbol{\alpha} \neq \mathbf{0}$, such that $\sum \alpha_i \boldsymbol{u}^i = \mathbf{0}$, then there exists a vector $\boldsymbol{v}$ such that $\boldsymbol{v}^T\boldsymbol{u}^i \geq 1$, $i = 1, \ldots, r$. Thus, if C is a pointed cone (not necessarily polyhedral), there exists a vector $\boldsymbol{v}$ such that for every unit-vector $\boldsymbol{u} \in C$, $\boldsymbol{v}^T\boldsymbol{u} \geq 1$. It follows that $\boldsymbol{v} \in C^+$ and there exists a ball B, centered at $\boldsymbol{v}$, such that for every $\boldsymbol{w} \in B$ and $\boldsymbol{u} \in C$ ($\boldsymbol{u} \neq \mathbf{0}$) we have $\boldsymbol{w}^T\boldsymbol{u} > 0$. This implies that $B \subset C^+$. ▮

Proposition 2.5.4

$$\dim(C^+) = \dim(L^\perp) .$$

Proof: It follows from Proposition 2.5.3 that C_p^+ is of full dimension in the space $\operatorname{lin} C_p$. Recall that $\operatorname{lin} C_p \subset L^+$. The proof follows from Proposition 2.5.2 part ii. ▮

Proposition 2.5.5 *If $\boldsymbol{\lambda} \in \operatorname{rel\,int}(C^+)$, then for all $\boldsymbol{c} \in C_p$ ($\boldsymbol{c} \neq \mathbf{0}$), $\boldsymbol{\lambda}^T\boldsymbol{c} > 0$.*

Proof: From the proof of Proposition 2.5.3 and Proposition 2.5.4 it follows that there exists a vector $\boldsymbol{v}$ such that for every unit-vector $\boldsymbol{u} \in C_p$, $\boldsymbol{v}^T\boldsymbol{u} \geq 1$, and for every $\boldsymbol{w} \in L$, $\boldsymbol{v}^T\boldsymbol{w} = 0$. The set C^+ is full dimensional relative to $L^\perp$. Therefore, if $\boldsymbol{\lambda}^T\boldsymbol{c} = 0$ for some $\boldsymbol{c} \in C_p$ ($\boldsymbol{c} \neq \mathbf{0}$), then $\boldsymbol{\lambda} \notin \operatorname{rel\,int} C^+$. ▮

Let $C = \mathrm{CIRC}(G)$ (see Definition 2.4.4). Let L and C_p be as in Proposition 2.5.1. Let $\mathcal{K}$ be as in Definition 2.4.2.

Proposition 2.5.6

$$\mathcal{K} = C^+$$

Proof: Immediate from Proposition 2.4.5. ∎

Proposition 2.5.7 *For every* $\lambda \in \operatorname{relint}(\mathcal{K})$, *the set* $\mathrm{ORTH}(G, \lambda)$ *is equal to the linear subspace* L.

Proof: It follows from Propositions 2.5.5 and 2.5.6, that if $\lambda \in \operatorname{relint} \mathcal{K}$ and $c \in C$ are orthogonal, then $c \in L$. On the other hand, since $\mathcal{K} \subseteq L^{\perp}$ (see Proposition 2.5.2), if $c \in L$ and $\lambda \in \mathcal{K}$, then $\lambda^T c = 0$. ∎

2.6 Applications of the zero-cycle detection algorithm

We first introduce some notation for the discussion of periodic graphs (see Section 2.1). For a given $G = (V, E, f)$ where $f : E \to Z^d$ and $V = \{1, \ldots, n\}$, denote by $G^{\infty} = (V^{\infty}, E^{\infty})$ the infinite periodic graph that is defined by G as explained in Section 2.1. We refer to G^{∞} as a d-dimensional periodic graph. Formally,

$$V^{\infty} = Z^d \times V = \{(z, i) \ : \ z \in Z^d, i \in V\},$$

$$E^{\infty} = Z^d \times E = \{(z, e) \ : \ z \in Z^d, e \in E\}.$$

If $e = (i, j)$ we also identify the edge (z, e) with the pair $((z, i), (z + f(e), j))$.

The zero-cycle detection algorithm of Section 2.4 computes the decomposition tree of an input graph $G = (V, E, f)$ and the separating vectors for all the subgraphs sitting at the nodes of this tree. Recall that this computation can be performed by solving polynomially many LP programs. In Section 2.4 we presented strongly polynomial time solution when the dimension d is fixed. We discuss two problems which can be solved easily when the decomposition tree and the separating vectors are given.

2.6.1 Scheduling

The first application is the problem of scheduling a system of uniform recurrence equations. The problem was raised by Karp, Miller and Winograd [39] and algorithms that solve it were given in [39, 56, 55]. These algorithms are stated in terms of solving systems of linear inequalities and therefore do not establish strong polynomiality. We show that the knowledge of the decomposition tree and the separating vectors enables us to produce an immediate solution. Hence, we obtain strongly polynomial complexity bounds.

A system of uniform recurrence equations is a finite set of relations among functions $F_i : Z^d \rightarrow R$ $(i = 1, \dots, n)$,

$$F_i(\boldsymbol{z}) = \psi_i(F_1(\boldsymbol{z} - \boldsymbol{c}_{i1}), \dots, F_n(\boldsymbol{z} - \boldsymbol{c}_{in})) .$$

(The definition can be easily extended to accommodate the case where the value of some F_i is related directly to more than one value of some F_j.) Such a system can be modeled by a finite graph $G = (V, E, \boldsymbol{f})$, where the functions F_i correspond to the vertices. It is called the "dependence graph" in [39]. This graph defines a periodic graph G^* whose vertices correspond to the function values $F_i(\boldsymbol{z})$, $(i = 1, \dots, n,\ \boldsymbol{z} \in Z^d,\ \boldsymbol{z} \geq \boldsymbol{0})$. The direct dependencies among function values are modeled by the edges of G^* as follows. If $e = (i, j) \in E$ and $\boldsymbol{f}(e) = \boldsymbol{a}$, then the evaluation of $F_i(\boldsymbol{z})$ requires the knowledge of $F_j(\boldsymbol{z} - \boldsymbol{a})$ (and having all the required knowledge is sufficient). For simplicity, suppose it takes one time unit to evaluate the ψ_i's, that is, given all the required knowledge, it takes one time unit to calculate the function value.

For "efficient" parallel evaluation of the function values, one would like to find large sets of "independent" values, that is, sets of values that can be computed simultaneously. Here, two values are independent if there is no directed path in G^* between their corresponding vertices. A set of values is called independent if every two members of the set are independent. The problem of finding a maximal independent set of values is not easy, since the problem of deciding whether there exists a directed path in G^*, from $(\boldsymbol{z}^1, \boldsymbol{i}^1)$ to $(\boldsymbol{z}^2, \boldsymbol{i}^2)$ is $\mathcal{NP}$-Complete, even for one-dimensional periodic graphs (see [51]).

A subspace $S_i \subset V_i$ is said to be *independent* if the values $F_i(z)$ $(z \in S_i)$ are independent. Interestingly, one can compute in polynomial time maximal independent subspaces [39, 56]. Let $V_i = Z^d \times \{i\}$, $i = 1, \ldots, n$. A maximal independent subspace S_i gives a partition of V_i into independent "isomorphic" flats S_v $(v \in S_i^{\perp})$, where $S_v = (v, 0) + S_i = \{(v + z, i) \mid z \in S_i\}$.

The algorithm for maximal independent subspaces finds for each i, $i = 1, \ldots, n$, a matrix M_i of dimensions $(d - \dim(S_i)) \times d$, whose rows are linearly independent, and whose null space is S_i. Following [56], the matrix M_i is called the *scheduling matrix* of i.

An algorithm that computes the scheduling matrix for the special case where the decomposition tree of G is of depth one was given in [39]. In this special case, assuming that G is strongly connected, the scheduling matrix would be the same for all i. In fact, $M = M_1 = \cdots = M_n$ consists of a single vector $v \in R^d$, which is computed by solving a set of linear programming problems. Obviously, the null space of v is of dimension $d - 1$. Any solution of the set of linear programs used in [39] is in the interior of the set $\{v \mid (\forall c \in \text{CIRC})(v^T c \geq 0)\}$. Observe that every such v is a separating vector (see Definition 2.4.2) for G. Moreover, it is a witness since the decomposition tree has depth one. A more formal statement follows.

Proposition 2.6.1 *Suppose $G = (V, E, f)$ is strongly connected and G^* has no cycles. If the decomposition tree of G is of depth one and v is a separating vector (and hence a witness) for G, then the null space of v is a maximal independent subspace.*

Proof: The null space of v has dimension $d - 1$. Therefore, if it is independent, it must be maximal. It remains to show that the null space of v is independent. First, we claim that for any subspace S, if $S \cap \text{CIRC} = \{0\}$, then for all i $(i = 1, \ldots, n)$ and for all $u \in S^{\perp}$, the set $\{(u + z, i) \mid z \in S\}$ is independent. To prove this claim, assume to the contrary that for some $b \neq 0$ in S and some i, there exists a directed path in G^* from (u, i) to $(u + b, i)$. Thus, there is a cycle in G with vector weight b, which implies $b \in \text{CIRC}$, and hence a contradiction. Second, we claim that the intersection of the null space of any witness v with CIRC is equal to

$\{0\}$. To prove the second claim, observe that if witness exists, then the cone CIRC of possible circulation values is pointed. Therefore, since ORTH(G) is the lineality space of CIRC(G) (see Theorem 2.4.6), we have dim(ORTH(G)) = 0. Observe that ORTH(G) is the intersection of the null space of any separating vector with CIRC (see Definition 2.4.4). Assuming the second claim holds, the first claim implies that $S(\boldsymbol{v})$ is an independent subspace. This concludes the proof of the proposition. ∎

Roychowdhury and Kailath [56] generalized the result of [39] and gave an algorithm which computes the scheduling matrices for any dependence graph G, where the decomposition tree is not necessarily of depth one. In the general case, the scheduling matrices M_i ($i = 1, \ldots, n$) need not be all identical, or even of the same dimension. Their algorithm first computes the decomposition tree of G, along with the separating vectors of the subgraphs sitting at the nodes of the decomposition tree. Subsequently, the algorithm uses these separating vectors to construct the scheduling matrices. The latter construction is trivial (see Definition 2.6.2 and Proposition 2.6.3).

The algorithm of Roychowdhury and Kailath [56] (like the algorithm for the depth one case of [39]) is based on solving $O(m)$ sets of linear inequalities and therefore, does not establish strong polynomiality. Recall that the zero-cycle detection algorithm computes the decomposition tree of G along with a collection of separating vectors that correspond to the subgraphs of G sitting at the nodes of the decomposition tree. Hence, the results obtained here imply that the scheduling matrices can be computed within the time bounds of the zero-cycle detection algorithm, that is, in $\mathcal{NC}$ and strongly polynomial time.

When the decomposition tree of G and the separating vectors at its nodes are given, it is easy to compute the scheduling matrices [56]:

Definition 2.6.2 Let $G = (V, E, \boldsymbol{f})$ be a dependence graph, where $V = \{1, \ldots, n\}$. Consider the decomposition tree of G, and the separating vectors of the subgraphs sitting at the nodes of the tree. For each vertex $i \in V$, consider the set of subgraphs that are sitting in the decomposition tree and of which i is a member. This set of subgraphs corresponds to a path in the decomposition tree. Define the *path* of a vertex i to be the ordered set of subgraphs along this path.

Proposition 2.6.3 *For a given dependence graph $G = (V, E, f)$, the scheduling matrix M_i of a vertex i is the matrix whose rows are the separating vectors of the subgraphs along the path of i.*

2.6.2 Strong connectivity

Another application of the zero-cycle detection algorithm is the following. Given a dependence graph $G = (V, E, f)$, compute the strongly connected components of G^{∞}, that is, find graphs $G_i = (V_i, E_i, f_i)$ such that the graphs G_i^{∞} are isomorphic to each of the strongly connected components of G^{∞}. The problem of strong connectivity on periodic graphs was first raised by Orlin [51]. However, his paper is concerned only with one-dimensional periodic graphs (i.e., when $f : E \rightarrow Z$ is a scalar function). Orlin gave an algorithm for the one-dimensional case that does not seem to generalize to higher dimensions. This is not surprising since the strong connectivity problem is obviously at least as hard as zero-cycle detection in G: the periodic graph G^{∞} does not have a cycle if and only if it has no nontrivial strongly connected components. We show that the zero-cycle detection algorithm can be used to compute the strongly connected components of G^{∞}. More specifically, we prove that the strongly connected components are the connected components of the subgraphs sitting at the leaves of the decomposition tree.

The relation between the decomposition tree of the dependence graph G and the strongly connected components of G^{∞} is given by the following proposition:

Proposition 2.6.4 *The strongly connected components of G^{∞} are precisely the connected components of the graphs G_i^{∞}, where G_i is any subgraph of G sitting at a leaf of the decomposition tree of G.*

Proof: The proof is immediate from the following two claims:
The first claim is that every strongly connected component of G^{∞} must be contained in some G_i^{∞}. This is obvious since all zero-cycles of G must be contained in one of the G_i's.

The second claim is that every connected component of a G_i^* is strongly connected. It suffices to show that every path $(e_1, e_2, \ldots, e_\ell)$ in G_i is part of a zero-cycle. Since every e_j $(j = 1, \ldots, \ell)$ participates in a zero-cycle C_i, the cycle $\bigcup_{j=1}^{\ell} C_i$ is a zero-cycle which contains the path. ∎

Remark 2.6.5 A strongly connected component $S \subset V^*$ of G^* is such that if $(\boldsymbol{a}, i), (\boldsymbol{b}, i) \in S$, then $(\boldsymbol{a} + \alpha(\boldsymbol{a} - \boldsymbol{b}), i) \in S$ for any integer α.

It follows from Proposition 2.6.4 that for a given dependence graph $G = (V, E, \boldsymbol{f})$, we can compute a collection of dependence graphs $\hat{G}_i$ $(i = 1, \ldots, r)$, such that the graphs $(\hat{G}_i)^*$ are isomorphic to the strongly connected components of G^*. This is done as follows.

Algorithm 2.6.6 [Strongly connected components of G^*]

i. Compute the decomposition tree of G. Denote by G_i $(i = 1, \ldots, r)$ the subgraphs sitting at the leaves of the decomposition tree.

ii. For each G_i, compute a dependence graph $\hat{G}_i$, such that $(\hat{G}_i)^*$ is isomorphic to each of the the connected components of G_i^*.

Step i of the algorithm involves the computation of the decomposition tree, that is, the zero-cycle detection algorithm. Step ii involves the computation of the connected components of the G_i^*'s. An algorithm for computing connected components of a periodic graph is given in [8, 12].

2.7 Concluding remarks

The obvious open question that arises is whether Problem 2.1.1, where the dimension d is part of the input, can be solved in strongly polynomial time, and whether it is in the class $\mathcal{NC}$. It is interesting to note the following:

Proposition 2.7.1 *The problem of detecting a zero cycle (Problem 2.1.1) is $\mathcal{P}$-complete, and also the general linear programming problem is reducible to it in strongly polynomial time.*

Proof: The general linear programming problem is equivalent (in strongly polynomial time and an $\mathcal{NC}$ reduction) to the problem of solving the following system:

$$(S) \qquad \begin{aligned} \boldsymbol{A}\boldsymbol{x} &= \boldsymbol{0} \\ \boldsymbol{0} \neq \boldsymbol{x} &\geq \boldsymbol{0} \,, \end{aligned}$$

where $\boldsymbol{A} \in R^{m \times n}$. Consider a network consisting of n parallel edges from vertex s to vertex t and one edge from t to s. (It is a trivial matter to avoid parallel edges if this is desired.) Associate with the i'th edge the weight-vector given by the i'th column of $\boldsymbol{A}$, and associate with the reverse edge the zero vector. The existence of a nontrivial zero circulation in this network is equivalent to the existence of a solution to the given system (S). This establishes our claim. ∎

In view of Proposition 2.7.1, the questions stated in the beginning of this section are equivalent to two famous and difficult open questions.

Recall that we considered zero-cycle which were not necessarily simple. Unfortunately, if simplicity of the cycle is added to the requirements, the problem becomes $\mathcal{NP}$-complete. Moreover, even the problem of recognizing whether a graph with scalar weights has a simple cycle with a total weight of zero is $\mathcal{NP}$-complete. This follows from the fact that the knapsack problem can be reduced to detecting a simple zero-cycle in a graph whose edges form a ring, where two consecutive vertices are connected with two parallel edges.

Chapter 3

Parametric minimum cycle

This chapter is concerned with the parametric extensions of the minimum cycle and the minimum cycle-mean problems. In this problems, we consider graphs with edge-weights which are linear functions of d parameters. The goal, roughly, is to find an assignment of the parameters such that the value of the optimal cycle is maximized.

Recall from Theorem 2.4.27 that Problem 2.1.1 can be solved within the same time bounds as these parametric extensions. The proof of Proposition 2.4.9, which establishes this relationship, was deferred to this chapter.

The algorithms presented in this chapter to solve the parametric minimum cycle problem introduce a general method. This method is applicable to parametric extensions of a large class of problems. More specifically, this method introduces a combinatorial way to optimize a concave function in fixed dimension, when we are given a piecewise affine algorithm (see Definition 3.1.2) that computes this function. The latter is discussed further in Chapter 4.

This chapter is organized as follows. In Section 3.1 we give some definitions and describe the general setup. We also give the proof of Proposition 2.4.9. In Section 3.2 we present a simplified algorithm for the parametric minimum cycle-mean problem. The goal of this presentation is to give the reader a sense of how the strongly polynomial time bounds are achieved. Details which are not essential for the qualitative result of strongly polynomial time bounds are avoided. The reader may skip Section 3.2, since the succeeding sections are independent. In Section 3.3 we give an algorithm for the problem of parametric minimum cycle with at most n edges. This section introduces additional ideas whose purpose is to improve the sequential and parallel time bounds.

3.1 Preliminaries

We start by giving some definitions and notations.

Definition 3.1.1

i. For a finite set $C \subset R^d$, denote by $L_C : R^d \to R$ the lower envelope of the linear functions that correspond to the vectors in C, $L_C(\boldsymbol{\lambda}) = \min_{\mathbf{c}\in C} \mathbf{c}^T\boldsymbol{\lambda}$. The vectors $\mathbf{c} \in C$, and interchangeably the linear functions $\mathbf{c}^T\boldsymbol{\lambda}$ are referred to as *pieces* of L_C. If for $\boldsymbol{\lambda}' \in R^d$ and $\mathbf{c} \in C$ we have $\mathbf{c}^T\boldsymbol{\lambda}' = L_C(\boldsymbol{\lambda}')$ we say that $\mathbf{c}$ is *active* at $\boldsymbol{\lambda}'$.

ii. Suppose $H \subset R^d$ is a flat, $F \subset R^d$ is the subspace parallel to it, and $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_r\} \subset R^d$ is a set of vectors. A *balancing combination* of C relative to H is a positive linear combination $\sum_{i=1}^{r} \alpha_i\mathbf{c}_i$ which is orthogonal to F. If $H = R^d$ we say that $\sum_{i=1}^{r} \alpha_i\mathbf{c}_i = \mathbf{0}$ is a balancing combination of C if $\alpha_1, \ldots, \alpha_r > 0$.

Let R^d_δ denote the set of vectors $\boldsymbol{\lambda} = (\lambda_1, \cdots, \lambda_d)^T \in R^d$ such that $\lambda_d = \delta$.

Definition 3.1.2 For $g : R^d \to R$ ($g : H \to R$ where $H \subset R^d$ is a hyperplane), we introduce the following definitions and notations:

i. Denote by $\Lambda_g \equiv \Lambda$ (possibly $\Lambda = \emptyset$) the set of vectors $\boldsymbol{\lambda} \in R^d$ ($\boldsymbol{\lambda} \in H$) where $g(\boldsymbol{\lambda})$ is maximized.

ii. Denote by $\mathcal{K}_g \equiv \mathcal{K}$ the set of $\boldsymbol{\lambda} \in R^d$ ($\boldsymbol{\lambda} \in H$) such that $g(\boldsymbol{\lambda}) \geq 0$. Also, denote by $\mathcal{K}_\delta$ the set of $\boldsymbol{\lambda} \in \mathcal{K} \cap R^d_\delta$.

iii. An algorithm that computes the function g is called *piecewise affine*, if the operations it performs on intermediate values that depend on the input vector are restricted to additions, multiplications by constants, comparisons, and making copies.

iv. When $g = L_C$ ($C \subset R^d$), we say that $g' = L_{C'}$ is a *weak approximation* of g, if the pieces of g' comprise a subset of the pieces of g ($C' \subset C$) and aff $\Lambda_g =$ aff $\Lambda_{g'}$. The function $g' = L_{C'}$ is a *minimal* weak approximation of g, if there is no proper subset C'' of C' such that $L_{C''}$ is a weak approximation of g.

We assume throughout that the function $g : R^{d-1} \to R$ is concave (sometimes we denote the range by R^d_δ). We also assume that g is given by a piecewise affine algorithm $\mathcal{A}$ that evaluates it at a given point. The parametric minimum cycle and parametric minimum cycle-mean problems are special cases of the following problem. The scheme presented here to solve Problem 3.1.3 for these cases, however, can be applied to any concave function g as above. This is discussed further in Chapter 4.

Problem 3.1.3 If $g(\lambda) > 0$ for some λ, then output such λ; otherwise, find $\lambda \in$ relint Λ. We sometimes add the following requirement. If $g \leq 0$, find a subset C of the pieces of g, such that L_C is a minimal weak approximation of g, and find a balancing combination of C relative to R^d_1. We refer to this last task as the "optional" part of the problem.

Intuitively, the set C certifies that the function g does not exceed its maximum value. The advantage of considering it is that while the number of pieces of g may be vary large, the size of a minimal weak approximation is at most $2d$.

We discuss Problem 3.1.3 where the function g results from parametric extensions of the minimum cycle and the minimum cycle-mean problems. In an instance of a parametric problem with $d-1$ parameters, edge weights are generalized to be linear functions of the parameters. For each assignment of values to the parameters (vectors in R^d_1), we get an instance of the nonparametric problem. The function g is defined as the mapping from assignments of values to the solution of the corresponding nonparametric problem.

Let $G = (V, E, \boldsymbol{f})$ be as in Problem 2.1.1. The vector-weights $\boldsymbol{f}(e)$ $(e \in E)$ are interpreted as linear functions of $d-1$ variables (parameters):

$$\boldsymbol{f}(e) = f_1(e)\lambda_1 + \cdots + f_{d-1}(e)\lambda_{d-1} + f_d(e) \ .$$

When $\boldsymbol{\lambda}$ is assigned with specific numerical values, we denote by $\boldsymbol{f}^T\boldsymbol{\lambda}$ the resulting set of scalar weights.

Parametric minimum cycle-mean :

Definition 3.1.4 Consider $G = (V, E, \boldsymbol{f})$, where for $(i,j) \in E$, $\boldsymbol{f}(i,j) = \boldsymbol{c}_{ij} \in R^d$,

i. For a subset of edges $E' \subset E$, denote by $\boldsymbol{f}(E')$ the $(d-1)$-variable linear function $(1/|E'|)\sum_{e\in E'} \boldsymbol{f}(e)$.

ii. Denote by $g(\boldsymbol{\lambda})$ the minimum cycle-mean[1] relative to the scalar-weights $\boldsymbol{\lambda}^T\boldsymbol{c}_{ij}$.

Problem 3.1.5 [Parametric Minimum Cycle-Mean]
For a given graph $G = (V, E, \boldsymbol{f})$, if $g(\boldsymbol{\lambda}) > 0$ for some $\boldsymbol{\lambda}$, then output one such $\boldsymbol{\lambda}$; otherwise, find $\boldsymbol{\lambda} \in \operatorname{rel\,int} \Lambda$.

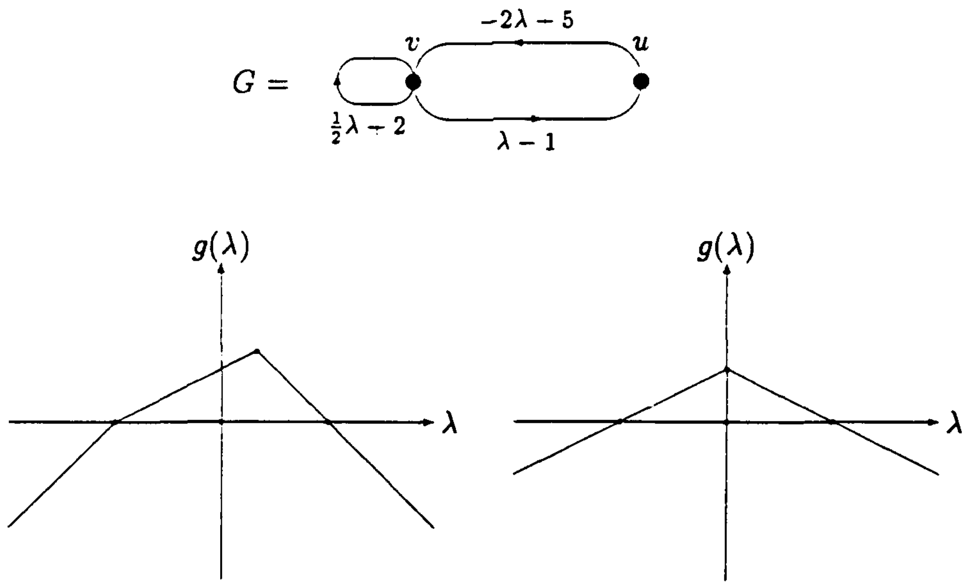
Parametric minimum cycle :

Definition 3.1.6 Consider $G = (V, E, \boldsymbol{f})$, where for $e \in E$, $\boldsymbol{f}(e) = \boldsymbol{c}_e \in R^d$,

i. For $E' \subset E$, denote by $\boldsymbol{f}(E')$ the $(d-1)$-variable linear function $\sum_{e\in E'} \boldsymbol{f}(e)$.

ii. Let $C = C(\boldsymbol{\lambda})$ denote a cycle of at most n edges which minimizes the total scalar weight $\boldsymbol{\lambda}^T\boldsymbol{c}_e$. Denote $g(\boldsymbol{\lambda}) = \boldsymbol{f}(C)^T\boldsymbol{\lambda}$.

[1]The *minimum cycle-mean* is the minimum, over all simple cycles, of the total weight of a cycle divided by the number of its edges.



Minimum cycle of at most 2 edges in G — $g(\lambda) = \min\{\frac{1}{2}\lambda + 2, \lambda + 4, -\lambda + 4\}$

Minimum cycle-mean in G — $g(\lambda) = \min\{\frac{1}{2}\lambda + 2, -\frac{1}{2}\lambda + 2\}$

Figure 3.1: Example of g for a graph G with 2-dimensional weights

Problem 3.1.7 [Parametric Minimum Cycle]
For a given graph $G = (V, E, \boldsymbol{f})$, if $g(\boldsymbol{\lambda}) > 0$ for some $\boldsymbol{\lambda}$, then output any such $\boldsymbol{\lambda}$; otherwise, find $\boldsymbol{\lambda}^* \in \operatorname{relint} \Lambda$ and a collection $C = \{C_1, \cdots, C_r\}$ of cycles , each of at most n edges, such that $L_{\{\boldsymbol{f}(C_i) \mid i \in \{1,\ldots,r\}\}}$ is a minimal weak approximation of g (see Definition 3.1.2), along with a balancing combination of the cycle values $\boldsymbol{f}(C)$ relative to R^d_1.

The function g defined for both problems above is of the form $g = L_C$, where C is the collection of all possible vector values of cycle-means (Problem 3.1.5) or cycles of at most n edges (Problem 3.1.7). Also note that g is concave and computable by a piecewise affine algorithm (see Figure 3.1 for an example of such functions). The parallel of the optional part in Problem 3.1.3 is omitted in the statement of Problem 3.1.5. In Problem 3.1.7, however, the optional part corresponds to the collection of cycles. We explain the purpose of considering the optional part. Recall that in Chapter 2 the problem of detecting zero-cycles was reduced to solving Problem 2.4.7. Solving the latter problem amounts to (i) computing a separating vector, which is needed to solve the decision problem, and (ii) finding a collection of cycles, which is used for computing an explicit zero-cycle when one does exist. Proposition 2.4.9 tied the solution of Problem 2.4.7 to solving instances of the parametric minimum cycle and the parametric minimum cycle-mean problems. The proof of Proposition 2.4.9 was deferred to the current chapter. We will see that in order to compute a separating vector (and hence decide existence of a zero-cycle) it suffices to consider Problems 3.1.5 and 3.1.7 where the optional part is omitted.

The remainder of this section is organized as follows. In Subsection 3.1.1 we define an important subproblem of Problem 3.1.3. In Subsection 3.1.2 we discuss properties of weak approximations and balancing combinations. We suggest to skip Subsection 3.1.2 in first reading. In Subsection 3.1.3 we give the proof of Proposition 2.4.9, and hence, completing the reduction of the zero-cycle detection problem to solving instances of Problems 3.1.5 and 3.1.7. The properties given in Subsection 3.1.2 are used for solving the "optional" part of Problem 3.1.3, that is, to solve the second part of Problem 2.4.7.

3.1.1 The oracle problem

We find the following subproblem useful for the solution of Problem 3.1.3. The goal of Problem 3.1.3 is to maximize a function over some domain. Intuitively, the following is a useful tool: decide on which side of a given query hyperplane (in the λ space) $g(\lambda)$ is either maximized or unbounded. We refer to a procedure that solves this problem as an *oracle*. Clearly, such an "oracle" would enable us to perform binary searches over the λ space. In order to achieve strongly polynomial bounds, however, we use a more sophisticated approach where the number of hyperplane queries needed is of the order of the number of comparisons done by $\mathcal{A}$. Each hyperplane query can be resolved by an oracle call. By using the multi-dimensional search techniques and exploiting the parallelism of $\mathcal{A}$, however, we can do better: We present a solution where the number of oracle calls performed is a polylog in the number of comparisons (hyperplane queries needed).

The function g is a concave piecewise linear mapping from R_1^d into R. Concave functions have the property that it can be effectively decided which side of a given hyperplane H contains the maximum of the function. The decision can be made by considering a neighborhood of the maximum of the function relative to H, searching for a direction of ascent from that point. This principle is explained in detail in [47] and is developed below for the special structure of our problem.

Problem 3.1.8 Given is a concave function $g : R_1^d \rightarrow R$ and a hyperplane H in the $\boldsymbol{\lambda}$-space.

i. Recognize whether there exists $\boldsymbol{\lambda} \in H$ such that $g(\boldsymbol{\lambda}) > 0$, and if so, output any such $\boldsymbol{\lambda}$; otherwise,

ii. find $\boldsymbol{\lambda} \in H \cap \operatorname{relint}(\Lambda)$, if such a $\boldsymbol{\lambda}$ exists, and generate a solution of Problem 3.1.3 for g; otherwise, if $H \cap \operatorname{relint} \Lambda = \emptyset$,

iii. recognize which of the two halfspaces determined by H either intersects $\operatorname{relint} \Lambda$, or has g unbounded on it.

We refer to a procedure that solves Problem 3.1.8 as an *oracle* and to the hyperplane H as the *query hyperplane*.

The method presented here solves instances of Problem 3.1.3 by running a "simulation" of the algorithm $\mathcal{A}$, where (i) additions and multiplication are replaced by vector operations, and (ii) comparisons are replaced by hyperplane queries. Problem 3.1.8 is solved by three recursive calls to instances of Problem 3.1.3 on functions of the form $g' : R_1^{d-1} \rightarrow R$. For a point $\boldsymbol{\lambda} \in R_1^d$, define $g_{\boldsymbol{\lambda}} = L_{C'}$ as the function whose pieces are all the pieces of $g = L_C$ which are active at $\boldsymbol{\lambda} \in R_1^d$ ($C' = \{\mathbf{c} \in C \mid \mathbf{c}^T\boldsymbol{\lambda} = g(\boldsymbol{\lambda})\}$). The functions to which the recursive calls are made are restrictions of functions of the form $g_{\boldsymbol{\lambda}}$ to hyperplanes. In Chapter 4 we show that a piecewise affine algorithm that computes g can be converted to a piecewise affine algorithm that computes g' and uses the same number of operations. Algorithms that solve Problem 3.1.8 for the parametric minimum cycle and the parametric minimum cycle-mean functions are given later in the current chapter.

3.1.2 Geometric lemmas

We discuss some properties of weak approximations and balancing combinations.

Proposition 3.1.9 *Suppose that $g = L_C$, where C is a finite set of vectors, $\boldsymbol{\lambda}^* \in \Lambda_g$, and $C' = \{\mathbf{c} \in C \mid \mathbf{c}^T\boldsymbol{\lambda}^* = g(\Lambda)\}$. The function $L_{C'}$ is a weak approximation of g.*

Proof: Consider g in a neighborhood of $\boldsymbol{\lambda}^*$. Since C is finite, there exists an open neighborhood N of $\boldsymbol{\lambda}^*$ such that $L_{C'}(\boldsymbol{\lambda}) = g(\boldsymbol{\lambda})$ for every $\boldsymbol{\lambda} \in N$. Consider a vector $\boldsymbol{\lambda}'$ and denote by L the open line segment determined by $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}^*$. Consider a point $\boldsymbol{\lambda}'' \in L \cap N$, and let $\mathbf{c} \in C'$ be a piece of g active at $\boldsymbol{\lambda}''$. Since $g(\boldsymbol{\lambda}'') \leq g(\boldsymbol{\lambda}^*)$ we have $g(\boldsymbol{\lambda}') \leq g(\boldsymbol{\lambda}^*)$. Suppose that $\boldsymbol{\lambda}' \notin \operatorname{aff} \Lambda$. We need to show that $L_{C'}(\boldsymbol{\lambda}') < g(\boldsymbol{\lambda}^*)$. For every $\boldsymbol{\lambda} \in L$, $g(\boldsymbol{\lambda}) < g(\boldsymbol{\lambda}^*)$, since otherwise we have $\boldsymbol{\lambda}' \in \operatorname{aff} \Lambda$. In particular the latter holds for $\boldsymbol{\lambda}''$. It follows that $L_{C'}(\boldsymbol{\lambda}') \leq \mathbf{c}^T\boldsymbol{\lambda}' < g(\boldsymbol{\lambda}^*)$. ∎

Corollary 3.1.10 *If $L_{C'}$ is a minimal weak approximation of g, we have $\mathbf{c}^T\boldsymbol{\lambda} = g(\Lambda)$ for all $\mathbf{c} \in C'$ and $\boldsymbol{\lambda} \in \operatorname{aff} \Lambda$. It follows that $\Lambda_{L_{C'}} = \operatorname{aff} \Lambda$.*

Proposition 3.1.11 *If $g = L_C$ and for some $\boldsymbol{\lambda} \in \operatorname{relint} \Lambda$ and $\mathbf{c} \in C$ we have $\mathbf{c}^T\boldsymbol{\lambda} = g(\Lambda)$, then $\mathbf{c}^T\boldsymbol{\lambda} = g(\boldsymbol{\lambda})$ for all $\boldsymbol{\lambda} \in \operatorname{aff} \Lambda$.*

The proof is immediate.

Proposition 3.1.12 *Suppose $g = L_{\tilde{C}}$, where $\tilde{C} \subset R^d$, is bounded from above. Let $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_r\} \subset \tilde{C}$ be a set of pieces of g. Under these conditions,*

i. The function L_C is a weak approximation of g if and only if $\operatorname{cone} C \supseteq (\operatorname{aff} \Lambda_g)^\perp$. If L_C is a minimal weak approximation of g, $\operatorname{cone} C = (\operatorname{aff} \Lambda_g)^\perp$.

ii. If L_C is a minimal weak approximation of g, then there exist positive numbers $\alpha_1, \ldots, \alpha_r$ such that $\sum_{i=1}^{r} \alpha_i \boldsymbol{c}_i = \mathbf{0}$ and hence $\operatorname{lin} C = (\operatorname{aff} \Lambda_g)^{\perp}$.

iii. If $\alpha_1, \ldots, \alpha_r$ as in (ii) are given explicitly, then every $\boldsymbol{c} \in (\operatorname{aff} \Lambda)^{\perp}$ can be expressed as a nonnegative linear combination of vectors in C using $O(d^2 r)$ operations.

Proof: The function $L_{\tilde{C}}$ is bounded from above on R^d, and hence is nonpositive. Therefore, $\mathbf{0} \in \Lambda, L_{\tilde{C}}(\Lambda) = 0$, and $\operatorname{aff} \Lambda = \operatorname{lin} \Lambda$ is a subspace. Part (i) follows from Proposition 2.4.10. The equality when L_C is a minimal weak approximation follows from Corollary 3.1.10. Parts (ii)-(iii) are direct consequences of Part (i) and Proposition 2.4.17. ∎

Remark 3.1.13 Suppose $H = \{\boldsymbol{\lambda} \in R^d \mid \boldsymbol{a}^T \boldsymbol{\lambda} = \beta\}$ $(\beta \geq 0)$ is a hyperplane, $F = \{\boldsymbol{\lambda} \in R^d \mid \boldsymbol{a}^T \boldsymbol{\lambda} = 0\}$ is the subspace parallel to H, and $\boldsymbol{\lambda}^* \in H$ is a point on the hyperplane. By applying standard methods, using $O(d^3)$ operations we can find an affine mapping M from H onto R^{d-1} that maps $\boldsymbol{\lambda}^*$ to $\mathbf{0}$. All such mappings are of the form $M : H \rightarrow R^{d-1}$, $M(\boldsymbol{\lambda}) = \boldsymbol{L}(\boldsymbol{\lambda} - \boldsymbol{\lambda}^*)$ where the matrix $\boldsymbol{L} \in R^{(d-1)\times d}$ is such that $\boldsymbol{L}(F) = \{\boldsymbol{L}\boldsymbol{\lambda} \mid \boldsymbol{\lambda} \in F\} = R^d$. Within the same time bound we can compute the matrices $\boldsymbol{L}^{-1} \in R^{d\times(d-1)}$ which map R^{d-1} onto the subspace F, and $\boldsymbol{F} \in R^{d\times d}$ such that for $\boldsymbol{y} \in R^d$, $\boldsymbol{F}\boldsymbol{y}$ is the projection of $\boldsymbol{y}$ into the subspace F.

Proposition 3.1.14 *Let $C \subset R^d$ be a finite set of vectors, and suppose $L_C : R^d \rightarrow R$ is bounded. Suppose we are given a hyperplane $H \subset R^d$, along with a point $\boldsymbol{\lambda}^* \in \Lambda_{L_C|_H}$. Under these conditions, in $O(d^3)$ operations we can compute an affine mapping $M : H \rightarrow R^{d-1}$ $(M(\boldsymbol{\lambda}^*) = \mathbf{0})$, and a linear mapping of the vectors in $C' = \{\boldsymbol{c} \in C \mid \boldsymbol{c}^T \boldsymbol{\lambda}^* = L_C(\Lambda)\}$ (the pieces of L_C which are "active" at $\boldsymbol{\lambda}^*$) into vectors in $\tilde{C} \subset R^{d-1}$. These mappings convert the problem of computing a minimal weak approximation of L_C restricted to H into an equivalent problem of computing a minimal weak approximation of $L_{\tilde{C}} : R^{d-1} \rightarrow R$.*

Proof: We use the notation of Remark 3.1.13. It follows from Proposition 3.1.9 that $L_{C'}$ is a weak approximation of $L_C|_H$, so we can consider only the vectors in the set C'. Let $\boldsymbol{K} = \boldsymbol{F}\boldsymbol{L}^{-1}$, and $\tilde{C} = \{\tilde{c} \mid c \in C'\}$ where $\tilde{c} = \boldsymbol{c}^T\boldsymbol{K}$. We will show that (i) the set $\tilde{C}$ is such that for all $\boldsymbol{\lambda} \in H, \boldsymbol{c} \in C'$, $\boldsymbol{c}^T\boldsymbol{\lambda} = \tilde{\boldsymbol{c}}^T M(\boldsymbol{\lambda}) + L_C(\boldsymbol{\lambda}^*)$, and (ii) for $Y \subset C$, the set $\tilde{Y} \subset \tilde{C}$ is such that $L_{\tilde{Y}}$ is a minimal weak approximation of $L_{\tilde{C}}$ if and only if L_Y is a minimal weak approximation of $L_C|_H$.

For a vector $\boldsymbol{y} \in R^d$ denote by $\boldsymbol{y}' = \boldsymbol{y}^T\boldsymbol{a}/\|\boldsymbol{a}\|$ the projection of $\boldsymbol{y}$ into the subspace spanned by $\boldsymbol{a}$, and denote by $\hat{\boldsymbol{y}} = \boldsymbol{F}\boldsymbol{y} = \boldsymbol{y} - \boldsymbol{y}'$ the projection of $\boldsymbol{y}$ on F. Observe that $F = \{\beta\boldsymbol{a} \mid \beta \in R\}^{\perp} \subset R^d$, and thus for all $\boldsymbol{x}, \boldsymbol{y} \in R^d$, $\boldsymbol{x}^T\boldsymbol{y} = \hat{\boldsymbol{x}}^T\hat{\boldsymbol{y}} + \boldsymbol{x}'\boldsymbol{y}'$, and for all $\boldsymbol{\lambda} \in H$ we have $\boldsymbol{\lambda}' = (\boldsymbol{\lambda}^*)'$.

For $\boldsymbol{c} \in C'$, define $\tilde{c} = \hat{\boldsymbol{c}}^T\boldsymbol{L}^{-1} = \boldsymbol{c}\boldsymbol{K}$. Consider vectors $\boldsymbol{c} \in C'$ and $\boldsymbol{\lambda} \in H$. Recall that $\boldsymbol{c}^T\boldsymbol{\lambda}^* = L_C(\boldsymbol{\lambda}^*)$, and therefore we need to show that $\boldsymbol{c}^T(\boldsymbol{\lambda} - \boldsymbol{\lambda}^*) = \tilde{\boldsymbol{c}}^T M(\boldsymbol{\lambda})$. Note that $\boldsymbol{c}^T(\boldsymbol{\lambda} - \boldsymbol{\lambda}^*) = \hat{\boldsymbol{c}}^T(\hat{\boldsymbol{\lambda}} - \hat{\boldsymbol{\lambda}^*}) + \boldsymbol{c}'(\boldsymbol{\lambda}' - (\boldsymbol{\lambda}^*)')$. Since $\boldsymbol{\lambda}' = (\boldsymbol{\lambda}^*)'$ for all $\boldsymbol{\lambda} \in H$, we have $\boldsymbol{c}^T(\boldsymbol{\lambda} - \boldsymbol{\lambda}^*) = \hat{\boldsymbol{c}}^T\boldsymbol{L}^{-1}\boldsymbol{L}(\boldsymbol{\lambda} - \boldsymbol{\lambda}^*) = \tilde{\boldsymbol{c}}^T M(\boldsymbol{\lambda})$. ∎

Proposition 3.1.15 *Under the conditions of Proposition 3.1.14 the following holds: A set of vectors $\{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_r\} \subset C'$, and a set of positive numbers $\alpha_1, \ldots, \alpha_r$, $\alpha_i > 0$ satisfy $\sum_{i=1}^r \alpha_i\tilde{\boldsymbol{c}}_i = \boldsymbol{0}$, if and only if $\sum_{i=1}^r \alpha_i\boldsymbol{c}_i = -\gamma\boldsymbol{a}$ and $\gamma \geq 0$. Moreover, $\gamma > 0$ if and only if $L_C|_H < 0$. The scalars α_i are independent of the particular choice of $\boldsymbol{\lambda}^*$ and the linear mappings.*

Proof: We continue to use the notation of the proof of Proposition 3.1.14. Recall that $\tilde{\boldsymbol{c}}L = \hat{\boldsymbol{c}}$, and thus $\sum_{i=1}^r \alpha_i\hat{\boldsymbol{c}}_i = (\sum_{i=1}^r \alpha_i\tilde{\boldsymbol{c}}_i)L = \boldsymbol{0}$. Hence, $\sum_{i=1}^r \alpha_i\boldsymbol{c}_i = \sum_{i=1}^r \alpha_i\hat{\boldsymbol{c}}_i + \sum_{i=1}^r \alpha_i\boldsymbol{c}_i' = \sum_{i=1}^r \alpha_i\boldsymbol{c}_i'$. Denote $\boldsymbol{c}_i' = c_i'\boldsymbol{a}$. We need to show that $\sum_{i=1}^r \alpha_i c_i' \leq 0$ and the inequality is strict if and only if $L_C(\boldsymbol{\lambda}^*) < 0$. Recall that for all $\boldsymbol{c} \in C'$, $\boldsymbol{c}^T\boldsymbol{\lambda}^* = L_C(\boldsymbol{\lambda}^*)$. Thus, $(\sum_{i=1}^r \alpha_i\boldsymbol{c}_i')^T\boldsymbol{\lambda}^* = (\sum_{i=1}^r \alpha_i c_i')\boldsymbol{a}^T\boldsymbol{\lambda}^* \leq 0$, and equality holds if and only if $L_C(\boldsymbol{\lambda}^*) = 0$. To conclude the proof, observe that $\boldsymbol{\lambda}^* \in H$, and therefore $\boldsymbol{a}^T\boldsymbol{\lambda}^* = \beta > 0$. ∎

Proposition 3.1.16 *Suppose we are given a set of vectors $C = \{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_r\} \subset R^d$ such that C spans R^d and a balancing combination of C (see Definition 3.1.1 part ii). By using $O(rd^2)$ operations we can find a minimal subset $C' \subset C$ ($|C'| \leq 2d$) that spans R^d and a balancing combination for C'.*

Proof: Assume that the vectors $\mathbf{c}_1, \ldots, \mathbf{c}_d$ span R^d. In order to find a solution we use an iterative step with the following properties. If $|C| > 2d$, we find a balancing combination for $C \setminus \bar{C}$ where $\bar{C} \subset \{\mathbf{c}_{d+1}, \ldots, \mathbf{c}_{2d+1}\}$ and $|\bar{C}| \geq 1$. It is easy to verify that after repeating this step at most $r - 2d$ times, we get a balancing combination for a subset C' of C, which spans R^d and is of size at most $2d$.

The iterative step is as follows. Solve the linear system of equalities $\sum_{i=d+1}^{2d+1} \beta_i \mathbf{c}_i = \mathbf{0}$. The vectors $\mathbf{c}_{d+1}, \ldots, \mathbf{c}_{2d+1}$ are linearly dependent, and hence this system is feasible. Assume without loss of generality that for at least one index i, $\beta_i > 0$. Let $I_1 = \{d+1 \leq i \leq 2d \mid \beta_i > 0\}$, $I_2 = \{d+1 \leq i \leq 2d \mid \beta_i < 0\}$, $\gamma = \min_{i \in I_1} \alpha_i/\beta_i$, and $\bar{C} = \{\mathbf{c}_i \mid i \in I_1, \alpha_i/\beta_i = \gamma\}$. We have $\sum_{i \in I_1 \cup I_2} \beta_i \mathbf{c}_i = \mathbf{0}$, and $\bar{C} \neq \emptyset$. Consider $\alpha' = (\alpha'_1, \ldots, \alpha'_r)$ where $\alpha'_i = \alpha_i$ ($i \notin I_1 \cup I_2$), $\alpha'_i = \alpha_i - \gamma\beta_i$ ($i \in I_1 \cup I_2$). It is easy to verify that $\alpha'_i \geq 0$ and $\alpha'_i = 0$ if and only if $\mathbf{c}_i \in \bar{C}$. It follows that $\sum_{\mathbf{c}_i \in C - \bar{C}} \alpha'_i \mathbf{c}_i = \sum_{\mathbf{c}_i \in C} \alpha_i \mathbf{c}_i - \gamma \sum_{i \in I_1 \cup I_2} \beta_i \mathbf{c}_i = \mathbf{0}$. Thus, the vectors in $C \setminus \bar{C}$ span R^d, and $\sum_{\mathbf{c}_i \in C - \bar{C}} \alpha'_i \mathbf{c}_i$ is a balancing combination.

The complexity is as follows. The first step amounts to computing d independent vectors $\{\mathbf{c}_1, \ldots, \mathbf{c}_d\}$, what can be done in $O(rd^2)$ time. In each iterative step we solve a system of linear equalities. This can be done in $O(d^3)$ operations, using Gaussian eliminations. Notice, however, that in two consecutive iterations, the linear systems have $d - |\bar{C}|$ columns in common. In such a case, when the matrix of the first problem is given in upper triangular form, the second system can be solved in only $O(|\bar{C}|d^2)$ operations. Thus, the total number of operations is $O(rd^2)$. ∎

Corollary 3.1.17 *Suppose $g = L_Y$ where ($Y \subset R^d$) is bounded from above. Assume we are given a set of vectors $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_r\}$ and a balancing combination of C such that (i) L_C is a weak approximation of g, and (ii) there exist $\lambda^* \in \operatorname{rel\,int} \Lambda$ such that for all $\mathbf{c} \in C$ we have $\mathbf{c}^T\lambda^* = g(\lambda^*)$. Under these conditions, in $O(rd^2)$ operations we can compute $C' \subset C$ such that $L_{C'}$ is a minimal weak approximation of g, and a balancing combination of C'. The size of such C' is at most* $2(d - \dim(\Lambda))$.

Proof: It follows from Corollary 3.1.10 and Propositions 3.1.11 that for all $\mathbf{c} \in C$, $\mathbf{c} \in (\operatorname{aff} \Lambda)^\perp$. It follows from Proposition 3.1.12 that the vectors C span the subspace

$(\operatorname{aff}\Lambda)^{\perp}$. To conclude the proof, note that the subspace $(\operatorname{aff}\Lambda)^{\perp}$ is of dimension $d - \dim(\Lambda)$. ∎

Proposition 3.1.18 *Suppose $g = L_C$ has the property that there exist a vector $\boldsymbol{\lambda}^* \in$ relint Λ such that $\mathbf{c}^T\boldsymbol{\lambda}^* = g(\Lambda)$ for all $\mathbf{c} \in C$. Suppose we are given:*

i. Hyperplanes H_δ $(\delta \in \{-1,0,1\})$, where $H_\delta = \{\boldsymbol{\lambda} \mid \boldsymbol{a}^T\boldsymbol{\lambda} = \alpha + \delta\}$ for some vector $\boldsymbol{a}$ and $\alpha \in R$, and H_0 contains $\boldsymbol{\lambda}^*$.

ii. Sets of vectors $C_\delta = \{\mathbf{c}_1^\delta, \ldots, \mathbf{c}_{r_\delta}^\delta\} \subset C$ and $\boldsymbol{\alpha}^\delta \in R_+^{r_\delta}$ $(\delta \in \{-1,0,1\})$ such that L_{C_δ} is a minimal weak approximation of $g|H_\delta$, and $\sum_{i=1}^{r_\delta} \alpha_i^\delta \mathbf{c}_i^\delta$ is a balancing combination of C_δ relative to H_δ $(\delta \in \{-1,0,1\})$.

By using $O(d^3 \sum_{\delta\in\{-1,0,1\}} r_\delta)$ operations we can compute a set of vectors $\bar{C}$ and a balancing combination of $\bar{C}$ relative to R_1^d, such that $\bar{C} \subset \bigcup_{\delta\in\{-1,0,1\}} C_\delta$ and $L_{\bar{C}}$ is a minimal weak approximation of g.

Proof: We give a description of an algorithm. In the first step the algorithm computes a set $C' \subset C$ of size at most $\sum_{\delta\in\{-1,0,1\}} r_\delta$ and a balancing combination of C', such that $L_{C'}$ is a weak approximation of g. This is done as follows.

i. If $\Lambda \subset H_0$, then $C' = \bigcup_{\delta\in\{0,1,-1\}} C_\delta$ is such that $L_{C'}$ is a weak approximation of g. To find a balancing combination, the algorithm computes $\beta_\delta > 0$ $(\delta \in \{-1,0,1\})$ such that $\sum_{\delta\in\{-1,0,1\}} \beta_\delta \sum_{i=1}^{r_\delta} \alpha_i^\delta \mathbf{c}_i^\delta = \mathbf{0}$. Otherwise,

ii. $H_\delta \cap \Lambda \neq \emptyset$ (for either $\delta = 1$ or $\delta = -1$). It turns out that the set $C' = C_\delta$ is such that $L_{C'}$ is a weak approximation of g. Hence, the algorithm chooses $\boldsymbol{\alpha}^\delta$ as the coefficients of a balancing combination.

Assuming that C' is a weak approximation of g the algorithm proceeds as follows. In case ii, $\bar{C} = C'$ is obviously a minimal weak approximation of g. In case i it follows from Corollary 3.1.17 that by using $O(d^3 \sum_{\delta\in\{-1,0,1\}} r_\delta)$ operations we can compute $\bar{C} \subset C$ and a balancing combination for $\bar{C}$, such that $L_{\bar{C}}$ is a minimal weak approximation of g.

What remains is to prove that $L_{C'}$ is indeed a weak approximation of g. Denote by $\Lambda_\delta \subset H_\delta$ the set of maximizers of g restricted to H_δ. By using an appropriate translation of the coordinates, we can transform the problem so that $C \subset R^{d-1}$, $g : R^{d-1} \to R$, $H_\delta = \{\boldsymbol{\lambda} \in R^{d-1} \mid \boldsymbol{a}^T\boldsymbol{\lambda} = \delta\}$, $\boldsymbol{\lambda}^* = \mathbf{0}$, and $g(\Lambda) = 0$. This transformation preserves the property of a subset being a weak approximation or having a balancing combination relative to a given flat. A balancing combination in the transformed problem corresponds to a balancing combination relative to R^d_1 in the original one. Note that in the transformed problem, $\mathbf{0} \in \Lambda$, so the flat aff Λ is a subspace.

Let $\boldsymbol{\lambda}^*_\delta \in \operatorname{relint} \Lambda_\delta$. Observe that g is linear, that is, for all $\alpha > 0$, $g(\alpha\boldsymbol{\lambda}) = \alpha g(\boldsymbol{\lambda})$. Thus, in case i $\Lambda = \Lambda_0$. In case ii consider the intersection of Λ with the open halfspace $\{\boldsymbol{\lambda} \mid \delta(\boldsymbol{a}^T\boldsymbol{\lambda}) > 0\}$. The set Λ is convex. Thus, if is not empty, it must contain a relative interior point of Λ. The hyperplane H_δ is contained in this halfspace and hence this intersection is not empty, and $\dim \Lambda_\delta = \dim \Lambda - 1$. Note that $\mathbf{0} \notin \operatorname{aff} \Lambda_\delta$. Thus, the flat $\operatorname{aff} \Lambda_\delta \oplus \{\beta\boldsymbol{\lambda}^*_\delta \mid \beta \in R\}$ must be contained in aff Λ. On the other hand this flat is of the same dimension as aff Λ and therefore equality holds. Consider any $\boldsymbol{\lambda}' \in R^{d-1}$ such that $\boldsymbol{\lambda}' \notin \operatorname{aff} \Lambda$. In order to prove that $L_{C'}$ weakly approximates g we need to show that there exists a $\boldsymbol{c} \in C'$ such that $\boldsymbol{c}^T\boldsymbol{\lambda}' < 0$.

First, we prove case i. If $\boldsymbol{a}^T\boldsymbol{\lambda}' = 0$ then we are done, since $\boldsymbol{\lambda} \in H_0$ and $C_0 \subset C'$. Otherwise, assume without loss of generality that $\boldsymbol{a}^T\boldsymbol{\lambda}' = \beta > 0$. Consider the vector $\tilde{\boldsymbol{\lambda}} = \boldsymbol{\lambda}/\beta$. The vector $\tilde{\boldsymbol{\lambda}}$ lies on the line determined by $\mathbf{0}$ and $\boldsymbol{\lambda}$, and is such that $\boldsymbol{a}^T\tilde{\boldsymbol{\lambda}} = 1$. Since aff Λ is a flat, if it contains two points on a line, it contains the whole line. Therefore, $\mathbf{0} \in \operatorname{aff}\Lambda$ and $\boldsymbol{\lambda}' \notin \operatorname{aff}\Lambda$ imply that $\tilde{\boldsymbol{\lambda}} \notin \operatorname{aff}\Lambda$. It follows that there exists a $\boldsymbol{c} \in C_1$ such that $\tilde{\boldsymbol{\lambda}}^T\boldsymbol{c} < 0$. Finally, observe that $\boldsymbol{\lambda}'^T\boldsymbol{c} = (\beta\tilde{\boldsymbol{\lambda}})^T\boldsymbol{c} < 0$.

To prove case ii, define $\boldsymbol{\lambda}''$ to be the vector $\boldsymbol{\lambda}'' = \boldsymbol{\lambda}' + \beta\boldsymbol{\lambda}^*_\delta$ where $\beta = (\delta - \boldsymbol{a}^T\boldsymbol{\lambda}')/(\boldsymbol{a}^T\boldsymbol{\lambda}^*_\delta)$. Note that $\boldsymbol{a}^T\boldsymbol{\lambda}'' = \delta$ and thus $\boldsymbol{\lambda}'' \in H_\delta$. Moreover, $\boldsymbol{\lambda}'' \notin \operatorname{aff}\Lambda_\delta$, since otherwise $\boldsymbol{\lambda}' \in \operatorname{aff}\Lambda$. Thus, there exists a $\boldsymbol{c} \in C_\delta$ such that $\boldsymbol{c}^T\boldsymbol{\lambda}'' < 0$. We have $\boldsymbol{\lambda}^*_\delta \in \operatorname{aff}\Lambda$ and therefore $\boldsymbol{\lambda}''$ has the same projection on $(\operatorname{aff}\Lambda)^\perp$ as $\boldsymbol{\lambda}'$. It follows from Proposition 3.1.11 that for all $\boldsymbol{c}' \in C_\delta$, $\boldsymbol{c}' \in (\operatorname{aff} \Lambda)^\perp$. Hence, $\boldsymbol{c}^T\boldsymbol{\lambda}' = \boldsymbol{c}^T\boldsymbol{\lambda}'' < 0$

To find a balancing combination, recall from Proposition 3.1.15 that there exist numbers $\gamma_1 > 0, \gamma_{-1} < 0$ such that $\sum_{i=1}^{r_\delta} \alpha_i^\delta \boldsymbol{c}_i^\delta = \gamma_\delta \boldsymbol{a}$. Hence, we choose $\beta_\delta = |\gamma_{1-\delta}|$, $\beta_0 = 1$. It is easy to verify that

$$\sum_{\delta \in \{-1,0,1\}} \beta_\delta \sum_{i=1}^{r_\delta} \alpha_i^\delta \boldsymbol{c}_i^\delta = \boldsymbol{0}\ .$$

∎

3.1.3 Back to zero-cycles

We are now ready to give the deferred proof from Chapter 2.

Proof of Proposition 2.4.9: Observe that the weight of a minimum weight cycle in a graph with the scalar weights $\alpha\boldsymbol{\lambda}^T\boldsymbol{f}$ $(\alpha > 0)$ is linear in α. If the graph G has a witness $\boldsymbol{\lambda}$ then either $\lambda_d = 0$ or the vector $(1/\lambda_d)\boldsymbol{\lambda}$ is a witness as well. Therefore, we can restrict our search on the $\boldsymbol{\lambda}$-space to vectors $\boldsymbol{\lambda} \in R^d$ where $\lambda_d \in \{-1, 0, 1\}$. Denote

$$\mathcal{K}_\delta = \mathcal{K} \cap \{\boldsymbol{y} : y_d = \delta\} \quad (\delta \in \{-1, 0, 1\})\ .$$

Run the parametric minimum cycle (resp., parametric minimum cycle-mean) algorithm on G as defined in Problem 3.1.7 (resp., Problem 3.1.5) three times with the following $(d-1)$-variable linear functions. For $\delta = -1, 0, 1$, we associate with the edges the linear functions $\boldsymbol{f}^T\boldsymbol{\lambda}_\delta$ where $\boldsymbol{\lambda}_\delta = (\lambda_1, \ldots, \lambda_{d-1}, \delta)^T$. If a witness for G exists, it must be found during at least one of the three runs. Otherwise, the maximum of the function g in each of the three subproblems is nonpositive. Thus, for all three instances, the corresponding sets [illegible] $(\delta \in \{-1, 0, 1\})$ consist of vectors for which $g = 0$. The algorithm computes nonzero vectors $\boldsymbol{\lambda}_\delta^* \in \operatorname{rel\,int} \mathcal{K}_\delta$, $\delta \in \{-1, 0, 1\}$ (if $\mathcal{K}_\delta \neq \boldsymbol{0}$), along with collections $C^{(\delta)}$ (of size $|C^{(\delta)}| = O(d)$) of cycles such that $L_{\{\boldsymbol{f}(C) \mid C \in C^{(\delta)}\}}$ is a minimal weak approximation of g restricted to the hyperplane $\lambda_d = \delta$, and $\boldsymbol{\alpha}^\delta$ are coefficients of balancing combinations of C^δ relative to $\lambda_d = \delta$. To find the separating vector $\boldsymbol{\lambda}$ we proceed as follows. (i) If $\mathcal{K}_\delta \neq \emptyset$ for

$\delta \in \{-1, 1\}$, we choose λ to be λ^*_δ, and then

$$\lambda = \lambda^*_\delta \in \operatorname{rel int} \mathcal{K}_\delta \subset \operatorname{rel int} \mathcal{K} \ .$$

(ii) If $\mathcal{K}_{-1} = \mathcal{K}_1 = \emptyset$ and $\mathcal{K}_0 \neq \{0\}$, we choose λ to be a nonzero vector $\lambda^*_0 \in \operatorname{rel int} \mathcal{K}_0$, and so

$$\lambda = \lambda^*_0 \in \operatorname{rel int} \mathcal{K}_0 = \operatorname{rel int} \mathcal{K} \ .$$

(iii) The remaining case is $\mathcal{K}_{-1} = \mathcal{K}_1 = \emptyset$ and $\mathcal{K}_0 = \{0\}$. Here we conclude that $\mathcal{K} = \{0\}$, so there is no separating vector.

As done in the proof of Proposition 3.1.18, using $O(d^3)$ operations, we can find a set $\mathcal{C}$ of cycles and a balancing combination for the cycle values $\{f(C) \mid C \in \mathcal{C}\}$, such that the lower envelope of the set of cycle values is a minimal weak approximation of the function g. To conclude the proof observe that the size of a minimal weak approximation is at most $O(2d)$ (see Corollary 3.1.17), and that cone$\mathcal{C} \supseteq$ ORTH(G) (see Corollary 2.4.11). ∎

3.2 Algorithm for parametric min cycle-mean

In this subsection we sketch a strongly polynomial time algorithm for the parametric minimum cycle-mean problem. As mentioned before, this algorithm is simpler than the parametric minimum cycle algorithm. However, its time bounds are worse. The purpose of discussing it here is to give the reader some intuition and explain some of the main ideas behind the strongly polynomial bounds. Thus, only the ideas that are essential for strongly polynomial time bounds are introduced. Two simplifications are made. First, the multi-dimensional search technique is not discussed. It improves the time complexity, but is not essential for strongly polynomial bounds. The second simplification is that the algorithm only decides the existence of a zero-cycle. Therefore, the collection $\mathcal{C}$ (see Problem 2.4.7) need not be computed, and it suffices to be able to compute a witness or a separating vector.

Remark 3.2.1 The minimum cycle-mean relative to a set of scalar weights can be found in $O(|E|\cdot|V|)$ time by an algorithm due to Karp [38]. With n^3 parallel processors, the minimum cycle-mean can be computed in $O(\log^5 n)$ time (see [45]).

Proposition 3.2.2 *The value of $g(\boldsymbol{\lambda})$ can be described as*

$$g(\boldsymbol{\lambda}) = \max_{\pi_1,\ldots,\pi_n} \min_{i,j} \left(-\pi_i + \pi_j - \boldsymbol{\lambda}^T \mathbf{c}_{ij}\right) .$$

Proof: Given $\boldsymbol{\lambda} \in R^d_1$, consider the following linear programming problem:

$$(P) \quad \begin{array}{ll} \text{Minimize} & \sum_{i,j} (\boldsymbol{\lambda}^T \mathbf{c}_{ij}) x_{ij} \\ \text{subject to} & \sum_j (x_{ij} - x_{ji}) = 0 \qquad (i = 1,\ldots,n) \\ & \sum_{i,j} x_{ij} = 1 \\ & \boldsymbol{x} \geq \mathbf{0} \, . \end{array}$$

Obviously, (P) has an optimal solution. Now, every feasible solution $\boldsymbol{x}$ of (P) is a convex combination of feasible solutions $\boldsymbol{x}'$, where $E(\boldsymbol{x}')$ is a simple cycle. Since $\sum x'_{ij} = 1$, the value of the objective function at each $\boldsymbol{x}'$ is precisely the mean weight of the cycle determined by $\boldsymbol{x}'$. Hence (P) has an optimal solution of the form $\boldsymbol{x}'$. In other words, the optimal value of (P) is equal to $g(\boldsymbol{\lambda})$. Consider the dual of (P):

$$(D) \quad \begin{array}{ll} \underset{\pi,t}{\text{Maximize}} & t \\ \text{subject to} & \pi_i - \pi_j + t \leq -\boldsymbol{\lambda}^T \mathbf{c}_{ij} \, . \end{array}$$

Our claim follows from the fact that the optimal value of (D) is equal to $g(\boldsymbol{\lambda})$. ∎

Corollary 3.2.3 *The problem of maximizing $g(\boldsymbol{\lambda})$ can be formulated as a linear programming problem:*

$$(\tilde{P}) \quad \begin{array}{ll} \underset{\pi,t,\lambda}{\text{Maximize}} & t \\ \text{subject to} & \pi_i - \pi_j + \boldsymbol{\lambda}^T \mathbf{c}_{ij} + t \leq 0 \, . \end{array}$$

The dual of $(\tilde{P})$ *is the following zero-circulation problem:*

$$(\tilde{D}) \quad \begin{array}{ll} \text{Minimize} & \sum_{i,j} d_{ij} x_{ij} \\ \text{subject to} & \sum_{j} (x_{ij} - x_{ji}) = 0 \qquad (i = 1, \ldots, n) \\ & \sum_{i,j} x_{ij} = 1 \\ & \sum_{i,j} x_{ij} c'_{ij} = \mathbf{0} \\ & \mathbf{x} \geq \mathbf{0} \; . \end{array}$$

Where $\mathbf{c}_{ij} = (\mathbf{c}'_{ij}, d_{ij})$, $\mathbf{c}'_{ij} \in Q^{d-1}$.

Remark 3.2.4 In view of Corollary 3.2.3, the problem of maximizing $g(\boldsymbol{\lambda})$ can be solved by an extension of the algorithm for linear programming in fixed dimension [47] and its improvements [5, 19]. Note that only the dimension of the $\boldsymbol{\lambda}$-space is fixed, whereas the number of π_i's varies. However, we will work recursively on the space of $\boldsymbol{\lambda}$'s where at the base of the recursion ($d = 1$) we have a problem of the form of (D), i.e., a problem with fixed scalar-weights which is equivalent to the usual minimum cycle-mean problem (with no parameters).

Before discussing the algorithm for Problem 3.1.5 we present the respective "oracle" algorithm.

Problem 3.2.5 Given $G = (V, E, \mathbf{f})$ and a hyperplane H in the $\boldsymbol{\lambda}$-space, solve Problem 3.1.8 for g relative to H.

Problem 3.2.5 (an oracle call) is solved by recursive calls to an algorithm for Problem 3.1.5 on input graphs with vector weights of a lower dimension as we argue below.

Theorem 3.2.6 *Problem 3.2.5 can be solved by three recursive calls to an algorithm for Problem 3.1.5 in dimension* $d - 1$. *The complexity of the additional computation is dominated by the calls to Problem 3.1.5.*

Proof: Consider Problem 3.1.5 subject to $\boldsymbol{a}^T\boldsymbol{\lambda} = \alpha$ (and $\boldsymbol{\lambda} \in R_1^d$). This is in fact a $(d-1)$-dimensional version of the original problem restricted to $H = \{\boldsymbol{\lambda} \in R_1^d | \boldsymbol{a}^T\boldsymbol{\lambda} = \alpha\}$. If g is unbounded on H, then this fact is detected; otherwise, suppose $\boldsymbol{\lambda}^{(0)}$ is in the relative interior of the set of maximizers of $g(\boldsymbol{\lambda})$ subject to $\boldsymbol{\lambda} \in H$. Suppose we also have corresponding values $\pi_1^0, \ldots, \pi_n^0$ and $t^{(0)} = g(\boldsymbol{\lambda}^{(0)})$. We wish to recognize whether $\boldsymbol{\lambda}^{(0)}$ is also a relative interior point of the set of global maxima (i.e., relative to R_1^d). If not, we wish to decide whether for all $\boldsymbol{\lambda}^* \in R_1^d$ such that $g(\boldsymbol{\lambda}^*) \geq g(\boldsymbol{\lambda}^{(0)})$, necessarily $\boldsymbol{a}^T\boldsymbol{\lambda}^* > \alpha$, or whether for all of them $\boldsymbol{a}^T\boldsymbol{\lambda}^* < \alpha$; these are all the possible cases. Let E^0 denote the set of edges (i,j) such that

$$-\pi_i^0 + \pi_j^0 - (\boldsymbol{\lambda}^0)^T\boldsymbol{c}_{ij} = t^0 \ .$$

Notice that the subgraph G^0 of G induced by E^0 contains all the cycles of G whose cycle-mean is minimum relatively to the weights $(\boldsymbol{\lambda}^0)^T\boldsymbol{c}_{ij}$, and only such cycles. In order to determine whether $\boldsymbol{\lambda}^0$ is in the relative interior of the global maxima (and if not determine which side of H contains it), we can consider the behavior of g on two hyperplanes close to H. These hyperplanes are parallel to H, one on each side. We consider the local maxima relative to these planes. In order to avoid the problem of how close to get, we can consider just the pieces of g which are active at $\boldsymbol{\lambda}^0$. This is equivalent to considering only cycles whose cycle-mean is minimum relative to $(\boldsymbol{\lambda}^0)^T\boldsymbol{c}_{ij}$, namely, the cycles of the subgraph G^0. We now solve two problems on G^0 of maximizing t subject to

$$\pi_i - \pi_j + \boldsymbol{\lambda}^T\boldsymbol{c}_{ij} + t \leq 0 \qquad ((i,j) \in E^0) \ ,$$

where in one of the problems we also include the constraint $\boldsymbol{a}^T\boldsymbol{\lambda} = \alpha - 1$, and in the other we include the constraint $\boldsymbol{a}^T\boldsymbol{\lambda} = \alpha + 1$. Both problems can be solved as $(d-1)$-dimensional problems since one of the λ_i's can be eliminated. Denote the optimal values of these problems by $t^{(1)}$ and $t^{(-1)}$. Only one of the optimal values can be greater than $t^{(0)}$. If this is the case, or if one of $t^{(1)}, t^{(-1)}$ equals $t^{(0)}$ and the other is smaller, then the side of the hyperplane that contains $\operatorname{rel\,int}\Lambda$ is determined. Otherwise, if either both are less than, or both are equal to $t^{(0)}$, then $t^{(0)}$ is the global optimal value. The base of the recursion is the 1-dimensional problem (with no parameters at all). ∎

A sketch of the algorithm for Problem 3.1.5 is given below.

Algorithm 3.2.7 Consider a minimum cycle-mean algorithm that uses only comparisons, additions, and divisions (multiplications) by scalars as primitive operations. Define the corresponding *lifted* algorithm to operate on edge weights that are linear functions, defined by $f(e)$ $(e \in E)$. The lifted algorithm maintains a collection $\mathcal{H}$ of open halfspaces which is initialized as the empty set. Additions and divisions by scalars are naturally "lifted" to additions and scalar divisions of linear functions. Comparisons are more intricate. To "compare" two linear functions, we compute the hyperplane H such that the result of a comparison is uniform throughout each of the halfspaces defines by H. We then perform an oracle call on H. If the oracle call did not result in a global solution, a halfspace h is found (one of the sides of H) which contains the maxima of g. The result of the comparison is the relation between the linear functions that holds throughout h. The halfspace h is then added to $\mathcal{H}$ and the algorithm continues. If the algorithm terminates and no oracle call resulted in a solution, a separating vector (or a witness) is found by considering a point in the intersection of all the open halfspaces in $\mathcal{H}$.

3.3 Algorithm for parametric minimum cycle

This section introduces an algorithm for the parametric minimum cycle problem. This problem is essentially an instance of Problem 3.1.3 for a particular family of functions. In Chapter 4 we use the general lines of this algorithm to solve Problem 3.1.3 for any concave function g given by a piecewise affine algorithm.

We suggest an oracle algorithm which relies on making recursive calls to instances of Problem 3.1.7, where there are fewer parameters, but the weights are more complex.

In order to facilitate the recursion, the problem is generalized to the *extended* parametric minimum cycle problem (EPMC). The algorithm presented here solves the EPMC problem. Let us start with a definition of this problem.

Extended parametric minimum cycle. Let $G = (V, E, \boldsymbol{w}, \boldsymbol{f})$ be a graph with two sets of vectors associated with the edges, i.e., for every $e \in E$, $\boldsymbol{f}(e) \in R^d$ and $\boldsymbol{w}(e) \in R^\ell$. We identify $\boldsymbol{f}(e)$ with the $(d-1)$-dimensional linear function:

$$\boldsymbol{f}(e) = f_1(e)\lambda_1 + \cdots + f_{d-1}(e)\lambda_{d-1} + f_d(e) \ .$$

The *weight* of an edge e is the $(\ell+1)$-tuple $(w_1(e), \ldots, w_\ell(e), \boldsymbol{f}(e))$. The definitions given in 3.1.6 are extended as follows.

Definition 3.3.1 Consider a graph $G = (V, E, \boldsymbol{w}, \boldsymbol{f})$, where for $e \in E$, $\boldsymbol{f}(e) = \boldsymbol{c}_e \in R^d$.

i. A cycle C (not necessarily simple) is called ***w**-minimal* if the value $\boldsymbol{w}(C) = \sum_{e \in C} \boldsymbol{w}(e)$ (where edges are counted as many times as they occur on the cycle) is minimal relative to the lexicographic order on R^ℓ.

ii. Let $C = C(\boldsymbol{\lambda})$ denote a $\boldsymbol{w}$-minimal cycle of at most n edges which minimizes the weight $\sum_{e \in C} \boldsymbol{\lambda}^T \boldsymbol{c}_e$. Denote $g(\boldsymbol{\lambda}) = \boldsymbol{f}(C)^T \boldsymbol{\lambda}$. Note that the first ℓ coordinates of the vector weight of a minimum cycle (i.e., the values given by $\boldsymbol{w}$) are independent of $\boldsymbol{\lambda}$; $g(\boldsymbol{\lambda})$ gives only the $(\ell+1)$st coordinate.

Remark 3.3.2 For a graph with vector weights in R^ℓ, the minimum, relative to the lexicographic order, among cycles of length less than or equal to n can be computed in one application of an all-pairs shortest path algorithm. This takes $O(\ell|E| \cdot |V|)$ time sequentially, or $O(\log^2 n)$ time using ℓn^3 processors in parallel. Therefore, the function $g(\boldsymbol{\lambda})$ can be evaluated by a piecewise affine algorithm (see Definition 3.1.2). Note that $g(\boldsymbol{\lambda}) = L_{\bar{C}}$, where $\bar{C} = \{\boldsymbol{f}(C) | C \text{ is a } \boldsymbol{w}\text{-minimal cycle}\}$.

Problem 3.3.3 [Extended Parametric Minimum Cycle]
Given is $G = (V, E, \boldsymbol{w}, \boldsymbol{f})$ as above. If $g(\boldsymbol{\lambda}) > 0$ for some $\boldsymbol{\lambda}$, then output any such $\boldsymbol{\lambda}$; otherwise, find $\boldsymbol{\lambda} \in \operatorname{relint} \Lambda$ and a collection $\mathcal{C} = \{C_1, \ldots, C_r\}$ $(r \le 2d)$ of $\boldsymbol{w}$-minimal cycles, where each consists of at most n edges, such that the lower envelope $L_{\{\boldsymbol{f}(C) | C \in \mathcal{C}\}}$ of their cycle values is a minimal weak approximation of g, and find a balancing combination of the cycle values $\boldsymbol{f}(C_1), \ldots, \boldsymbol{f}(C_r)$ relative to R^d_1.

The EPMC algorithm presented below performs "oracle" calls. The oracle algorithm recursively solves instances of the EPMC problem on G with sets of vector weights $\boldsymbol{w}' : E \to R^{\ell+1}$ and $\boldsymbol{f}' : E \to R^{d-1}$. The dimension of the $\boldsymbol{f}$ weights, which corresponds to the number of parameters, decreases in the recursive calls. In order to solve an instance of the parametric minimum cycle problem (Problem 3.1.7), we start with an instance of EPMC where $\boldsymbol{f}$ gives the vector edge weights and $\boldsymbol{w}$ is a set of null vectors. At the base of the recursion the weights $\boldsymbol{f}$ are null vectors, and the problem is reduced to the non-parametric problem of computing minimum cycle relative to the lexicographic order on d-tuples.

In Subsection 3.3.3 we propose Algorithm 3.3.14 for Problem 3.3.3. The algorithm executes calls to the oracle problem (Problem 3.1.8) relative to g. An algorithm for the oracle problem is given in Subsection 3.3.1. A call to the oracle is a costly operation. Therefore, one wishes to solve many hyperplane queries with a small number of oracle calls. In Subsection 3.3.2 we discuss the multi-dimensional search technique (introduced in [47]). By applying it, we are able to reduces the number of oracle calls performed to a polylog in the number of hyperplane queries.

3.3.1 Hyperplane queries

For a given a hyperplane H of R^d_1, we wish to solve Problem 3.1.8 for g relative to H. If $H \cap \operatorname{relint} \Lambda \neq \emptyset$, we solve Problem 3.3.3, that is, we find $\boldsymbol{\lambda} \in \operatorname{relint} \Lambda$, the collection $\mathcal{C}$, and a balancing combination of $\mathcal{C}$.

Problem 3.3.4 Given is $G = (V, E, \boldsymbol{w}, \boldsymbol{f})$ and a hyperplane $H = \{\boldsymbol{\lambda} \in R^d_1 \mid \boldsymbol{a}\boldsymbol{\lambda} = \alpha\}$ in the $\boldsymbol{\lambda}$-space. Solve Problem 3.1.8 for g relative to H.

Theorem 3.3.5 *Problem 3.3.4 can be solved by an algorithm which performs three calls to instances of Problem 3.3.3 where $\boldsymbol{f}$ is $(d-1)$-dimensional. The time complexity of the additional computation is dominated by these calls: It can be done sequentially in $C|E| + D$ time for some constants $C = O(d)$ and $D = O(d^3)$. In parallel, it can be done in constant $B = O(d^3)$ time using $O(m + n^3)$ processors.*

Proof: Consider Problem 3.3.3 subject to $\boldsymbol{a}^T\boldsymbol{\lambda} = \alpha$ and $\boldsymbol{\lambda} \in R_1^d$. This is in fact a $(d-1)$-dimensional version of the original problem restricted to the hyperplane $H = \{\boldsymbol{\lambda} \in R_1^d \mid \boldsymbol{a}^T\boldsymbol{\lambda} = \alpha\}$. If g is unbounded on H, then this fact is detected; otherwise, suppose $\boldsymbol{\lambda}^{(0)}$ is in the relative interior of the set of maximizers of $g(\boldsymbol{\lambda})$ subject to $\boldsymbol{\lambda} \in H$, and we get the collection $C^{(0)}$ and a balancing combination of $C^{(0)}$ relative to H. Suppose $t^{(0)} = g(\boldsymbol{\lambda}^{(0)})$. We wish to recognize whether $\boldsymbol{\lambda}^{(0)}$ is also a relative interior point of the set of global maxima (i.e., relative to R_1^d). If not, then we wish to decide whether for all $\boldsymbol{\lambda}^* \in R_1^d$ such that $g(\boldsymbol{\lambda}^*) \geq g(\boldsymbol{\lambda}^{(0)})$, necessarily $\boldsymbol{a}^T\boldsymbol{\lambda}^* > \alpha$, or whether for all of them $\boldsymbol{a}^T\boldsymbol{\lambda}^* < \alpha$. These are all the possible cases. Consider $G' = (V, E, \boldsymbol{w}', \boldsymbol{f})$, where $\boldsymbol{w}' = (\boldsymbol{w}, \boldsymbol{f}^T\boldsymbol{\lambda}^{(0)})$. Note that all the minimum cycles of G' correspond to minimum cycles with value $t^{(0)}$ at $\boldsymbol{\lambda}$ of G. We solve Problem 3.3.3 twice on G', where in one of the problems we also include the constraint $\boldsymbol{a}^T\boldsymbol{\lambda} = \alpha - 1$, and in the other we include the constraint $\boldsymbol{a}^T\boldsymbol{\lambda} = \alpha + 1$. Both problems can be solved as $(d-1)$-dimensional problems since one of the λ_i's can be eliminated. Denote the optimal values of these problems by $t^{(\delta)}$, the corresponding collections of cycles by C^δ, and let $\boldsymbol{\alpha}^\delta$ be coefficients of a balancing combination of C^δ ($\delta \in \{-1, 1\}$). Only one of the optimal values can be greater than $t^{(0)}$. If this is the case, or if one of $t^{(1)}, t^{(-1)}$ equals $t^{(0)}$ and the other is smaller, then the side of the hyperplane that contains relint Λ is determined. Otherwise, if either both are less than, or both are equal to $t^{(0)}$, then $t^{(0)}$ is the global optimal value. In the latter case $\boldsymbol{\lambda}^{(0)} \in \operatorname{relint} \Lambda$. It follows from Proposition 3.1.9 that the pieces of g which are active in a minimal weak approximation have the value $t^{(0)}$ at $\boldsymbol{\lambda}^{(0)}$. Thus, a minimal weak approximation of the function g' (see Definition 3.3.1) which corresponds to G' is a minimal weak approximation of the "original" g which corresponds to the graph G. The conditions of Proposition 3.1.18 and Corollary 3.1.17 hold for the function g'. Hence, in $O(d^3)$ operations we can construct a minimal weak approximation of g, and find a corresponding balancing combination. The base of the recursion is the one-dimensional problem (with no parameters at all) where C consists of a single cycle with minimal value, and the balancing combination is the value of this cycle. ∎

3.3.2 Employing multi-dimensional search

The multi-dimensional search problem was defined and used in [47] for solving linear programming problems in fixed dimension.

Problem 3.3.6 [multi-dimensional search]
Suppose there exists an unknown convex set $X \subseteq R^d$, and an oracle is available such that for any query hyperplane H in R^d, the oracle tells whether $X \cap H = \emptyset$; if so, then the oracle tells which of the open halfspaces determined by H contains X. For m given query hyperplanes, determine the location of X relative to each of the hyperplanes, or find any hyperplane (not necessarily one of the given ones) which intersects X.

The following theorem was proven in [47]:

Theorem 3.3.7 *The solution of Problem 3.3.6 relative to some $m/2$ of the given hyperplanes can be found by using $\gamma \equiv \gamma(d)$ oracle calls. The additional computation can be done sequentially in $O(\gamma(d)m)$ time, and on m parallel processors in $O(\gamma(d)\log m)$ time. From [5, 19] $\gamma(d) = O((5/9)^d 3^{d^2})$.*

Corollary 3.3.8 *Problem 3.3.6 can be solved by using $O(\gamma(d)\log m)$ oracle calls and $O(\gamma(d)m)$ additional time (see [47]).*

Remark 3.3.9 The procedure described in [47] can be parallelized so that the additional computation (besides the $O(\gamma(d)\log m)$ oracle calls) is done by m parallel processors in $O(\gamma(d)\log^2 m)$ time.

Definition 3.3.10 We define a partial order on $R^d \setminus \{0\}$ as follows. For any pair of distinct vectors $\boldsymbol{a}_1, \boldsymbol{a}_2 \in R^d$, denote

$$H = H(\boldsymbol{a}_1, \boldsymbol{a}_2) = \{\boldsymbol{\lambda} \in R^d_1 : \boldsymbol{a}_1^T \boldsymbol{\lambda} = \boldsymbol{a}_2^T \boldsymbol{\lambda}\} .$$

If g is unbounded on $H(\boldsymbol{a}_1, \boldsymbol{a}_2)$ or if $H(\boldsymbol{a}_1, \boldsymbol{a}_2) \cap \text{rel int}\, \Lambda \neq \emptyset$, then we write $\boldsymbol{a}_1 <>_\Lambda \boldsymbol{a}_2$. Otherwise, g can be unbounded on at most one of the open subspaces determined by

H, and also relint Λ can intersect at most one of these open halfspaces. We denote $a_1 <_\Lambda a_2$ (respectively, $a_1 >_\Lambda a_2$) if there exists a $\lambda \in$ relint Λ such that $a_1^T\lambda < a_2^T\lambda$ (respectively, $a_1^T\lambda > a_2^T\lambda$), in which case the same holds for all these λ's, or if g is unbounded on the halfspace determined by the inequality $a_1^T\lambda < a_2^T\lambda$ (respectively, $a_1^T\lambda < a_2^T\lambda$). We also use the notation $<_P$ for a similar partial order relative to any set P.

Problem 3.3.11 Given are finite sets $A_1, \ldots, A_r$ of nonzero vectors, where $A_i = \{a_1^i, \ldots, a_{s_i}^i\}$ ($a_j^i \in R^d$) and $s = \sum s_i$. We wish either to find a minimal element, with respect to the partial order $<_\Lambda$, in each of the sets A_i, or (if we encounter two incomparable elements) to reduce the problem to a lower dimension. More specifically, we need to find either one of the following:

i. A collection of closed halfspaces whose intersection P contains relint Λ, and indices $1 \le m_i \le s_i$ ($i = 1, \ldots, r$) such that for every $1 \le i \le r$ and every $1 \le j \le s_i$, $j \ne m_i$, we have $a_{m_i}^i <_\Lambda a_j^i$ and $a_{m_i}^i <_P a_j^i$.

ii. A hyperplane H such that either g is unbounded on H or $H \cap$ relint $\Lambda \ne \emptyset$.

Proposition 3.3.12 *Problem 3.3.11 can be solved using $O(\gamma(d-1)\log s)$ oracle calls plus either*

i. $O(\gamma(d-1)\log^2 s)$ parallel time on $O(s)$ processors, or

ii. $O(\gamma(d-1)s\log s)$ sequential time.

(Where $\gamma(d-1)$ is as in Theorem 3.3.7.)

Proof: The underlying algorithm is an extension of the multi-dimensional search procedure for Problem 3.3.6 mentioned in Theorem 3.3.7. Here the set X is either relint Λ or (if the latter is empty) a domain where the function g is unbounded. Thus, the case where X intersects the query hyperplane corresponds to the case where either g is unbounded on the query hyperplane, or the latter intersects rel int Λ; the case where X is contained in one of the open halfspaces corresponds to the case where either rel int Λ is contained in the halfspace, or g is unbounded on the halfspace (but bounded on the hyperplane). Also, note that the flat R_1^d on which the multi-dimensional search is done, is of dimension $d-1$.

We first explain how to recognize for a given pair of distinct vectors $\boldsymbol{a}_i, \boldsymbol{a}_j$ whether $\boldsymbol{a}_1 <_\Lambda \boldsymbol{a}_2$, $\boldsymbol{a}_2 <_\Lambda \boldsymbol{a}_1$ or $\boldsymbol{a}_1 <>_\Lambda \boldsymbol{a}_2$. Consider the hyperplane $H = H(\boldsymbol{a}_i, \boldsymbol{a}_j) \subset R_1^d$ (see Definition 3.3.10). Suppose H is the query hyperplane presented to an oracle which recognizes the location of the set rel int Λ relative to H (in the sense of Problem 3.3.4). In particular, if $H \cap \text{rel int}\,\Lambda \neq \emptyset$, then the oracle discovers this fact and returns a $\boldsymbol{\lambda} \in H \cap \text{rel int}\,\Lambda$. Similarly, if g is unbounded on H, then the oracle reports this fact and provides a $\boldsymbol{\lambda}$ such that $g(\boldsymbol{\lambda}) > 0$. In the remaining cases the oracle reports either $\boldsymbol{a}_1 <_\Lambda \boldsymbol{a}_2$ or $\boldsymbol{a}_2 <_\Lambda \boldsymbol{a}_1$.

The multi-dimensional search algorithm computes, adaptively, $O(\gamma(d-1)\log s)$ hyperplane queries. If any of these hyperplanes either intersects rel int Λ or has g unbounded on it, then this fact is reported, and the present problem is considered solved. Otherwise, each of the hyperplanes determines a closed halfspace such that the intersection P of all these halfspaces has the following property: either g is unbounded on the interior of P, or rel int Λ is nonempty and contained in the interior of P. Moreover, vectors $\boldsymbol{a}_{m_1}, \ldots, \boldsymbol{a}_{m_r}$ are found, such that for every i, $\boldsymbol{a}^i_{m_i} <_P \boldsymbol{a}^i_j$, for all $1 \leq j \leq s_i, j \neq m_i$.

We implement the algorithm as follows. View the dimension d as fixed. It was shown in [5] and [19] that by using a constant number of oracle calls (which, however, grows exponentially with the dimension) one can locate X relative to at least half of the hyperplanes. A similar scheme can be applied here. We apply $O(\log s)$ phases. First, for each i $(1 \leq i \leq r)$ we match the members of A_i into $s_i/2$ arbitrary pairs. This is done with at most $s/2$ processors. We then calculate the corresponding (at most $s/2$) hyperplanes $H(\boldsymbol{a}_i, \boldsymbol{a}_j)$ (see Definition 3.3.10). In a constant number $\gamma(d-1)$ of oracle calls and $O(\log s)$ time we can locate rel int Λ relative to half of these $s/2$ hyperplanes; unless one of these hyperplanes turns out to be a valid output (in the sense of (ii) in Problem 3.3.11). We now drop one vector from every pair for which the location relative to $\operatorname{rel\,int} \Lambda$ has been found. The same is repeated with the remaining $3s/4$ vectors, and so on. Altogether, we run in $O(\log s)$ phases, each of which takes $O(\gamma(d-1)\log s)$ time on $O(s)$ processors, and $O(\gamma(d-1))$ oracle calls. The sequential time bound is $O(\gamma(d-1)s\log s)$ plus $O(\gamma(d-1)\log s)$ oracle calls. In parallel, using $O(s)$ processors, the time is $O(\gamma(d-1)\log^2 s)$ plus $O(\gamma(d-1)\log s)$ oracle calls. ∎

3.3.3 Algorithm for extended parametric minimum cycle

The algorithm described below solves Problem 3.3.3. It finds a vector $\boldsymbol{\lambda} \in \operatorname{rel\,int} \Lambda$, unless $g(\boldsymbol{\lambda}) > 0$ for some $\boldsymbol{\lambda}$, in which case the algorithm outputs such a $\boldsymbol{\lambda}$. It also returns a collection C of $\boldsymbol{w}$-minimal cycles such that the lower envelope $L_{\{\boldsymbol{f}(C)|\ C \in C\}}$ of the linear functions defined by $\boldsymbol{f}(C)$ is a minimal weak approximation of g. The number of cycles in C is at most $2d$.

Definition 3.3.13 Consider a scalar minimum cycle algorithm, where the only primitive operations on expressions that depend on the edge weights are additions, multiplications by scalars, and comparisons. We define the corresponding ***lifted algorithm*** for input graphs of the form $G = (V, E, \boldsymbol{f}, \boldsymbol{w})$. The weight of an edge $e \in E$ on these input graphs is an $(\ell+1)$-tuple $(w_1(e), \ldots, w_\ell(e), \boldsymbol{f}(e))$, where $\boldsymbol{f}(e) \in R^d$ is viewed as a $(d-1)$-dimensional linear function and $\boldsymbol{w}(e) \equiv (w_1(e), \ldots, w_\ell(e))$. The lifted algorithm is an extension of the scalar minimum cycle algorithm that operates on such $(\ell+1)$-tuples instead of scalars. The extension of the operations of addition and multiplication by a scalar is straightforward, namely, given tuples $(\boldsymbol{w}_1, \boldsymbol{f}_1)$, $(\boldsymbol{w}_2, \boldsymbol{f}_2)$ their sum is $(\boldsymbol{w}_1 + \boldsymbol{w}_2, \boldsymbol{f}_1 + \boldsymbol{f}_2)$, and the multiplication by a scalar α is $(\alpha\boldsymbol{w}_1, \alpha\boldsymbol{f}_1)$. Comparisons are made with respect to a lexicographic partial order on the $(\ell+1)$-tuples. It is only a partial order since in the $(\ell+1)$st coordinate we have the partial order $<_\Lambda$ (see Definition 3.3.10). To compare two $(\ell+1)$-tuples $(\boldsymbol{w}_1, \boldsymbol{f}_1)$ and $(\boldsymbol{w}_2, \boldsymbol{f}_2)$, we first compare lexicographically the first ℓ scalar coordinates. If the comparison is not resolved there, we need to compare the linear functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$. For this purpose, the lifted algorithm computes the hyperplane $H(\boldsymbol{f}_1, \boldsymbol{f}_2)$ and solves Problem 3.3.4 (hyperplane query) relative to H. A set of hyperplane queries is resolved by performing "oracle" calls (see Subsection 3.3.2). The lifted algorithm maintains a set $\mathcal{H}$ of closed halfspaces which initially is empty. The hyperplane query decides whether or not the vectors are comparable. If they are, it decides whether $\boldsymbol{f}_1 <_\Lambda \boldsymbol{f}_2$. If $\boldsymbol{f}_1 <>_\Lambda \boldsymbol{f}_2$, then the lifted algorithm halts since an oracle call resulted in a solution to Problem 3.3.3. Otherwise, the resolved hyperplane query tells us which of the halfspaces defined by $H(\boldsymbol{f}_1, \boldsymbol{f}_2)$ (see Definition 3.3.10) contains the set $\operatorname{relint} \Lambda$, so this hyperplane is added to $\mathcal{H}$.

Algorithm 3.3.14 [Extended parametric minimum cycle]

Step 1. Run the lifted minimum cycle algorithm, collecting into $\mathcal{H}$ all the halfspaces resulting from oracle calls where comparisons are resolved. Either some oracle call resulted in a global solution, or otherwise, the algorithm terminates normally. Denote by C_M the minimum cycle found.

Step 2. Denote by P the intersection of the halfspaces in $\mathcal{H}$.

i. Compute $\boldsymbol{\lambda}^* \in \operatorname{relint} P$. This amounts to a linear programming problemwith $d-1$ variables and $|\mathcal{H}|$ constraints, and hence it can be solved in $O(|\mathcal{H}|)$ sequential time [47]. Note that the size of $\mathcal{H}$ is bounded by the number of oracle calls.

ii. If the function $L_{\boldsymbol{f}(C_M)}$ is not constant on R_1^d, that is, not all the coefficients $f_1(C_M), f_2(C_M), \ldots, f_{d-1}(C_M)$ equal zero, then g is unbounded. Otherwise,

iii. consider $g(\boldsymbol{\lambda}^*) = f_d(C_M)$.

- If $g(\boldsymbol{\lambda}^*) > 0$, then output $\boldsymbol{\lambda}^*$ and stop. Otherwise,
- the function $L_{\boldsymbol{f}(C_M)}$ is a weak approximation of g, and $P = \Lambda$. Hence, $\boldsymbol{\lambda}^* \in \operatorname{relint} \Lambda$. Output $\boldsymbol{\lambda}^*$ and $C = \{C_M\}$.

3.3.4 Correctness

If an oracle call results in a solution in Step 1 of Algorithm 3.3.14, then correctness follows by induction on the dimension (see also the discussion under Hyperplane Queries). We now assume that no oracle call resulted in a solution in Step 1. In this case, a collection $\mathcal{H}$ of closed halfspaces is obtained. Recall that if an oracle call on a hyperplane H did not result in a solution then the returned halfspace h has the following properties: (i) if the function g is bounded then $\Lambda \subset h$ but $\Lambda \not\subset H$, (ii) if the function g is unbounded, then it must be bounded on the hyperplane H, and unbounded on the halfspace h. Let P be the polyhedron $P = \bigcap_{h \in \mathcal{H}} h$. It follows that if g is bounded then $P \supset \Lambda$, and if g is unbounded then it must be bounded outside and on the boundary of P. Note that P must be full dimensional ($\dim P = d-1$), for if not then it must be contained in one of the query hyperplanes, which contradicts the previous statement.

Observe that for all pairs $\boldsymbol{a}_1, \boldsymbol{a}_2$ of vectors compared by the lifted minimum cycle algorithm, one of the following must hold: either $\boldsymbol{a}_1 <_\Lambda \boldsymbol{a}_2$ and $\boldsymbol{a}_1 <_P \boldsymbol{a}_2$, or $\boldsymbol{a}_2 <_\Lambda \boldsymbol{a}_1$ and $\boldsymbol{a}_2 <_P \boldsymbol{a}_1$. The latter is obvious when we perform an oracle call for each hyperplane query, and it is easy to see that it still holds when we employ the multidimensional search technique (see Problem 3.3.11 and Proposition 3.3.12) and solve these hyperplane queries by a smaller number of oracle calls. Thus, the vector value $\boldsymbol{f}(C_M)$ of the minimum cycle found by the algorithm must be such that $\boldsymbol{f}(C_M) <_\Lambda \boldsymbol{f}(C)$ and hence $\boldsymbol{f}(C_M) <_P \boldsymbol{f}(C)$ for any $\boldsymbol{w}$-minimal cycle C. It follows that $g(\boldsymbol{\lambda}) = \boldsymbol{f}(C_M)^T\boldsymbol{\lambda}$ for all $\boldsymbol{\lambda} \in P$. Thus, g is unbounded if and only if $\boldsymbol{f}(C_M)$ is not a constant, and the correctness of step ii follows. To show the correctness of step iii assume that $\boldsymbol{f}(C_M)$ is a constant, and thus $g = f_d(\boldsymbol{\lambda})$ for all $\boldsymbol{\lambda} \in P$. Since $P \supset \Lambda$ we have $P = \Lambda$. It follows that $\boldsymbol{\lambda}^* \in \operatorname{relint} \Lambda$, $\operatorname{aff} \Lambda = R^d_1$, and $L_{\boldsymbol{f}(C_M)}$ is a minimal weak approximation of g.

3.3.5 Complexity

The complexity of the algorithm is related to the number of oracle calls. We would like to resolve many hyperplane queries by performing only a polylogarithmic number of oracle calls. Thus, it is advantageous to group together many comparisons that could be done "in parallel" and employ the multi-dimensional search techniques discussed in Proposition 3.3.12.

Theorem 3.3.15 *Algorithm 3.3.14 can be implemented with complexity as follows (where $m = |E|$ and $n = |V|$).*

i. $O(\log^{2d} n + \log^d m)$ *parallel time on* $O(n^3 + m)$ *processors.*

ii. $O(m(\log^{2d} n + \log^d m))$ *sequential time, when* $m = \Omega(n^3 \log n)$.

iii. $O(\log^{2d} n(n^3 + m))$ *sequential time, when* $m = O(n^3 \log n)$ *and* $m = \Omega(n^2)$.

iv. $O(n^3 \log^{2(d-2)} n + nm \log^{2(d-1)} n)$ *sequential time, when* $m = O(n^2)$.

Proof: The problem of all-pairs shortest path can be solved in $O(\log^2 n)$ time using n^3 processors by the Floyd-Warshall algorithm [13]. The algorithm for this problem runs in $O(\log n)$ phases. During the first phase, the minimal among all the parallel edges is determined for each pair of vertices which are linked with at least one edge. In general, the minimal value in a set is computed for $O(n^2)$ sets, each with $O(n)$ elements. More precisely, during phase ℓ, for each ordered pair (i, j) of vertices we find d^ℓ_{ij}, the length of shortest path from i to j consisting of at most 2^ℓ edges. We use the relation $d^{\ell+1}_{ij} = \min\{d^\ell_{ij}, \min_k\{d^\ell_{ik} + d^\ell_{kj}\}\}$. To find a minimum cycle, we run one more phase, where we compute a minimum of the diagonal elements in the distance matrix. The complexity of this last phase is dominated by the other phases. Each phase can be implemented in one application of Problem 3.3.11, with $s = m$ for the first phase, and $s = n^3$ for the remaining $O(\log n)$ phases. The complexity is analyzed in Proposition 3.3.12.

Denote by T_d and R_d, respectively, the sequential time complexity and the parallel time complexity with $n^3 + m$ processors, of the d-dimensional problem. Recall from Theorem 3.3.5 and Remark 3.3.9 that the time complexity of one oracle call is $3T_{d-1} = O(T_{d-1})$ on a single processor and $3R_{d-1} = O(R_{d-1})$ on $O(n^3 + m)$ processors. When $d = 1$, an oracle call can be implemented simply by a scalar minimum cycle algorithm. We derive recursion formulas for R_i and T_i. The oracle calls are executed sequentially. First we derive an expression for the parallel complexity when $d = 1$. Note that the problem can be solved by employing n^3 processors for $O(\log^2 n)$ time plus m processors for $O(\log m)$ time. Thus, on $O(n^3 + \min\{m, m\log m/\log^2 n\})$ processors, we have $R_1 = O(\log^2 n + \log m)$. It follows that

$$R_d = O(\log^3 n + \log^2 m + (\log^2 n + \log m)R_{d-1}),$$

which proves i.

The sequential complexity for $d = 1$ is $T_1 = O(\min\{nm, m + n^3\})$. Parts ii-iv of the proposition follow from the recursion:

$$T_d = O(n^3 \log n + m + (\log^2 n + \log m)T_{d-1}).$$

The above analysis applies only to the complexity of Step 1 of the algorithm. In Step 2 we compute $\lambda \in \operatorname{relint} P$. This is done by solving a linear program with $O(\mathcal{H})$ equations and $O(d)$ variables. The size of $\mathcal{H}$ is at most the number of oracle calls performed, that is, $|\mathcal{H}| = O(\log^2 n + \log m)$. Therefore the complexity of Step 2 is dominated by that of Step 1. ∎

Remark 3.3.16 There are hidden constant factors in the complexities stated in Theorem 3.3.15, which depend on the dimension d. First, there is an extra ctor of d on the serial time complexity and the number of parallel processors, since most "operations" are done on vectors in R^d and take d time units. Second, there is an $O(3^d\gamma(d-1))$ factor on the time complexities. The factor $\gamma(d-1) = O((5/9)^d 3^{(d-1)^2})$ is due to the multi-dimensional search (see Subsection 3.3.2), and the factor of $O(3^d)$ is due to the fact that an oracle call involves three calls to a problem of lower dimension. It follows that the constant factor in the serial time complexity is $O(d3^d\gamma(d-1))$. In the parallel case there is a factor of $O(3^d\gamma(d-1))$ for the time complexity, and a factor of $O(d)$ on the number of processors.

Chapter 4

Convex optimization in fixed dimension

In Chapter 3 we introduced a technique which enabled us to solve the parametric minimum cycle problem with a fixed number of parameters in strongly polynomial time. In the current chapter we present this technique as a general tool. In order to allow for an independent reading of this chapter, we repeat some of the definitions and propositions given in Chapter 3. Some proofs are not repeated, however, and instead we supply the interested reader with appropriate pointers.

Suppose $\mathcal{Q} \subset R^d$ is a convex set given as an intersection of k halfspaces, and let $g : \mathcal{Q} \rightarrow R$ be a concave function that is computable by a piecewise affine algorithm (i.e., roughly, an algorithm that performs only multiplications by scalars, additions, and comparisons of intermediate values which depend on the input). Assume that such an algorithm $\mathcal{A}$ is given and the maximal number of operations required by $\mathcal{A}$ on any input (i.e., point in $\mathcal{Q}$) is T. We show that under these assumptions, for any fixed d, the function g can be maximized in a number of operations polynomial in k and T. We also present a general framework for parametric extensions of problems where this technique can be used to obtain strongly polynomial algorithms. Norton, Plotkin, and Tardos [50] applied a similar scheme and presented additional applications.

4.1 Introduction

A *convex optimization problem* is a problem of minimizing a convex function g over a convex set $S \subset R^d$. Equivalently, we can consider maximizing a concave function.

We consider the problem of maximizing a concave function, where the dimension of the space is fixed. We also assume that the function g is given by a piecewise affine algorithm (see Definition 4.2.2) which evaluates it at any point.

The results of this chapter can be extended easily to the case where the range of g is R^ℓ for any $\ell \geq 1$. We then define the notions of maximum and concavity of g with respect to the lexicographic order as follows. We say that a function $g : Q \subset R^d \to R^\ell$ is *concave* with respect to the lexicographic order $\leq_{\text{lex}}$ if for every $\alpha \in [0,1]$ and $\boldsymbol{x}, \boldsymbol{y} \in Q$,

$$\alpha g(\boldsymbol{x}) + (1-\alpha) g(\boldsymbol{y}) \leq_{\text{lex}} g(\alpha \boldsymbol{x} + (1-\alpha)\boldsymbol{y}) .$$

Applications where the range of g is R^2 were given in [10].

In Section 4.2 we define the problem. In Section 4.3 we introduce the subproblem of hyperplane queries, which is essential for the design of our algorithm. In Section 4.4 we discuss the multi-dimensional search technique which we utilize for improving our time bounds. In Section 4.5 we introduce the optimization algorithm. In Sections 4.6 and 4.7 we prove the correctness and analyze the time complexity of the algorithm. In Section 4.8 we discuss applying the technique introduced in Chapters 3 and 4 to obtain strongly polynomial time algorithms for parametric extensions of other problems.

4.2 Preliminaries

Let R^d_1 denote the set of vectors $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_d)^T \in R^d$ such that $\lambda_d = 1$. For $\boldsymbol{\lambda} \in R^{d-1}$, denote by $\hat{\boldsymbol{\lambda}} \in R^d_1$ the vector $\hat{\boldsymbol{\lambda}} = (\boldsymbol{\lambda}, 1)$. For a halfspace F, denote by ∂F the boundary of F, i.e., ∂F is a hyperplane.

Definition 4.2.1 For a finite set $C \subset R^{d+1}$, denote by $L_C : R^{d+1} \to R$ the minimum envelope of the linear functions that correspond to the vectors in C, i.e.,

$$L_C(\boldsymbol{\lambda}) = \min_{\boldsymbol{c} \in C} \boldsymbol{c}^T \boldsymbol{\lambda} \ .$$

Denote by $\hat{L}_C : R^d \to R$ the function given by

$$\hat{L}_C(\boldsymbol{\lambda}) = L_C(\hat{\boldsymbol{\lambda}}) \ .$$

The vectors in $C \subset R^{d+1}$ are called the *pieces of* g. For a piece $\boldsymbol{c} \in C$ and a vector $\boldsymbol{\beta} \in R^d$ such that $\boldsymbol{c}^T \hat{\boldsymbol{\beta}} = g(\boldsymbol{\beta})$, we say that $\boldsymbol{c}$ is *active* at $\boldsymbol{\beta}$.

Definition 4.2.2 For a function $g : Q \to R$, where $Q \subset R^d$:

i. Denote by Λ_g (or Λ, for brevity) the set of maximizers of $g(\boldsymbol{\lambda})$. The set Λ may be empty.

ii. An algorithm that computes the function g (i.e., for $\boldsymbol{\lambda} \in Q$ returns the value $g(\boldsymbol{\lambda})$ and otherwise stops or returns an arbitrary value) is called *piecewise affine*, if the operations it performs on intermediate values that depend on the input vector are restricted to additions, multiplications by constants, comparisons, and copies.

iii. For a piecewise affine algorithm $\mathcal{A}$, denote by $T(\mathcal{A})$ and $C(\mathcal{A})$ the maximum number of operations and the maximum number of comparisons, respectively, performed by $\mathcal{A}$. We assume that this numbers are finite.

iv. If $g = \hat{L}_C$ for some $C \subset R^{d+1}$, we say that $g' = \hat{L}_{C'}$ is a ***weak approximation*** of g, if the set of pieces of g' is a subset of the set of pieces of g ($C' \subset C$), and the affine hulls aff Λ_g and aff $\Lambda_{g'}$ are equal. The function $g' = \hat{L}_{C'}$ is a *minimal* weak approximation of g, if there is no $C'' \subset C'$ such that $\hat{L}_{C''}$ is a weak approximation of g.

Remark 4.2.3 Suppose that $\mathcal{A}$ is a piecewise affine algorithm. Consider the computation tree (i.e., the tree consisting of all possible computation paths) of $\mathcal{A}$. Observe that all the intermediate values along a computation path, including the final output, can be expressed as linear functions (i.e., are of the form $a^T\hat{\lambda}$) of the input vector. These linear functions can be easily computed and maintained during a single execution of the algorithm. These linear functions map the input vectors whose computation path coincides so far with that same path to the corresponding value. Moreover, the linear function which corresponds to the final output at a single execution is a piece which is active at the input vector.

Remark 4.2.4 Suppose $g : Q \subset R^d \to R$ is concave and computable by a piecewise affine algorithm $\mathcal{A}$. It is easy to see that there exists a finite set $C \subset R^{d+1}$ such that g coincides with $\hat{L}_C$.

Definition 4.2.5 Suppose $Q = F_1 \cap \cdots \cap F_k$ is the intersection of k closed halfspaces and $g : Q \to R$, $g = \hat{L}_C$, is concave and computable by a piecewise affine algorithm.

i. For $\beta \in R^d$, denote

$$Q_\beta = \bigcap \{F_i \ : \ \beta \in \partial F_i\} \ .$$

Denote by $g_\beta : Q_\beta \subset R^d \to R$ the function whose pieces are all the pieces of g which are active at β,

$$g_\beta(\lambda) = \min_{c \in C}\{c^T\hat{\lambda} \ : \ c^T\hat{\beta} = g(\beta)\} \ .$$

Note that $g_\beta = \hat{L}_{C'}$ where $C' = \{c \in C \mid c^T\hat{\beta} = g(\beta)\}$. See Figure 4.1 for an example. Later, we describe an algorithm for evaluating g_β.

ii. For a given sequence of vectors $\beta_1, \ldots, \beta_\ell \in R^d$, denote by $g_{\beta_1 \cdots \beta_\ell}$ the function $((\ldots(g_{\beta_1})_{\beta_2})\ldots)_{\beta_\ell}$. Note that $g_{\beta_1 \cdots \beta_\ell}$ depends on the order of the β_i's.

iii. Suppose an ℓ-dimensional flat $S \subset R^d$, is represented as a set of solutions of a linear system of equations. There exists an affine mapping M from R^ℓ onto S, which can be computed in $O(d^3)$ time. Denote $Q^S = \{\lambda \in R^\ell \mid M(\lambda) \in Q\}$, and define $g^S : Q^S \to R$ by $g^S = g \circ M$.

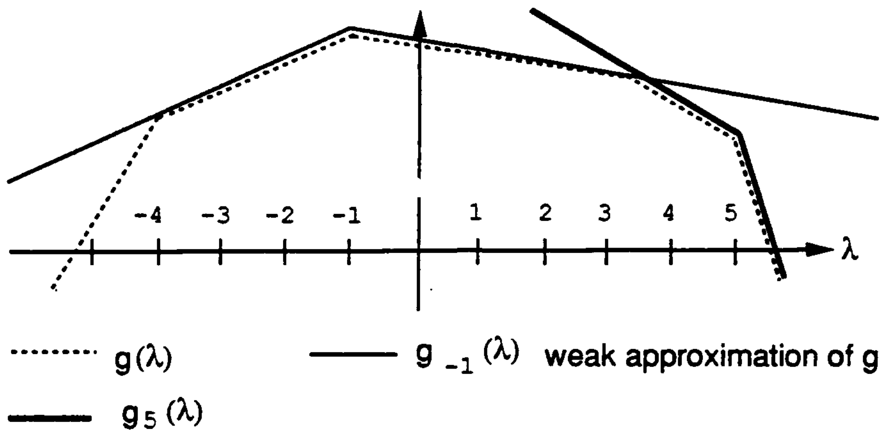


Figure 4.1: Examples of restrictions of g

Proposition 4.2.6 *Let $\mathcal{A}$ be a piecewise affine algorithm for evaluating $g : \mathcal{Q} \rightarrow R$, where $\mathcal{Q} \subset R^d$ is given as the intersection of k halfspaces. By modifying $\mathcal{A}$ we can obtain the following piecewise affine algorithms:*

i. For any given vector $\beta \in \mathcal{Q}$, we obtain an algorithm $\mathcal{A}_\beta$ for evaluating g_β so that $T(\mathcal{A}_\beta) = T(\mathcal{A}) + dC(\mathcal{A})$ and $C(\mathcal{A}_\beta) = C(\mathcal{A})$.

ii. For any ℓ-dimensional flat $S \subset R^d$, represented as the set of solutions of a system of linear equations, we obtain an algorithm $\mathcal{A}^S$ for evaluating g^S so that $T(\mathcal{A}^S) = T(\mathcal{A}) + O(\ell d)$ and $C(\mathcal{A}^S) = C(\mathcal{A})$.

Proof: Part ii is straightforward, since we can choose the algorithm $\mathcal{A}^S$ to be a composition of the appropriate affine mapping and $\mathcal{A}$. We discuss the construction of the algorithm $\mathcal{A}_\beta$ for part i. Consider an input vector $\lambda \in \mathcal{Q}_\beta$. Let $\epsilon > 0$ be such that for all ϵ' ($0 < \epsilon' \leq \epsilon$), $\beta + \epsilon'(\lambda - \beta) \in \mathcal{Q}$, and the set of pieces of g which are active at $\beta + \epsilon'(\lambda - \beta)$ is equal to the set of pieces which are active at $\beta + \epsilon(\lambda - \beta)$. It is immediate to see that such an ϵ always exists. It follows from the definition that $g_\beta(\lambda)$ is the value of λ at the linear pieces of g which are active at $\beta + \epsilon(\lambda - \beta)$. The algorithm $\mathcal{A}_\beta$, when executed with an input λ, follows the computation path of $\mathcal{A}$ which corresponds to the input $\beta + \epsilon(\lambda - \beta)$. The algorithm computes the linear functions associated with the intermediate values of this path (see Remark 4.2.3). Recall that the linear function which corresponds to the final value is a piece of g which is active at $\beta + \epsilon(\lambda - \beta)$. Hence, the value of $g_\beta(\lambda)$ is obtained by substituting λ in this linear function. In order to follow the desired run of $\mathcal{A}$, the algorithm $\mathcal{A}_\beta$ mimics the work of $\mathcal{A}$ on additions and multiplications by scalars, keeping track of linear functions rather than just numerical values. When the run of $\mathcal{A}$ reaches a comparison (branching point), $\mathcal{A}_\beta$ does as follows. Without loss of generality we assume that the branching is according to the sign of the linear function $\boldsymbol{a}^T\hat{\boldsymbol{x}}$. In order to decide what to do at a branching point, $\mathcal{A}_\beta$ has to determine the sign of $\boldsymbol{a}^T\hat{\boldsymbol{x}}$ at the point $\boldsymbol{x} = \boldsymbol{\beta} + \epsilon(\boldsymbol{\lambda} - \boldsymbol{\beta})$. Since ϵ is not given explicitly, the decision cannot be made directly by substitution. The decision is made as follows. The algorithm first computes $\alpha = \boldsymbol{a}^T\hat{\boldsymbol{\beta}}$. If $\alpha \neq 0$, then obviously for any vector $\boldsymbol{y}$, for sufficiently small number $\epsilon > 0$ $\boldsymbol{a}^T(\widehat{\boldsymbol{\beta} + \epsilon \boldsymbol{y}})$ has the same sign as α. In particular this holds for $\boldsymbol{y} = \boldsymbol{\lambda} - \boldsymbol{\beta}$ and the sign is detected. Otherwise, if $\alpha = 0$, it follows that $\boldsymbol{a}^T(\boldsymbol{\beta} + \epsilon\widehat{(\boldsymbol{\lambda} - \boldsymbol{\beta})}) = \epsilon \boldsymbol{a}^T\hat{\boldsymbol{\lambda}}$. Hence $\boldsymbol{a}^T\hat{\boldsymbol{\lambda}}$ has the same sign as $\boldsymbol{a}^T(\boldsymbol{\beta} + \epsilon\widehat{(\boldsymbol{\lambda} - \boldsymbol{\beta})})$. It remains to compute the sign of $\boldsymbol{a}^T\hat{\boldsymbol{\lambda}}$ and branch accordingly. It is easy to verify that $\mathcal{A}_\beta$ evaluates the function g_β for any vector $\boldsymbol{\lambda} \in Q_\beta$, and performs the stated number of operations. ▮

Proposition 4.2.7 *If $\boldsymbol{\beta} \in \Lambda_g$ then g_β is a weak approximation of g (see Proposition 3.1.9 for a proof).*

The goal is to solve the following problem:

Problem 4.2.8 The input of this problem consists of a polyhedron $Q = F_1 \cap \cdots \cap F_k$, given as the intersection of k closed halfspaces and a piecewise affine algorithm $\mathcal{A}$ for evaluating a concave function $g : Q \rightarrow R$. Decide whether or not g is bounded. If so, then find a $\boldsymbol{\lambda}^* \in \operatorname{relint} \Lambda$. We refer to the following as the "optional" part of the problem: If g is bounded, then find a subset C of the set of pieces of g, such that $\hat{L}_C$ is a minimal weak approximation of g, and $|C| \leq 2d$.

The set C may be viewed as a *certificate* for the fact that the maximum of the function g does not exceed $g(\boldsymbol{\lambda}^*)$. In the current chapter we do not discuss the details of solving the optional part of the problem. See Chapter 3 for an existence proof and an algorithm which finds such a set.

We propose an algorithm for Problem 4.2.8. In any fixed dimension d, the total number of operations performed by this algorithm is bounded by a polynomial in $T(\mathcal{A})$ and k. The algorithm is based on solving instances of a subproblem, which we call *hyperplane query*: For a given hyperplane H_0, decide on which side of H_0 the function g is either unbounded or attains its maximum. A procedure for hyperplane queries is called an *oracle*. Obviously, an oracle can be utilized to perform a binary search over the polyhedron $\mathcal{Q}$. However, in order to attain an exact solution within time bounds that depend only on d, T, and k, we use the oracle in a more sophisticated way. The number of hyperplane queries needed by the algorithm, and hence the number of oracle calls, is bounded by the number of comparisons performed by $\mathcal{A}$. We later discuss applying the multi-dimensional search technique, what allows us to do even better. By exploiting the parallelism of $\mathcal{A}$, the number of oracle calls can in some cases be reduced to a polylogarithm of the number of hyperplane queries.

The function g is a concave piecewise linear mapping. Concave functions have the property that it can be effectively decided which side of a given hyperplane H_0 contains the maximum of the function. If the domain of g does not intersect H_0, then the answer is the side of H_0 which contains the domain of g. Otherwise, the decision can be made by considering a neighborhood of the maximum of the function relative to H_0, searching for a direction of ascent from that point. This principle is explained in detail in [47].

For a hyperplane $H_0 \subset R^d$, we wish to decide on which side of H_0 the set $\operatorname{relint} \Lambda$ lies. By solving a linear program with d variables and $k+1$ constraints, we determine whether or not $H_0 \cap \mathcal{Q} = \emptyset$, and if so, we determine which side of H_0 contains $\mathcal{Q}$. It follows from [47] that this can be done in $O(k)$ time. If $H_0 \cap \mathcal{Q} \neq \emptyset$, then the oracle problem solves the original problem, when g is restricted to H_0. If g is unbounded on H_0 the oracle reveals that. If $\Lambda = \emptyset$, or if $\operatorname{relint} \Lambda$ is either contained in H_0 or extends into both sides of H_0 (i.e., $H_0 \cap \operatorname{relint} \Lambda \neq \emptyset$), then we find $\lambda \in H_0 \cap \operatorname{relint} \Lambda$ and the oracle will actually solve Problem 4.2.8.

Problem 4.2.9 Given are a set $Q = F_1 \cap \cdots \cap F_k$, a piecewise affine algorithm $\mathcal{A}$ which evaluates a concave function $g : Q \to R$, and a hyperplane H_0 in R^d. Do as follows:

i. If $Q \cap H_0 = \emptyset$, recognize which of the two halfspaces determined by H_0 contains Q. Otherwise,

ii. recognize whether or not g is bounded on H_0. If it is, then

iii. find $\lambda \in H_0 \cap \operatorname{rel\,int} \Lambda$ if such λ exists, and solve Problem 4.2.8 relative to g. Otherwise, if $H_0 \cap \operatorname{rel\,int} \Lambda = \emptyset$, then

iv. recognize which of the two halfspaces determined by H_0 has either a nonempty intersection with $\operatorname{rel\,int} \Lambda$, or has g unbounded on it.

A procedure for solving Problem 4.2.9 will be called an *oracle* and the hyperplane H_0 will be called the *query hyperplane*. Problem 4.2.8 is solved by running a modification of the algorithm $\mathcal{A}$, where additions and multiplications are replaced by vector operations and comparisons are replaced by hyperplane queries. Problem 4.2.9 is solved by three recursive calls to instances of Problem 4.2.8 of the form $(\mathcal{A}^H, Q^H)$, $(\mathcal{A}^H_\beta, Q^H_\beta)$, where $\beta \in R^d$, and H is a hyperplane (see Definitions 4.2.5 and 4.2.6). Note that these algorithms compute, respectively, the functions $g^H : Q^H \to R$, $g^H_\beta : Q^H_\beta \to R$, where Q^H and Q^H_β are subsets of R^{d-1}. Hence, the recursive calls are made to instances of lower dimension.

In Section 4.5 we propose Algorithm 4.5.2 for Problem 4.2.8. The algorithm executes calls to the oracle problem (Problem 4.2.9) relative to g. An algorithm for the oracle problem is given in Section 4.3. A call to the oracle is costly. Therefore, one wishes to solve many hyperplane queries with a small number of oracle calls. In Section 4.4 we discuss the multi-dimensional search technique (introduced in [47]).

4.3 Hyperplane queries

For a hyperplane $H \subset R^d$, we solve Problem 4.2.9 for g relative to H.

Theorem 4.3.1 *Problem 4.2.9 can be reduced to the problem of solving three instances of Problem 4.2.8 on functions defined on an intersection of at most k closed halfspaces in R^{d-1}. The time complexity of the additional computation is $O(d^3)$.*

Proof: We solve Problem 4.2.8 with the function g^H, where $H = \{\boldsymbol{\lambda} \in R^d \ \ \boldsymbol{a}^T\hat{\boldsymbol{\lambda}} = \alpha\}$. If g is unbounded on H, then this fact is detected; otherwise, suppose $\boldsymbol{\lambda}^{(0)}$ is in the relative interior of the set of maximizers of $g(\boldsymbol{\lambda})$ subject to $\boldsymbol{\lambda} \in H$, and we get the collection $C^{(0)}$. Let $t^{(0)} = g(\boldsymbol{\lambda}^{(0)})$. We wish to recognize whether $\boldsymbol{\lambda}^{(0)}$ is also a relative interior point of the set of global maxima (i.e., relative to R^d). If not, then we wish to decide whether for all $\boldsymbol{\lambda}^{*} \in R^d$ such that $g(\boldsymbol{\lambda}^{*}) \geq g(\boldsymbol{\lambda}^{(0)})$, necessarily $\boldsymbol{a}^T\boldsymbol{\lambda}^{*\prime} > \alpha$, or whether for all of them $\boldsymbol{a}^T\boldsymbol{\lambda}^{*\prime} < \alpha$. These are the two possible cases. Consider the function $g_{\lambda^{(0)}}$. We solve Problem 4.2.8 on two restrictions of $g_{\lambda^{(0)}}$ to hyperplanes (see Proposition 4.2.6), where in one case it is restricted to $H^{(1)} = \{\boldsymbol{\lambda} \mid \boldsymbol{a}^T\hat{\boldsymbol{\lambda}} = \alpha - 1\}$, and in the other to $H^{(-1)} = \{\boldsymbol{\lambda} \mid \boldsymbol{a}^T\hat{\boldsymbol{\lambda}} = \alpha + 1\}$. Note that the domains $Q^{H^{(\delta)}}_{\lambda^{(0)}} (\delta \in \{-1, 1\})$ are $(d-1)$-dimensional. Denote the respective optimal values of $g^{H^{(\delta)}}_{\lambda^{(0)}}$ by $t^{(\delta)}$ $(\delta \in \{-1, 1\})$, and let C^{δ} be the respective minimal weak approximations. Only one of the optimal values $t^{(1)}, t^{(-1)}$ can be greater than $t^{(0)}$. If this is the case, or if one of $t^{(1)}, t^{(-1)}$ equals $t^{(0)}$ and the other is smaller, then the side of the hyperplane that contains relint Λ is determined. Otherwise, if both values are less than or both values are equal to $t^{(0)}$, then $t^{(0)}$ is the global optimal value. In the latter case $\boldsymbol{\lambda}^{(0)} \in$ relint Λ. It follows from Proposition 4.2.7 that the pieces of g active in a minimal weak approximation have the value $t^{(0)}$ at $\boldsymbol{\lambda}^{(0)}$. Thus, a minimal weak approximation of the function $g_{\lambda^{(0)}}$ is a minimal weak approximation of g. It follows from analysis done in Chapter 3 that by using $O(d^3)$ operations we can construct a minimal weak approximation of $g_{\lambda^{(0)}}$. Furthermore, the number of pieces involved in a minimal weak approximation is at most $2d$. ∎

As an example, consider an application of the algorithm described in the proof to decide on which side of the hyperplane $H = \{2\}$ the function $g(\lambda) = \min\{\lambda/5 + 2, -4\lambda + 12.5\}$ is maximized (see Figure 4.2). Note that maximizing a function $f : R \to R$ on a hyperplane corresponds to evaluating it at a single point. Therefore, the maximum value of g on H is 2.4. The algorithm considers the restriction $g_2 = \lambda/5+2$,

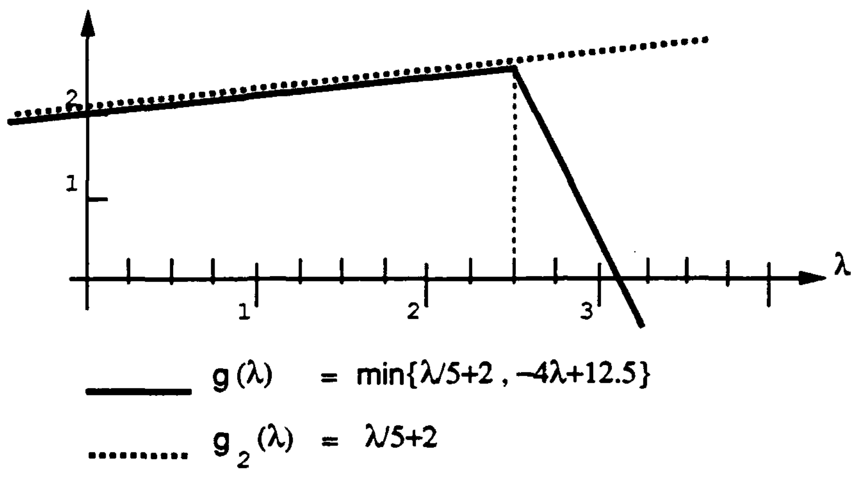


Figure 4.2: Example: hyperplane query at $H = \{2\}$

and maximizes it on the hyperplanes $H^{(1)} = \{1\}$ and $H^{(-1)} = \{3\}$. The corresponding maxima are $t^{(1)} = 2.2$ and $t^{(-1)} = 2.6$, and hence, the algorithm concludes that the maximizers of g are contained in the halfspace $\{\lambda \in R | \lambda > 2\}$. Observe that this conclusion could not have been made if the algorithm considered the values of g, rather than the values of the restriction g_2, at the hyperplanes $\{1\}$ and $\{3\}$.

4.4 Employing multi-dimensional search

The definitions and propositions stated in this section appeared in Section 3.3.2. They are presented here to allow for an independent reading of this chapter. For proofs, the reader is referred to Section 3.3.2. The multi-dimensional search problem was defined and used in [47] for solving linear programming problems in fixed dimension. In this section we employ it to achieve better time bounds.

Definition 4.4.1 We define a partial order on $R^d \setminus \{0\}$, relative to a concave function $g : Q \to R$, where $Q \subset R^{d-1}$ is a nonempty polyhedral set. For any pair of distinct vectors $a_1, a_2 \in R^d$, denote

$$H = H(a_1, a_2) = \{\lambda \in R^{d-1} : a_1^T \hat{\lambda} = a_2^T \hat{\lambda}\} .$$

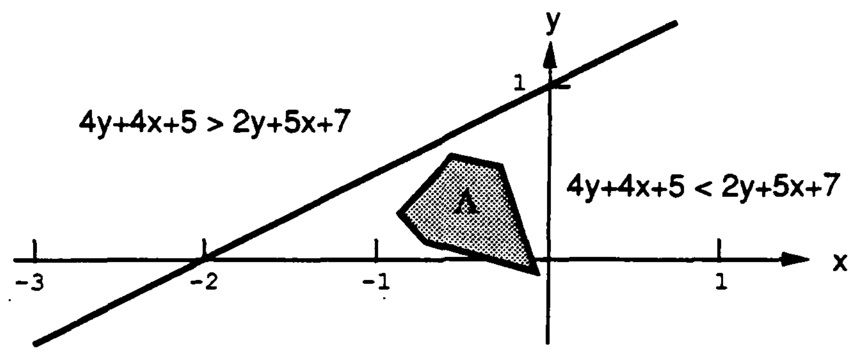


Figure 4.3: An example where $(4,4,5) <_\Lambda (2,5,7)$

If g is unbounded on $H(\boldsymbol{a}_1, \boldsymbol{a}_2)$ or if $H(\boldsymbol{a}_1, \boldsymbol{a}_2) \cap \operatorname{rel\,int} \Lambda \neq \emptyset$, then we write $\boldsymbol{a}_1 <>_\Lambda \boldsymbol{a}_2$. Otherwise, g can be unbounded on at most one of the open halfspaces determined by H, and also rel int Λ can intersect at most one of these open halfspaces. If g is undefined on H (i.e., $Q \cap H = \emptyset$), then Q is contained in one of these halfspaces. We denote $\boldsymbol{a}_1 <_\Lambda \boldsymbol{a}_2$ (respectively, $\boldsymbol{a}_1 >_\Lambda \boldsymbol{a}_2$) if there exists a $\hat{\boldsymbol{\lambda}} \in \operatorname{rel\,int} \Lambda$ such that $\boldsymbol{a}_1^T \hat{\boldsymbol{\lambda}} < \boldsymbol{a}_2^T \hat{\boldsymbol{\lambda}}$ (respectively, $\boldsymbol{a}_1^T \hat{\boldsymbol{\lambda}} > \boldsymbol{a}_2^T \hat{\boldsymbol{\lambda}}$), in which case the same holds for all these $\boldsymbol{\lambda}$'s, or if g is unbounded on the halfspace determined by the inequality $\boldsymbol{a}_1^T \hat{\boldsymbol{\lambda}} < \boldsymbol{a}_2^T \hat{\boldsymbol{\lambda}}$ (respectively, $\boldsymbol{a}_1^T \hat{\boldsymbol{\lambda}} < \boldsymbol{a}_2^T \hat{\boldsymbol{\lambda}}$). See Figure 4.3 for an example. We also use the notation $<_P$ for a similar partial order relative to any set P.

Problem 4.4.2 Given are finite sets $A_1, \ldots, A_r$ of nonzero vectors, where $A_i = \{\boldsymbol{a}_1^i, \ldots, \boldsymbol{a}_{s_i}^i\}$ ($\boldsymbol{a}_j^i \in R^d$) and $s = \sum s_i$. We wish either to find a minimal element, with respect to the partial order $<_\Lambda$, in each of the sets A_i, or (if we encounter two incomparable elements) to reduce the problem to a lower dimension. More specifically, we need to do either one of the following:

i. Find a collection of closed halfspaces whose intersection P contains rel int Λ, and indices $1 \le m_i \le s_i$ ($i = 1, \ldots, r$) as follows. For every $1 \le i \le r$ and every $1 \le j \le s_i$, $j \neq m_i$, we have $\boldsymbol{a}_{m_i}^i <_\Lambda \boldsymbol{a}_j^i$ and $\boldsymbol{a}_{m_i}^i <_P \boldsymbol{a}_j^i$.

ii. Find a hyperplane H such that either g is unbounded on H or $H \cap \operatorname{rel\,int} \Lambda \neq \emptyset$.

Proposition 4.4.3 *Problem 4.4.2 can be solved using $O(\gamma(d-1)\log s)$ oracle calls plus additional computation which can be performed in either*

i. $O(\gamma(d-1)\log^2 s)$ *parallel time on* $O(s)$ *processors, or*

ii. $O(\gamma(d-1)s\log s)$ *sequential time.*

The function $\gamma(d)$ arises from the multi-dimensional search [47]. It follows from [5, 19] that $\gamma(d) = 3^{O(d^2)}$.

4.5 The algorithm

The algorithm described below solves Problem 4.2.8. It finds a vector $\boldsymbol{\lambda}^* \in \operatorname{relint} \Lambda$, unless g is unbounded. It also returns a collection C of pieces of g whose minimum envelope $\hat{L}_C$ is a minimal weak approximation of g. The number of vectors in C is at most $2d$.

Definition 4.5.1 For a piecewise affine algorithm $\mathcal{A}$, we define the corresponding *lifted computation*. The lifted computation is a run of the algorithm on a set of inputs. The computation is done symbolically on linear functions instead of on scalars. It follows the path on the computation tree of $\mathcal{A}$ that corresponds to input vectors which are [illegible] The additions and scalar multiplications are trivially generalized to operations [illegible]ear functions. When a comparison is done between $\boldsymbol{f}_1^T\hat{\boldsymbol{\lambda}}$ and $\boldsymbol{f}_2^T\hat{\boldsymbol{\lambda}}$, it is resolved according to the partial order $<_\Lambda$. We compute the hyperplane $H(\boldsymbol{f}_1, \boldsymbol{f}_2)$ and solve Problem 4.2.9 (hyperplane query) relative to H. The hyperplane query decides whether or not the vectors are comparable. If they are, it decides whether $\boldsymbol{f}_1 <_\Lambda \boldsymbol{f}_2$. If $\boldsymbol{f}_1 <>_\Lambda \boldsymbol{f}_2$, then the lifted computation halts since an oracle call resulted in a solution to Problem 4.2.8. Otherwise, the resolved hyperplane query tells us which of the halfspaces defined by $H(\boldsymbol{f}_1, \boldsymbol{f}_2)$ contains the set $\operatorname{relint} \Lambda$, and the comparison is resolved.

Sets of independent comparisons performed by $\mathcal{A}$ correspond to sets of independent hyperplane queries. Recall from Section 4.4 that a set of independent hyperplane queries can be solved by performing a logarithmic number of "oracle" calls. The lifted computation maintains a set $\mathcal{H}$ of closed halfspaces which is initially empty. Whenever an oracle call is executed the resulting halfspace is added to $\mathcal{H}$.

Algorithm 4.5.2 [Find a vector $\boldsymbol{\lambda} \in \operatorname{relint} \Lambda_g$]

Step 1. Run the lifted computation, collecting into $\mathcal{H}$ all the halfspaces resulting from oracle calls where comparisons are resolved. If the computation halts, then some comparison is not resolved but a global solution is found, so stop. Otherwise, denote by $\boldsymbol{m} = (m_1, \ldots, m_{d+1})^T \in R^{d+1}$ the piece of g that corresponds to the computation path followed.

Step 2. Denote by P the intersection of the halfspaces in $\mathcal{H}$.

i. Compute $\boldsymbol{\lambda}^* \in \operatorname{relint} P \cap Q$. This amounts to a linear programming problem with d variables and $|\mathcal{H}|$ constraints, and hence it can be solved in $O(|\mathcal{H}|)$ sequential time [47]. Note that the size of $\mathcal{H}$ is bounded by the number of oracle calls.

ii. If $\hat{L}_{\{\boldsymbol{m}\}}$ is not constant on R^d, that is, not all of $m_1, m_2, \ldots, m_d$ equal zero, then g is unbounded. Otherwise,

iii. consider $g(\boldsymbol{\lambda}^*) = m_{d+1}$. The function $\hat{L}_{\{\boldsymbol{m}\}}$ is a weak approximation of g, and $P = \Lambda$. Hence, $\boldsymbol{\lambda}^* \in \operatorname{relint} \Lambda$. Output $\boldsymbol{\lambda}^*$ and $C = \{\boldsymbol{m}\}$.

4.6 Correctness

If an oracle call results in a solution during Step 1 of Algorithm 4.5.2, then correctness follows by induction on the dimension. We now assume that no oracle call resulted in a solution during Step 1. In this case, a collection $\mathcal{H}$ of closed halfspaces is obtained. Recall that if an oracle call on a hyperplane H did not result in a solution, then the halfspace F returned has the following properties: (i) if the function g is bounded then $\Lambda \subset F$ but $\Lambda \not\subset H$, (ii) if the function g is unbounded, then it must be bounded on the hyperplane H, and unbounded on the halfspace F. Let P be the polyhedron $P = \bigcap_{F \in \mathcal{H}} F$. It follows that if g is bounded then $P \supset \Lambda$, and if g is unbounded then it must be bounded outside and on the boundary of P. Note that P must be of full dimension ($\dim P = d$), for if not, then it must be contained in one of the query hyperplanes, which contradicts the previous statement.

Observe that for all pairs $\boldsymbol{a}_1, \boldsymbol{a}_2$ of vectors compared by the lifted computation, one of the following must hold: either $\boldsymbol{a}_1 <_\Lambda \boldsymbol{a}_2$ and $\boldsymbol{a}_1 <_P \boldsymbol{a}_2$, or $\boldsymbol{a}_2 <_\Lambda \boldsymbol{a}_1$ and $\boldsymbol{a}_2 <_P \boldsymbol{a}_1$. The latter is obvious when we call the oracle to resolve each hyperplane query, and it is easy to see that it still holds when we employ the multi-dimensional search technique (see Problem 4.4.2 and Proposition 4.4.3) and solve these hyperplane queries by a smaller number of oracle calls. Thus, the piece $\boldsymbol{m}$ (the maximizer) found by the lifted computation must satisfy $\boldsymbol{m} <_\Lambda \boldsymbol{c}$ and hence $\boldsymbol{m} <_P \boldsymbol{c}$ for all pieces $\boldsymbol{c}$ of g. It follows that $g(\boldsymbol{\lambda}) = \boldsymbol{m}^T\hat{\boldsymbol{\lambda}}$ for all $\boldsymbol{\lambda} \in P$. Thus, g is unbounded if and only if $\hat{L}_{\{\boldsymbol{m}\}}$ is not constant, and the correctness of step ii follows. To show the correctness of step iii assume that $\hat{L}_{\{\boldsymbol{m}\}}$ is constant, and thus $g = m_{d+1}$ for all $\boldsymbol{\lambda} \in P$. Since $P \supseteq \Lambda$ we have $P = \Lambda$. It follows that $\boldsymbol{\lambda}^* \in \operatorname{relint} \Lambda$, $\operatorname{aff} \Lambda = R^d$, and $\hat{L}_{\{\boldsymbol{m}\}}$ is a minimal weak approximation of g.

4.7 Complexity

Consider the algorithm $\mathcal{A}$. Suppose that the $C(\mathcal{A})$ comparisons performed by $\mathcal{A}$ can be divided into r phases, where C_i independent comparisons are performed during phase i ($i = 1, \ldots, r$). It follows from Proposition 4.4.3, that the lifted computation can be implemented in such a way that it performs $\gamma(d) \sum_{i=1}^{r} \lceil \log C_i \rceil$ oracle calls. It follows from Theorem 4.3.1 that each oracle call involves three recursive calls to instances of Problem 4.2.8 of lower dimension. The piecewise affine algorithms that correspond to these instances have the same number of comparisons as $\mathcal{A}$, divided into phases in the same way, and $O(d)$ times more operations. Thus, the total number of operations needed for the lifted computation is

$$d^3\gamma(d)kT(\mathcal{A})(\sum_{i=1}^{r}\lceil\log C_i\rceil)^d .$$

The number of parallel phases needed in the above computation is bounded by the product of the number of phases of the algorithm $\mathcal{A}$ with $\sum_{i=1}^{r}\lceil\log C_i\rceil)^d$. If the algorithm $\mathcal{A}$ is inherently sequential, then the total number of operations is $O(kT(\mathcal{A})C(\mathcal{A})^d)$.

4.8 Parametric extensions of problems

The technique described in this chapter was employed in Chapter 3 to get algorithms for the parametric extensions of the minimum cycle and the minimum cycle-mean problems. This technique can be applied to a variety of other problems, where we consider a strongly polynomial algorithm for a problem and obtain a strongly polynomial algorithm for a parametric extension of the problem (when the number of parameters is fixed). We state the conditions where this technique is applicable and present applications.

Definition 4.8.1 [Parametric extensions]

i. A ***problem*** $S : \mathcal{P} \rightarrow R$ is a mapping from a set $\mathcal{P}$ of instances into the set of real numbers. We say that $S(P)$ is the ***solution*** of the problem for the instance $P \in \mathcal{P}$. Suppose that every instance $P \in \mathcal{P}$ has a ***size*** $\|P\|$ associated with it. The size of an instance is not necessarily defined to be the number of bits in its representation. It may be any natural parameter (for example, the number of edges in a weighted graph).

ii. Let $\mathcal{A}$ be an algorithm that computes $S(P)$. Denote by $T_{\mathcal{A}}(P)$ the number of elementary operations the algorithm performs on the instance P. The algorithm $\mathcal{A}$ is ***polynomial*** if $T_{\mathcal{A}}(P) = O(p(\|P\|))$ for some polynomial $p(\bullet)$.

iii. A *d-parametric extension* $P^d = (\mathcal{M}, \mathcal{Q})$ of $\mathcal{P}$ is defined as follows, where $\mathcal{Q} \subset R^d$ is a polyhedron given as an intersection of k halfspaces, and $\mathcal{M} : \mathcal{Q} \rightarrow \mathcal{P}$ is a mapping from points $\lambda \in \mathcal{Q}$ to instances of $\mathcal{P}$. The extension P^d corresponds to a subset of instances $\{\mathcal{M}(\lambda) \mid \lambda \in \mathcal{Q}\} \subset \mathcal{P}$. We refer to $\mathcal{M}(\lambda) \in \mathcal{P}$ as the instance of $\mathcal{P}$ *induced* by λ. For an extension P^d, we define $g : \mathcal{Q} \rightarrow R$ as a mapping from vectors $\lambda \in \mathcal{Q}$ to the solution of the corresponding induced instance $g(\lambda) = S(\mathcal{M}(\lambda))$. A *solution of the parametric extension* P^d is defined as follows. Consider the maximum of $g(\lambda)$. If it is finite, a solution consists of the maximum and a vector $\lambda \in R^d$ that belongs to the relative interior of the set of vectors which maximize S. Formally, if $\mathcal{Q}$ is empty or if $S(\mathcal{M}(\lambda))$ is unbounded on $\mathcal{Q}$, these facts are recognized. Otherwise, a pair $(m, \lambda^*) \in R \times R^d$, where $m = \max_{\lambda \in \mathcal{Q}} g(\lambda)$, and $\lambda^* \in \operatorname{rel\,int}\{\lambda \mid g(\lambda) = m\}$ is computed. We denote $T = \max_{\lambda \in \mathcal{Q}} T_{\mathcal{A}}(\mathcal{M}(\lambda))$.

Theorem 4.8.2 *Let $S : \mathcal{P} \rightarrow R$ be a problem in the sense of Definition 4.8.1. Let $\mathcal{A}$ be an algorithm that evaluates S, and let $P^d = (\mathcal{M}, \mathcal{Q})$ (where $|\mathcal{Q}| = k$) be a corresponding parametric extension. We assume that*

i. the function g is concave,

ii. the mapping $\mathcal{M}$ is computable by a piecewise affine algorithm $\mathcal{A}_{\mathcal{M}}$ (see Definition 4.2.2) in less than T operations, and

iii. the combined algorithm which computes an instance $\mathcal{A}_{\mathcal{M}}(\lambda) \in \mathcal{P}$ and applies $\mathcal{A}$ to $\mathcal{A}_{\mathcal{M}}(\lambda)$, is piecewise affine.

Denote by C the maximum (over $\lambda \in \mathcal{Q}$) number of comparisons performed by the combined algorithm. Suppose the comparisons can be divided into r sets of sizes $C_1, \ldots, C_r$ ($C = \sum_{i=1}^r C_i$) such that the algorithm runs in r phases, where C_i independent comparisons are performed in phase i.

Under these conditions, the d-parametric extension P^d can be solved using

$$\beta(d)kT(\sum_{i=1}^{r} \lceil \log C_i \rceil)^d \text{ operations, where } \beta(d) = 3^{O(d^2)} .$$

Remark 4.8.3 In the above formulation we defined a problem as a mapping into the set of real numbers $S : \mathcal{P} \rightarrow R$. The results generalize to cases where the range of S is R^{ℓ} for $\ell > 1$ and the notions of maximum and concavity of g are defined with respect to the lexicographic order as discussed in the introduction.

Below we present some applications of Theorem 4.8.2. Additional applications were found by Norton, Plotkin, and Tardos [50].

Adding variables to LP's with two variables per inequality Linear programming problems with at most two variables in each constraint and in the objective function were shown to have a strongly polynomial time algorithm by Megiddo [46]. Lueker, Megiddo and Ramachandran [43] gave a polylogarithmic time parallel algorithm for the problem which uses a quasipolynomial number of processors. The best known time bounds for the problem are presented in Chapter 5. Cosares, using nested parametrization, extended Megiddo's strong polynomiality result to allow objective functions which have a fixed number of nonzero coefficients. This result can be further extended to include the following. For a fixed d, we consider linear programming problems as above, but we allow certain d additional variables to appear anywhere in the constraints and in the objective function without being "counted." This problem is a d-parameter extension of the two variables per constraint problem, where the "parameters" are the d additional variables. For each choice of values for the parameters we have a corresponding induced system with two variable per constraint. It is easy to verify that the conditions of Theorem 4.8.2 hold. Hence, this class of problems also has a strongly polynomial time algorithm, and a polylogarithmic time parallel algorithm which uses a quasipolynomial number of processors.

Parametric flow problems Theorem 4.8.2 was applied in [10] to generate strongly polynomial algorithms for parametric flow problems with a fixed number of parameters and to some constrained flow problems with a fixed number of additional constraints. Complementing results showing the P-completeness of these problems when the number of parameters is not fixed, were also given.

Chapter 5

Linear systems with two variables per inequality

We show that a system of m linear inequalities with n variables, where each inequality involves at most two variables, can be solved in $\tilde{O}(mn^2)$ time, and using randomization, in $\tilde{O}(n^3 + mn)$ expected time. Parallel implementations of these algorithms run in $\tilde{O}(n)$ time, where the deterministic algorithm uses $\tilde{O}(mn)$ processors and the randomized algorithm uses $\tilde{O}(n^2 + m)$ processors.

5.1 Introduction

In this chapter we consider the following class of linear systems:

Definition 5.1.1 A *TVPI* system is a system of linear inequalities where each inequality involves at most two variables (i.e., a system of the form $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$, where $\boldsymbol{A} \in R^{m \times n}$ is a matrix, $\boldsymbol{b} \in R^m$ is a real vector, and each row of $\boldsymbol{A}$ contains at most two nonzero entries). We denote the number of inequalities by m , and the variables by $x_1, \ldots, x_n$. We denote by $F = \{\boldsymbol{x} \in R^n | \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ the set of feasible solutions.
A TVPI system is called *monotone* if the two nonzero entries in each row have opposite signs. See Figure 5.1 for an example of a TVPI system.

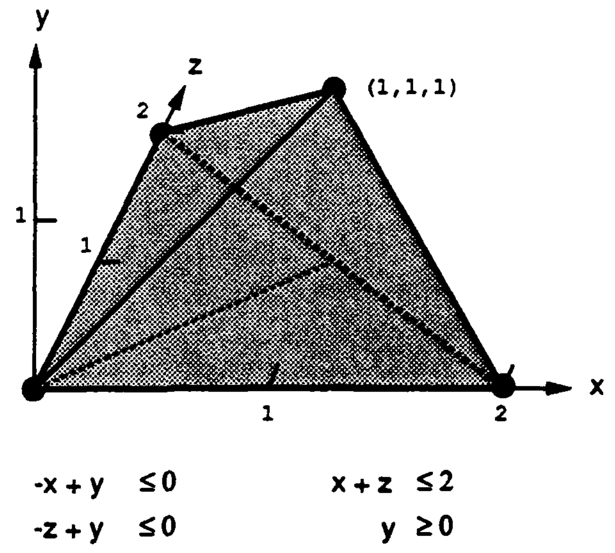


Figure 5.1: An example of a TVPI system with 3 variables

Our goal is to either find a point that satisfies all the inequalities or conclude that no such point exists. The structure of TVPI systems enables us to obtain specialized algorithms that are faster than known algorithms for solving general linear systems. The algorithms given here can also be adapted to find a solution that minimizes or maximizes a specific variable. We summarize previous work on solving TVPI systems. Shostak [57] characterized the set of solutions of TVPI systems and gave an algorithm that is exponential in the worst case. Nelson [49] gave an $n^{O(\log n)}$ algorithm. Aspvall and Shiloach [4] and Aspvall [3] proposed algorithms that perform $O(mn^3 I)$ and $O(mn^2 I)$ arithmetic operations, respectively, where I is the size of the binary encoding of the problem. Megiddo [46] proposed the first strongly polynomial time algorithm for the problem, which performs $O(mn^3 \log m)$ operations. The parallel implementation of Megiddo's algorithm runs in $O(n^3 \log m)$ time using $O(m)$ processors.

The algorithms presented here improve the sequential and parallel time bounds. We give an $O\left(mn^2(\log m + \log^2 n)\right)$ time deterministic algorithm, which has a parallel implementation that runs in $O\left(n(\log m + \log^2 n)\right)$ time using $O(mn)$ processors.

An additional improvement is obtained through using randomization: we give an algorithm that runs in $O\left(n^3 \log n + mn(\log^5 n + \log m \log^3 n)\right)$ expected time. A parallel implementation runs in $O\left(n(\log^5 n + \log^3 n \log m)\right)$ expected time using $O(n^2 + m)$ processors. The effort involved in translating these algorithms into actual programs is about the same as in the previously known algorithms, and the hidden constants in the time bounds are still reasonable. The space requirement of these algorithms is $O(n^2 + m)$. Hochbaum and Naor [32] found a new $O(mn^2 \log m)$ deterministic algorithm for the problem. Their algorithm, however, runs in $O(n^2 \log m)$ time in parallel, and it does not seem possible to combine it with the randomized approach to yield algorithms with better expected time.

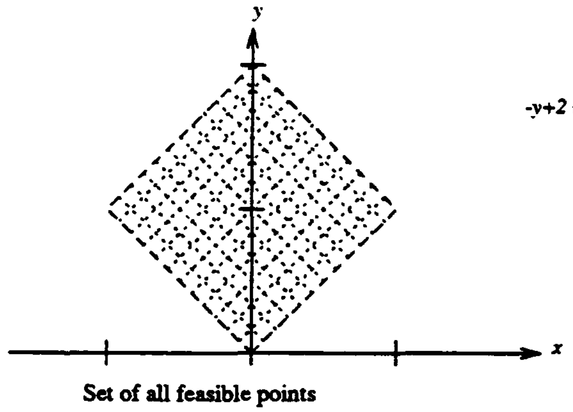
Section 5.2 gives the preliminaries. In particular we discuss the subproblem of *locating a value*: for a numerical value ξ and $1 \leq i \leq n$, decide whether a TVPI system with the addition of either $x_i \geq \xi$ or $x_i \leq \xi$ remains feasible. An $O(mn)$ algorithm ($O(n)$ time $O(m)$ processors in parallel) for locating a value was given by Aspvall and Shiloach [4].

In Section 5.3 we introduce a framework for solving TVPI systems and analyze the running time of a deterministic algorithm that relies on locating values. We use this framework to reduce solving TVPI systems to solving a polylogarithmic number of instances of the more general subproblem of *locating a pool of values*: for numerical values $\xi_1, \ldots, \xi_n$ choose bounds $s_1, \ldots, s_n$, where $s_i \in \{x_i \geq \xi_i, x_i \leq \xi_i\}$, such that the system with the additional constraints $s_1, \ldots, s_n$ remains feasible. Obviously, a pool of values can be located by n sequential applications of an algorithm that locates a single value. This yields an $O(mn^2)$ time sequential algorithm which in parallel, runs in $O(n^2)$ time using $O(m)$ processors.

In Section 5.4 we present better algorithms for locating a pool. We first give an algorithm that improves the parallel running time to $O(n)$ using $O(mn)$ processors. We also give an overview of a two stage $\tilde{O}(mn)$ time randomized algorithm. Sections 5.5 and 5.6 are concerned with the details of the two stages.

In Section 5.7 we discuss the special structure of monotone systems. Section 5.8 contains concluding remarks.

$$\begin{array}{lrcl} (1.) & y & \leq & x + 2 \\ (2.) & y & \geq & x \\ (3.) & y & \geq & -x \\ (4.) & y & \leq & -x + 2 \end{array}$$



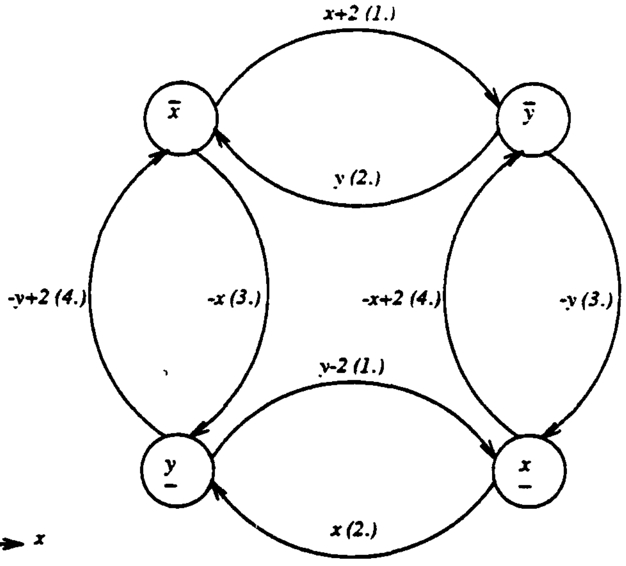


Figure 5.2: An example of a system of inequalities and the associated graph

5.2 Preliminaries

In this section we introduce some terminology, notation, and discuss the properties of TVPI systems. We define some basic constructs and operations. We also present a modification of an algorithm by Aspvall and Shiloach [4] for locating values.

5.2.1 The associated graph

We represent a TVPI system by a set of n intervals and a graph with $2n$ nodes and $2m$ edges. The graph is a natural representation of the system, where variables correspond to vertices and two-variable inequalities to edges between vertices of the participating variables. Directed paths in the graph yield new inequalities. For example, the edges who correspond to $x \geq 3y + 2$, $y \geq 2 - z$, and $z \geq 1 - 2w$ constitutes a directed path of length 3 which yields the inequality $x \geq 5 + 6w$. A formal definition of the associated graph follows.

Definition 5.2.1 Suppose we are given a TVPI system as in Definition 5.1.1. Without loss of generality, assume that the inequalities with a single variable (*bounds*) are summarized in the form of intervals $S_i = [a_i, b_i]$ ($-\infty \le a_i \le b_i \le \infty$, $i = 1, \ldots, n$). We consider a directed graph $G = (V, E)$ as follows. For each variable x_i, there are two vertices in G associated with x_i, namely, $V = \overline{V} \cup \underline{V}$ where $\overline{V} = \{\overline{v}_i \mid i = 1, \ldots, n\}$ and $\underline{V} = \{\underline{v}_i \mid i = 1, \ldots, n\}$. For $u \in V$, we define $u^{-1} \in V$ as follows. If $u = \underline{v}_i \in \underline{V}$, then $u^{-1} = \overline{v}_i$, and if $u = \overline{v}_i \in \overline{V}$, then $u^{-1} = \underline{v}_i$. The edges of G are associated with the inequalities that involve exactly two variables as follows. Each such inequality is represented by two edges, where each edge e is labeled with a certain linear function f_e. The edges corresponding to an inequality of the form

$$\gamma x_i \le \alpha x_j + \beta$$

are as follows:

i. If $\alpha > 0$ and $\gamma = 1$, we have an edge $e = (\overline{v}_j, \overline{v}_i)$ labeled $f_e(x) = \alpha x + \beta$ and an edge $e^{-1} = (\underline{v}_i, \underline{v}_j)$ labeled $f_{e^{-1}}(x) = \frac{1}{\alpha}x - \frac{\beta}{\alpha}$.

ii. If $\alpha < 0$ and $\gamma = 1$, we have an edge $e = (\underline{v}_j, \overline{v}_i)$ labeled $f_e(x) = \alpha x + \beta$ and an edge $e^{-1} = (\underline{v}_i, \overline{v}_j)$ labeled $f_{e^{-1}}(x) = \frac{1}{\alpha}x - \frac{\beta}{\alpha}$.

iii. If $\alpha > 0$ and $\gamma = -1$, we have an edge $e = (\overline{v}_j, \underline{v}_i)$ labeled $f_e(x) = -\alpha x - \beta$ and an edge $e^{-1} = (\overline{v}_i, \underline{v}_j)$ labeled $f_{e^{-1}}(x) = -\frac{1}{\alpha}x - \frac{\beta}{\alpha}$.

See Figure 5.2 for an example of such a system and the associated graph. See Figure 5.3 for the associated graph of the system shown in Figure 5.1. We assume throughout this chapter that a TVPI system is given by the associated graph and set of intervals.

For any one-to-one function f let f^{-1} denote the inverse function. In particular, if $f(x) = \alpha x + \beta$ and $\alpha \ne 0$, then $f^{-1}(x) = \frac{1}{\alpha}x - \frac{\beta}{\alpha}$. Note that for all $e \in E$, $f_{e^{-1}} = f_e^{-1}$.

Let G be as in Definition 5.2.1. A linear function associated with an edge (u, w) corresponds to an inequality in the original system: suppose that $u \in \{\underline{v}_i, \overline{v}_i\}$; if

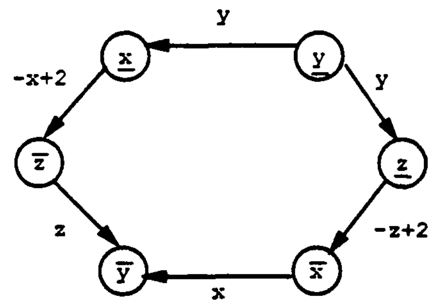


Figure 5.3: The associated graph of the TVPI system of Figure 5.1

$w = \overline{v}_j$ the inequality is $x_j \le f_e(x_i)$, and if $w = \underline{v}_j$ the inequality is $x_j \ge f_e(x_i)$. We define the linear function associated with a directed path p from u to w. The corresponding two variable inequality which results from treating the path as an edge (u, w), is implied by the original system.

Definition 5.2.2 Let $p = (e_1, \ldots, e_k)$ be a (directed) path in G.

i. For any path p, we define a linear function f_p, where $f_p = f_{e_k} \circ \cdots \circ f_{e_1}$. Note that if both ends of p lie either in $\overline{V}$ or in $\underline{V}$ then f_p is increasing. Otherwise, f_p is decreasing.

ii. We denote by p^{-1} the path $p = (e_k^{-1}, \ldots, e_1^{-1})$. Note that $f_{p^{-1}} = f_p^{-1}$ and hence the two paths p, p^{-1} correspond to the same inequality.

Consider, for example, the directed path from $\underline{y}$ to $\overline{z}$ in the graph of Figure 5.3. The linear function associated with this path is $f = 2 - y$, and the corresponding inequality is $z \le 2 - y$.

In particular, Definition 5.2.2 applies to cycles (closed paths) starting at distinguished vertices. Cycles play a special role since they give a relation that involves a single variable, from which a bound on this variable can be deduced. This is formalized in the following definition.

Definition 5.2.3 We extend the definition of a cycle to paths starting at v and ending either at v or at v^{-1}.

i. Let c be a cycle starting at one of the vertices $\overline{v}_i$ and $\underline{v}_i$, and ending at a vertex $\overline{v}_i$ (resp., $\underline{v}_i$). Let f_c be the associated linear function. The bound on x_i implied by the cycle c follows from the inequality $x \leq f_c(x)$ (resp., $x \geq f_c(x)$). Obviously, the implied inequality must hold for all feasible points. Note that if a cycle c starts at $\overline{v}_i$ and ends at $\underline{v}_i$ (resp., starts at $\underline{v}_i$ and ends at $\overline{v}_i$), then f_c is decreasing. Hence, c implies a lower bound (resp., an upper bound) on x_i.

ii. We say that the cycle c *contradicts* a value ξ_i of $\overline{x}_i$ (resp., $\underline{x}_i$) if the bound implied by c is of the form $x_i \geq \alpha$ (resp., $x_i \leq \alpha$) whereas $\xi_i < \alpha$ (resp., $\xi_i > \alpha$).

Consider the graph of Figure 5.2 and the cycle which consists of the path from $\underline{x}$ to $\overline{x}$. The function associated with this cycle is $f = -x + 2$ and the bound implied by the cycle is $x \leq -x + 2$ (i.e., $x \leq 1$).

5.2.2 Pushing bounds along edges

We maintain numerical values associated with the vertices of G; the value associated with $\overline{v}_i$ (resp., $\underline{v}_i$) is denoted by $\overline{x}_i$ (resp., $\underline{x}_i$). Intuitively, $\overline{x}_i$ and $\underline{x}_i$ correspond to upper and lower bounds, respectively, on the value of the variable x_i. We say that ξ is *tighter* than μ as a value of $\overline{x}_i$ (resp., $\underline{x}_i$), if $\xi < \mu$ (resp., $\xi > \mu$). We introduce the following operation which considers an edge (u, v) and updates the bound associated with v, as implied by the inequality associated with (u, v) and the bound associated with u.

Definition 5.2.4 A *push* through an edge e is defined as the following operation:

i. If $e = (\overline{v}_j, \overline{v}_i)$, $\overline{x}_i \leftarrow \min\{\overline{x}_i, f_e(\overline{x}_j)\}$.

ii. If $e = (\underline{v}_j, \underline{v}_i)$, $\underline{x}_i \leftarrow \max\{\underline{x}_i, f_e(\underline{x}_j)\}$.

iii. If $e = (\overline{v}_j, \underline{v}_i)$, $\underline{x}_i \leftarrow \max\{\underline{x}_i, f_e(\overline{x}_j)\}$.

iv. If $e = (\underline{v}_j, \overline{v}_i)$, $\overline{x}_i \leftarrow \min\{\overline{x}_i, f_e(\underline{x}_j)\}$.

A push is said to be *essential* if it actually modifies the value of $\underline{x}_i$ or $\overline{x}_i$.

Proposition 5.2.5 *Suppose $\underline{x}_i$ and $\overline{x}_i$ $(i = 1, \ldots, n)$ are initialized to any values and let X be the set of vectors $x = (x_1, \ldots, x_n)^T$ which (i) satisfy all the given inequalities and (ii) $\underline{x}_i \le x_i \le \overline{x}_i$ $(i = 1, \ldots, n)$. The set X is invariant under pushes.*

Proof: Consider any $x \in X$ and a push through an edge $e = (\overline{v}_j, \overline{v}_i)$. (The arguments for the other cases are similar.) Since x is a feasible solution we have $x_i \le f_e(x_j)$. Note that f_e is an increasing function when both ends of e lie either in $\overline{V}$ or in $\underline{V}$. Hence, from $x_j \le \overline{x}_j$, it follows that $x_i \le f_e(\overline{x}_j)$. Since $x_i \le \overline{x}_i$, we have $x_i \le \min\{\underline{x}_i, f_e(\underline{x}_j)\}$. ∎

It is easy to see that a vector x solves a given TVPI system if and only if (i) $x_i \in S_i$ $(i = 1, \ldots, n)$ and (ii) the set of values $\overline{x}_i = \underline{x}_i = x_i$ $(i = 1, \ldots, n)$ is invariant under pushes.

We define a new operation which amounts to pushing (see Definition 5.2.4) through all the edges simultaneously. This operation is used as a subroutine in algorithms presented later in this section.

Definition 5.2.6 Consider the graph G with some set of values at the vertices, $\underline{x}_i = \underline{x}_i^k$, $\overline{x}_i = \overline{x}_i^k$ $(i = 1, \ldots, n)$. A *push phase* on G is assigning, at the respective vertices, the set of values $\underline{x}_i^{k+1}$, $\overline{x}_i^{k+1}$ $(i = 1, \ldots, n)$ defined as follows:

$$\overline{x}_i^{k+1} \leftarrow \min\left\{\overline{x}_i^k, \min_{e \in \text{out}\{\overline{v}_i\}}\left\{f_e(\overline{x}_j^k) \text{ if } e = (\overline{v}_i, \overline{v}_j), \text{ or } f_e(\underline{x}_j^k) \text{ if } e = (\overline{v}_i, \underline{v}_j)\right\}\right\}$$

$$\underline{x}_i^{k+1} \leftarrow \max\left\{\underline{x}_i^k, \max_{e \in \text{out}\{\underline{v}_i\}}\left\{f_e(\overline{x}_j^k) \text{ if } e = (\underline{v}_i, \overline{v}_j), \text{ or } f_e(\underline{x}_j^k) \text{ if } e = (\underline{v}_i, \underline{v}_j)\right\}\right\}$$

Consider repeated applications of push phases. The initial set of values is denoted by $\underline{x}_i = \underline{x}_i^0$, $\overline{x}_i = \overline{x}_i^0$ $(i = 1, \ldots, n)$, and $\underline{x}_i^k$ and $\overline{x}_i^k$ denote the respective values of $\underline{x}_i$

and $\bar{x}_i$ after the termination of the k'th push phase.
Consider pairs consisting of a value and a vertex, $(\underline{x}_j^k, \underline{v}_j)$ and $(\bar{x}_j^k, \bar{v}_j)$ (where $k \geq 0$ and $1 \leq j \leq n$). The *predecessor* of a pair (ξ, v) is the pair (ξ', v') such that $e = (v', v)$ is the edge through which the essential push determined the value ξ, and $\xi' = f_e^{-1}(\xi)$ is the corresponding value at v'. If the predecessor is not uniquely defined we choose arbitrarily among the qualified pairs. A pair (ξ, v) does not have a predecessor if and only if ξ is the initial value at v.
The *essential path* associated with a pair (ξ, v) consists of the sequence of pairs $(\xi_1, v_1), \ldots, (\xi_\ell, v_\ell)$ and the corresponding sequence of edges $e_1, \ldots, e_{\ell-1}$ such that (i) (ξ_i, v_i) is the predecessor of (ξ_{i+1}, v_{i+1}) and e_i is the edge through which the essential push occurs ($i = 1, \ldots, \ell - 1$), (ii) $(\xi_\ell, v_\ell) = (\xi, v)$, and (iii) (ξ_1, v_1) does not have a predecessor.

It is easy to see that the essential paths which correspond to $(\underline{x}_i^\ell, \underline{v}_i)$ and $(\bar{x}_i^\ell, \bar{v}_i)$ ($i = 1, \ldots, n$) can be found within the same time bounds of performing ℓ push phases, simply by keeping track of all essential push operations.

We extend the notion of a push through an edge to directed paths. Let $p = (e_1, \ldots, e_k)$ be a (directed) path in G. A *push along* p is defined to be the composition of k successive pushes, through the edges $e_1, \ldots, e_k$.

5.2.3 Properties of the feasible region

Consider the bounds on the variables which are implied by directed cycles. We discuss the relation of these bounds to the feasible region. We first give the following definitions.

Definition 5.2.7 i. For each variable x_i, let $[\underline{x}_i^*, \bar{x}_i^*] \subseteq [a_i, b_i]$ be the set of values which are not contradicted by any simple cycle. Note that the two cycles c and c^{-1} imply the same bound. It follows that in order to find $\bar{x}_i^*$ (resp., $\underline{x}_i^*$), it suffices to consider cycles ending at $\bar{v}_i$ (resp., $\underline{v}_i$).

ii. Denote by $[x_i^{\min}, x_i^{\max}] \subseteq [\underline{x}_i^*, \overline{x}_i^*]$ the largest interval such that (1) for all simple paths p from a vertex $\overline{v}_j$ (resp., $\underline{v}_j$) to $\overline{v}_i$ we have $x_i^{\max} \leq f_p(\overline{x}_j^*)$ (resp., $x_i^{\max} \leq f_p(\underline{x}_j^*)$) and, (2) for all simple paths p from a vertex $\overline{v}_j$ (resp., $\underline{v}_j$) to $\underline{v}_i$ we have $x_i^{\min} \geq f_p(\overline{x}_j^*)$ (resp., $x_i^{\min} \geq f_p(\underline{x}_j^*)$).

iii. An interval $I \subset R$ is *infeasible* with respect to the variable x_i if $I \cap [x_i^{\min}, x_i^{\max}] = \emptyset$. A value $\overline{x}_i$ (resp., $\underline{x}_i$) is *infeasible* if $\overline{x}_i < x^{\min}$ (resp., $\underline{x}_i > x^{\max}$).

iv. A value $\overline{x}_i$ (resp., $\underline{x}_i$) is *consistent* with an interval $[a, b]$ if $\overline{x}_i > b$ (resp., $\underline{x}_i < a$).

Remark 5.2.8 When the values $\overline{x}_i^*$ and $\underline{x}_i^*$ $(i = 1, \ldots, n)$ are given, we can compute $x_i^{\min}$ and $x_i^{\max}$ as follows. Initialize the values at the vertices to be $\overline{x}_i^0 = \overline{x}_i^*$ and $\underline{x}_i^0 = \underline{x}_i^*$ $(i = 1, \ldots, n)$. Perform $2n$ push phases. It follows from Definition 5.2.7 part ii that $x_i^{\min} = \underline{x}_i^n$ and $x_i^{\max} = \overline{x}_i^n$. This procedure runs in $O(mn)$ time and is used in Megiddo's algorithm [46].

For example, the TVPI system of Figure 5.2 has $[\underline{x}^*, \overline{x}^*] = [-1, 1] = [x^{\min}, x^{\max}]$, and $[\underline{y}^*, \overline{y}^*] = [0, 2] = [y^{\min}, y^{\max}]$. The TVPI system of Figure 5.3 has $[\underline{x}^*, \overline{x}^*] = [\underline{z}^*, \overline{z}^*] = [-\infty, +\infty]$, $[\underline{y}^*, \overline{y}^*] = [y^{\min}, y^{\max}] = [0, 1]$, and $[x^{\min}, x^{\max}] = [z^{\min}, z^{\max}] = [0, 2]$.

The following key observation is due to Shostak:

Proposition 5.2.9 [57] *If the system is feasible, then the interval $[x_i^{\min}, x_i^{\max}]$ is the projection of the set of solutions on the x_i-axis. Otherwise, $x_i^{\min} > x_i^{\max}$ for some $1 \leq i \leq n$.*

The proof follows from considering the possible structure of minimal sets of inequalities which imply a bound on a variable. Proposition 5.2.9 and the procedure described in Remark 5.2.8 assert that it suffices to consider all *simple* paths and cycles in the associated graph. Shostak presented an exponential time algorithm that essentially examines all directed simple cycles [57].

The following corollary states that if two bounds are feasible separately but not simultaneously, then there exist a simple path that implies an inequality which asserts that the bounds are not simultaneously feasible.

Corollary 5.2.10 *Let S be a set of TVPI constraints, and $s_1 \in \{x_i \geq \alpha, x_i \leq \alpha\}$ $s_2 \in \{x_j \geq \beta, x_j \leq \beta\}$ be two bounds. Let $u_i = \overline{v}_i$ if s_1 is an upper bound, and $u_i = \underline{v}_i$ otherwise; and let $u_j = \overline{v}_j$ if s_2 is an upper bound, and $u_j = \underline{v}_j$ otherwise. Suppose $S \cup \{s_1\}$, and $S \cup \{s_2\}$ are feasible systems, but $S \cup \{s_1, s_2\}$ is not feasible. There exist a simple path p from u_i to u_j^{-1} such that (i) $f_p(\alpha) < \beta$, if s_2 is a lower bound, and (ii) $f_p(\alpha) > \beta$, if s_2 is an upper bound. Moreover, if p is the tightest (simple) path from u_i to u_j^{-1} relative to $x_i = \alpha$, then if s_2 is an upper (resp., lower) bound, $x_j = f_p(\alpha)$ is the smallest (resp., largest) value of x_j subject to $S \cup \{s_1\}$.*

Proof: Suppose that both s_1, s_2 are upper bounds (similar for the other cases). Consider $b_j \equiv x_j^{\min}$ under the system $S \cup \{s_1\}$ and $b_j' \equiv x_j^{\min}$ under the system S. We have $b_j' < \beta$ and $b_j > \beta$. By definition, $x_j^{\min}$ is either (a) implied by a cycle or a cycle and a path (i.e., implied only by inequalities which involve exactly two variables), or (b) implied by a single variable inequality involving some variable x_j and a (possibly empty) directed simple path. Since the system S contains the same two variable inequalities as $S \cup \{s_1\}$, b_j is determined by a system of type (b) consisting of the bound s_1 and a simple path p. ∎

5.2.4 Characterizations of TVPI polyhedra

Proposition 5.2.9 characterizes the feasible region in terms of the associated graph. We characterize polyhedra which comprise sets of solutions of TVPI systems (*TVPI polyhedra*). Polyhedra which can not be expressed as such are *non-TVPI polyhedra.*

Proposition 5.2.11 *Consider a polyhedron $P \subset R^n$. The following statements are equivalent:*

i. The polyhedron P is a TVPI polyhedron.

ii. For a bounded convex set $P' = P \cap B'$, where B' is a box, denote $a_i = \min_{x \in P'} x_i$, $b_i = \max_{x \in P'} x_i$ $(i = 1, \ldots, n)$, and denote by $B = \times_{i=1}^{n}[a_i, b_i]$ the tightest bounding box for P'. Every P' as above is such that $\frac{1}{2}a + \frac{1}{2}b \in P'$.

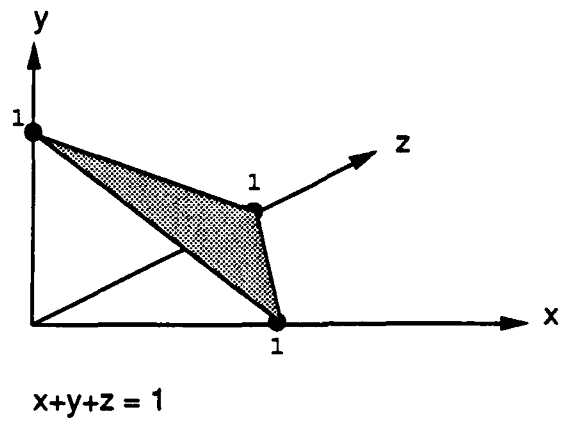


Figure 5.4: An example of a non-TVPI polyhedron

iii. *Every set S of bounds (i.e., conditions of the form $x_i \geq \alpha$, $x_i \leq \alpha$, $x_i = \alpha$) has the following property. Denote by $B_S \in R^n$ the set of all feasible vectors for S. $P \cap B = \emptyset$ if and only if there exist a set $S' \subset S$, where $|S'| \leq 2$ such that $P \cap B_{S'} = \emptyset$.*

iv. *Same as property iii, but S contains only equations.*

Proof: We first show that property i implies properties ii–iv.

i $\Rightarrow$ ii Was proved by Lueker, Megiddo, and Ramachandran [43].

i $\Rightarrow$ iii Suppose P is a TVPI polyhedron, let S_P be a set of TVPI constraints which define P and let $P_{\{i\}} = \hat{x}_i^{\min}, \hat{x}_i^{\max}$ $(i = 1, \ldots, n)$ be the corresponding projections. Let S be a set of bounds such that the combined TVPI system $S_P \cup S$ is infeasible. Note that the associated graphs of S_P and $S_P \cup S$ are identical. Denote by $x_i^{\min}, x_i^{\max}$ $(i = 1, \ldots, n)$ the bounds defined by the system $S \cup S_P$. By definition, each of $x_i^{\min}, x_i^{\max}$ $(i = 1, \ldots, n)$ which does not coincide with the respective end point of the interval $P_{\{i\}}$, is determined by a single bound from S and a simple path. Proposition 5.2.9 implies that the system $S_P \cup S$ is infeasible if and only if for some variable x_i, $x_i^{\min} > x_i^{\max}$. Hence, at least one of the two bounds $x_i^{\min}, x_i^{\max}$ does not coincide with an end point of P_i. Consider the bounds $S' \subset S_P \cup S$ ($|S'| \leq 2$) which determine $x_i^{\min} > x_i^{\max}$. Obviously, the system $S_P \cup S'$ is infeasible.

iii $\Rightarrow$ iv Obvious.

It remains to prove that ii–iv $\Rightarrow$ i. Note that properties i–iv are invariant under scalings and translations of coordinates.

Figure 5.4 gives an example of a non-TVPI polyhedron. It is easy to see that for this polyhedron property ii does not hold (consider the bounding box $0 \leq x \leq 1$, $0 \leq y \leq 1$, $0 \leq z \leq 1$), and property iv does not hold (consider $S = \{x = 0, y = 0, z = 0\}$).

Assume that P is a non-TVPI polyhedron. There exist at least one face F such that the following holds. Let $\boldsymbol{A} \in R^{m \times n}, \boldsymbol{b} \in R^n$ be such that $\{\boldsymbol{x} \in R^n | \ \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}\}$ is the affine hull of F. Consider the process of eliminating variables from the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, until no two equations involve the same variable. At least one of these equations involves more than two variables. Consider such a face F. Without loss of generality (by scaling coordinates) we can assume that the equation is $\sum_{i=1}^{\ell} x_i = 1$ (where $\ell \geq 3$), and for all points $\boldsymbol{x} \in P$, $\sum_{i=1}^{\ell} x_i \leq 1$. Denote by H the hyperplane $H = \{\boldsymbol{x} | \ \sum_{i=1}^{\ell} x_i = 1\}$. It is easy to see that the construction is such that the projection $P' \subset H$ of P on H is full-dimensional, that is, $\operatorname{aff} P' = H$. Consider a point $\hat{\boldsymbol{x}} \in \operatorname{rel\,int} P'$.

iv $\Rightarrow$ i By definition, there exist a small enough $\epsilon \geq 0$ such that for $i = 1, \ldots, \ell$, there exist $\boldsymbol{x}^i \in P$ such that $x_i^i = \hat{x}_i - (\ell - 1)\epsilon$, and $x_j^i = \hat{x}_j + \epsilon$ ($j = 1, \ldots, i-1, i+1, \ldots, \ell$). Consider the set of equations $S = \{x_j = \hat{x}_j + \epsilon | \ j = 1, \ldots, \ell\}$. Obviously, there is no point in P which satisfies S. On the other hand, there is a point which satisfies every subset of $\ell - 1 \geq 2$ constraints. Hence, property iv does not hold for P.

ii $\Rightarrow$ i By definition, there exist a small enough $\epsilon \geq 0$ such that for $i = 1, \ldots, \ell$, there exist $\boldsymbol{x}^i \in P$ such that $x_i^i = \hat{x}_i + (\ell - 1)\epsilon$, and $x_j^i = \hat{x}_j - \epsilon$ ($j = 1, \ldots, i-1, i+1, \ldots, \ell$). Consider the box B defined by the intervals $[\hat{x}_j - \epsilon, \hat{x}_j + (\ell - 1)\epsilon]$

$(i = 1, \ldots, \ell)$. Since $x^i \in B \cap P$, B is a bounding box for $B \cap P$. However, there is no $x \in P$ for which $x_j = \hat{x}_j + (\ell - 2)\epsilon$ $(j = 1, \ldots, \ell)$. It follows that property ii does not hold for P.

∎

5.2.5 Locating values

We discuss two procedures which are used later in the paper and are based on repetitive applications of push phases and examining essential paths.

One of the procedures solves the following problem.

Problem 5.2.12 [Locate a value] For a given TVPI system, a number ξ and a variable x_i, locate ξ with respect to the interval $[x_i^{\min}, x_i^{\max}]$, that is, decide whether (i) $\xi_i < x_i^{\min}$, (ii) $\xi_i > x_i^{\max}$, or (iii) $\xi_i \in [x_i^{\min}, x_i^{\max}]$.

We refer to solving Problem 5.2.12 as *locating* the value ξ_i. Aspvall and Shiloach [3, 4] proved the following:

Proposition 5.2.13 *Problem 5.2.12 can be solved in $O(mn)$ operations.*

Aspvall and Shiloach utilized this result to solve TVPI systems: by solving instances of Problem 5.2.12 their algorithm conducts a binary search which finds a point $\xi_i \in [x_i^{\min}, x_i^{\max}]$ $(1 \le i \le n)$. They showed that finding such a point ξ_i can be done in $O(mnI)$ operations where I is the number of bits in the binary representation of the input. This yielded their $O(mn^2I)$ algorithm for TVPI systems: the algorithm performs n steps, where in step i a point $\xi_i \in [x_i^{\min}, x_i^{\max}]$ is found, and the equation $x_i = \xi_i$ is added to the system. If the original system is feasible, then $\xi \in F$.

In Section 5.3 we present a $\tilde{O}(mn^2)$ deterministic algorithm for solving TVPI systems, which is based on locating values in the time bounds stated in Proposition 5.2.13. The material contained in the remainder of the current section is needed for achieving the randomized bound of $\tilde{O}(n^3 - mn)$ and improving the parallel complexity of solving TVPI systems. We present and analyze an algorithm for locating values which is also applicable to the *reveal* problem which is defined later. The underlying computation amounts to $O(n)$ applications of push phases where essential paths are maintained.

Certificates of infeasibility: Suppose an interval $I = [a, b]$ is infeasible with respect to x_i (equivalently, either $[-\infty, b]$ or $[a, \infty]$ is infeasible). It follows from Proposition 5.2.9 that there exist a vertex w, a simple path p from $u \in \{\underline{v}_i, \overline{v}_i\}$ to w, and a disjoint simple cycle starting at w such that the bound on w implied by c and the path p produce a bound on x_i which does not intersect I. The path p and the cycle c form a minimal subsystem which implies the infeasibility of I. Note that the cycle c may be empty in which case the infeasibility follows from the path p and a bound on the variable associated with w. We refer to such a system as a *certificate* for I. If c starts and ends at u, we say that the certificate is *closed*. If c ends at u^{-1} or c is empty, we say that the certificate is *open*. When we refer to a certificate, we interchangeably mean the set of edges $E' = p \cup c$, the set E'^{-1} of the reversed edges, or the corresponding set of inequalities.

Classifying values: We classify values of variables according to the types of their certificates. Consider an interval I with respect to x_i. If I has a closed certificate we say that I is *strongly infeasible*. If only open certificates exist, we say that I is *weakly infeasible*. Otherwise, if no certificate exists, $I \cap [x_i^{\min}, x_i^{\max}] \neq \emptyset$ and we say that I is feasible.
In particular this definitions apply when I consists of a single point. For $1 \leq i \leq n$, we denote by $[x_i^{\min *}, x_i^{\max *}]$ the set of all feasible and weakly infeasible values of x_i. Note that $x_i^{\min *}, x_i^{\max *}$ $(1 \leq i \leq n)$ are independent of the single variable inequalities. We extend the definitions of the concepts above from intervals to values at vertices of G: A property (having a closed/open certificate, weak/strong infeasibility) holds for a value ξ at a vertex $\underline{v}_i$ (resp., $\overline{v}_i$) if it holds for the interval $[\xi, \infty]$ (resp., $[-\infty, \xi]$) with respect to x_i.

We present two corollaries of Propositions 5.2.5 and 5.2.9. Consider a feasible TVPI system and the associated graph. Suppose a sequence of push phases is performed.

Corollary 5.2.14 *If the values at the vertices are initially consistent (see Definition 5.2.7), they remain consistent after any number of push operations.*

Corollary 5.2.15 *Consider a vertex $u = \underline{v}_i$ (resp., $u = \overline{v}_i$) and a value $\xi = \underline{x}_i^\ell$ (resp., $\xi = \overline{x}_i^\ell$) for some ℓ. Let (ξ_j, v_j) $(j = 1, \ldots, k \leq \ell)$, where $(\xi_k, v_k) \equiv (\xi, u)$, be the pairs comprising the essential path (see Definition 5.2.6) associated with (ξ, u). If ξ is infeasible (resp., strongly infeasible) at u, then each of the pairs (ξ_j, v_j) $(1 \leq j < k)$ is such that ξ_j is an infeasible (resp., strongly infeasible) value of v_j.*

Consider an essential path which contains a cycle (a vertex appears more than once). The following proposition enables us to extract information from the cycle. It states that either all values following the start pair of the cycle are consistent (the path does not add information) or all values preceding the last pair of the cycle are strongly infeasible (an infeasible value is detected). Moreover, by considering the updating cycle we can determine which of the two situations occurs.

Proposition 5.2.16 *Let ξ be a value at $u = \overline{v}_i$ (resp., $u = \underline{v}_i$). Suppose a cycle c, which starts and ends at u, is such that $f_c(\xi) < \xi$ (resp., $f_c(\xi) > \xi$). Then, either (i) ξ is strongly infeasible at u and the cycle c is a closed certificate, or (ii) if $F \neq \emptyset$, ξ is consistent at u.*

Proof: The bound implied by the cycle c is $f_c(y) \geq y$ (resp., $f_c(y) \leq y$). It suffices to show that it holds either (i) only for values y such that $y \geq \xi$, or (ii) only for values y such that $y \leq \xi$. The function $f_c(y) - y$ is linear and thus can change sign only once. It cannot be a positive (resp., negative) constant, since this contradicts $f_c(\xi)$ being tighter than ξ. If the function is a negative (resp., positive) constant, the cycle implies that the system is infeasible (and therefore ξ is contradicted by the cycle). Otherwise, consider the value y^* such that $f_c(y^*) = y^*$. It is immediate that the bound implied by c and the number ξ are on opposite sides of y^*. ∎

The following algorithm is applied to a set of values $\underline{x}_i^0$, $\overline{x}_i^0$ (for $1 \le i \le n$). The algorithm performs a sequence of push phases, and keeps track of essential paths. When the algorithm detects a non simple essential path, it either terminates or discards the path. If the algorithm terminates as a result of such a path, a closed certificate is found for one of the initial values. A vertex v is *active* at a particular point in the execution of the algorithm if the current value at v resulted from an essential path which is not discarded.

Algorithm 5.2.17

i. For $i = 1, \ldots, n$, initialize the values at $\underline{v}_i$ and $\overline{v}_i$ as $\underline{x}_i^0$ and $\overline{x}_i^0$, respectively.

ii. For $k = 1, \ldots, 2n$ do steps iii–vii. If the algorithm did not terminate, determine that none of the initial values is strongly infeasible, and stop.

iii. Perform a push phase. Denote the values at the vertices after the k'th push phase by $\underline{x}_j^k$, $\overline{x}_j^k$ $(1 \le k \le 2n)$.
Keep track of all essential push operations which result from active vertices. If there are no such essential pushes or no active vertices, stop and determine that none of the initial values is strongly infeasible.
(We assume that ambiguities about the edge which carried the essential push are resolved consistently according to some ordering on the edges.)

iv. If $k \neq 2^j$ for all integers j and $k \neq 2n$, go to step iii (next iteration).

v. Optional: If for some j, $\underline{x}_j^k > \overline{x}_j^k$, then stop.

vi. For each active vertex v, let p_v be the essential path that corresponds to the last update of the value at v.
If p_v contains a cycle (i.e., some vertex appears in more than one pair), execute step vii. After considering all vertices, go to step iii (next iteration).

vii. Consider the last simple cycle c on p_v. Denote by u the vertex where the cycle starts and ends. Denote by ξ the value associated with u at its first occurrence on the cycle. Check whether the bound implied by c contradicts ξ. If there is no contradiction, ξ is consistent (see Proposition 5.2.16), discard all essential paths originating from the first pair (u, ξ) of the cycle (all correspond to consistent values according to Corollary 5.2.14); proceed to consider the next vertex. Otherwise, if c contradicts ξ, ξ is strongly infeasible. It follows from Corollary 5.2.15 that so are all values along the path p_v prior to the pair $(u, f_c(\xi))$ (the last occurrence of u on p_v). In particular, the initial value at the first vertex w of p_v is strongly infeasible. Stop.

Complexity: The underlying computation of the algorithm is the Bellman-Ford single-source shortest path computation [13], where we maintain information needed to construct the paths and test for cycles.
The algorithm terminates after $\ell \leq 2n$ iterations, in which case, it requires $O(m\ell)$ sequential time, and $O(\ell)$ time using $O(m)$ processors on a CRCW PRAM.

If the algorithm terminates at step vii, it finds a closed certificate for the initial value at w. If the algorithm terminates during the ℓ'th iteration, the closed certificate found is of size $\theta(\ell)$.

Proposition 5.2.18 *Consider an execution of Algorithm 5.2.17, where step v is skipped.*

i. At least one of the initial values is strongly infeasible if and only if the algorithm terminates at step vii.

ii. Suppose exactly one of the initial values, ξ at the vertex w, is strongly infeasible. The algorithm terminates at step vii with the same closed certificate regardless of the initial values at other vertices. We refer to this certificate as the certificate *of ξ.*

Proof: We first prove part i. The "if" direction is immediate. Suppose an initial value ξ at a vertex w has a closed certificate consisting of a path p of length $|p|$ to u_0

and a cycle c of length r which starts and ends at u_0. After at most $|p|$ iterations, the value at u_0 is at least as tight as $f_p(\xi)$ and hence, has a closed certificate consisting of c. Similarly, after $|p| + r$ iterations the same holds for all the vertices of c. Let $e_0, \ldots, e_r$ and $u_0, \ldots, u_r$, respectively, be the edges and vertices of c. We claim that all iterations for which $k \geq \ell$ result in an update of at least one of the values at $u_0, \ldots, u_r$. Assume the claim is true. Suppose the algorithm did not terminate during any of the $2n$ iterations. It follows that at least one of the paths p_{u_j} $(1 \leq j \leq r)$ is of length $2n$, and hence, contains a cycle. Since the value at u_j is strongly infeasible, it follows from Corollary 5.2.15 that w is the first vertex of p_{u_j}. It follows from Proposition 5.2.16 that each cycle in p_{u_j} contradicts the value at the start vertex.

What remains is to prove the claim. Assume the contrary. Let $\xi_0, \ldots, \xi_r$ be the respective values at $u_0, \ldots, u_r$. In particular, none of the edges e_k had an essential push. Hence, for all $j = 1, \ldots, r$, $f_{e_j}(\xi_j)$ is not tighter than $\xi_{j+1 \bmod r}$. It follows that $f_{e_1 \cdots e_r}(\xi_0) = f_c(\xi_0)$ is less tight than ξ_0. This is a contradiction for c being a closed certificate for ξ_0.

We prove part ii. Consider an initialization where ξ at w is the only strongly infeasible value. It follows from part i that the algorithm terminates in step vii, and finds an essential path p which starts at w and terminates in a cycle. All values along the path are strongly infeasible. Suppose that an initial path starting at a vertex other than w updates a value at a vertex in p. It follows from Corollary 5.2.15 that the updated value can not be tighter or as tight as the value resulting from p. The proof follows. ∎

Remark 5.2.19 Consider a run of the algorithm where step v is performed.

i. Suppose that the algorithm terminates at step v, where it detects $\underline{x}_j^k > \overline{x}_j^k$. Consider the initial pairs (u, ξ) and (w, μ) of the two essential paths which determined $(\underline{v}_j, \underline{x}_j^k)$ and $(\overline{v}_j, \overline{x}_j^k)$. It is not necessarily true that one of the initial pairs is infeasible. We can conclude, however, that the two corresponding bounds can not be satisfied simultaneously by a feasible vector.

ii. Consider a run of the algorithm where at least one of the initial values has an open certificate of size ℓ which contains a nontrivial cycle. The following is immediate: the algorithm terminates at either step vii or step v within ℓ iterations.

Locating a value: We apply Algorithm 5.2.17 to solve Problem 5.2.12.

Consider a value ξ_i. We show how to decide whether or not $\xi_i > x_i^{\max}$ (the case $\xi_i < x_i^{\min}$ is similar).

Algorithm 5.2.20 [Locate a value with respect to $x_i^{\max}$]
Perform a run of Algorithm 5.2.17, where step v is enabled, for the following input of values:
$\underline{x}_j^0 = a_j$ for $j \neq i$, $\overline{x}_j^0 = b_j$ for $j = 1, \ldots, n$, and $\underline{x}_i^0 = \xi_i$.
Conclude as follows:

- If the algorithm stopped at step iii (no active vertices), determine that $\xi_i \leq x_i^{\max}$.
- If the algorithms stopped at step v, consider the essential paths associated with $\underline{x}_j^k > \overline{x}_j^k$. If in neither path, the initial pair is $(\underline{v}_i, \xi_i)$, conclude that the system is infeasible. Otherwise, conclude that $\xi_i > x_i^{\max}$.
- Suppose the algorithm terminated at step vii. If $w = \underline{v}_i$, determine that $\xi_i > x_i^{\max}$. Otherwise, if $w \neq \underline{v}_i$, the system is infeasible.
- If the algorithm terminated at step ii, determine that $\xi_i < x_i^{\max}$.

The correctness follows immediately from Proposition 5.2.18 part i, and Remark 5.2.19.

Reveal a strongly infeasible value: We discuss applying Algorithm 5.2.17 to solve the following problem.

Problem 5.2.21 [Reveal a strongly infeasible value]
Given are values ξ_i, $a_i \le \xi_i \le b_i$ $(i \in I)$ for the respective variables x_i $(i \in I)$, where $I \subset \{1, \dots, n\}$. Do one of the following:

i. Conclude that all the values are feasible or weakly infeasible.

ii. Find a strongly infeasible value ξ_j and a closed certificate.

Algorithm 5.2.22 [Reveal]
Perform a run of Algorithm 5.2.17, where step v is skipped, for the input values: $\overline{x}_i = \underline{x}_i = \xi_i$ $(i \in I)$, and $\overline{x}_i = b_i$, $\underline{x}_i = a_i$ $(i \notin I)$.

The following is a corollary of Proposition 5.2.18:

Corollary 5.2.23 *Algorithm 5.2.22 solves Problem 5.2.21. If none of the initial values is strongly infeasible, the algorithm requires $O(mn)$ operations. Otherwise, the algorithm terminates after $\ell < 2n$ iterations with a closed certificate of size $\theta(\ell)$. If exactly one of the values ξ_i $(i \in I)$ is strongly infeasible, the algorithm terminates with the same closed certificate regardless of the other values.*

5.3 The basic algorithm

In this section we present a framework for solving TVPI systems. This framework allows us to state an algorithm for TVPI systems in terms of solving instances of Problem 5.2.12 (locating single values). The framework is stated in Subsection 5.3.1, and the correctness proof is given in Subsection 5.3.2. In Subsection 5.3.3 we reduce solving a TVPI system to locating $O(n(\log^2 n + \log m))$ values. Since locating a single value requires $O(mn)$ time ($O(n)$ time in parallel) (see Proposition 5.2.13), we obtain an $O(mn^2(\log^2 n + \log m))$ deterministic algorithm for solving TVPI systems, which runs in $\tilde{O}(n^2)$ time in parallel.
In Subsection 5.3.4 we introduce the problem of *locating a pool of values*. A key for further improvements is reducing the solution of a TVPI system to locating $O(\log^2 n+$

$\log m$) pools of values. A pool of values can be located naively by sequentially locating n single values. In Section 5.4 we present faster parallel and sequential algorithms for locating a pool, which yield better algorithms for solving TVPI systems.

5.3.1 The framework

We first describe an idea introduced by Megiddo [46], which is the key in obtaining strongly polynomial time bounds. Consider the associated graph of some TVPI system. Every directed path in the associated graph corresponds to a two-variable inequality. Consider two inequalities which correspond to two paths between the same pair of vertices (v_i, v_j), where $v_i \in \{\overline{v}_i, \underline{v}_i\}$. These inequalities are linear, hence, there exists a number a such that for all $x_i \leq a$ one of the inequalities implies the other, and vice versa for $x_i \geq a$. If we focus only on feasible points x for which $x_i \geq a$ (similarly $x_i \leq a$), one of the inequalities is redundant.
The algorithm eliminates paths and simultaneously restricts the feasible region. When "comparing" two paths, the decision about which one to eliminate is done as follows. First, the number a, as above, is computed. The redundant path is determined by locating a with respect to feasible values of x_i (see Problem 5.2.12).
Megiddo applied the above idea in an algorithm which basically performed n single-source shortest path Bellman-Ford type computations, where comparisons amount to locating values. The framework presented here is based on performing an all-pairs shortest path Floyd-Warshall [13] type computation which allows us to apply further ideas.

The following definition formalizes the concept of comparing paths.

Definition 5.3.1

i. Suppose that p_1 and p_2 are two directed paths from u to v, and from v to w, respectively. Denote by p_1p_2 the path from u to w obtained by concatenating p_1 and p_2.

ii. Suppose that p_1 and p_2 are two paths from u to w. Let $I \subset R$ be an interval. We say that the path p_1 is at least as *tight* as p_2 relative to I (denote it by $p_1 \prec_I p_2$) if either $w \in \underline{V}$ and $f_{p_1}(\xi) \geq f_{p_2}(\xi)$ for all $\xi \in I$, or $w \in \overline{V}$ and $f_{p_1}(\xi) \leq f_{p_2}(\xi)$ for all $\xi \in I$.

iii. Suppose $p_1, p_2, \ldots, p_k$ are paths from u to w. If for some i, $p_i \prec_I p_j$ for all j, we write $p_i = \min_{\prec_I}\{p_1, p_2, \ldots, p_k\}$. Note that when I is a single point, $\min_{\prec_I}$ is well-defined.

The algorithm maintains a set of intervals S_i $(1 \leq i \leq n)$, and a path p_{uw} from u to w for every pair of vertices (u, w). The algorithm runs in $\lceil \log_2 2n \rceil$ phases. During each phase, the paths and the intervals are considered for possible updates. Denote by S_i^k $(1 \leq i \leq n)$ and p_{uw}^k $(\{u, w\} \subseteq V)$ the intervals and paths, respectively, at the beginning of the k'th phase. The algorithm has the following properties: (i) The set $\times_{i=1}^n S_i^k$ contains a feasible point, and (ii) the path p_{uw}^k is the tightest path from u to w, of length at most 2^k, relative to the interval S_i^k (where $u \in \{\overline{v}_i, \underline{v}_i\}$).

The algorithm is based on solving instances of the following problem:

Problem 5.3.2 Given are a graph G and a set of intervals $S_1, \ldots, S_n$ as in Definition 5.2.1. For every ordered pair $(u, v) \in V \times V$ of vertices we are given $p_{uv}^1, \ldots, p_{uv}^{k_{uv}}$, a collection of directed paths from u to v in G. The goal is to find a set of n intervals $I_1, \ldots, I_n$ and select a path $p_{uv}^* \in \{p_{uv}^1, \ldots, p_{uv}^{k_{uv}}\}$ for every pair (u, v) of vertices, where

i. $F \neq \emptyset \Rightarrow (\times_{i=1}^n I_i) \cap F \neq \emptyset$, and

ii. $u \in \{\underline{v}_i, \overline{v}_i\} \Rightarrow p_{uv}^* = \min_{\prec_{I_i}}\{p_{uv}^1, \ldots, p_{uv}^{k_{uv}}\}$.

We later suggest algorithms for Problem 5.3.2. The following algorithm uses it as a subroutine. It first initializes the tightest-paths matrix by selecting the tightest edge out of every set of multiple edges (step ii). The rest of the algorithm consists of $\lceil \log 2n \rceil$ update phases of the tightest-paths matrix (step iv). During each phase, the algorithm considers n^2 sets of n paths (one for each pair of vertices). It then selects the tightest path in each set. In the last step, the tightest path matrix is used to compute a feasible vector.

Algorithm 5.3.3 [Solve TVPI systems]

i. [Initialization] Construct S and G as in Definition 5.2.1.

ii. [Initialize tightest paths matrix] For each pair of vertices consider the set of paths of length 1, i.e., all the multiple edges. Solve Problem 5.3.2 relative to these paths.
For $(u,v) \in V \times V$: $p^0_{uv} \leftarrow p^*_{uv}$. For $i = 1,\ldots,n$: $S_i \leftarrow S_i \cap I_i$, $S^0_i \leftarrow S_i$.

iii. For $k = 1,\ldots,\lceil \log_2 2n \rceil$, execute step iv. To continue, go to step v.

iv. For each pair $(u,v) \in V \times V$, consider the set of paths

$$\left\{p^{k-1}_{uv}\right\} \cup \left\{p^{k-1}_{uw} p^{k-1}_{wv} | w \in V \setminus \{u,v\}\right\} .$$

Solve Problem 5.3.2 relative to these sets of paths.
For $(u,v) \in V \times V$: $p^k_{uv} \leftarrow p^*_{uv}$. For $i = 1,\ldots,n$: $S_i \leftarrow S_i \cap I_i$, $S^k_i \leftarrow S_i$.

v. Denote $p_{\underline{i}\underline{j}} \equiv p^{\lceil \log 2n \rceil}_{\underline{v}_i,\underline{v}_j}$, $p_{\underline{i}\bar{j}} \equiv p^{\lceil \log 2n \rceil}_{\underline{v}_i,\bar{v}_j}$, $p_{\bar{i}\underline{j}} \equiv p^{\lceil \log 2n \rceil}_{\bar{v}_i,\underline{v}_j}$, $p_{\bar{i}\bar{j}} \equiv p^{\lceil \log 2n \rceil}_{\bar{v}_i,\bar{v}_j}$ (for $1 \le i \le n$, $1 \le j \le n$).
For $i = 1,\ldots,n$:
$S_i \leftarrow S_i \cap \{x | f_{p_{\underline{i}\underline{i}}}(x) \le x\}$, $S_i \leftarrow S_i \cap \{x | f_{p_{\underline{i}\bar{i}}}(x) \ge x\}$, $S_i \leftarrow S_i \cap \{x | f_{p_{\bar{i}\underline{i}}}(x) \le x\}$.

vi. For $i = 1,\ldots,n$, compute intervals $S'_i = [a'_i, b'_i]$ as follows:

$$a'_i \leftarrow \max\left\{a_i, \max_j f_{p_{\bar{j}\underline{i}}}(b_j), \max_j f_{p_{\underline{j}\underline{i}}}(a_j)\right\}$$

$$b'_i \leftarrow \min\left\{b_i, \min_j f_{p_{\bar{j}\bar{i}}}(b_j), \min_j f_{p_{\underline{j}\bar{i}}}(a_j)\right\}$$

vii. Compute a feasible solution $\hat{x}$ as follows, for $i = 1,\ldots,n$:

$$\hat{x}_i \in \left[\max\left\{a'_i, \max_{j<i} f_{p_{\bar{j}\underline{i}}}(\hat{x}_j), \max_{j<i} f_{p_{\underline{j}\underline{i}}}(\hat{x}_j)\right\}, \min\left\{b'_i, \min_{j<i} f_{p_{\bar{j}\bar{i}}}(\hat{x}_j), \min_{j<i} f_{p_{\underline{j}\bar{i}}}(\hat{x}_j)\right\}\right]$$

5.3.2 Correctness

We prove the following:

i. For $1 \leq i \leq n$, $S_i^0 \supset S_i^1 \supset \cdots \supset S_i^{\lceil \log_2 2n \rceil} \supset S_i'$.

ii. If $\times_{i=1}^n S_i^0$ contains feasible points, then so do $\times_{i=1}^n S_i^k$ (for all $k > 0$) and $\times_{i=1}^n S_i'$.

iii. p_{uw}^k ($u \in \{\underline{v}_i, \overline{v}_i\}$, $w \in V$) is the tightest path (relative to S_i^k) from u to w, of length less than or equal to 2^k.

iv. If the system is feasible, the process described in step vii results in a feasible vector $\hat{\boldsymbol{x}}$.

Claims i and ii follow directly from the statement of the algorithm.

We prove Claim iii. Consider paths p_1, p_2 between the same pair of vertices. Assume they originate at $\{\underline{v}_j, \overline{v}_j\}$). It follows from the definition of $\prec$ that if $p_1 \prec p_2$ relative to S_j^k, then $p_1 \prec p_2$ relative to S_j^{k+1}. Consider paths of length at most 2^{k+1} from u to v. It suffices to show that when $p_{uv} = p_{uw}p_{wv}$ and $p'_{uv} = p'_{uw}p'_{wv}$, if $p_{uw} \prec_I p'_{uw}$ and $p_{wv} \prec_I p'_{wv}$, then $p_{uv} \prec_I p'_{uv}$. The latter is straightforward.

We prove claim iv. We first show that $S_i' \subset [x_i^{\min}, x_i^{\max}]$ $(i = 1, \ldots, n)$. We claim that $S_j^{\lceil \log 2n \rceil} \subset [\underline{x}_j^*, \overline{x}_j^*]$. If the latter holds, it follows from Proposition 5.2.9 that $S_i' \subset [x_i^{\min}, x_i^{\max}]$. Assume the contrary, that is, for some $1 \leq j \leq n$, $S_j^{\lceil \log 2n \rceil} \not\subset [\underline{x}_j^*, \overline{x}_j^*]$. Consider a point $\eta \in S_j^{\lceil \log 2n \rceil} \setminus [\underline{x}_j^*, \overline{x}_j^*]$. Assume that $\eta > \overline{x}_j^*$ (the case where $\eta < \underline{x}_j^*$ is similar). It follows that there exists a simple cycle c, such that either of the following is true:

i. The cycle c, starts and ends at $\overline{v}_j$, and is such that $f_c(\eta) < \eta$.

ii. The cycle c, starts at $\underline{v}_j$ and ends at $\overline{v}_j$, and is such that $f_c(\eta) < \eta$.

Denote $c' = p_{\overline{v}_j \overline{v}_j}$ (for case (i)) and $c' = p_{\underline{v}_j \overline{v}_j}$ (for case (ii)). It follows from claim iii that the cycle c' is the tightest simple cycle relative to $S_j^{\lceil \log 2n \rceil}$. Therefore, $f_{c'}(\eta) \leq$

$f_c(\eta)$. On the other hand, due to step v of the algorithm $f_{c'}(\eta) \geq \eta$. This is a contradiction. We had shown that $S'_i \subset [x_i^{\min}, x_i^{\max}]$ $(i = 1, \ldots, n)$.

We conclude the proof of Claim iv. Consider the computation performed in step vii. For $i = 1, \ldots, n$ consider the set of intervals $\hat{S}^i_\ell$ $(1 \leq \ell \leq n)$ defined as follows. If $\ell < i$, $\hat{S}^i_\ell = \{\hat{x}_\ell\}$. Otherwise,

$$\hat{S}^i_\ell = \left[\max\left\{a'_\ell, \max_{j<i} f_{p_{\bar{j}\underline{\ell}}}(\hat{x}_j), \max_{j<i} f_{p_{j\underline{\ell}}}(\hat{x}_j)\right\}, \min\left\{b'_\ell, \min_{j<i} f_{p_{\bar{j}\bar{\ell}}}(\hat{x}_j), \min_{j<i} f_{p_{\underline{j}\bar{\ell}}}(\hat{x}_j)\right\}\right] .$$

We claim that for $i = 1, \ldots, n$ the following holds. For each $\mu \in \hat{S}^i_j$ there exist a feasible vector ξ such that $\xi_j = \mu$ and $\xi_\ell \in \hat{S}^i_\ell$ $(\ell \neq j)$. To conclude the proof of property iv, it suffices to prove the claim. We prove the claim by induction on i.
In the base case $\hat{S}^1_\ell = S'_\ell$ $(1 \leq \ell \leq n)$. We show that for every $\xi_j \in S'_j$ there exists a feasible solution x such that $x_j = \xi_j$ and $x_\ell \in S'_\ell$ for $\ell \neq j$. Assume the contrary. It follows from Proposition 5.2.11 that there exist two bounds $s_1 \in \{x_j \geq \xi_j, x_i \leq \xi_j\}$ and $s_2 \in \{x_\ell \leq b'_\ell, x_\ell \geq a'_\ell\}$ such that the system subject to $x_k \in S'_k$ $(1 \leq k \leq n)$ with the additional bounds s_1, s_2 is infeasible. Assume that s_1 is $x_j \geq \xi_j$ and s_2 is $x_\ell \leq b'_\ell$ (the other cases are treated similarly). It follows from Corollary 5.2.10 that there exists a simple path p from $\bar{v}_\ell$ to $\bar{v}_j$ such that $f_p(b'_\ell) < \xi_j$. In contrast, it follows from property iii and the computation of step vi that $f_{p'}(b'_\ell) \geq b'_j$ for all paths p' from $\bar{v}_\ell$ to $\bar{v}_j$, hence a contradiction.
We prove the correctness of the induction step. Consider the step which determines $\hat{x}_i$. Assume that the claim is true for previous steps. The induction hypothesis asserts that subject to the constraints $x_\ell \in \hat{S}^i_\ell$ $(1 \leq \ell \leq n)$, we have $[x_\ell^{\min}, x_\ell^{\max}] = \hat{S}^i_\ell \equiv [\hat{a}^i_\ell, \hat{b}^i_\ell]$. For $1 \leq \ell \leq n$, let $\tilde{x}_\ell^{\min}$, $\tilde{x}_\ell^{\max}$ be the respective values of $x_\ell^{\min}$, $x_\ell^{\max}$ subject to $x_\ell \in \hat{S}^i_\ell$ $(1 \leq \ell \leq n)$ and the additional constraint $x_i = \hat{x}_i$. We need to show that for $1 \leq \ell \leq n$, $[\tilde{x}_\ell^{\min}, \tilde{x}_\ell^{\max}] = \hat{S}^{i+1}_\ell$. The direction $\hat{S}^{i+1}_\ell \supset [\tilde{x}_\ell^{\min}, \tilde{x}_\ell^{\max}]$ is obvious. Consider $\tilde{x}_j^{\max}$ (the arguments for $\tilde{x}_j^{\min}$ are similar). If $\tilde{x}_j^{\max} = \hat{b}^i_j$ we are done. Otherwise, it follows from Proposition 5.2.9 that there exist a simple path p from $v_i \in \{\bar{v}_i, \underline{v}_i\}$ to $\underline{v}_j$ such that $f_p(\hat{x}_i) = \tilde{x}_j^{\max}$. It follows from property iii that either $p' = p_{\bar{i}\underline{j}}$ or $p' = p_{\underline{i}\underline{j}}$ is as tight as p. Hence, $f_{p'}(\hat{x}_i) = \tilde{x}_j^{\max} = \hat{b}^{i+1}_j$.

Remark 5.3.4 Note that step vii can be replaced by choosing $\hat{x}_i = (a'_i + b'_i)/2$, $1 \leq i \leq n$. This follows from the fact that for the system subject to $x_i \in S'_i$

$1 \leq i \leq n$, we have $[x_i^{\min}, x_i^{\max}] = S_i'$ $(1 \leq i \leq n)$, and from Proposition 5.2.11 part ii.

5.3.3 Complexity of the naive implementation

The complexity is dominated by the calls to an algorithm for Problem 5.3.2. We present a naive algorithm for the problem, which is based on locating single values (see Problem 5.2.12). The algorithm consists of n sequential stages as follows. At the i'th stage the interval I_i is computed and a tightest path is found for each of the $O(n)$ sets of paths which emanate from either one of $\{\underline{v}_i, \overline{v}_i\}$. In the proceeding stages we consider the system with the additional constraint $x_i \in I_i$.

We discuss stage i. We consider $O(n)$ sets of paths with the goal of choosing a tightest path in each set. This is done in $O(\log k^{(i)})$ iterations, where in each iteration the total number of paths which need to be considered reduces from r to $n+3(r-n)/4$. Initially, $r = k^{(i)}$. After the stage terminates, $r = n$ and each set contains a single path (the tightest path). Each iteration is as follows. First, we pair up paths which belong to the same pair of vertices. Each such pair corresponds to a comparison between the two paths that needs to be resolved. For each pair, we compute the intersection of the two linear functions which correspond to the two paths. Each "comparison" amounts to locating the intersection point with respect to the interval $[x_i^{\min}, x_i^{\max}]$. We solve one instance of Problem 5.2.12 to locate the median of these $r/2$ intersections. By doing this, half the comparisons are resolved, and the number of remaining paths is at most $n + 3(r-n)/4$.

The following proposition is immediate.

Proposition 5.3.5 *Stage i can be performed by an $O(k^{(i)} + mn \log k^{(i)})$ algorithm, where $k^{(i)} = \sum_v (k_{u^{-1}v} + k_{uv})$ $(u = \overline{v}_i)$. A parallel algorithm runs in $O(n \log k^{(i)})$ time using $O(m + k^{(i)})$ processors on a CRCW PRAM.*

We discuss the resulting complexity of Algorithm 5.3.3. In step ii, the total number of paths $(\sum_{i=1}^{n} k^{(i)})$ is the number of edges in the graph. Hence the number of operations is $O(mn^2 \log m)$. In step iv, $k^{(i)} = n^2$ $(1 \leq i \leq n)$. Hence, the number of operations in each execution of step iv is $O(n^3 + mn^2 \log n)$. Step iv is performed $\lceil \log_2 2n \rceil$

times. It follows that the total number of operations is $O\left(mn^2(\log m + \log^2 n)\right)$. On a CRCW PRAM the algorithm runs in $O\left(n^2(\log m + \log^2 n)\right)$ time, using $O(m + n^2)$ processors.

5.3.4 Solving TVPI systems by locating pools of values

Consider the "naive" algorithm for solving Problem 5.3.2. The algorithm consists of n stages which are performed sequentially. We present a different algorithm where the stages are performed "concurrently" with interleaving iterations: the algorithm consists of $O(\max_{1\leq i\leq n} \log k^{(i)})$ phases, where phase i comprises the i'th iterations at each of the n stages.

An iteration of stage $1 \leq j \leq n$ amounts to locating a value of x_j. Performing a phase amounts to solving an instance of the following problem:

Problem 5.3.6 [Locate a pool of values] Given are a graph G and a set of intervals $S_1, \ldots, S_n$ as in Definition 5.2.1. Also given are a set of values ξ_i $(i \in I)$ for the corresponding variables x_i $(i \in I)$, where $I \subset \{1, \ldots, n\}$. The goal is to find a set of intervals J_i $(i \in I)$, such that (i) $F \neq \emptyset \Rightarrow F \cap \{x \in R^n | \bigwedge_{i\in I}\ x_i \in J_i\} \neq \emptyset$, and (ii) $\xi_i \notin$ interior J_i $(i \in I)$.

We give a more elaborate description of the algorithm which reduces Problem 5.3.2 to locating pools of values. The correctness is straightforward. The algorithms performs interleaving executions of the "n" stages. This is done in $O(\log \max_i k^{(i)})$ phases. Denote by $I \subset \{1, \ldots, n\}$ $(|I| = \ell)$ the set of stages which did not terminate at the current phase (initially, $I = \{1, \ldots, n\}$). Phase j is as follows:

i. For each $\ell \in I$: compute the value ξ_ℓ of x_ℓ arising from an iteration of stage ℓ. This computation requires $O(n + (3/4)^j \sum_{i\in I} k^{(i)})$ time, and requires logarithmic time in parallel with optimal speedup.

ii. Locate the pool (solve Problem 5.3.6) ξ_ℓ ($\ell \in I$).

iii. For $\ell \in I$, $S_\ell \leftarrow S_\ell \cap J_\ell$.

It follows that Problem 5.3.2 is reduced to solving $O(\max_i \log k^{(i)})$ instances of Problem 5.3.6 and $O(n \max_i \log k^{(i)} + \sum_{1\le i\le n} k^{(i)})$ additional time. The problem of solving TVPI systems is therefore reduced to solving $O(\log^2 n + \log m)$ instances of Problem 5.3.6 and $O(m + n^3 \log n)$ additional computation. Note that, in parallel, the additional computation can be done in logarithmic time with optimal speedup.

Problem 5.3.6 can be solved naively by sequentially solving $|I|$ instances of Problem 5.2.12 (locating a single value). This requires mn^2 time, and $O(n^2)$ time in parallel. In Section 5.4 we give algorithms which improve over this bound.

5.4 Algorithms for locating a pool

The problem of solving TVPI systems was reduced to locating $O(\log^2 n + \log m)$ pools (Problem 5.3.6) and $O(n^3 \log n + m)$ additional computation.

We had shown that a pool can be located in mn^2 time, and $O(n^2)$ time in parallel. In this section we discuss two algorithms for locating a pool. In Subsection 5.3.6 we present an algorithm which runs in $O(mn^2)$ sequential time, and $O(n)$ time in parallel with optimal speedup. In Subsection 5.4.2 we overview a randomized $O(mn \log^3 n)$ expected time algorithm. A parallel implementation runs in $\tilde{O}(n)$ expected time using $O(m)$ processors. The details of the randomized algorithm are given in Sections 5.5 and 5.6.

5.4.1 $O(n)$ time using $O(nm)$ processors

In this subsection we prove the following.

Proposition 5.4.1 *Problem 5.3.6 can be solved on a CRCW PRAM in $O(n)$ time using $O(m|I|)$ processors.*

Corollary 5.4.2 *Algorithm 5.3.3 has a parallel implementation on a CRCW PRAM which runs in $O\left(n(\log^2 n + \log m)\right)$ time and uses $O(mn)$ processors.*

The following algorithm solves Problem 5.3.6 within the time bounds stated in Proposition 5.4.1.

Algorithm 5.4.3 [locate a pool]

i. Locate, in parallel, the values ξ_i $(i \in I)$.
   For $i \in I$ do as follows:
   - If $\xi_i \geq x_i^{\max}$, determine $J_i \leftarrow \{z | z \leq \xi_i\}$.
   - If $\xi_i \leq x_i^{\min}$, determine $J_i \leftarrow \{z | z \geq \xi_i\}$.

   Let $I' \subset I$ be the set of indices such that $x_i^{\min} < \xi_i < x_i^{\max}$ $(i \in I')$.

ii. **For each $i \in I'$ perform the following computation:**
   Initialize the values at the vertices of G to $\overline{x}_i = \underline{x}_i = \xi_i$ and $\overline{x}_j = \infty$, $\underline{x}_j = -\infty$ $(j \neq i)$. Apply $2n$ push phases. For $1 \leq j \leq n$ denote the final values at the nodes $\underline{v}_j$, $\overline{v}_j$, respectively, by $\underline{x}_j^i, \overline{x}_j^i$.

iii. **Construct a graph H as follows: The graph H has $|I'|$ nodes w_i $(i \in I')$. There is an edge between w_i and w_j if and only if $\xi_j < \underline{x}_j^i$ or $\xi_j > \overline{x}_j^i$.**

iv. **Compute a maximal independent set (corresponds to nodes $I^+ \subset I'$) in H.**
   Choose intervals J_i $(i \in I')$ as follows:
   If $i \in I^+$, $J_i \leftarrow \{\xi_i\}$.
   If $i \in I' \setminus I^+$, then:
   - **If $\min_{j \in I^+} \overline{x}_i^j < \xi_i$, $J_i \leftarrow \{z | z \leq \xi_i\}$.**
   - **If $\max_{j \in I^+} \underline{x}_i^j > \xi_i$, $J_i \leftarrow \{z | z \geq \xi_i\}$.**

We prove the correctness of the algorithm. It follows from Corollary 5.2.10 that $F \cap \{x | x_i = \xi_i \ \wedge \ x_j = \xi_j\} \neq \emptyset$ if and only if $\underline{x}_j^i \leq \xi_j \leq \overline{x}_j^i$. Hence, the graph H captures the dependencies between pairs of values. Proposition 5.2.11 (see equivalence of properties i and iv) implies that a set of single variable equations is feasible if and only if every pair is feasible. Hence, the set of intervals J_i $(i \in I')$ solves Problem 5.3.6.

5.4.2 Overview of a $\tilde{O}(mn)$ algorithm

In this subsection we present an overview of a faster randomized algorithm for locating a pool of values. We prove the following:

Theorem 5.4.4 *Problem 5.3.6 can be solved (i) sequentially, in an expected number of $O(mn\log^3 n)$ operations, and (ii) on a CRCW PRAM, in $O(n\log^3 n)$ expected time, using $O(m)$ processors.*

Consequently, TVPI systems can be solved in an expected number of

$$O\left(n^3\log n + mn(\log^5 n + \log m \log^3 n)\right)$$

operations.

A single value can be located in $O(mn)$ steps by using Algorithm 5.2.20. In the previous section we solved Problem 5.3.2 more efficiently by inferring from locating a value of one variable about other values of the same variable. The randomized approach presented in this section enables us to infer about values of other variables as well.

Consider a pool of values $\xi_1, \ldots, \xi_n$ for the corresponding variables. These values are classified into 3 groups as follows:

i. $\xi_i \in [x_i^{\min}, x_i^{\max}]$ (ξ_i is feasible)

ii. $\xi_i \notin [x_i^{\min}, x_i^{\max}]$, but $\xi_i \in [x_i^{\min *}, x_i^{\max *}]$ (ξ_i is weakly infeasible)

iii. $\xi_i \notin [x_i^{\min *}, x_i^{\max *}]$ (ξ_i is strongly infeasible)

We suggest a two stage algorithm for Problem 5.3.6. In the first stage the algorithm locates all the strongly infeasible values ξ_i ($i \in I'$), and determines whether $\xi_i < x_i^{\min *}$ or $\xi_i > x_i^{\max *}$. The respective intervals are determined to be $J_i = [-\infty, \xi_i]$ if $\xi_i > x_i^{\max *}$, and $J_i = [\xi_i, \infty]$ if $\xi_i < x_i^{\min *}$. In the second stage the algorithm solves an easier special case of Problem 5.3.6 where the values are guaranteed to be either feasible or weakly infeasible. In Section 5.5 we present an algorithm which solves the first stage in $O(mn\log^3 n)$ time, and in $O(n\log^3 n)$ time using $O(m)$ processors on a CRCW PRAM. In Section 5.6 we present an algorithm for the second stage which computes a solution in time $O(mn\log^2 n)$, and in $O(n\log^2 n)$ parallel time using $O(m)$ processors on a CRCW PRAM.

From combining the above results we get algorithms for Problem 5.3.6 with the running times stated in Theorem 5.4.4.

5.5 Locating the strongly infeasible values

We present an algorithm for the following problem:

Problem 5.5.1 [Determine the strongly infeasible values] Given are values ξ_i ($i \in I$) for the corresponding variables x_i ($i \in I$), where $I \subset \{1, \ldots, n\}$. The goal is to determine for each $i \in I$ whether $\xi_i < x^{\min *}$, $\xi_i > x^{\max *}$, or $\xi_i \in [x^{\min *}, x^{\max *}]$.

We prove the following:

Proposition 5.5.2 *Problem 5.5.1 can be solved (i) sequentially, in an expected number of $O(mn\log^3 n)$ operations, and (ii) on a CRCW PRAM, in $O(n\log^3 n)$ expected time, using $O(m)$ processors.*

For purposes of analysis we classify the strongly infeasible values of variables according to the sizes of their certificates:

Definition 5.5.3 A strongly infeasible value ξ_i of a variable x_i is *ℓ-big* if the certificate is of size at most ℓ. A value is $(\ell_1, \ell_2]$-big if it is ℓ_2-big but not ℓ_1-big. We interchangeably refer to strongly infeasible values as *big*. Note that all strongly infeasible values are $2n$-big.

We explain the motivation for this classification. The algorithm which determines all the big values is based on a tradeoff between the following two properties. These properties are stated more formally and proved later. The first one "favors" big values with small certificates: For any ℓ-big value ξ it takes $O(m\ell)$ operations to find the certificate. Hence, a decision procedure which locates an ℓ-big value requires only $O(m\ell)$ operations. The second property favors values with large certificates: We introduce the procedure "check" that considers a big value ξ along with an associated certificate E'. The "check" procedure can on average locate many other big values whose certificates intersect E'.

5.5.1 The algorithm

The description of the algorithm includes calls to the following three procedures:

- The first procedure "reveal" (Problem 5.2.21) considers a set of k values, and either finds a big value which belongs to the set, or concludes that all these values are feasible or weakly infeasible. Algorithm 5.2.22 solves "reveal". Recall (see Proposition 5.2.23) that the algorithm takes $O(mn)$ time. If the input set contains big values, the algorithm terminates after $O(m\ell)$ operations (where $\ell \leq n$) with a closed certificate of size $\theta(\ell)$. If the original set contained exactly one big value ξ_j, the certificate found by the algorithm is the certificate of ξ_j.

- The second procedure "check" considers an ℓ-big value and the corresponding certificate. The "check" procedure uses the certificate to produce upper and lower bounds on the feasible regions of other variables. This gives rise to decisions regarding other values in the pool. An $O(m\ell)$ time "check" algorithm is given in Subsection 5.5.3. In Subsection 5.5.4 we discuss properties of the "check" algorithm. We show that at least half of the $(\ell, 2\ell]$-big values in the pool have the following property: When a "check" is applied to any of them, it determines at least $\ell/(6n)$ of the other $(\ell, 2\ell]$-big values in the pool.

- The third procedure considers the set of unlocated values I and computes a crude estimate r^* to the number of big values r. The estimate r^* is such that $1/2 \leq r^*/r \leq 2$ with probability at least $1/2$. In Subsection 5.5.5 we present an "estimate" algorithm which performs $O((\log\log n)^2)$ calls to the "reveal" procedure. The time complexity is dominated by these calls.

The following algorithm determines all big values in a pool of $|I| = \hat{n}$ values. The algorithm performs iterations where each iteration determines some of the big values. The set I and the number $\hat{n}$ are updated accordingly. Denote by $r \leq \hat{n}$ be the (unknown) number of big values in I. Appropriate values for the constants C_1, C_2, and C are given later.

Algorithm 5.5.4 [Determine all the big values]

◇ Loop A: Steps i-ix

i. Apply the "reveal" procedure to the set I. If there are no big values, stop.

ii. Compute an estimate r^* for r.
Reset the following two counters: $tt \leftarrow 0$ [number of operations];
$tb \leftarrow 0$ [number of big values discarded] (in the current iteration of Loop A).

◇ Loop B: Steps iii-ix

iii. If either (i) $tb \geq r^*/4$ (successful iteration of Loop A), or (ii) $tt \geq Cmn\log^2 n$ (unsuccessful iteration of Loop A), go to Step i (next iteration).

iv. $b \leftarrow 0$ [the number of big values discarded in the current iteration of Loop B].

v. If $r^* \geq \hat{n}/4$, $s \leftarrow 1$. Otherwise, $s \leftarrow \lfloor \hat{n}/r^* \rfloor$. Choose $k = \lceil C_1 \log n \rceil$ random samples $S_1, \ldots, S_k$ of size s (with returns) from the pool I.

vi. Execute "concurrently" k runs of "reveal" (see Algorithm 5.2.22) applied to the sets $S_1, \ldots, S_k$ as follows:

vii. Initialize copy i according to the set of values S_i. Set $\ell \leftarrow 0$ [current phase number]. Set $K \leftarrow \{1, \ldots, k\}$ [set of "active" runs].

◇ Loop C: Steps viii-ix

viii. Perform an additional iteration (push phase) to the runs in K. set $\ell \leftarrow \ell + 1$. Let $K' \subset K$ be the possibly empty subset of the runs, for which certificates are found. Apply "check" operations to these certificates, and discard from I all the big values which are determined. Increment b and tb accordingly, and set $K \leftarrow K \setminus K'$.

ix. If either $\ell = 2n + 1$ or $b \geq \lceil C_2 \ell r^* / (n \log n) \rceil$ then $tt \leftarrow tt + C_1 \ell m \log n$, go to Step iii [next iteration of Loop B]. Otherwise, go to Step viii.

In Subsection 5.5.2 we prove the following:

Proposition 5.5.5 *There exist constants C_1,C_2, and C as follows. If the estimate r^* is such that $1/2 \leq r^*/r \leq 2$, then with probability $1/2$, the current iteration of Loop A terminates after determining $r^*/4$ big values (the iteration is successful).*

It follows that with probability 1/2, each iteration of Loop A determines at least 1/8 of the big values in the pool. Hence, the expected number of iterations performed until all big values are determined is at most $6 \log n$. Each iteration of Loop A runs in $O(mn \log^2 n)$ time. Hence, the expected time in which Algorithm 5.5.4 terminates is $O(mn \log^3 n)$. This concludes the proof of Proposition 5.5.2.

5.5.2 Probability for a successful iteration of Loop A

In this subsection we prove Proposition 5.5.5.

Denote by r_i ($i = 1, \ldots, \lceil \log n \rceil$) the number of $(2^{i-1}, 2^i]$-big values in I.

Definition 5.5.6 A sample $S \subset I$ of values from the pool is *good* if:

i. The sample contains exactly one big value ξ_j. Denote the size of the certificate of ξ_j by $2^{i-1} < \ell \leq 2^i$.

ii. ξ_j determines (using "check") at least $2^{i-1} r_i / (6n)$ big values in I.

iii. $r_i \geq r/(2\log n)$.

Note that since $r = \sum_{i=1}^{\lceil \log n \rceil} r_i$, property (iii) must hold for at least 3/4 of the big values. It follows from the analysis done in Subsection 5.5.4 (for the "check" procedure) that property (ii) holds for half the $(2^{i-1}, 2^i]$-big values (for all i). Hence, properties (ii) and (iii) hold for 3/8 of the big values.

The following proposition motivates the definition of good samples:

Proposition 5.5.7 *Consider an application of the reveal algorithm to a good sample, followed by a check to the certificate found. There exist a number $\ell = 2^i$ ($1 \leq i \leq \lceil \log n \rceil$), such that by using $O(m\ell)$ operations we can locate at least $\ell r/(12n\log n) + 1$ big values.*

Proof: Denote by ξ_j the big value in the sample, and by $2^{i-1} < \ell \leq 2^i$ the size of the certificate of ξ_j. Consider an application of the reveal algorithm to the sample. It follows from Proposition 5.2.18 part ii that the certificate is found using $O(m\ell)$ operations. A check procedure applied to the certificate locates at least $2^{i-1}r_i/(6n)$ additional big values from the pool. Note that $r_i \geq r/(2\log n)$, hence at least $\ell r/(12n\log n) + 1$ big values are located. ∎

Proposition 5.5.8 *Consider a number s such that (i) $s = 1$ if $r \geq \hat{n}/2$ and sometimes when $r \geq \hat{n}/8$, and (ii) otherwise, $c = rs/\hat{n}$ is such that $0.5 \leq c \leq 2$. Consider a random sample $S \subset I$ of s elements (with returns). There exists a constant ρ_1 such that S contains exactly one big value with probability at least ρ_1.*

Proof: The probability that S contains exactly one big value is

$$P = s\frac{r}{\hat{n}}\left(1 - \frac{r}{\hat{n}}\right)^{s-1} .$$

Assume that $s = 1$. It follows that $8r > \hat{n}$ and hence $P \geq 1/8$. Otherwise, $2r < n$ and $P \geq c/4^c \geq 1/32$. ∎

Corollary 5.5.9 *Suppose that* $1/2 \leq r^*/r \leq 2$, *and consider a random sample* $S \subset I$ *(with returns) of size* s, *where* $s = 1$ *when* $r^* \geq \hat{n}/4$ *and* $s = \lfloor \hat{n}/r^* \rfloor$ *otherwise. The sample* S *is good with probability* $\rho \geq 3\rho_1/8$.

Corollary 5.5.10 *Let* $C_1 = -1/\log(1-\rho)$. *Consider* $\lceil C_1 \log n \rceil$ *randomly chosen samples (with returns) of size* s. *If* $1/2 \leq r^*/r \leq 2$, *the probability that none of the samples is good is smaller than* $1/n$.

Proof: The probability that no sample is good is $p = (1-\rho)^{\lceil C_1 \log n \rceil}$. It follows that $p \leq 1/n$ when $C_1 \geq -1/\log(1-\rho)$. ∎

Assume that $1/2 \leq r^*/r \leq 2$, and choose $C_2 = 1/12$. We compute the expected number of steps performed until $r^*/4$ big values are located.

Definition 5.5.11 An iteration of Loop B is *beneficial* if $b \geq \lceil C_2 \ell r^*/(n \log n) \rceil$ when the iteration terminates (at step ix).

A beneficial iteration requires $O(C_1 m \ell \log n)$ operations, and each unbeneficial iteration requires $O(C_1 mn \log n)$ operations.
It follows from Proposition 5.5.7 and Corollary 5.5.10 that $(n-1)/n$ of all iterations of Loop B are beneficial. Hence, the expected number of operations performed by unbeneficial iterations is not bigger than the expected number of operations performed by beneficial iterations.

It follows that it suffices to count the expected number of operations performed by beneficial iterations until the number of big values is reduced by a half (hence, at least $r^*/4$ big values are located).
Denote by t_i ($i = 1, \ldots, \lceil \log n \rceil$) the number of beneficial iterations which ended within $(2^{i-1}, 2^i]$ phases. The total number of steps performed by beneficial iterations is $ss = C_1 \sum_{i=1}^{\lceil \log n \rceil} t_i 2^i m \log n$. By definition, half the big values are eliminated when $\Pi_{i=1}^{\lceil \log n \rceil}(1 - C_2 2^{i-1}/(n \log n))^{t_i} = 1/2$. By applying a log operation to both sides, it follows that $\sum_{i=1}^{\lceil \log n \rceil}(t_i C_2 2^{i-1}/(n \log n)) \leq 2$. The total number of operations of beneficial iteration until half the big values are decided is therefore $ss \leq 8C_1/C_2 mn \log^2 n$. The

expected number of operations in beneficial and unbeneficial operations is at most $ss \leq 16C_1/C_2 mn\log^2 n$. We choose $C = 20C_1/C_2$. It follows that with probability 1/2, at least half of the big values are located before the current iteration of Loop A performs $Cmn\log^2 n$ operations. This concludes the proof of proposition 5.5.5.

5.5.3 The "check" procedure

In this subsection we give an algorithm for performing "checks." The "check" algorithm is applied to a $(2^{k-1}, 2^k]$-big value ξ_i and the certificate ξ_i, and provides information from which it can be concluded that certain other values are also big.

We explain how a closed certificate for a value of one variable is used to obtain information about other variables:

Remark 5.5.12 Suppose ξ_i is a big value of x_i. The certificate G_i consists of a path from v_i to a vertex u and a cycle which starts and ends at u, such that $v_i = \overline{v}_i$ if $\xi_i > x_i^{\max *}$, and $v_i = \underline{v}_i$ if $\xi_i < x_i^{\min *}$. For a vertex w in G_i, define $\hat{\xi}_i(w) = f_p(\xi_i)$, where p is the path in G_i from v_i to w. Define $\hat{\xi}_i(v_i) = \xi_i$. It follows from Corollary 5.2.15 that for every vertex w of G_i, the value $\hat{\xi}_i(w)$ at w is strongly infeasible.

The "check" algorithm is described below. The input is a $(2^{k-1}, 2^k]$-big value ξ_i of x_i along with the certificate G_i.

Algorithm 5.5.13 [check]

i. For each vertex $v \in G_i$, compute the big value $\xi(v)$ (as in Remark 5.5.12).

ii. Initialize the values of $\overline{x}_1, \ldots, \overline{x}_n$ and $\underline{x}_1, \ldots, \underline{x}_n$ as follows. If $\overline{v}_j \in G_i$, set $\underline{x}_j = \xi(\overline{v}_j)$; otherwise, if $\overline{v}_j \notin G_i$, set $\underline{x}_j = -\infty$. If $\underline{v}_j \in G_i$, set $\overline{x}_j = \xi(\underline{v}_j)$; otherwise, if $\underline{v}_j \notin G_i$, set $\overline{x}_j = \infty$.

iii. Perform 2^k push phases.

iv. Make conclusions as follows. If for some value η we have $\eta \geq \overline{x}_j$ (resp., $\eta \leq \underline{x}_j$), conclude that $\eta > x_j^{\max *}$ (resp., $\eta < x_j^{\min *}$).

The initial values at the vertices are consistent (see Remark 5.5.12), and hence (see Corollary 5.2.14), the final values are consistent. It follows that the conclusions made by the "check" algorithm are correct.

Definition 5.5.14 If in step iv of Algorithm 5.5.13 we deduce that a value η is big, we say that ξ_i *2^k-locates* η.

5.5.4 Properties of the "check" procedure

This subsection establishes the following theorem:

Theorem 5.5.15 *Consider a collection of b $(2^{k-1}, 2^k]$-big values. At least half the values have the following property. If a "check" is applied to any of them, it results in 2^k-locating at least $b2^{k-1}/(6n))$ other values from the collection.*

The following proposition analyzes the dependencies among values of different variables.

Proposition 5.5.16 *Let ξ_i and ξ_j be big values of x_i and x_j, respectively. Let the respective certificates G_i, G_j, and the nodes v_i, v_j be as in Remark 5.5.12. Suppose that $|G_i| \leq \ell$, $|G_j| \leq \ell$ for some ℓ. If the sets of vertices participating in G_i and G_j intersect, at least one of the values ξ_i and ξ_j is ℓ-located by the other.*

Proof: Suppose $u \in \underline{V}$ (similar for $u \in \underline{V}$) participates both in G_i and in G_j. Let p_i be the directed path in G_i from v_i to u and let p_j be a directed path in G_j from v_j to u. (see Figure 5.5 for an illustration). We claim that if $f_{p_i}(\xi_i) \leq f_{p_j}(\xi_j)$ then ξ_i ℓ-locates ξ_j; otherwise, ξ_j ℓ-locates ξ_i. Assume that $f_{p_i}(\xi_i) \leq f_{p_j}(\xi_j)$, and consider an application of "check" to ξ_i. The "check" algorithm determines that $f_{p_i}(\xi_i)$ is a strongly infeasible value of u. The algorithm initializes u^{-1} to the consistent value $f_{p_i}(\xi_i)$ and applies ℓ push phases. Consider the path p_j^{-1} from u^{-1} to v_j^{-1}. The value at v_j^{-1} after at most ℓ push phases must be tighter than $\xi_j^{-} \equiv f_{p_j^{-}}(f_{p_i}(\xi_i))$. Note that ξ_j^{-} is tighter than ξ_j at v_j^{-1}, since $\xi_j = f_{p_j^{-1}}(f_{p_j}(\xi_j))$ and $f_{p_j}(\xi_j)$ is tighter than $f_{p_i}(\xi_i)$ at u^{-1}. Hence, ξ_i ℓ-locates ξ_j. ∎

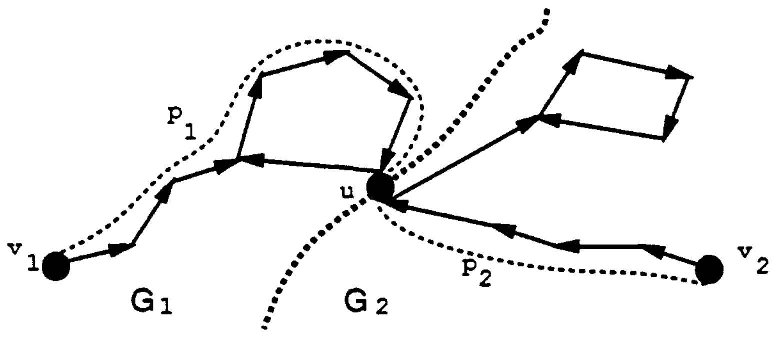


Figure 5.5: Dependence of intersecting certificates

Definition 5.5.17 Consider a pool of b $(2^{k-1}, 2^k]$-big values. The *influence* graph of this pool is defined as follows. Each big value in the pool corresponds to a vertex in the influence graph. If one value 2^k-locates another, then there is a directed edge from the vertex of the former to the vertex of the latter.

Theorem 5.5.15 is an immediate corollary of the following theorem:

Theorem 5.5.18 *In the influence graph of a pool of b $(2^{k-1}, 2^k]$-big values, at least half of the vertices have an out-degree greater than $\frac{1}{6}b2^{k-1}/n - 1$.*

We first prove two propositions which are used in the proof of the theorem.

Proposition 5.5.19 *In the intersection graph of b subsets of $\{1, \ldots, n\}$ with cardinality ℓ, the sum of the degrees is at least $b^2\ell/n - b$.*

Proof: Let s_i $(i = 1, \ldots, n)$ be the number of subsets containing i. Obviously, $\sum_{i=1}^{n} s_i = b\ell$. The sum of the degrees plus the number of vertices (subsets) is at least $\sum_{i=1}^{n} s_i^2/\ell$. This expression is minimized when $s_i \in \{\lfloor b\ell/n \rfloor, \lceil b\ell/n \rceil\}$ $(i = 1, \ldots, n)$. It follows that the sum of degrees is at most $\lceil b^2\ell/n \rceil - b$. ▮

Proposition 5.5.20 *In any orientation of the edges of the intersection graph of b subsets of $\{1, \ldots, n\}$ with cardinality ℓ, at least half the vertices have out-degrees greater than $\frac{1}{6}b\ell/n - 1$.*

Proof: It follows from Proposition 5.5.19 that the sum of out-degrees (i.e., the total number of edges) in any subgraph induced by b' vertices is at least $\frac{1}{2}(b'^2\ell/n - b')$. Suppose, to the contrary, that there exists a subset U of cardinality $|U| = b' = b/2$ where the out-degree of each vertex is less than or equal to $\frac{1}{6}b\ell/n - 1$. The total number of edges (which is the same as the sum of out-degrees) in the subgraph induced by these vertices is $\frac{1}{3}b'^2\ell/n - b'$, hence a contradiction since it is less that $\frac{1}{2}(b'^2\ell/n - b')$. ∎

Proof of Theorem 5.5.18: The cardinality of a certificate of a $(2^{k-1}, 2^k]$-big value is greater than or equal to $\ell = 2^{k-1}$. It follows from Proposition 5.5.16 that if two big values have intersecting certificates, then in the influence graph there is an edge between the corresponding vertices. Hence, the influence graph contains an intersection graph of b subsets of $\{1, \ldots, n\}$ of cardinality ℓ. Thus, the proof follows from Proposition 5.5.20. ∎

5.5.5 Estimating the number of big values

In this subsection we prove the following proposition:

Proposition 5.5.21 *An estimate r^* such that $1/2 \leq r^*/r \leq 2$ with probability at least $1/2$, can be found using $O((\log\log n)^2)$ "reveal" operations.*

The problem of computing an estimate can be stated in the following terms. We are given a collection of $\hat{n}$ stones, where r of them are radioactive. The number r is not known. We are equipped with a primitive gauge that when exposed to a set of stones can determine if at least one of them is radioactive (i.e., "reveal" operation). A *good estimate* to the number of radioactive stones is a number r^* such that $1/2 \leq r^*/r \leq 2$. The goal is to compute a good estimate which is correct with probability at least 1/2, by using at most $O((\log\log \hat{n})^2)$ gauge readings. We first give some useful propositions.

Proposition 5.5.22 *Denote by $p_0(s)$ the probability that a sample (with returns) of size s contains no radioactive stones and let $c \equiv rs/\hat{n}$. Under these conditions, (i) $p_0(s) < e^{-c}$, and (ii) if $r/\hat{n} \leq 0.5$, $p_0(s) > 4^{-c}$.*

Proof:

$$p_0(s) = \left(1 - \frac{r}{\hat{n}}\right)^s = \left(\left(1 - \frac{r}{\hat{n}}\right)^{\frac{\hat{n}}{r}}\right)^c$$

It follows that $p_0(s) \leq e^{-c}$ and if $r/\hat{n} \leq 0.5$, $p_0 \geq 4^{-c}$. ∎

Corollary 5.5.23 *Suppose $p_0(s)$ is known. We can conclude as follows:*

i. *If* $0.0625 \leq p_0(s) \leq 0.6$, *then* $1/2 \leq c \leq 2$.

ii. *If* $0.0725 \leq p_0(s) \leq 0.59$, *then* $0.53 \leq c \leq 1.89$.

iii. *If* $p_0(s) \leq 0.0825$, *then* $c \geq 1.8$.

iv. *If* $p_0(s) \geq 0.58$, *then* $c \leq 0.544$.

Proposition 5.5.24 *Let B' be the r.v. which corresponds to the number of non-radioactive samples out of $B = 5000\lceil\log\log\hat{n}\rceil$ random samples of size s.*

$$Prob\,\{|B'/B - p_0(s)| \geq 0.01\} \leq 1/(2\lceil\log\log\hat{n}\rceil)\ .$$

Proof: B' is the number of successes of B Bernoulli trials with success probability $p_0(s)$, and hence, has a binomial distribution. Therefore B' has expected value $p_0(s)B$ and variance $Bp_0(s)(1 - p_0(s))$ (see [23] for background). It follows from Chebyshev inequality that $Prob\,\{|B' - p_0(s)B| \geq 0.01B\} \leq 2500B/B^2 = 2500/B = 1/(2\lceil\log\log\hat{n}\rceil)$. A smaller constant can be achieved by using the normal approximation [23]. ∎

Denote by $\boldsymbol{p}'_0(s)$ the r.v. B'/B, and by $p'_0(s)$ a value of $\boldsymbol{p}'_0(s)$.

We now describe the algorithm that computes the estimate r^*. The algorithm is based on performing a binary search on the $\lfloor\log\hat{n}\rfloor$ possible estimate values 2^1, 2^2, $2^4, \ldots, 2^{\lfloor\log\hat{n}\rfloor}$.

Algorithm 5.5.25 [Compute an estimate]

i. If $p'_0(2) \leq 0.35$, stop and return the estimate $r^* = 2^{\lfloor \log \hat{n} \rfloor}$.

ii. Initialize $a = 1$, $b = \lfloor \log \hat{n} \rfloor$. Repeat the following until $a = b$:

iii. Set $j = \lceil (a+b)/2 \rceil$, and $s = 2^j$.

- If $0.0725 \leq p'_0(s) \leq 0.59$, stop and return the estimate $r^* = \hat{n}/s$.
- If $p'_0(s) < 0.0725$, set $b = j$ and go to step iii.
- If $p'_0(s) > 0.59$, set $a = j$ and go to step iii.

We show that with probability at least 0.5, $1/2 \leq r^*/r \leq 2$. Consider the first step. If $r \geq \hat{n}/2$, $p_0(2) \leq 1/4$. Hence, $Prob\{p'_0(2) \leq 0.35\} \leq 1/\log\log \hat{n}$. If $r \leq \hat{n}/4$, $p_0(2) \geq 9/16$. Hence, $Prob\{p'_0(2) \geq 0.47\} \leq 1/\log\log \hat{n}$. It follows that if $r \geq \hat{n}/2$, the algorithm stops with the correct estimate with probability at least $1 - 1/(2\lceil \log\log \hat{n} \rceil)$. If $r \leq \hat{n}/4$ the probability that the algorithm stops with an incorrect estimate is smaller than $1/(2\lceil \log\log \hat{n} \rceil)$.

The algorithm performs $\log\log \hat{n}$ iterations. Each iteration terminates with further restriction of the set $a, a+1, \ldots, b$. Consider a single iteration. It follows from Corollary 5.5.23 and Proposition 5.5.24 that if at the beginning of the iteration, one of the values $2^a, 2^{a+1}, \ldots, 2^b$ is a good estimate, then with probability $(1 - 1/(2\lceil \log\log \hat{n} \rceil))$ this is still the case when the iteration ends.

Suppose $\log\log \hat{n} \geq 2$. The probability that when the algorithm terminates 2^a is a good estimate is at least

$$(1 - 1/(2\lceil \log\log \hat{n} \rceil))^{\lceil \log\log \hat{n} \rceil} \geq 0.5 \ .$$

The algorithm performs $O(\log\log \hat{n})$ iterations, where each iteration computes an appropriate value of p'_0. Each iteration requires $O(\log\log \hat{n})$ "gauge readings" (see Proposition 5.5.24). Hence, the total number of "gauge readings" performed by the algorithm is $O((\log\log \hat{n})^2)$.

5.6 Locating a pool of weakly infeasible values and feasible values

In this section we present an algorithm which asserts the following:

Theorem 5.6.1 *Problem 5.3.6, where the values ξ_i $(i \in I)$ are guaranteed to be either feasible or weakly infeasible, can be solved (i) sequentially, in $O(mn\log^2 n)$ expected time, and (ii) on a CRCW PRAM, in $O(n\log^2 n)$ expected time, using $O(m)$ processors.*

The following is a key property of feasible and weakly infeasible values:

Proposition 5.6.2 *Let ξ_i and ξ_j be feasible or weakly infeasible values of x_i and x_j, respectively. If there exists a path p from $v_j \in \{\underline{v}_j, \overline{v}_j\}$ to $v_i \in \{\underline{v}_i, \overline{v}_i\}$ such that $f_p(\xi_j)$ is tighter than ξ_i at the vertex v_i, there exist such a simple path p'.*

Proof: We assume that $v_j = \overline{v}_j$, $v_i = \overline{v}_i$ (the proof for the other cases is similar). Consider the path p from v_j to v_i. Suppose a vertex u occurs more than once on p. Consider two of the occurrences of u. Let $p = p_1p_2p_3$, where p_1 is the prefix of p until the first occurrence and p_3 is the suffix of p starting from the second occurrence of u. If $f_{p_1p_2}(\xi_j)$ is not tighter than $f_{p_1}(\xi_j)$ at u, the path $p' = p_1p_3$ is at least as tight as p. Consider iterating the above process. It follows that there exist a path $\hat{p}$ from v_j to v_i which is at least as tight as p with respect to $x_j = \xi_j$, and when a vertex appears more than once, the value at subsequent occurrences is always tighter. We claim that each vertex may occur at most once in $\hat{p}$. Assume the contrary, and consider the first cycle c along $\hat{p}$. Let u be the vertex where c starts and ends, and let $p' = p_1c$ be the corresponding prefix of p. It follows from Proposition 5.2.16 that either $f_{p_1}(\xi_j)$ is infeasible at u, or $f_{p_1}(\xi_j)$ is consistent at u. If $f_{p_1}(\xi_j)$ is infeasible, p' is a closed certificate for $[-\infty, \xi_j]$ with respect to x_j and we get a contradiction to the assumption that ξ_j is not strongly infeasible. Otherwise, $f_{p_1}(\xi_j)$ is consistent at u. It follows that for every prefix p_2 of $\hat{p}$ which contains p_1 and ends at a vertex u', the value $f_{p_2}(\xi_j)$ is consistent at u'. Consider the last cycle

c' along $\hat{p}$, and let $p' = c'p_3$ be the corresponding suffix of $\hat{p}$. It follows that p' is a closed certificate for $[\xi_i, \infty]$ at v_i. This is a contradiction to ξ_i not being strongly infeasible. ∎

We present two procedures which are subroutines of the algorithm for locating a pool. The first procedure is applied to a subset of the values and returns a partial solution. The second procedure extends a partial solution as much as possible without restricting the feasible region further. We discuss the first procedure in detail. The input is a subset of values ξ_i $(i \in I')$, where $I' \subset I$. The procedure determines intervals J_i $(i \in I'')$, where $I'' \subset I'$ is a subset of the values in the input set. The intervals J_i $(i \in I'')$ constitute a partial solution for Problem 5.3.6, that is, there exist a set of intervals J_i $(i \in I \setminus I'')$ such that J_i $(i \in I)$ is a solution of Problem 5.3.6.

Algorithm 5.6.3 [Find a partial solution]

i. Initialize the values at the vertices of G as follows:
$\overline{x}_i = \underline{x}_i = \xi_i$ $(i \in I')$, $\overline{x}_i = b_i$, $\underline{x}_i = a_i$ $(i \notin I')$,

ii. Apply $2n$ push phases.

iii. For $i \in I'$ do as follows:

- If $\xi_i = \overline{x}_i = \underline{x}_i$, set $J_i = \{\xi_i\}$.
- If $\xi_i = \overline{x}_i < \underline{x}_i$, set $J_i = [\xi_i, \infty]$.
- If $\xi_i = \underline{x}_i > \overline{x}_i$, set $J_i = [-\infty, \xi_i]$.
- Otherwise, J_i is not determined.

Let $I'' \subset I'$ be the set of all $i \in I'$ for which J_i is determined. For $i \in I''$ do: $S_i \leftarrow S_i \cap J_i$.

Denote by F^* the feasible region $(1 \leq i \leq n)$ prior to applying Algorithm 5.6.3. Consider an application of the algorithm. We show that if $F^* \neq \emptyset$ then the system remains feasible, that is, $F \neq \emptyset$. The following proposition proves a necessary condition for correctness: If $F^* \neq \emptyset$, then for all $i \in I''$, $F^* \cap \{x | x_i \in J_i\} \neq \emptyset$.

Proposition 5.6.4 *Suppose $\underline{x}_i = \xi_i$ (resp., $\overline{x}_i = \xi_i$) is weakly infeasible. There exist a simple path p from $\underline{v}_i$ to $\overline{v}_i$ (resp., $\overline{v}_i$ to $\underline{v}_i$), such that $f_p(\xi_i) < \xi_i$ (resp., $f_p(\xi_i) > \xi_i$).*

Proof: It suffices to prove the existence of a not necessarily simple path p with the above properties. The existence of a simple path would follow from Proposition 5.6.2. We assume $\underline{x}_i = \xi_i$ is weakly infeasible (the other case is proved similarly). There exist $1 \le j \le n$ and two simple paths emanating from $\underline{v}_i$, p_1 to $\overline{v}_j$ and p_2 to $\underline{v}_j$ such that $f_{p_1}(\xi_i) < f_{p_2}(\xi_i)$. Consider the path $p = p_1 p_2^{-1}$ from $\underline{v}_i$ to $\overline{v}_i$. We claim that $f_p(\xi_i) < \xi_i$. To prove the claim note that $f_p(\xi_i) = f_{p_2^{-1}} \circ f_{p_1}(\xi_i)$. Since the end vertices of p_2^{-1} both lie in $\underline{V}$, the function $f_{p_2^{-1}}$ is monotone increasing. Hence, $f_{p_2^{-1}} \circ f_{p_1}(\xi_i) < f_{p_2^{-1}} \circ f_{p_2}(\xi_i) = \xi_i$. ∎

We show that the intervals J_i ($i \in I''$) constitute a partial solution:

Proposition 5.6.5 $F = F^* \cap \{x \mid \bigwedge_{i \in I''} x_i \in J_i\} \neq \emptyset$.

Proof: It follows from Proposition 5.6.4 that J_i is feasible for all $i \in I''$. It follows from Proposition 5.2.11 part iii that it suffices to show that for every pair $\{i, j\} \in I''$, $F^* \cap \{x | x_i \in J_i,\ x_j \in J_j\} \neq \emptyset$. The latter follows from Corollary 5.2.10. ∎

We discuss the procedure which extends a partial solution. Suppose we applied Algorithm 5.6.3 and as a result, for $i \in I''$, we updated the intervals S_i and determined J_i. The following algorithm may determine the interval J_i for some additional values $i \in I \setminus I''$.

Algorithm 5.6.6 [Extend a partial solution]

i. Initialize the values at the vertices of G as follows. $\underline{x}_i = a_i$ and $\overline{x}_i = b_i$ ($i \in I$).

ii. Apply $2n$ push phases.

iii. For $i \in I \setminus I'$ conclude as follows:

- If $\overline{x}_i \le \xi_i$, set $J_i = \{-\infty, \xi_i\}$.
- If $\underline{x}_i \ge \xi_i$, set $J_i = \{\xi_i, \infty\}$.

Let I'' be the set of $i \in I \setminus I'$ such that J_i was determined.
For $i \in I''$ do: $S_i \leftarrow S_i \cap J_i$.

Remark 5.6.7 Proposition 5.2.5 asserts that the feasible region is not changed by the updates of the intervals S_i done by Algorithm 5.6.6. In particular, if the feasible region was non empty prior to applying the algorithm it remains non empty.

5.6.1 The algorithm

The following algorithm considers feasible and weakly infeasible values ξ_i $(i \in I)$, and solves Problem 5.3.6 for these values. The solution consists of a set of intervals J_i $(i \in I)$. The algorithm performs iterations, where in each iteration some intervals J_i $(i \in I' \subset I)$ are determined. The intervals J_i $(i \in I')$ are incorporated into the system, we set $I \leftarrow I \setminus I'$, and the problem is reduced to locating the values ξ_i $(i \in I)$.

Algorithm 5.6.8 [Locate a pool]

i. Repeat the following until $I = \emptyset$.

ii. For $\ell = 1, 2, 4, \ldots, 2^{\lfloor \log n \rfloor}$, execute steps iii–v:

iii. Choose (with returns) ℓ random elements from I. Let $I' \subset I$ be the set of chosen elements.

iv. Apply Algorithm 5.6.3 to I'. The algorithm determines J_i for $i \in I'' \subset I'$.

v. Apply Algorithm 5.6.6 and determine J_i for additional values $\hat{I} \in I$. Set $I \leftarrow I \setminus \{I'' \cup \hat{I}\}$.

It follows from Proposition 5.6.5 and Remark 5.6.7 that if the problem was feasible, it remains feasible in any point during the execution of the algorithm. hence, when the algorithm terminates, the intervals J_i $(i \in I)$ constitute a solution for Problem 5.3.6.

5.6.2 Complexity

For values ξ_i $(i \in I)$, consider a solution for Problem 5.3.6, that is, intervals J_i $(i \in I)$ such that $J_i \in \{[-\infty, \xi_i], [\xi_i, \infty]\}$ and $\times_{i \in I} J_i \cap F_I \neq \emptyset$.
We designate vertices v_i $(i \in I)$ according to the choice of intervals J_i $(i \in I)$: If $J_i = [-\infty, \xi_i]$, $v_i = \overline{v}_i$. If $J_i = [\xi_i, \infty]$, $v_i = \underline{v}_i$.

We define a relation on the set I, with respect to the solution J_i $(i \in I)$:

Definition 5.6.9 Let $i \in I$, $j \in I$ be two elements.

i. We say that j *locates* i if there exist a path p from v_j to v_i such that $f_p(\xi_j)$ is at least as tight as ξ_i on v_i.

ii. We say that i *interferes* with j if there exists a path p from v_i^{-1} to v_j^{-1} such that $f_p(\xi_i)$ is tighter than ξ_j on v_j^{-1}.

iii. We define the relation $\prec$ on I as follows. $j \prec i$ if and only if i interferes with j. (Note that when $i \neq j$, i locates j if and only if j interferes with i.)

iv. For an element $i \in I$, denote by $\text{pred}(i) = \{j \in I | j \preceq i\}$ the set of elements that locate i, and by $\text{succ}(i) = \{j \in I | j \succ i\}$ the set of elements which interfere with i.

The following is a corollary of Proposition 5.6.2:

Corollary 5.6.10 *If a path p as in Definition 5.6.9 parts i and ii exists, then there must exist a simple path with the same properties.*

The following proposition relates the locate and interfere relations to the algorithms presented earlier. The proof follows from Corollary 5.6.10.

Proposition 5.6.11 *Let $i \in I$, $j \in J$ be two elements.*

i. Suppose that j locates i. If $S_j \subset J_j$, then an application of Algorithm 5.6.6 would determine J_i.

ii. Suppose that i interferes with j. Consider an application of Algorithm 5.6.3 where $\{i, j\} \subset I'$. The interval J_j will not be determined as a result of such a run. Moreover, if $j \in I'$, the interval J_j is determined in this run if and only if for all values $i \in I'$, i does not interfere with j.

We characterize the values located in an iteration of Algorithm 5.6.8, in terms of pred and succ:

Corollary 5.6.12 *Consider an execution of steps iv–v. An interval J_k (for $k \in I$) is determined as a result if and only if* $\text{pred}(k) \cap I' \neq \emptyset$ *and* $\text{succ}(k) \cap I' = \emptyset$.

Proposition 5.6.13 *The relation $\prec$ is a partial order on I.*

Proof: The transitivity of $\prec$ is immediate. We need to show that $\prec$ has no cycles. If there is a cycle, it follows from the transitivity that for some $j \in I$, j interferes with j. The path p is in this case a simple cycle which constitutes a closed certificate for ξ_j. This contradicts the assumption that ξ_j is not strongly infeasible. ∎

Note that each iteration of steps iii–v results in determining at least one interval, and therefore, the algorithm terminates within $O(n)$ iterations. In order to prove Theorem 5.6.1, however, we need to show that the expected number of iterations is only $O(\log^2 n)$. We first discuss the chances of an interval J_i ($i \in I$) to be determined in an iteration:

Proposition 5.6.14 ***Consider an execution of steps iii–v. If an element $i \in I$ is such that $|\text{succ}(i)| = 0$, then J_i is determined with probability at least ρ_0, where $\rho_0 > 0$ is some constant, at the steps where $\ell \geq |I|/|\text{pred}(i)|$. Otherwise, if $|\text{pred}(i)| > |\text{succ}(i)|/3$, then J_i is determined with probability at least ρ_0 at the step where***

$$\frac{|I|}{|\text{succ}(i)|} \leq \ell \leq \frac{2|I|}{|\text{succ}(i)|} .$$

Proof: We prove the part where $|\text{succ}(i)| > 0$. The first part is similar and simpler. It follows from Corollary 5.6.12 that J_i is determined with probability

$$P = \mathit{Prob}\{\text{pred}(i) \cap I' \neq \emptyset \ \wedge \ \text{succ}(i) \cap I' = \emptyset\} \ .$$

We need to show that there exist a constant ρ_0 such that $P > \rho_0$. Since the sets $\text{succ}(i)$ and $\text{pred}(i)$ are disjoint, it suffices to show that

$$P_1 = \mathit{Prob}\{\text{pred}(i) \cap I' \neq \emptyset\} > \rho_1, \text{ and } P_2 = \mathit{Prob}\{\text{succ}(i) \cap I' = \emptyset\} > \rho_2,$$

for some constants $\rho_1 > 0$, $\rho_2 > 0$.

$$P_1 = 1 - \left(1 - \frac{\text{pred}(i)}{|I|}\right)^{\ell}, \quad P_2 = \left(1 - \frac{\text{succ}(i)}{|I|}\right)^{\ell} \ .$$

Consider ℓ as above. It follows that $P_2 \geq 0.06$ and $P_1 \geq 0.28$. ∎

Proof of Theorem 5.6.1: Algorithm 5.6.8 terminates with a correct solution. We claim that the expected number of iterations of step ii is $O(\log n)$. Each execution of step ii amounts to $O(\log n)$ calls to Algorithms 5.6.3 and 5.6.6, and hence, Algorithm 5.6.8 amounts to an expected number of $O(\log^2 n)$ applications of Algorithms 5.6.3 and 5.6.6. Recall that Algorithms 5.6.3 and 5.6.6 run sequentially, in $O(mn)$ time, and on a CRCW PRAM, in $O(n)$ time using $O(m)$ processors. It follows that the expected running time of Algorithm 5.6.8 is as stated in the theorem. What remains is to prove the claim. Consider the quantity

$$s(I) = \sum_{i \in I} |\text{pred}(i)| = |I| + \sum_{i \in I} |\text{succ}(i)| \ .$$

Initially, $s \leq n^2$. It suffices to show that an execution of step ii reduces s by at least $s/7$, with probability at least 1/2. If $s \leq 7|I|/4$,

$$\sum_{\{i:|\text{succ}(i)|=0\}} |\text{pred}(i)| > s/7 \ .$$

Otherwise, if $s > 7|I|/4$, we have $\sum_{i \in I} |\text{succ}(i)| \geq 3 \sum_{i \in I} |\text{pred}(i)|/7$, and hence,

$$\sum_{\{i:|\text{pred}(i)|>|\text{succ}(i)|/3\}} |\text{pred}(i)| > 2s/7 \ .$$

The proof follows from Proposition 5.6.14. ∎

5.7 Monotone systems

Consider the following two operations on vectors in R^n:

$$\begin{aligned} \boldsymbol{x} \vee \boldsymbol{y} &= (\max\{x_1, y_1\}, \ldots, \max\{x_n, y_n\})^T \quad \text{(the } \textit{join} \text{ of } \boldsymbol{x} \text{ and } \boldsymbol{y}\text{)} \\ \boldsymbol{x} \wedge \boldsymbol{y} &= (\min\{x_1, y_1\}, \ldots, \min\{x_n, y_n\})^T \quad \text{(the } \textit{meet} \text{ of } \boldsymbol{x} \text{ and } \boldsymbol{y}\text{)} . \end{aligned}$$

Consider a TVPI system as in Definition 5.1.1, where the matrix $\boldsymbol{A}^T$ (resp., $-\boldsymbol{A}^T$) is a *pre-Leontief* matrix, that is, each row of $\boldsymbol{A}$ contains at most one positive (resp., negative) entry. The set of solutions of such a system constitutes a semilattice relative to the $\vee$ (resp., $\wedge$) operation (see Cottle and Veinott [15]). It follows that if all the variables are bounded from above (resp., below), there exists a solution where all the variables are maximized (resp., minimized) simultaneously, hence, the vector $\boldsymbol{x}^{\max}$ (resp., $\boldsymbol{x}^{\min}$) is feasible. Also, the associated graph of such a system (see Definition 5.2.1) does not contain edges from $\underline{V}$ to $\overline{V}$ (resp., from $\overline{V}$ to $\underline{V}$).

A TVPI system where the two non-zero coefficients in each inequality have opposite signs (equivalently, both $\boldsymbol{A}^T$ and $-\boldsymbol{A}^T$ are pre-Leontief) is called *monotone*. The set of solutions of monotone systems constitutes a lattice relative to the $\vee$ and $\wedge$ operations. Note that the converse is also true: if a polyhedral set constitutes a lattice relative to the $\vee$ and $\wedge$ operations, then it can be represented as the set of solutions of some monotone system. The converse is a direct consequence of the following result due to Veinott [61]: a polyhedral set is a semilattice relative to the $\vee$ (resp., $\wedge$) operation if and only if it constitutes the set of solutions of some linear system $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ (resp., $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$), where the matrix $\boldsymbol{A}^T$ (resp., $-\boldsymbol{A}^T$) is pre-Leontief. Consider, for example, the non-monotone system in Figure 5.6. The points $(5,1)$ and $(3,4)$ are feasible, but the point $(5,1) \wedge (3,4) = (3,1)$ is not. Monotone systems are related to generalized network flow problems: the linear programming dual of the uncapacitated generalized transshipment problem is monotone. In Chapter 6 we present algorithms for generalized network flow problem that exploit this relationship.

The algorithms presented here for solving the feasibility of TVPI systems can be adapted to find the vectors $\boldsymbol{x}^{\min}$, $\boldsymbol{x}^{\max}$ for monotone systems. Note that in the associated graph of a monotone system there are no edges passing between the two

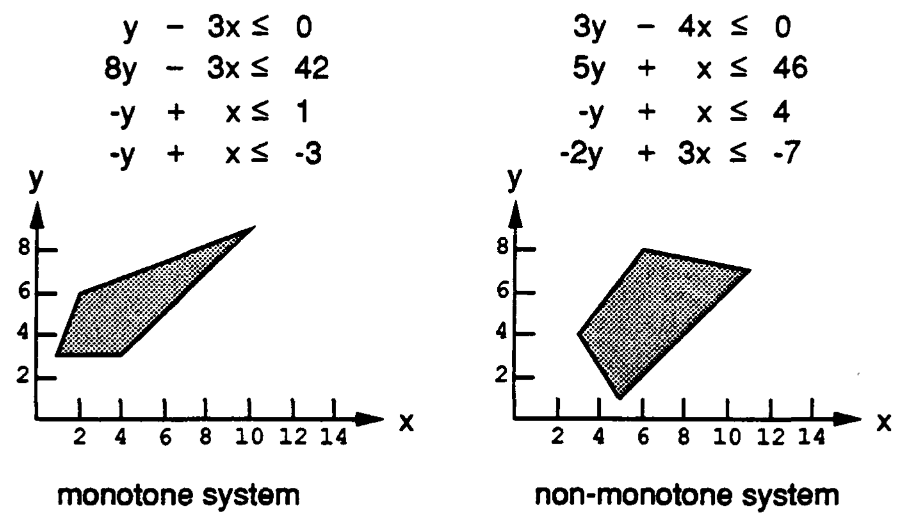


Figure 5.6: Examples of a monotone system and a non-monotone system

sets of vertices $\overline{V}$ and $\underline{V}$. It follows that all certificates are either closed certificates or simple paths, and all infeasible values must be strongly infeasible.

We consider an application of Algorithm 5.3.3 to a monotone system, with the goal of finding the point $x^{\max}$. It suffices to rely on a simpler notion of locating values: *locating* the value ξ means locating it with respect to $x_i^{\max}$ (rather than determining its position relative to $x_i^{\min}$ as well). Problem 5.3.6 amounts to the following:

Problem 5.7.1 [Locate a pool of values] Given are values ξ_i $(i \in I)$ for the corresponding variable x_i $(i \in I)$. Locate these values, i.e., determine if $\xi_i > x_i^{\max}$ $(i \in I)$. Choose intervals J_i $(i \in I)$ as follows. $J_i = [\xi_i, \infty]$, if $\xi_i \le x_i^{\max}$ and $J_i = [-\infty, \xi_i]$, if $\xi_i \ge x_i^{\max}$ $(i \in I)$.

It is easy to see that the intervals J_i are such that $\{x \mid \bigwedge_{i \in I} x_i \in J_i\}$ contains the simultaneous maximum.

Consider step vii of Algorithm 5.3.3. The interval S_i' are feasible for x_i $(1 \le i \le n)$. Hence, for $1 \le i \le n$, $b_i' = x_i^{\max}$ (x_i is unbounded if and only if $b_i' = \infty$). If all variables are bounded, $b' = x^{\max}$. Otherwise, the algorithm supplies dependencies that allow us to compute feasible solutions when we determine some variables and maximize others.

Remark 5.7.2 An algorithm which computes $\boldsymbol{x}^{\max}$ (resp., $\boldsymbol{x}^{\min}$) for feasible monotone systems can be applied to compute $\boldsymbol{x}^{\max}$ (resp., $\boldsymbol{x}^{\min}$) for TVPI systems where the matrix $\boldsymbol{A}^T$ (resp., $-\boldsymbol{A}^T$) is pre-Leontief. Consider a system $\boldsymbol{Ax} \leq \boldsymbol{b}$ where at most one of the two nonzero coefficients in each row is positive. Obtain a monotone system $\boldsymbol{A}'\boldsymbol{x} \leq \boldsymbol{b}'$ by omitting all rows of $\boldsymbol{A}$ where there are two negative entries. Compute the vector $\boldsymbol{y}$ which maximizes all variables in the resulting monotone system. If the original system $\boldsymbol{Ax} \leq \boldsymbol{b}$ is feasible, $\boldsymbol{y}$ is the vector which maximizes all variables.

5.8 Concluding remarks

In this chapter we considered systems of linear inequalities, where each inequality has at most two nonzero coefficients (TVPI systems). We presented algorithms which solve the feasibility problem, that is, either find a point which satisfies all the inequalities or conclude that no such point exists. We gave a $\tilde{O}(mn^2)$ deterministic algorithm and a $\tilde{O}(n^3 + mn)$ expected time randomized algorithm. The complexities of the respective parallel implementations are: $\tilde{O}(n)$ time using $O(mn)$ processors, and $\tilde{O}(n)$ expected time using $O(n^2 + m)$ processors. Although the analysis of these algorithms seems quiet lengthy, the algorithms themselves are simple. The underlying computation amounts the basic Bellman-Ford and Floyd-Warshall [13] shortest path algorithms where only data structures are used.

We give some comments and suggestions for further research:

The time complexity of the randomized algorithm involves many logarithmic factors ($\log^5 n$). We believe that a more careful analysis may eliminate some of these factors. Another obvious question is whether the $\tilde{O}(n^3 + mn)$ bound can be achieved deterministically.

We discuss the possibility of improving over a $\tilde{O}(n^3 + mn)$ bound. The $\tilde{O}(n^3)$ factor results from the all-pairs shortest part Floyd-Warshall computation and is inherent from our basic framework. A different approach, however, may yield a $\tilde{O}(mn)$ algorithm.
The feasibility problem of monotone systems includes as a very special case the problem of detecting existence of negative-weight directed cycles in a graph with n nodes, m edges, and real weights associated with the edges. The best known bound for detecting negative weight cycles is $O(mn)$, and hence, we believe it is improbable that a $\tilde{o}(mn)$ algorithm exists for solving TVPI systems. We sketch the reduction. Consider a weighted graph $G = (V, E, w)$, where $w : E \rightarrow R$. The corresponding monotone system is as follows. For each node $v \in V$ assign a variable x_v. For each edge $e = (u, v) \in E$ assign the inequality $x_v - x_u \leq w(e)$. It follows from Proposition 5.2.9 that G contains a negative weight cycle if and only if the system is infeasible.

Our algorithms solve the feasibility problem of TVPI systems. They can be adapted, however, to (i) find a solution which maximizes a specific variable (ii) and find the lexicographic maximum. Questions that remain open regard finding an optimal solution relative to an arbitrary linear objective function. It is not known whether a strongly polynomial time algorithm exists for the problem, or whether we can achieve a time bound which is better than specializations of existing general LP algorithms. A partial result is due to Cosares [14] who showed that when the objective function has a fixed number of nonzero entries, the problem can be solved in strongly polynomial time bounds. The degree of the polynomial, however, grows linearly with the number of nonzero entries.

We discuss the parallel complexity of solving TVPI systems. Lueker, Megiddo, and Ramachandran [43] showed that the problem of finding an optimal solution relative to a general objective function is $\mathcal{P}$-complete. It is not known, however, whether the feasibility problem is $\mathcal{P}$-complete. The algorithms presented in this chapter have a parallel running time of $\tilde{O}(n)$. These are the best known parallel time bound achievable by a polynomial number of processors. Lueker, Megiddo, and Ramachandran [43] gave an algorithm for the problem which runs in polylogarithmic time, but uses $n^{O(\log n)}$ processors.

Chapter 6

Algorithms for generalized network flows

A *generalized network* is a digraph $G = (V, E)$ given together with positive *flow multipliers* a_e $(e \in E)$ associated with the edges. The multiplier a_e $(e \in E)$ is interpreted as a gain factor (when $a_e > 1$) or a loss factor (when $a_e < 1$) of flow along the edge e; when x_e units of flow "enter" the edge e, $a_e x_e$ units "leave". Generalized network flows are also known in the literature as *flows with losses and gains.* They can be used to model many situations that arise in financial analysis [25, 26, 42].

The uncapacitated generalized transshipment problem (UGT) is defined on a generalized network, where costs are given for the edges and supplies or demands are given for the nodes. The goal is to find a flow of minimum cost, which satisfies the supplies/demands. Adler and Cosares [1] gave an algorithm for solving restricted instances of UGT where there are many sources and no sinks. Their algorithm is based on a solution for the linear programming dual, which turns out to be a monotone TVPI system, and hence, the results of Chapter 5 imply better time bounds for restricted UGT.

In the generalized circulation problem (GC) we consider a generalized network where demands (nonnegative numbers) are given for the nodes and capacity constraints are given for the edges. The goal is to find a feasible flow which maximizes the proportion of satisfied demands. Goldberg, Plotkin, and Tardos [26] presented an algorithm for the seemingly more general capacitated generalized transshipment problem without costs. Their algorithm is based on solving an instance of GC which has a single demand node (the *source*) and performs $O(mn)$ additional computation. We present a scheme for solving generalized circulation problems by iteratively relaxing the capacity constraints, solving an instance of UGT on the same network with costs which "capture" the capacities, scaling the flow to a feasible one, and replacing the capacities by the residual capacities relative to this flow.

This scheme introduces a general method of approximating a solution to linear programming problems in the following situation. For given matrices $\boldsymbol{A} \in R^{n\times m}$, $\boldsymbol{U} \in R^{\ell\times m}$ and vectors $\boldsymbol{b} \in R^n$, $\boldsymbol{d} \in R^\ell$, where $\boldsymbol{U}$, $\boldsymbol{d}$ have nonnegative entries, maximize t subject to: (i) $\boldsymbol{Ax} = t\boldsymbol{b}$, $\boldsymbol{x} \geq \boldsymbol{0}$, and (ii) $\boldsymbol{Ux} \leq \boldsymbol{d}$. Condition (ii) can be viewed as generalized capacity constraints. We assume that for $\mathbf{c} \geq \mathbf{0}$, it is "easy" to minimize $\boldsymbol{c}^T\boldsymbol{x}$ subject to $\boldsymbol{Ax} = \boldsymbol{b}$, $\boldsymbol{x} \geq \boldsymbol{0}$. Denote by t^* the maximal value of the objective function. We will show that a feasible solution $\boldsymbol{x}' \in R^n$, $t' \in R$ such that $t' \geq t^*/\ell$ can be found by solving a single instance of the "easy" problem.

Consider the generalized circulation problem with the relaxed goal of computing a flow which satisfies a proportion of the demands which approximates to a constant factor the best achievable proportion. For the relaxed problem, the scheme described above yields a strongly polynomial time algorithm, which is also the fastest known (on some ranges) algorithm. This scheme also yields an algorithm for obtaining an optimal solution, which is the fastest known on some ranges.

In Section 6.1 we define the UGT problem and review the Adler and Cosares [1] algorithm. In Section 6.2 we introduce the approximation algorithm and apply it to the generalized circulation problem. In Section 6.3 we introduce bidirected generalized networks and discuss the UGT and generalized circulation problems on these networks. Section 6.4 contains concluding remarks.

Note that for instances of the problems mentioned above we need to consider cases where $m = \omega(n^2)$. The algorithms presented here handle multiple edges within the stated time bounds.

6.1 Generalized transshipment problem

Problem 6.1.1 [Uncapacitated Generalized Transshipment (UGT)]
Given are a generalized network $G = (V, E)$, edge-costs c_e ($e \in E$), and supplies (or demands) b_i ($1 \le i \le n$) for the nodes. Find a nonnegative flow function $\boldsymbol{x} = (x_e)$ such that

$$\text{for every } i, \sum_{e \in \text{in}(i)} a_e x_e - \sum_{e \in \text{out}(i)} x_e = b_i \ ,$$

so that the cost $\sum_{e \in E} c_e x_e$ is minimized.

When $b_i > 0$ (resp., $b_i < 0$), we refer to the i'th node as a *source* (resp., *sink*). The linear programming dual has the following form. Find values for $\pi_1, \ldots, \pi_n$ which maximize $\sum_{i=1}^n b_i \pi_i$, subject to the monotone inequalities

$$\text{for all } e \in E\text{:} \quad \pi_i - a_e \pi_j \le c_e \quad \text{where } e \text{ is from } i \text{ to } j \ .$$

In this section we consider restricted instances of UGT where there are either only sources ($\boldsymbol{b} \ge \mathbf{0}$) or only sinks ($\boldsymbol{b} \le \mathbf{0}$). Adler and Cosares [1] proposed a scheme for solving a subset of the LP problems where each variable appears in at most two inequalities. In particular, the scheme is applicable to restricted UGT instances. They showed that these instances can be solved using a single application of Megiddo's algorithm for TVPI systems [46]. An application of the faster algorithms for TVPI systems presented in Chapter 5 can be used instead. Hence, restricted UGT instances can be solved deterministically in $O\left(mn^2(\log m + \log^2 n)\right)$ time, and in $O\left(n^3 \log n + mn(\log m \log^3 n + \log^5 n)\right)$ expected time.

We characterize the problems for which the scheme of [1] is applicable. Consider an LP problem, $\mathcal{P}$, of the following form. minimize $\boldsymbol{c}^T\boldsymbol{x}$, subject to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, $\boldsymbol{x} \ge 0$, where $\boldsymbol{A} \in R^{n \times m}$ contains at most two non zero entries in each column. Denote by $\boldsymbol{x}^* \in R^m$ the optimal solution of $\mathcal{P}$. Note that the LP dual of $\mathcal{P}$ amounts to optimizing an arbitrary objective function subject to a TVPI system. The scheme of [1] is applicable to $\mathcal{P}$ if $\boldsymbol{x}^* = \sum_{i:b_i \ne 0} \boldsymbol{x}^{(i)}$, where $\boldsymbol{x}^{(i)} \ge \mathbf{0}$ maximizes $\boldsymbol{c}^T\boldsymbol{x}$ subject to $\boldsymbol{A}\boldsymbol{x} = b_i \boldsymbol{e}^i$.

We sketch the ideas used in their scheme. Denote by S the TVPI constraints $A^T\pi \leq c$. Let $\pi_i^{\min}$ (resp., $\pi_i^{\max}$) be the minimum (resp., maximum) value of π_i subject to S.
If $\pi_i^{\max} = \infty$ (resp., $\pi_i^{\min} = -\infty$), then $\mathcal{P}$ is feasible only if $b_i \leq 0$ (resp., $b_i \geq 0$).
If $b_i \neq 0$, a vector $x^{(i)}$ as defined above can be constructed from a minimal subset of constraints from S which implies (i) $\pi_i \leq \pi_i^{\max}$ if $b_i > 0$, or (ii) $\pi_i \geq \pi_i^{\min}$ if $b_i < 0$. The edges which correspond to such a minimal system comprise a generalized augmenting path [29] of flow to the i'th node (i.e., a flow generating cycle and a path from a node on the cycle to the i'th node). The vector $x^{(i)}$ is obtained by considering the flow values at the edges when pushing flow along this augmenting path.

It is easy to see that the scheme of [1] is applicable to restricted UGT instances. Consider a UGT instances where $b \geq 0$ (the arguments are similar for $b \leq 0$). The constraints in S are monotone, and therefore, by a single application of the algorithm of Chapter 5 we determine $\pi_i^{\max}$ for $i = 1, \ldots, n$ (see Section 5.7). If $\pi_i^{\max} < \infty$, the algorithm also computes a minimal subset of constraints from S which asserts that $\pi_i \leq \pi_i^{\max}$. The vector $x^{(i)}$ can be constructed by using this information. Conclude as follows. If $\pi_i^{\max} = \infty$ and $b_i > 0$, the UGT system is not feasible. Otherwise, $x^* = \sum_{i:b_i>0} x^{(i)}$ is a solution.

6.2 Generalized circulation

Definition 6.2.1 Consider a generalized network $G = (V, E)$, where demands $b_i \geq 0$ ($1 \leq i \leq n$) are given for the nodes and capacities $c_{ij} \geq 0$ (possibly $c_{ij} = \infty$) are given for the the edges.

i. A *generalized flow* is a flow function $x = (x_e)$ ($x_e \geq 0$) which satisfies the following. There exist a scalar $\hat{t}(x) \equiv \hat{t}$ such that for every $1 \leq i \leq n$, $\sum_{e\in \text{in}(i)} a_e x_e - \sum_{e \in \text{out}(i)} x_e = \hat{t} b_i$ (the flow x satisfies a proportion $\hat{t}(x)$ of the demands).

ii. A generalized flow is *feasible* if $x_e \leq c_e$ for all edges.

Vaidya, 89 [60]	$O(n^2 m^{1.5} \log(n\gamma))$
Kapoor and Vaidya, 88 [36]	$O(n^{2.5} m^{1.5} \log(n\gamma))$
Goldberg, Plotkin and Tardos, 88 [26]	$O(n^2 m^2 \log n \log^2 \gamma)$

Table 6.1: Some previous results on generalized circulation

Problem 6.2.2 [Generalized Circulation]
Given are a generalized network, demands, and capacities as above. Find a feasible generalized flow $\boldsymbol{x}^*$ such that $\hat{t}(\boldsymbol{x}^*)$ is maximized. Denote $t^* \equiv \hat{t}(\boldsymbol{x}^*)$.

We refer to t^* as the *optimal value*. A feasible generalized flow $\boldsymbol{x}$ is:
i. *optimal* if $\hat{t}(\boldsymbol{x}) = t^*$, and ii. ϵ*-optimal* if $\hat{t}(\boldsymbol{x})/t^* \geq 1 - \epsilon$.

Vaidya [60] gave an $O(n^2 m^{1.5} \log(n\gamma))$ time algorithm for the problem, were γ is an upper bound on the numerators and denominators of the capacities, multipliers, and costs. Vaidya's bound is based on a specialization of his currently fastest known general-purpose linear programming algorithm and relies on the highly impractical fast matrix multiplication algorithms. The previously fastest known algorithm, due to Kapoor and Vaidya [36], does not rely on fast matrix multiplication, and has a bound worse by a factor of $\sqrt{n}$. A new result by Murray [48] is based on a different specialization of Vaidya's LP algorithm to generalized flow. Murray's generalized circulation algorithm matches the bound of [60] and does not rely on fast matrix multiplication. The algorithms of [36, 48, 60] are applicable to the more general min-cost generalized flow problem. A different algorithm, of more combinatorial nature, was given by Goldberg, Plotkin and Tardos [26]. These results are summarized in Table 6.1.

6.2.1 A generalized circulation algorithm

The algorithms discussed above are designed to find an optimal flow. We introduce an algorithm for Problem 6.2.2 which is based on iteratively obtaining a $(1 - 1/m)$-optimal flow and then considering the problem on the residual network. Hence, an

Computing an ϵ-optimal flow:	
expected time	$O\left(m \log \epsilon^{-1} \left(n^3 \log n + mn(\log m \log^3 n - \mathrm{lo}$[illegible]$)\right)\right)$
deterministic	$O\left(m^2 n^2 \log \epsilon^{-1}(\log m + \log^2 n)\right)$
Computing the optimal solution:	
expected time	$\tilde{O}\left(\lvert t^* \rvert (mn^3 + m^2 n)\right)$
deterministic	$\tilde{O}(\lvert t^* \rvert m^2 n^2)$

Table 6.2: Bounds for generalized circulation

ϵ-optimal flow can be computed using $O(m \log \epsilon^{-1})$ iterations.
Note that when ϵ is a constant, $O(m)$ iterations suffice. The optimal value can be found within $O(m|t^*|)$ iterations, where $|t^*|$ is the number of bits of accuracy required. Note that $|t^*| \leq m \log(n\gamma)$, where γ is the largest enumerator/denominator of a capacity or a multiplier in the network.

We discuss the complexity of each iteration. In Subsection 6.2.2 we introduce an approximation method that allows us to find a $(1 - 1/m)$-optimal feasible generalized flow by solving a single UGT instance on the same generalized network, where the capacity constraints are relaxed and costs are introduced. It follows, that each iteration of our algorithm amounts to solving an instance of the restricted UGT problem.

The resulting deterministic and randomized bounds for computing an ϵ-optimal generalized circulation are summarized in Table 6.2. Our algorithm is more practical than [60]. When the algorithm is used to find an approximate solution, we achieve strongly polynomial time bounds which are also strictly better than [26, 36], and better than [60] on some ranges (e.g., when the size of the binary encoding of capacities and multipliers is large). The algorithm also yields improved bounds for some ranges (e.g., when we know the number of bits in the binary encoding of t^* is small) for obtaining an optimal solution.

Note that when $\epsilon = 1/q(m, n)$, where q is a polynomial, an ϵ-optimal flow can be found in strongly polynomial time bounds. It is still not known whether a strongly polynomial time algorithm exists for finding an optimal solution. This question is of a particular interest because generalized circulation is one of the simplest classes of linear programming problems for which no strongly polynomial algorithms are known [26, 59].

6.2.2 Obtaining an approximation

Consider linear programming problems of the following form. Given are matrices $\boldsymbol{A} \in R^{n\times m}$, $\boldsymbol{U} \in R^{\ell\times m}$, and vectors $\boldsymbol{b} \in R^n$, $\boldsymbol{d} \in R^\ell$, where $\boldsymbol{U} \geq \boldsymbol{0}$, $\boldsymbol{d} > \boldsymbol{0}$. The goal is to maximize $\hat{t}$, subject to $\boldsymbol{Ax} = \hat{t}\boldsymbol{b}$, $\boldsymbol{x} \geq 0$, and $\boldsymbol{Ux} \leq \boldsymbol{d}$. We refer to the constraints $\boldsymbol{Ux} \leq \boldsymbol{d}$ as *generalized capacity constraints*. A vector $\boldsymbol{x} \geq 0$, such that $\boldsymbol{Ax} \propto \boldsymbol{b}$ ($\boldsymbol{Ax}$ is proportional to $\boldsymbol{b}$) and $\boldsymbol{Ux} \leq \boldsymbol{d}$ is called *feasible*. For a feasible vector $\boldsymbol{x}$, denote by $\hat{t}(\boldsymbol{x})$ the scalar $\hat{t}$ such that $\boldsymbol{Ax} = \hat{t}\boldsymbol{b}$. Denote by t^* the optimal solution, and by $\boldsymbol{x}^*$ some vector where $\hat{t}(\boldsymbol{x}^*) = t^*$. A feasible vector $\boldsymbol{x}$ is *ϵ-optimal* if $\hat{t}(\boldsymbol{x})/t^* \geq 1 - \epsilon$.

Suppose that for $\boldsymbol{0} \leq \boldsymbol{c} \in R^m$ it is "easy" to compute a vector $\boldsymbol{x} \geq \boldsymbol{0}$ which minimizes $\boldsymbol{c}^T\boldsymbol{x}$, subject to $\boldsymbol{Ax} = \boldsymbol{b}$. We refer to problems of this form as *uncapacitated* instances. An instance of the original problem is referred to as a *capacitated* instance. Note that when the capacitated problem is an instance of generalized circulation, $\boldsymbol{U}$ is a diagonal matrix with at most $m = |E|$ rows. The corresponding uncapacitated problem is an instance of UGT on the same network, where only demand nodes are present.

We presents an algorithm for constructing a $(1/\ell)$-optimal vector $\boldsymbol{y}$. The algorithm amounts to solving a single instance of the uncapacitated problem. Consider the cost function

$$p(\boldsymbol{x}) = \sum_{i=1}^{\ell} \boldsymbol{U}_{i\bullet}\boldsymbol{x}/d_i \ .$$

It is easy to verify that this function is linear and has the following properties:

i. If $\boldsymbol{x}$ is feasible then $p(\boldsymbol{x}) \leq \ell$.

ii. If $\boldsymbol{x} \geq \boldsymbol{0}$, $\boldsymbol{Ax} \propto \boldsymbol{b}$, and $p(\boldsymbol{x}) \leq 1$, then $\boldsymbol{x}$ is feasible.

iii. If $p(\boldsymbol{x}^*) > 0$, then $p(\boldsymbol{x}^*) \geq 1$ (since for some i, $\boldsymbol{U}_{i\bullet}\boldsymbol{x}^* = d_i$).

Consider a vector $\boldsymbol{y} \geq \mathbf{0}$, $\boldsymbol{Ay} \propto \boldsymbol{b}$, $p(\boldsymbol{y}) = 1$ which maximizes $\hat{t}(\boldsymbol{y})$. Note that the vector $\ell\boldsymbol{y}$ maximizes $\hat{t}(\boldsymbol{x})$ for all vectors $\boldsymbol{x} \geq \mathbf{0}$ such that $p(\boldsymbol{x}) \leq \ell$ and $\boldsymbol{Ax} \propto \boldsymbol{b}$. In particular $\hat{t}(\ell\boldsymbol{y}) \geq t^*$, and hence, $\hat{t}(\boldsymbol{y}) \geq t^*/\ell$. Such a vector $\boldsymbol{y}$ can be obtained by normalizing a vector $\boldsymbol{x} \geq 0$ which minimizes $p(\boldsymbol{x})$ subject to $\boldsymbol{Ax} = \boldsymbol{b}$. Also, $\boldsymbol{y}$ is feasible and therefore provides the desired approximation. A formal description of the algorithm follows.

Algorithm 6.2.3 [Compute a $(1/\ell)$-optimal vector]

i. Solve the following instance of the uncapacitated problem: Minimize $p(\boldsymbol{x})$ subject to $\boldsymbol{x} \geq \mathbf{0}$ and $\boldsymbol{Ax} = \boldsymbol{b}$.
   If it is infeasible, then stop and claim that $\boldsymbol{x} = \mathbf{0}$ is the only feasible vector of the capacitated instance. Otherwise, let $\boldsymbol{x}$ be the solution.

ii. If $p(\boldsymbol{x}) = 0$, stop; the vectors $r\boldsymbol{x}$ are feasible for all $r \geq 0$, and the capacitated problem is unbounded.

iii. Otherwise, when $p(\boldsymbol{x}) \neq 0$, compute the largest number r (must be bounded), such that $r\boldsymbol{Ux} \leq \boldsymbol{d}$. Claim that $r\boldsymbol{x}$ is $(1/\ell)$-optimal.

Correctness: Consider the vector $\boldsymbol{x}$ computed in step i of the algorithm. Note that $\boldsymbol{x}$ is such that $\hat{t}(\boldsymbol{x}) = 1$. Hence, $\hat{t}(r\boldsymbol{x}) = r\hat{t}(\boldsymbol{x}) = r$.

Proposition 6.2.4 *$p(\boldsymbol{x}) = 0$ if and only if the capacitated problem is unbounded.*

Proof: Suppose that $p(\boldsymbol{x}) = 0$. Observe that for all $r \geq 0$, $p(r\boldsymbol{x}) = 0$, and hence, $r\boldsymbol{x}$ is feasible. Also, for all $r \geq 0$, $\hat{t}(r\boldsymbol{x}) = r$. Hence, the problem is unbounded. Suppose the problem is unbounded. There exist a vector $\boldsymbol{x}$ such that $r\boldsymbol{x}$ is feasible for all $r \geq 0$. It follows that $p(r\boldsymbol{x}) = rp(\boldsymbol{x}) \leq \ell$ for all $r \geq 0$. Hence, $p(\boldsymbol{x}) = 0$. ∎

The following proposition concludes the correctness proof.

Proposition 6.2.5 *If t^* is bounded, then $r \geq t^*/\ell$.*

Proof: For $k \geq 0$, denote

$$\begin{aligned} R(k) &= \max\{\hat{t}(y) \mid p(y) \leq k, Ay \propto b, y \geq 0\} \text{ , and} \\ R^*(k) &= \max\{\hat{t}(y) \mid p(y) \leq k, y \text{ is feasible}\} \text{ .} \end{aligned}$$

Obviously, (i) R and R^* are increasing functions, (ii) $R \geq R^*$, (iii) for every $a > 0$, $R(ak) = aR(k)$, and (iv) $t^* \leq R^*(\ell)$.

The vector x (computed in step i) is such that $R(p(x)) = 1$, and hence $R(p(rx)) = r$. Since $p(x) > 0$, $U_{i\bullet}rx = d_i$ for some $1 \leq i \leq \ell$. Hence, $p(rx) \geq 1$. It follows that

$$t^* \leq R^*(\ell) \leq R(\ell) \leq R(\ell p(rx)) = \ell r R(p(x)) = \ell r \text{ .}$$

∎

6.3 Bidirected generalized networks

In the previous sections we discussed generalized networks where the flow multipliers are positive numbers. We refer to the edges in these networks as tail-head edges. Tail-head edges contribute nonnegative amount of flow at the tail end of the edge and a proportional nonpositive amount at the head end. In bidirected generalized networks we allow two additional types of edges: two-head edges and two-tail edges. The properties of these edge types are shown in Figure 6.1. Note that a 2-tail edge can be viewed as a tail-head edge with a negative multiplier. Bidi--cted generalized networks are a generalization of bidirected networks (see [42]). In biderected networks the multipliers associated with the edges are always unity. Bidirected networks were first considered by Edmonds [20] who related them to non-bipartite matching theory. In this section we apply the methods discussed in previous sections to flow problems on bidirected generalized networks.

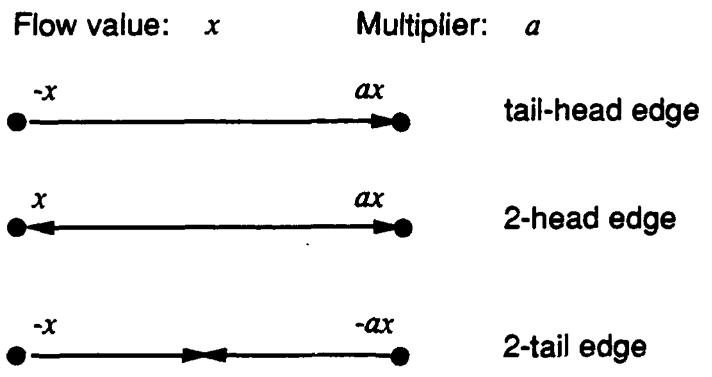


Figure 6.1: Edge types of a bidirected generalized network

6.3.1 UGT on bidirected networks

We discuss applying the Adler-Cosares scheme for solving UGT on bidirected networks. A bidirected UGT problem has the form: Minimize $\boldsymbol{c}^T\boldsymbol{x}$, subject to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, $\boldsymbol{x} \geq 0$, where $\boldsymbol{b} \in R^n$, $\boldsymbol{c} \in R^m$, and $\boldsymbol{A} \in R^{n \times m}$ has at most two non-zero entries per column. Note that head-tail edges correspond to columns where the two entries have opposite signs, 2-head edges correspond to columns with two positive entries, and 2-tail edges correspond to columns with two negative entries. The LP dual is the problem of maximizing $\boldsymbol{b}^T\boldsymbol{y}$, subject to the TVPI system $\boldsymbol{A}^T\boldsymbol{y} \leq \boldsymbol{c}$. Recall that when only head-tail edges are present, the dual has monotone constraints. When head-tail and 2-tail edges are allowed, $\boldsymbol{A}^T$ is pre-Leontief (see Subsection 5.7), and hence, there exist a vector which maximizes all variables. The Adler-Cosares scheme is applicable when $\boldsymbol{b} \geq 0$. Similarly, when head-tail and 2-head edges are present $-\boldsymbol{A}^T$ is pre-Leontief, and hence, there exist a vector which minimizes all variables. The Adler-Cosares scheme is applicable when $\boldsymbol{b} \leq 0$. When all 3 types of edges are present, the dual comprises a general TVPI system. The algorithm of Chapter 5 can find a vector which maximizes/minimizes a single variable. The Adler-Cosares scheme is applicable to problems for which $\boldsymbol{b} = \pm \boldsymbol{e}^i$. The UGT instances for which the Adler-Cosares scheme is applicable are listed in Table 6.3. These instances can

Allowed edge types	*structure of the dual* $A^T y \leq c$	*- supply/demand vectors* *- TVPI needed*
head-tail	monotone TVPI	$b \leq 0$ or $b \geq 0$ monotone (maximize $\pm e^T y$)
head-tail, 2-tail	A^T is pre-Leontief	$b \geq 0$ monotone (maximize $e^T y$)
head-tail, 2-head	$-A^T$ is pre-Leontief	$b \leq 0$ monotone (minimize $e^T y$)
head-tail, 2-tail, 2-head	general TVPI	$b_j = \pm 1$, $b_i = 0$ $(i \neq j)$ general (maximize/minimize y_j)

Table 6.3: Solving UGT on bidirected generalized networks

be solved using a single application of the algorithms of Chapter 5.

6.3.2 Generalized circulation on bidirected networks

We consider applying the approximation algorithm of Section 6.2 to generalized circulation problem where the underlying network is bidirected. Recall that the approximation algorithm iteratively computes a feasible flow and in the following iteration considers the residual graph. Since 2-tail edges give rise to 2-head edges in the residual graph (and vice versa), we only consider networks where all three edge types are present. Note that when all three edge types are present the Adler-Cosares scheme applies to UGT instances where there is a single source or a single sink (that is, $b = e_i$ or $b = -e_i$ for some $1 \leq i \leq n$).

It follows that the approximation scheme presented in Section 6.2 can be used to solve bidirected generalized circulation instances where $b = \pm e_i$ for some $1 \leq i \leq n$.

6.4 Concluding remarks

In this chapter we presented algorithms for the uncapacitated generalized transshipment (UGT) problem and the generalized circulation (GC) problem. We also considered the UGT and GC problems on bidirected generalized networks. To solve UGT, we combined a scheme by Adler and Cosares [1], which reduces the restricted UGT problem where either only demand nodes or only supply nodes are present to solving the LP dual, with the algorithms given in Chapter 5. The combination yielded better time bounds for restricted UGT instances.

In order to utilize the UGT algorithms for solving the capacitated GC problem, we introduced an iterative approximation algorithm. In each iteration, we consider a UGT instance, with costs which "capture" the relaxed capacities. The solution of this UGT instance yields an approximate solution for the GC instance. The next iteration considers the residual graph.

We comment on the parallel running times of the algorithms mentioned above. The parallel complexity of the algorithms of Chapter 5 is $\tilde{O}(n)$ using $O(mn)$ processors for the deterministic bound and $O(m+n^2)$ processors for the randomized bound. The algorithms for the restricted UGT instances have the same complexity. The approximation algorithm for GC runs in $\tilde{O}(mn \log \epsilon^{-1})$ time using $O(mn)$ processors for the deterministic bound and $O(m+n^2)$ processors for the randomized bound.

A problem which remains open regards the existence of a strongly polynomial for the unrestricted UGT problem (where many sources and sinks are allowed). The LP dual of UGT is a monotone system with a general objective function; what brings as back to an open question from Chapter 5. The following reduction (similar to [52]) demonstrates the difficulty of the problem. An instance of capacitated generalized transshipment can be reduced in linear time to an instance of UGT with $3m$ edges and $n+2m$ nodes. The reduction is as follows. Consider an instance of generalized circulation on a network $G=(V,E)$ where $d_v \in R$ is the supply/demand at v ($v \in V$), and a_e,u_e,c_e are the multiplier, capacity, and cost, respectively, of the edge e ($e \in E$). The corresponding UGT instance $G' = (V \cup W \cup W', E')$ preserves the supplies and demands at the nodes V. Each edge $e \in E$ has corresponding two nodes $w_e \in W$,$w'_e \in W'$ and three edges in E' which form an undirected path. The node w_e has demand u_e and w'_e has supply $-u_e$. Suppose $e=(v_1,v_2)$, the corresponding edges are (i) (v_1,w) with multiplier 1 and cost c_e, (ii) (w',w) with multiplier 1 and cost 0, and (iii) (w',v_2) with multiplier a_e and cost 0.

Chapter 7

Conclusion

In Chapter 2 we presented an NC and strongly polynomial algorithm to detect cycles in fixed dimensional periodic graphs. This algorithm is based on a strongly polynomial algorithm given in Chapter 3 for the parametric minimum cycle problem with fixed number of parameters. In Chapter 4 we presented the algorithm of Chapter 3 as a general tool for achieving strongly polynomial time bounds. In Chapter 5 we gave faster algorithms for linear systems of inequalities, where at most two variables appear in each inequality. In Chapter 6 we introduced algorithms for some generalized network flow problems which utilize the results of Chapter 5. In particular we obtained a faster algorithm for approximating the optimal generalized circulation, which is also the first strongly polynomial algorithm for the problem. The existence of this approximation algorithm is particularly interesting since it is still open whether there exist a strongly polynomial algorithm for computing the optimal solution.

These results are interesting in and of themselves, but our work is a step towards attacking an important open problem of the field: finding a strongly polynomial algorithm for general linear programming.

Bibliography

[1] I. Adler and S. Cosares. Strongly polynomial algorithms for linear programming problems with special structure. *Oper. Res.*, 1991. To appear.

[2] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM J. Comput.*, 18:939–954, 1989.

[3] B. Aspvall. *Efficient algorithms for certain satisfiability and linear programming problems.* PhD thesis, Department of Computer Science, Stanford University, Stanford, Ca., 1980.

[4] B. Aspvall and Y. Shiloach. A polynomial time algorithm for solving systems of linear inequlities with two variables per inequality. *SIAM J. Comput.*, 9:827–845, 1980.

[5] K. L. Clarkson. Linear programming in $O(n \times 3^{d^2})$ time. *Information Processing Let.*, 22:21–27, 1986.

[6] E. Cohen and N. Megiddo. Strongly polynomial and NC algorithms for detecting cycles in dynamic graphs. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 523–534. ACM, 1989.

[7] E. Cohen and N. Megiddo. Maximizing concave functions in fixed dimension. Technical Report RJ 7656 (71103), IBM Almaden Research Center, San Jose, CA 95120-6099, August 1990.

[8] E. Cohen and N. Megiddo. Recognizing properties of periodic graphs. Technical Report RJ 7246 (68013), IBM Almaden Research Center, San Jose, CA 95120-6099, January 1990.

[9] E. Cohen and N. Megiddo. Strongly polynomial time and NC algorithms for detecting cycles in periodic graphs. Technical Report RJ 7587 (70764), IBM Almaden Research Center, San Jose, CA 95120-6099, July 1990.

[10] E. Cohen and N. Megiddo. Complexity analysis and algorithms for some flow problems. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 120–130. ACM-SIAM, 1991.

[11] E. Cohen and N. Megiddo. Improved algorithms for linear inequalities with two variables per inequality. In *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 145–155. ACM, 1991.

[12] E. Cohen and N. Megiddo. Recognizing properties of periodic graphs. *Applied Geometry and Discrete Math.*, The "Victor Klee Festschrift", 1991. To appear.

[13] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms.* McGraw-Hill Book Co., New York, 1990.

[14] S. Cosares. *On the complexity of some primal-dual linear programming pairs.* PhD thesis, Department of IEOR, University of California, Berkeley, Ca., 1988.

[15] R. W. Cottle and A. F. Veinott Jr. Polyhedral sets having a least element. *Math. Prog.*, 3:238–249, 1972.

[16] J. B. Dantzig. *Linear programming and extensions.* Princeton Univ. Press, Princeton, NJ, 1962.

[17] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[18] D. P. Dobkin, R. J. Lipton, and S. P. Reiss. Linear programming is Log-Space hard for P. *Information Processing Let.*, 8(2):96–97, 1978.

[19] M. E. Dyer. On a multidimensional search technique and its application to the Euclidean one-center problem. *SIAM J. Comput.*, 15:725–738, 1986.

[20] J. Edmonds. An introduction to matchings. In *Engineering Summer Conference*. The Univ. of Michigan, Ann Arbor, 1967. Mimeographed notes.

[21] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1972.

[22] J. Farkas. Theorie der einfachen ungleichungen. *J. Reine und Angewandte Math.*, 124:1–27, 1902.

[23] W. Feller. *An introduction to probability theory and its applications*. John Wiley & Sons, New York, 1950.

[24] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Annual ACM Symposium on Theory of Computing*, pages 114–118. ACM, 1978.

[25] F. Glover, J. Hultz, D. Klingman, and J. Stunz. Generalized networks: a fundamental computer-based planning tool. *Management Science*, 24(12), August 1978.

[26] A. V. Goldberg, S. K. Plotkin, and É. Tardos. Combinatorial algorithms for the generalized circulation problem. In *Proc. 29th IEEE Annual Symposium on Foundations of Computer Science*, pages 432–443. IEEE, 1988.

[27] A. V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. In *Proc. 18th Annual ACM Symposium on Theory of Computing*, pages 136–146. ACM, 1986.

[28] A. V. Goldberg and R.E. Tarjan. Finding minimum cost circulations by successive approximation. *Math. of Oper. Res.*, 15:430–466, 1990. To appear.

[29] M. Gondran and M. Minoux. *Graphs and Algorithms*. John Wiley & Sons, New York, 1984.

[30] B. Grötschel, L. Lovasz, and Schrijver A. *Geometric algorithms and combinatorial optimization*. Springer-Verlag, New York, 1988.

[31] B. Grünbaum. *Convex polytopes*. Interscience-Wiley, London, 1967.

[32] D. Hochbaum and J. Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. Manuscript, 1991.

[33] K. Iwano. Some problems on doubly periodic infinite graphs. Technical Report CS-TR-078-87, Princeton University, 1987.

[34] K. Iwano and K. Steiglitz. Testing for cycles in infinite graphs with periodic structure. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 46–53. ACM, 1987.

[35] K. Iwano and K. Steiglitz. Planarity testing of doubly connected periodic infinite graphs. *Networks*, 18:205–222, October 1988.

[36] K. Kapoor and P. M. Vaidya. Speeding up Karmarkar's algorithm for multicommodity flows. *Math. Prog.*, 1991. To appear.

[37] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[38] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[39] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. Assoc. Comput. Mach.*, 14:563–590, 1967.

[40] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20:191–194, 1979.

[41] K. S. Kosaraju and G. F. Sullivan. Detecting cycles in dynamic graphs in polynomial time. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 398–406. ACM, 1988.

[42] E. L. Lawler. *Combinatorial optimization: networks and matroids.* Holt, Reinhart, and Winston, New York, 1976.

[43] G. S. Lueker, N. Megiddo, and V. Ramachandran. Linear programming with two variables per inequality in poly log time. *SIAM J. Comput.*, 19(6):1000–1010, 1990.

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

[45] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. Assoc. Comput. Mach.*, 30:337–341, 1983.

[46] N. Megiddo. Towards a genuinely polynomial algorithm for linear programming. *SIAM J. Comput.*, 12:347–353, 1983.

[47] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. Assoc. Comput. Mach.*, 31:114–127, 1984.

[48] S. Murray. An interior point conjugate gradient approach to the generalized flow problem with costs and the multicommodity flow problem dual. Manuscript, 1991.

[49] C. G. Nelson. An $n^{O(\log n)}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical Report AIM-319, Stanford University, 1978.

[50] C. H. Norton, S. A. Plotkin, and É. Tardos. Using separation algorithms in fixed dimension. In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 377–387. ACM-SIAM, 1990.

[51] J. B. Orlin. Some problems in dynamic/periodic graphs. In W. R. Pullyblank, editor, *Progress in combinatorial optimization*, pages 273–293, Orlando, Florida, 1984. Academic Press.

[52] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 377–387. ACM, 1988.

[53] N. Pippenger. On simultaneous resource bounds. In *Proc. 20th IEEE Annual Symposium on Foundations of Computer Science*, pages 307–311. IEEE, 1979.

[54] J. Renegar. A polynomial time algorithm, based on newtons method, for linear programming. *Math. Prog.*, 40:59–94, 1988.

[55] V. P. Roychowdhury. *Derivation, Extensions, and Parallel Implementation of Regular Iterative Algorithms.* PhD thesis, Department of Electrical Engineering, Stanford University, Stanford, Ca., 1989.

[56] V. P. Roychowdhury and T. Kailath. Study of parallelism in regular iterative algorithms. In *Proc. of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 367–376. ACM, 1990.

[57] R. Shostak. Deciding linear inequalities by computing loop residues. *J. Assoc. Comput. Mach.*, 28:769–779, 1981.

[58] É. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.

[59] É. Tardos. A strongly polynomial algorithm to solve combinatorial linear programs. *Oper. Res.*, 34:250–256, 1986.

[60] P. M. Vaidya. Speeding-up linear programming using fast matrix multiplication. In *Proc. 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 332–337. IEEE, 1989.

[61] A. F. Veinott Jr. Representation of general and polyhedral subsemilattices and sublattices of product spaces. *Linear Algebra and its Applications*, 114/115:681–704, 1989.