2013

Optimal certifying algorithms for linear and lattice point feasibility in a system of UTVPI constraints

Piotr Jerzy Wojciechowski
West Virginia University

Optimal certifying algorithms for linear and lattice point feasibility in a system of UTVPI constraints

by

Piotr Jerzy Wojciechowski

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

James Mooney, Ph.D.
Hong-Jian Lai, Ph.D.
K. Subramani, Ph.D., Chair

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2013

Keywords:
algorithm; linear programming; integer programming; feasibility; UTVPI

Abstract

Optimal certifying algorithms for linear and lattice point feasibility in a system of UTVPI constraints

by

Piotr Jerzy Wojciechowski
Master of Science in Computer Science

West Virginia University

K. Subramani, Ph.D., Chair

This thesis is concerned with the design and analysis of time-optimal and space-optimal, *certifying* algorithms for checking the linear and lattice point feasibility of a class of constraints called Unit Two Variable Per Inequality (UTVPI) constraints. In a UTVPI constraint, there are at most two non-zero variables per constraint, and the coefficients of the non-zero variables belong to the set $\{+1, -1\}$. These constraints occur in a number of application domains, including but not limited to program verification, abstract interpretation, and operations research. As per the literature, the fastest known certifying algorithm for checking lattice point feasibility in UTVPI constraint systems ([1]), runs in $O(m \cdot n + n^2 \cdot \log n)$ time and $O(n^2)$ space, where m represents the number of constraints and n represents the number of variables in the constraint system. In this paper, we design and analyze new algorithms for checking the linear feasibility and the lattice point feasibility of UTVPI constraints. Both of the presented algorithms run in $O(m \cdot n)$ time and $O(m + n)$ space. Additionally they are certifying in that they produce satisfying assignments in the event that they are presented with feasible instances and refutations in the event that they are presented with infeasible instances. The importance of providing certificates cannot be overemphasized, especially in mission-critical applications. Our approaches for both the linear and the lattice point feasibility problems in UTVPI constraints are fundamentally different from existing approaches for these problems (as described in the literature), in that our approaches are based on new insights on using well-known inference rules.

Acknowledgements

First and foremost I would like to thank my advisor Dr. K. Subramani whose insights and advice made this thesis possible. I also thank Dr. Hong-Jian Lai and Dr. James Mooney for agreeing to be on my thesis committee.

Last but not least, I would like to thank my family for being there to support me in my research efforts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we propose new certifying algorithms for checking the linear and lattice point (integer) feasibility of a conjunction of Unit Two Variable Per Inequality (UTVPI) constraints. A UTVPI constraint is a linear constraint of the form: $a \cdot x + b \cdot y \leq d$, where $a, b \in \{-1, 0, 1\}$ and d is an integer constant. A conjunction of such constraints is called a UTVPI constraint system. Observe that UTVPI systems subsume difference constraint systems [2], since in the latter, a and b must have opposite signs.

UTVPI constraints occur in a number of problem domains including but not limited to program verification [1], abstract interpretation [3, 4], real-time scheduling [5] and operations research. Indeed many software and hardware verification queries are naturally expressed using this fragment of integer linear arithmetic, i.e., the case in which the solutions of a UTVPI system are restricted to be integral. We note that when the goal is to model indices of an array or queues in hardware or software, rational solutions are unacceptable [1]. Other application areas include spatial databases [6] and theorem proving. When the range restrictions on a and b are removed, i.e., they are permitted to be arbitrary integers, then the constraint system is called a Two Variable Per Inequality (TVPI) system. Checking integer feasibility in TVPI systems is known to be weakly **NP-complete** [7].

This thesis deals with both the linear feasibility problem and the integer feasibility problem in UTPVI systems. Our algorithms are based on the following ideas, which

to the best of our knowledge have not been discussed in the literature:

1. We propose a new constraint network structure for UTVPI constraints that is similar to the constraint network structure for difference constraints [8], but incorporates many features that are unique to UTVPI constraint systems (see Section 2). This constraint structure enables the extraction of both linear and lattice point solutions.

2. We present theorems of the alternative for the recognition of the linear and integer feasibility of UTVPI constraints, which are similar in spirit to Farkas' lemma for a system of linear constraints. These theorems are crucial from the perspective of designing certifying algorithms [9].

The algorithms that we present run in $O(m \cdot n)$ time and use $O(m+n)$ space on a UTVPI constraint system with n variables over m constraints. For the case of integer feasibility this is a marked improvement over the current state-of-the-art certifying algorithm which runs in $O(m \cdot n + n^2 \cdot \log n)$ time and $O(n^2)$ space [1]. We note that the fastest known strongly polynomial time algorithm for checking linear (and hence, integer) feasibility in difference constraints is the Bellman-Ford procedure (or one of its variants), which runs in $O(m \cdot n)$ time and $O(m + n)$ space. It follows that our algorithms for linear and integer feasibility checking in a UTVPI constraint system are optimal, since UTVPI constraints subsume difference constraints. It is important to note that unlike difference constraints linear feasibility does not imply lattice point feasibility in UTVPI constraints (see Section 2).

We reiterate the fact that our algorithms are certifying, i.e., in the event that the given UTVPI system is feasible, we provide a satisfying assignment and in the event that it is infeasible, we provide a refutation, which explains the infeasibility. The nature of the satisfying assignment and the nature of the refutation depends on whether we are interested in linear feasibility or integer feasibility. Even algorithms that can be proven correct, suffer the risk of being implemented incorrectly. One of the more famous examples of this phenomenon is the error discovered in the planarity testing

algorithm of the LEDA software [10]. Consequently, there is widespread interest in the design and development of certifying algorithms, i.e., algorithms which provide certificates that validate the answer that is provided. For instance, an algorithm for graph planarity testing could provide a planar embedding when it declares a graph to be planar, and a subgraph of the input graph that is homeomorphic to $K_{3,3}$ or K_5, in the event that it declares the graph to be non-planar (Kurtowski's theorem). It is understood that the implementations of algorithms for verifying a planar embedding and checking homeomorphism to $K_{3,3}$ and K_5 are trivial enough to be checked by a simple, provably correct implementation.

The important contributions of this thesis are as follows:

(i) A new characterization of linear infeasibility in UTVPI constraint systems,

(ii) A new characterization of integer infeasibility in UTVPI constraint systems,

(iii) An optimal (time and space) certifying algorithm (**LA**) for checking linear feasibility in UTVPI constraint systems, and

(iv) An optimal (time and space) certifying algorithm (**IA**) for checking integer feasibility in UTVPI constraint systems.

The rest of this thesis is organized as follows: Section 2 formally specifies the problem under consideration. A theorem of the alternative for linear feasibility in UTVPI constraint systems is discussed in Section 3.1. In Section 3.2, we detail a theorem of the alternative for integer feasibility in UTVPI constraint systems. These theorems exactly characterize linear and integer feasibility in UTVPI systems and can be used to extract refutations in the event of infeasibility. Section 4 describes the motivation for our work, as well as related work in the literature. Our algorithm for the linear feasibility problem in UTVPI systems is presented in section 5. The proof of correctness of this algorithm is detailed in Section 6. Section 7 details the new algorithm for the lattice point feasibility problem in UTVPI constraint systems.

A detailed proof of correctness of this algorithm is provided in Section 8. Section 9 describes the working of our lattice-point algorithm on a sample UTVPI system. We conclude in Section 10 by summarizing our contributions and outlining avenues for future research.

Chapter 2

Statement of Problem

In this section, we formally define the linear and integer feasibility problems in UTVPI constraints and also define the various terms that will be used in the rest of the thesis.

Definition 2.0.1 *A constraint of the form $a_i \cdot x_i + a_j \cdot x_j \leq c_{ij}$ is said to be a Unit Two Variable Per Inequality (UTVPI) constraint if $a_i, a_j \in \{-1, 0, +1\}$ and $c_{ij} \in \mathbb{Z}$.*

Definition 2.0.2 *A constraint of the form $x_i \leq c_i$ or $x_i \geq c_i$ where $c_i \in \mathbb{Z}$ is called an absolute constraint.*

Absolute constraints are the subset of UTVPI constraints where one of the coefficients (a_i or a_j) is 0. They can be converted into constraints of the form: $a_i \cdot x_i + a_j \cdot x_j \leq c_{ij}$, where both a_i and a_j are non-zero (see Section 2.1).

Definition 2.0.3 *The constant which bounds a UTVPI constraint is called the defining constant.*

For instance, the defining constant for the constraint $x_1 - x_2 \leq 9$ is 9. *Example (1):*

Definition 2.0.4 *A conjunction of UTVPI constraints is called a UTVPI constraint system and can be represented in matrix form as $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$. If the constraint system has m constraints over n variables, then $\mathbf{A}$ has dimensions $m \times n$.*

UTVPI constraints are also known as Generalized 2SAT constraints [11] and are the invariants of the octagon abstract domain in [3].

Observe that a UTVPI system defines a polyhedron in n-dimensional space. Given such a system, we are interested in the following questions:

(i) Is the defined polyhedron non-empty? This problem is called the *Linear Feasibility problem* (LF).

(ii) Does the defined polyhedron enclose a lattice point? This problem is called the *Integer Feasibility problem* (IF).

Our goal is to design certifying algorithms for the LF and IF problems. In other words, our algorithms should produce models (satisfying solutions) for feasible instances and refutations for infeasible instances. Our algorithms incorporate the following six properties of UTVPI constraints

(i) A UTVPI system has a constraint network presentation, analogous to the constraint network representation of a Difference Constraint System (see Chapter 24 of [8]).

(ii) A UTVPI system is linear feasible if and only if the corresponding constraint network does not contain certain types of cycles (see Section 3.1).

(iii) A UTVPI system is integer feasible if and only if the corresponding constraint network does not contain certain types of cycles (see Section 3.2).

(iv) Fourier-Motzkin with rounding (FMR) is a sound and complete procedure for detecting integer feasibility in UTVPI constraints ([11]).

(v) A certificate of linear (and integer) infeasibility consists of at most $2 \cdot n$ constraints (see Section 3.1 and Section 3.2).

(vi) Given a solution to the LF problem in a UTVPI system, we can obtain a lattice point solution (or establish that none exists) by a rounding procedure in $O(m \cdot n)$ time (see Sections 7 and 8).

While integer feasibility in a UTVPI system immediately implies linear feasibility, the converse is not true. For instance, consider the UTVPI system defined by the following constraints:

$$\begin{aligned} x_1 + x_2 &\geq 1 \\ -x_1 + x_2 &\geq 0 \\ x_1 - x_2 &\geq 0 \\ -x_1 - x_2 &\geq -1 \end{aligned} \tag{2.1}$$

It is clear that System (2.1) has no lattice point (integer) solution. However, it contains the fractional point $(\frac{1}{2}, \frac{1}{2})$ and is thus non-empty.

2.1 Constraint Network Presentation

Let $\mathbf{U} : \mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$ denote the UTVPI constraint system and let $\mathbf{X}$ denote the set of all (fractional and integral) solutions to $\mathbf{U}$. Corresponding to this constraint system we construct the constraint network $\mathbf{G} = \langle V, E, \mathbf{c} \rangle$ as follows.

For each variable x_i create a vertex in V. For ease of reference, both the variable and its corresponding node are referred to as x_i in this thesis.

Constraints are represented as edges using the following rules:

(a) A constraint of the form $x_i - x_j \leq c_{ij}$ is represented as a directed edge from the node x_j to the node x_i having weight c_{ij}. These edges are called "gray" edges and are represented by $\overset{c_{ij}}{\leftarrow}$ where c is the weight.

(b) A constraint of the form $-x_i - x_j \leq c_{ij}$ is represented by an undirected "black" edge ($\overset{c_{ij}}{\blacksquare}$).

(c) A constraint of the form $x_i + x_j \leq c_{ij}$ is represented by an undirected "white" edge ($\overset{c_{ij}}{\square}$) respectively.

A $(k-1)$-path in our constraint network, is a sequence of k vertices, x'_1, x'_2, $\ldots$ x'_k, and $(k-1)$ edges e_1, e_2, $\ldots$ e_{k-1}, such that e_i is the edge corresponding to one of the constraints between x'_i and x'_{i+1} in the UTVPI constraint system.

For a k-path to be considered valid, it must have the following property: For every i from 2 to $k-1$, the coefficients of x'_i in the constraints corresponding to the edges e_i and $e_{(i-1)}$ have opposite signs.

The path defined by the sequence of vertices x_1, x_2, x_3, x_4 and the sequence of edges $x_1 \overset{c_{1,2}}{\blacksquare} x_2$, $x_2 \overset{c_{2,3}}{\blacksquare} x_3$, $x_3 \overset{c_{3,4}}{\square} x_4$ is $x_1 \overset{c_{1,2}}{\blacksquare} x_2 \overset{c_{2,3}}{\blacksquare} x_3 \overset{c_{3,4}}{\square} x_4$. However this path is not valid because the the coefficients of x_2 in the constraints corresponding to the edges $x_1 \overset{c_{1,2}}{\blacksquare} x_2$ and $x_2 \overset{c_{2,3}}{\blacksquare} x_3$ have the same sign; indeed, both of these constraints are of the form $-x_i - x_j \leq c_{ij}$. *Example (2):*

A *closed walk* is simply a valid $(k-1)$-path for which $x_1 = x_k$. In this thesis, we refer to closed walks as cycles. Note that a cycle, as defined above can consist of edges and vertices that occur more than once. Thus, the notion of a cycle in this

thesis differs from the notion of a cycle in a constraint network corresponding to a difference constraint system.

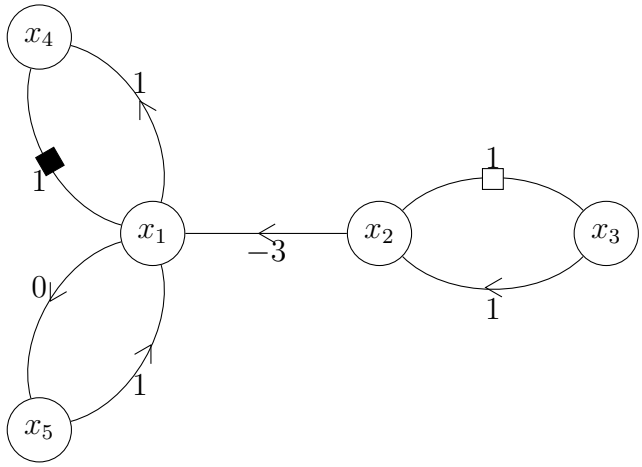

Figure 2.1: Example Constraint Network (without node x_0)

Suppose we have the system of constraints

1. $x_1 - x_2 \leq -3$
2. $-x_1 + x_4 \leq 1$
3. $-x_1 - x_4 \leq 1$
4. $x_1 - x_5 \leq 1$
5. $-x_1 + x_5 \leq 0$
6. $x_2 + x_3 \leq 1$
7. $x_2 - x_3 \leq 1$

Then, as we can see in Figure 2.1 the 8-path

$$x_1 \overset{-3}{\leftarrow} x_2 \overset{1}{\square} x_3 \overset{1}{\rightarrow} x_2 \overset{-3}{\rightarrow} x_1 \overset{0}{\rightarrow} x_5 \overset{1}{\rightarrow} x_1 \overset{1}{\rightarrow} x_4 \overset{1}{\blacksquare} x_1$$

forms a cycle even though the nodes x_1 and x_2 and the edge $x_2 \overset{-3}{\rightarrow} x_1$ are used multiple times. *Example (3):*

Since the "white" and "black" edges are directionless we will need to treat the "gray" edges as directionless as well. As we will show in section 2.2, $x_i \overset{c_{ij}}{\leftarrow} x_j \overset{c_{jk}}{\square} x_k \overset{c_{kl}}{\rightarrow} x_l$ is a valid path from x_i to x_l but requires gray edges to be traversed in both directions.

We add a node x_0 to the network. This node will be the starting point which our algorithms will utilize for traversing the network. Without of loss of generality, we assume that node x_0 is assigned the value 0. This gives us a point of reference and allows us to determine values for the remaining variables. For each node x_i in the network we add the four edges $x_0 \overset{n\cdot C}{\square} x_i$, $x_0 \overset{n\cdot C}{\blacksquare} x_i$, $x_0 \overset{n\cdot C}{\rightarrow} x_i$, and $x_i \overset{n\cdot C}{\rightarrow} x_0$ where C is the largest absolute weight of any edge in the network. These edges allow every vertex to be reached from x_0 using the reachability technique employed by our algorithms, without *introducing* infeasibility into the system. As discussed in Section 3.1, a UTVPI system is infeasible if and only if there exists a cycle of negative weight. Observe that any cycle that is introduced by the addition of x_0, must use x_0 and therefore, at least one edge that enters x_0 and at least one edge that leaves x_0. However, these edges have such a large weight $(n \cdot C)$, that the weight of such a cycle cannot be negative, unless a negative weight cycle existed in the network to begin with. This is clarified further in Lemma 6.0.3 of Section 6.

The newly added edges also permit the addition of absolute constraints. An absolute constraint $x_i \leq c$ is converted into a pair of constraints: $x_i + x_0 \leq c$ and $x_i - x_0 \leq c$, which are added to the UTVPI system (after the absolute constraint is deleted from the system). The corresponding edges are added to the constraint network by changing the weight of the appropriate edges from x_0. In the preceding example this would mean changing the weights of the edges corresponding to the constraints $x_0 \square x_i$ and $x_0 \rightarrow x_i$ to c.

We will now argue that the above replacement strategy is solution-preserving, i.e., if the original UTVPI system is feasible, then it stays feasible after the replacement. Likewise, if the original system is infeasible, then it stays infeasible after the replacement.

Let $\mathbf{P_1} : \mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$ denote a UTVPI system with $x_1 \leq c$ denoting an absolute

constraint in this system. We consider the following cases:

(i) $\mathbf{P_1}$ is non-empty - We can set $x_0 = 0$, thus after replacement the constraints $x_1 + x_0 \leq c$ and $x_1 - x_0 \leq c$ both become $x_1 \leq c$ thus the system remains feasible with $x_0 = 0$ part of a satisfying assignment.

(ii) $\mathbf{P_1}$ is empty - Observe that if there exists a subsystem of $\mathbf{P_1}$ that is infeasible and which does not include the constraint $x_1 \leq c$, then it stays infeasible after the replacement. Let us therefore consider the case in which the constraint $x_1 \leq c$ is part of the only infeasible subsystem of $\mathbf{P_1}$. In this case, we add $x_1 + x_0 \leq c$ and $x_1 - x_0 \leq c$, to produce the constraint $2 \cdot x_1 \leq 2 \cdot c$ which is equivalent to the original constraint. Thus replacing $x_1 \leq c$ does not affect the infeasibility of the system.

Consider the following constraint system.

$$\begin{aligned} x_1 + x_3 &\leq 0 \\ x_2 - x_3 &\leq -7 \\ x_4 - x_2 &\leq 3 \\ -x_1 - x_4 &\leq 5 \\ x_1 &\leq 6 \end{aligned} \tag{2.2}$$

The resulting network is shown in Figure 2.2.

We now contrast our constraint network construction with the representation in [3], which was the basis of the network construction in [1].

The [3] network construction produces what is called a potential network, constructed as follows:

For each variable, two nodes (a positive version and a negative version) are added to the constraint network. For instance, corresponding to the variable x_i, we create the nodes x_i^+ and x_i^-. Each constraint is replaced by a pair of equivalent constraints is found. For instance, a difference constraint $x_i - x_j \leq c$ is equivalent to the two constraints $x_i^+ - x_j^+ \leq c$ and $x_j^- - x_i^- \leq c$. The exception is for absolute constraints,

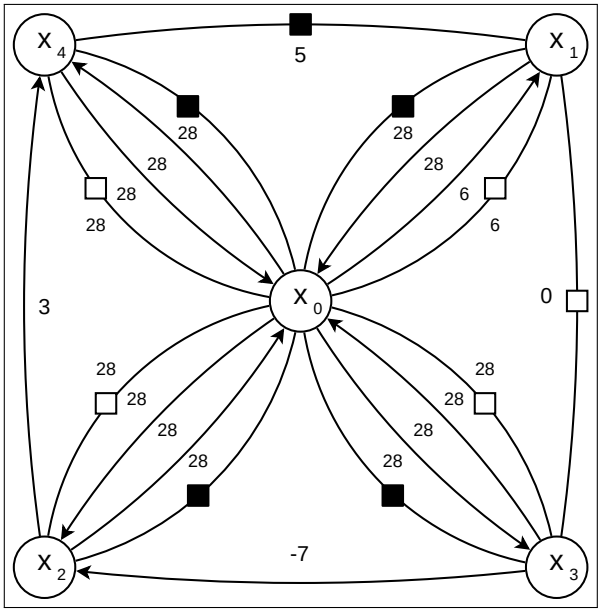

Figure 2.2: Example constraint network.

each of which is simply converted to a single equivalent constraint. For instance, $x_i \leq c$ yields $x_i^+ - x_i^- \leq 2 \cdot c$. Once all the equivalent constraints have been determined, they are represented in a constraint network, as discussed in [8]. It is thus seen that the network constructed as per [3] has $2 \cdot n$ vertices (assuming n variables in the constraint system) and up to $2 \cdot m$ edges (assuming m constraints in the original constraint system). The resultant graph is called the potential graph. Figure 2.3 shows the potential graph, corresponding to System (2.2).

It is important to note that even if the constraint system consisted solely of difference constraints, our constraint network differs from the one proposed in [8] (for instance, the weights on the edges from x_0 to the other nodes are not 0).

Our method differs from [3] and [1] in several respects:

(a) Our constraint network contains undirected edges, with special rules on how to follow them. Accordingly, we are not limited by the direction of the edges as used in the potential network.

(b) We are able to retain the information related to the constraint types themselves in the networks they produce. In other words, our network directly reflects the

original input.

(c) Rather than converting an entire system of constraints into a format comprising exclusively difference constraints, we directly handle all forms of constraints. This allows us to easily reproduce a sequence of constraints based on the edges in a subnetwork, such as a path. This is useful for producing a certificate of infeasibility.

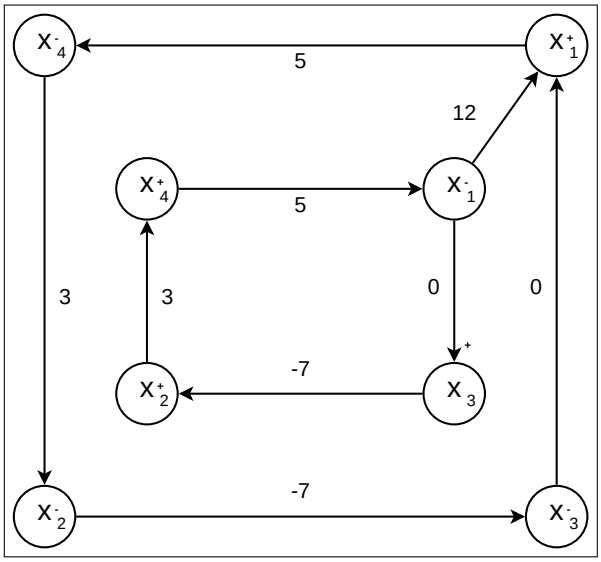

Figure 2.3: Example potential graph.

Whereas [3] and [1] have one inference rule (the addition of difference constraints), our algorithms have **four** inference rules, corresponding to edge reductions which are explained below.

2.2 Edge Reductions

We now introduce the notion of edge reductions.

Definition 2.2.1 *An edge reduction is an operation which determines a single edge equivalent to a two-edge path and represents the addition of the two UTVPI constraints which correspond to the edges in question. If this addition results in a UTVPI constraint, the reduction is said to be* valid. *Valid reductions correspond to the following transitive inference rule for UTVPI constraints:*

$$\frac{a \cdot x_i + b \cdot x_j \leq c_{ij} \qquad\qquad -b \cdot x_j + b' \cdot x_k \leq c_{jk}}{a \cdot x_i + b' \cdot x_k \leq c_{ij} + c_{jk}}$$

In the case of a valid reduction, since the resultant constraint is a valid UTVPI constraint, the path reduces to an edge corresponding to the sum of the two constraints.

The following table lists the valid edge reductions:

Constraints	Path					Reduction			Result
$x_j - x_i \leq a,\ x_k - x_j \leq b$	x_i	$\overset{a}{\rightarrow}$	x_j	$\overset{b}{\rightarrow}$	x_k	x_i	$\overset{a+b}{\rightarrow}$	x_k	$x_k - x_i \leq a + b$
$x_j - x_i \leq a,\ -x_k - x_j \leq b$	x_i	$\overset{a}{\rightarrow}$	x_j	$\overset{b}{\blacksquare}$	x_k	x_i	$\overset{a+b}{\blacksquare}$	x_k	$-x_k - x_i \leq a + b$
$x_j + x_i \leq a,\ x_k - x_j \leq b$	x_i	$\overset{a}{\square}$	x_j	$\overset{b}{\rightarrow}$	x_k	x_i	$\overset{a+b}{\square}$	x_k	$x_k + x_i \leq a + b$
$-x_j - x_i \leq a,\ x_k + x_j \leq b$	x_i	$\overset{a}{\blacksquare}$	x_j	$\overset{b}{\square}$	x_k	x_i	$\overset{a+b}{\rightarrow}$	x_k	$x_k - x_i \leq a + b$

Table 2.1: Valid Edge Reductions

Not all edge reductions are valid. For example, the reduction of the path $x_i \overset{c_{ij}}{\square} x_j \overset{c_{jk}}{\square} x_k$, corresponding to the constraints $x_i + x_j \leq c_{ij}$ and $x_j + x_k \leq c_{jk}$, is not valid since adding the constraints would produce the constraint $x_i + 2x_j + x_k \leq c_{ij} + c_{jk}$ which is not a UTVPI constraint. However, the reduction of the path $x_i \overset{c_{ij}}{\square} x_j \overset{c_{jk}}{\blacksquare} x_k$, corresponding to the constraints $x_i + x_j \leq c_{ij}$ and $-x_j - x_k \leq c_{jk}$, is valid since adding the constraints would produce the constraint $x_i - x_k \leq c_{ij} + c_{jk}$ which is a UTVPI constraint.

Reductions can also be applied to longer paths by repeatedly applying the two edge reductions until only one edge remains. A path P with k edges is said to reduce to an edge e if there exists a series of $(k-1)$ valid edge reductions which can be used to convert P to e. For instance, the path $x_1 \overset{c_1}{\square} x_2 \overset{c_2}{\blacksquare} x_3 \overset{c_3}{\square} x_4$ reduces to the edge $x_1 \overset{c_1+c_2+c_3}{\square} x_4$.

Chapter 3

Theorems of the Alternative

3.1 Linear Feasibility

In case of of difference constraints, it follows from Farkas' lemma, that we can construct a constraint network such that the original system of constraints is feasible if and only if the constructed network does not contain a negative cost cycle ([8]).

In this section, we demonstrate an analogous result between a UTVPI constraint system and its constraint network, which is constructed as per the specifications in Section 2. Recall that $\mathbf{U} : \mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$ denotes the UTVPI constraint system, $\mathbf{X}$ denotes the set of all (both fractional and integral) solutions to $\mathbf{U}$, and $\mathbf{G}$ is the constraint network created from $\mathbf{U}$.

Let $\mathbf{U}$ denote the following infeasible system of UTVPI constraints.

$$\begin{aligned} x_1 + x_2 &\leq 2 \\ x_1 + x_4 &\leq -1 \\ x_1 - x_4 &\leq -1 \\ x_3 - x_1 &\leq 0 \\ -x_1 - x_2 &\leq 2 \\ -x_1 - x_3 &\leq -3 \end{aligned} \tag{3.1}$$

The corresponding constraint network $\mathbf{G}$ (except for the node x_0) is shown in

Figure 3.1. *Example (4):*

We shall be using Example 3.1 to illustrate several of the lemmata and theorems in this section.

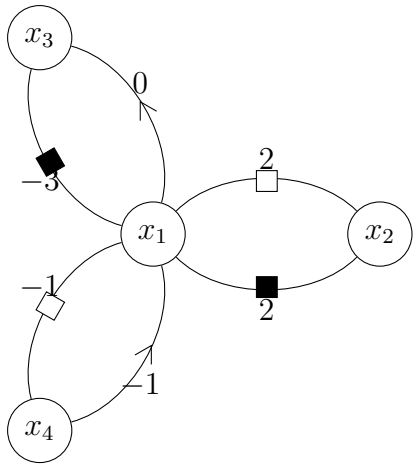

Figure 3.1: Example Constraint Network (without node x_0)

Theorem 3.1.1 *Either* $\mathbf{X}$ *is non-empty or (mutually exclusively) there exists one of the following paths in* $\mathbf{G}$*:*

(a) A path from a vertex x_i *to itself that can be reduced to a single gray edge of negative weight. This will be referred to as a path of type* (a)*.*

(b) A path of negative weight from a vertex x_i *to itself that consists of two sub-paths from* x_i *to itself, viz., a path which can be reduced to a single white edge and a path which can be reduced to a single black edge. This type of path will be referred to as a path of type* (b)*.*

For example in Figure 3.1, the cycle $x_1 \overset{-3}{\rightarrow} x_3 \overset{0}{\blacksquare} x_1 \overset{-1}{\square} x_4 \overset{-1}{\rightarrow} x_1$ is a path of type (b) because:

1. The cycle has negative weight,
2. The sub-cycle $x_1 \overset{-3}{\rightarrow} x_3 \overset{0}{\blacksquare} x_1$ can be reduced to the single black edge $x_1 \overset{-3}{\blacksquare} x_1$, and

3. The sub-cycle $x_1 \overset{-1}{\square} x_4 \overset{-1}{\rightarrow} x_1$ can be reduced to the single white edge $x_1 \overset{-2}{\square} x_1$.

Example (5):

To prove Theorem 3.1.1, we will first need to prove a number of lemmata which will build up to the desired result.

Lemma 3.1.1 *Edge reductions are associative. When reducing a path down to a single edge it does not matter in what order the reductions are performed.*

Proof: Since each reduction corresponds to the addition of two constraints, the lemma follows from the associativity of addition in inequalities. □

Consider the path $x_i \overset{c_{ij}}{\leftarrow} x_j \overset{c_{jk}}{\square} x_k \overset{c_{kl}}{\rightarrow} x_l$. If we first reduce the sub-path $x_i \overset{c_{ij}}{\leftarrow} x_j \overset{c_{jk}}{\square} x_k$, then the path $x_i \overset{c_{ij}}{\leftarrow} x_j \overset{c_{jk}}{\square} x_k \overset{c_{kl}}{\rightarrow} x_l$ reduces to $x_i \overset{c_{ij}+c_{jk}}{\square} x_k \overset{c_{kl}}{\rightarrow} x_l$ which in turn, reduces to the edge $x_i \overset{c_{ij}+c_{jk}+c_{kl}}{\square} x_l$.

Likewise, if we first reduce the sub-path $x_j \overset{c_{jk}}{\square} x_k \overset{c_{kl}}{\rightarrow} x_l$, then the path $x_i \overset{c_{ij}}{\leftarrow} x_j \overset{c_{jk}}{\square} x_k \overset{c_{kl}}{\rightarrow} x_l$ reduces to $x_i \overset{c_{ij}}{\leftarrow} x_j \overset{c_{jk}+c_{kl}}{\square} x_l$ which also reduces to the edge $x_i \overset{c_{ij}+c_{jk}+c_{kl}}{\square} x_l$.

Example (6):

Lemma 3.1.2 *If there is a path of type (b) from x_i to itself then there is a path of type (a) from x_i to itself.*

Proof: Because edge reductions are associative (Lemma 3.1.1), the path of type (b) can be reduced to a single white edge of weight c_1 followed by a single black edge of weight c_2. These edges can then be reduced to a single gray edge of weight $c_1 + c_2$. As the original path had negative weight, this resultant edge also has negative weight. Similarly, the edge goes from x_i to itself. This means that any path of type (b) is also a path of type (a). □

In Figure 3.1 the cycle $x_1 \overset{-3}{\rightarrow} x_3 \overset{0}{\blacksquare} x_1 \overset{-1}{\square} x_4 \overset{-1}{\rightarrow} x_1$ is a cycle of type (a). In Example 3.1 we showed that it it is a path of type (b) and that it can be reduced to the path $x_1 \overset{-3}{\blacksquare} x_1 \overset{-2}{\square} x_1$. This path can then be reduced to the path $x_1 \overset{-5}{\rightarrow} x_1$. *Example (7):*

Lemma 3.1.3 *If a path of type (a) exists, then $\mathbf{X}$ is empty.*

Proof: Since edge reductions correspond to additions of UTVPI constraints that produce other UTVPI constraints, the negative gray cycle corresponds to a series of UTVPI constraints that can be added to produce the constraint $x_i - x_i \leq c_i < 0$. However, this is an obvious contradiction. Thus, if a negative gray cycle exists in $\mathbf{G}$ then there is no assignment to x_i that satisfies this constraint. Thus $\mathbf{X}$ is empty. □

We have now shown one direction of the implication in Theorem 3.1.1. The following lemmata will help us show the other direction.

Lemma 3.1.4 *If* $\mathbf{X}$ *is empty then there exists a subset of constraints that can be added together (possibly with repeats) to produce a contradiction, namely a constraint of the form* $x_i - x_i \leq c < 0$.

Proof: If $\mathbf{X}$ is empty then by Farkas' Lemma there exists a rational vector $\mathbf{y} \geq 0$ such that $\mathbf{y^T} \cdot \mathbf{A} = \mathbf{0}$ and $\mathbf{y^T} \cdot \mathbf{b} < 0$. We can assume without loss of generality that $\mathbf{y} \in \mathbb{Z}^m$. Let U_j represent the j^{th} constraint of $\mathbf{U}$. We can create the set $S = \{U_j : y_j > 0,\ j = 1 \ldots m\}$, this is the set of constraints for which the corresponding element of $\mathbf{y}$ is non-zero. Summing the constraints of S with the constraint U_j appearing y_j times in the sum, for each $j = 1 \ldots m$, we get the constraint

$$x_i - x_i = 0 = \mathbf{y^T} \cdot \mathbf{A} \cdot \mathbf{x} \leq \mathbf{y^T} \cdot \mathbf{b} < 0$$

where x_i is one of the variables that is involved in a constraint in S. □

In System 3.1 all of the constraints can be added together, with no repeats, to produce the constraint $x_1 - x_1 \leq -1$. *Example (8):*

Lemma 3.1.5 *If* $\mathbf{X}$ *is empty then there exists a subset of constraints that can be added together (possibly with repeats), and an order to that addition, such that the result of the addition is* $x_i - x_i \leq c < 0$ *and at every point in the addition procedure a valid UTVPI constraint is maintained (allowing for constraints of the form* $x_i + x_i \leq c_i$*).*

Proof: Since $\mathbf{X}$ is empty, there is a set, S, of constraints and vector $\mathbf{v} > 0$ such that the the constraints in S can be added together, with each constraint S_i repeated

v_i times in the sum, to produce the constraint $x_i - x_i \leq c < 0$ (Lemma 3.1.4). We can assume without loss of generality that this pair is minimal, that is there is no $\mathbf{0} \leq \mathbf{v}' \leq \mathbf{v}$, $\mathbf{v}' \neq \mathbf{v}$, for which this property still holds.

Thus, we can construct a sequence of constraints T, which contains the constraints in S with each S_i appearing v_i times. Therefore adding all the constraints in T yields the constraint $x_i - x_i \leq 0$. Since the left hand side of this resultant constraint is simply 0, any variable introduced by adding one constraint must be canceled by the addition of some other constraint. Otherwise that variable would remain in the final sum. Utilizing this fact, the ordering of the set of constraints proceeds as follows.

(a) Start with the variable x_i which appears in at least one constraint.

(b) Let a constraint that uses x_i be the first constraint.

(c) Select as the next constraint one that eliminates the non-x_i variable introduced by the previous constraint.

(d) Repeat step (c), canceling each variable as it is introduced.

All constraints can be added in this fashion; however two situations need to be addressed:

(a) At some point, prior to adding the last constraint, the sum yields a constraint of the form $x_i - x_i \leq c \geq 0$ - In this case, the remaining constraints in T would add to $x_i - x_i \leq c < 0$ which contradicts the minimality of the pair (S, $\mathbf{v}$).

(b) At some point, prior to adding the last constraint, we get a constraint of the form $x_i - x_i \leq c < 0$ - Once again the minimality of the pair (S, $\mathbf{v}$) is contradicted.

Thus at every point in the addition sequence, a UTVPI constraint is maintained, with the allowed exception of constraints having the form: $x_i + x_i \leq c_{ii}$.

□

In System 3.1 we can start with x_1 and add the constraints as follows:

1. Start with constraint $x_1 + x_4 \leq -1$.

2. Add the constraint $x_1 - x_4 \leq -1$ to eliminate x_4 and produce the constraint $x_1 + x_1 \leq -2$

3. Add the constraint $x_3 - x_1 \leq 0$ to eliminate x_1 and produce the constraint $x_1 + x_3 \leq -2$

4. Add the constraint $-x_1 - x_3 \leq -3$ to eliminate x_3 and produce the constraint $x_1 - x_1 \leq -5$

Example (9):

Let us now examine how repeats occur.

Lemma 3.1.6 *If the network G has a path of type (a) then it has a path of type (a) in which no edge is used more than twice.*

Proof: Assume that there is a path of type (a) (say C), in which an edge is used more than twice, say three times. Note that a path of type (a) is equivalent to a cycle of negative weight that can be reduced to a single gray edge. This means that one of its defining vertices is used three times. We will argue that the existence of such a vertex (say x_i) leads to a contradiction.

Observe that the negative gray cycle C can be subdivided into sub-cycles each of which uses x_i only once, for convenience we will count the first and last verticies of a cycle as the same occurrence. Each sub-cycle is simply the part of the main cycle between, and including, two occurrences of the vertex x_i.

Because of how cycles are defined each of these sub-cycles can be reduced to the equivalent of a single white, black or gray edge from x_i to itself. Those sub-cycles which can be reduced to black edges shall be referred to as black sub-cycles. White and gray sub-cycles are defined similarly.

As stated before, when checking for unsatisfiability it suffices to search for negative cycles which can be reduced to a single gray edge. Thus, as in Lemma 3.1.2, each white sub-cycle can be paired with a black sub-cycle to produce a gray sub-cycle that uses x_i twice. As edge reductions are associative we can reduce the white cycle to a

single white edge and the black cycle to a single black edge. Thus the entirety can be reduced to a single gray edge. After each white sub-cycle is paired with a black sub-cycle there can be no remaining white or black sub-cycles. Otherwise the whole cycle would not reduce to a single gray edge. This is because the only reductions which produce a valid gray edge are two gray edges and a white edge with a black edge.

Thus the main cycle is equivalent to several gray sub-cycles each of which uses x_i no more than twice. Since the main cycle has a negative weight, at least one of these sub-cycles must also have negative weight. Thus we have found a gray cycle of negative weight which uses x_i at most twice.

□ If **U** is equal to the infeasible system of constraints

$$\begin{aligned} x_1 - x_2 &\leq -3 \\ -x_1 + x_4 &\leq 1 \\ -x_1 - x_4 &\leq 1 \\ x_2 + x_3 &\leq 1 \\ x_2 - x_3 &\leq 1 \\ x_5 - x_1 &\leq 0 \\ x_1 - x_5 &\leq 1 \end{aligned} \tag{3.2}$$

Then **G** (except for the node x_0) is shown in Figure 3.2.

In Figure 3.2 the negative cycle

$$x_1 \overset{-3}{\leftarrow} x_2 \overset{1}{\square} x_3 \overset{1}{\rightarrow} x_2 \overset{-3}{\rightarrow} x_1 \overset{0}{\rightarrow} x_5 \overset{1}{\rightarrow} x_1 \overset{1}{\rightarrow} x_4 \overset{1}{\blacksquare} x_1$$

uses x_1 three times. However it can be divided into the gray sub-cycle,

$$x_1 \overset{0}{\rightarrow} x_5 \overset{1}{\rightarrow} x_1$$

the white sub-cycle,

$$x_1 \overset{-3}{\leftarrow} x_2 \overset{1}{\square} x_3 \overset{1}{\rightarrow} x_2 \overset{-3}{\rightarrow} x_1$$

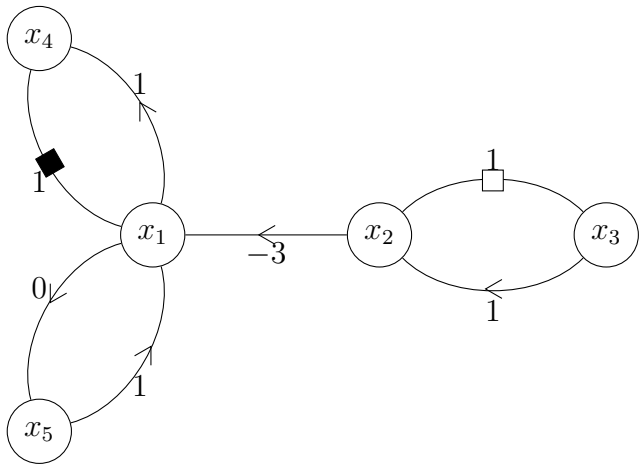

Figure 3.2: Example Constraint Network (without node x_0)

and the black sub-cycle

$$x_1 \overset{1}{\rightarrow} x_4 \overset{1}{\blacksquare} x_1$$

We can then combine the white and black cycles to form the gray cycle

$$x_1 \overset{-3}{\leftarrow} x_2 \overset{1}{\square} x_3 \overset{1}{\rightarrow} x_2 \overset{-3}{\rightarrow} x_1 \overset{1}{\rightarrow} x_4 \overset{1}{\blacksquare} x_1,$$

which is a negative cycle that uses x_1 twice. Also note that the edge $x_2 \overset{-3}{\rightarrow} x_1$ is used twice in this cycle and thus the constraint $x_1 - x_2 \leq -3$ appears twice in the corresponding sum of constraints. *Example (10):*

Lemma 3.1.7 *If* $\mathbf{X}$ *is empty then a path of type* (a) *exists.*

Proof: From the previous lemmata any inconsistency can be expressed as a series of constraints that can be added to get a constraint of the form $x_i - x_i \leq c < 0$ and such that at every point in the addition sequence a UTVPI constraint is maintained. Since edge reductions in $\mathbf{G}$ correspond to exactly such additions, such a series of constraints corresponds to a series of edges which can be reduced to a single gray loop of negative weight, namely a path of type (a). □

With the preceding lemmata proved, we now return to Theorem 1.

Theorem 1 1 *Either* $\mathbf{X}$ *is non-empty or (mutually exclusively) there exists one of the following paths in* $\mathbf{G}$*:*

(a) A path from a vertex x_i to itself that can be reduced to a single gray edge of negative weight. This will be referred to as a path of type (a).

(b) A path of negative weight from a vertex x_i to itself that consists of two sub-paths from x_i to itself, one which can be reduced to a single white edge and one which can be reduced to a single black edge. This will be referred to as a path of type (b).

Proof: As shown by the preceding lemmata, if a path of type (b) exists then a path of type (a) exists (Lemma 3.1.2). Thus it is enough to consider only paths of type (a). Similarly we showed that if $\mathbf{X}$ is empty then a path of type (a), or type (b), exists (Lemma 3.1.7) and that if a path of type (a), or (b), exists then $\mathbf{X}$ is empty (Lemma 3.1.3). Thus the theorem holds. □

3.2 Integer Feasibility

Theorem 3.1.1 applies when searching for a linear solution to the system of UTVPI constraints. We now present an analogous theorem that is useful, when searching for lattice point solutions.

Let $\mathbf{G}' = \langle V, E', \mathbf{c} \rangle$ denote the constraint network corresponding to the system of UTVPI constraints, $\mathbf{U}'$, formed by the addition of new absolute constraints to $\mathbf{U}$. A constraint $x_i \leq c_i$ where $c_i \in \mathbb{Z}$, is added to $\mathbf{U}'$, if

(1) There are two constraints in $\mathbf{U}$, which can be added to produce either $x_i + x_i \leq 2 \cdot c_i + 1$ or $x_i + x_i \leq 2 \cdot c_i$ (type (1)), or

(2) The addition of the constraint $-x_i \leq -c_i - 1$ causes $\mathbf{U}'$ to become infeasible (type (2)).

Constraints of the form $-x_i \leq c_i$ are added in similar fashion.

Let $\mathbf{X}'$ denote the set of feasible solutions to $\mathbf{U}'$.

Theorem 3.2.1 *Either the constraint system* $\mathbf{U}$ *encloses a lattice point or (mutually exclusively),* $\mathbf{G}'$ *contains a path from a vertex* x_i *to itself that can be reduced to a single gray edge of negative weight.*

To show this, we will first need to prove a number of lemmata which will build up to the desired result.

Lemma 3.2.1 *If* $\mathbf{G}'$ *contains a path from a vertex* x_i *to itself that can be reduced to a single gray edge of negative weight then* $\mathbf{X}'$ *is empty.*

Proof: If such a path exists, then it is a path of type (a) as described before. Thus from Lemma 3.1.3, we know that $\mathbf{X}'$ is empty. □

Lemma 3.2.2 *If* $\mathbf{u}$ *is a lattice point in* $\mathbf{X}$*, then* $\mathbf{u}$ *is a lattice point in* $\mathbf{X}'$ *as well.*

Proof: First observe that all the lattice points in $\mathbf{X}'$ are lattice points in $\mathbf{X}$, since $\mathbf{U}'$ is constructed by adding constraints to $\mathbf{U}$. Suppose that the constraint $x_j \leq c_j$ is in $\mathbf{U}'$, but not in $\mathbf{U}$. By the construction of $\mathbf{U}'$, either the constraint $x_j + x_j \leq 2 \cdot c_j + 1$ was deduceable from the constraints in $\mathbf{U}$ or the addition of $-x_j \leq -c_j - 1$ caused the system $(\mathbf{U}')$ to become infeasible. In the first case, any lattice point satisfying the original constraints must also satisfy $x_j + x_j \leq 2 \cdot c_j + 1$. Thus this lattice point must also satisfy $x_j \leq \lfloor \frac{2 \cdot c_j + 1}{2} \rfloor = c_j$. In the second case we have that no solution to $\mathbf{U}$ satisfies $-x_j \leq -c_j - 1$. Thus any lattice point satisfying the original constraints must also satisfy $-x_j \geq -c_j$, which is equivalent to $x_j \leq c_j$. Thus any lattice points in $\mathbf{X}$ must also be in $\mathbf{X}'$. □

Lemma 3.2.3 *If* $\mathbf{G}'$ *has a negative gray cycle, then* $\mathbf{X}$ *contains no lattice points.*

Proof: From Lemma 3.2.1, we know that if $\mathbf{G}'$ has a negative gray cycle, then $\mathbf{X}'$ is empty and so contains no lattice points. Thus, by Lemma 3.2.2, $\mathbf{X}$ cannot contain any lattice points either. □

We have now shown one direction of the implication. The following lemmata will help us show the other direction. We recall the two inference rules used in [1] viz., the transitive and tightening rules [12]. The transitive rule is

$$\frac{a \cdot x_i + b \cdot x_j \leq c_{ij} \qquad\qquad -b \cdot x_j + b' \cdot x_k \leq c_{jk}}{a \cdot x_i + b' \cdot x_k \leq c_{ij} + c_{jk}}$$

and the tightening rule is

$$\frac{a \cdot x_i + b \cdot x_j \leq c_{ij} \qquad\qquad a \cdot x_i - b \cdot x_j \leq c'_{ij}}{a \cdot x_i \leq \lfloor \frac{c_{ij}+c'_{ij}}{2} \rfloor} \tag{3.3}$$

As shown in [1], both the above rules are lattice point preserving.

Lemma 3.2.4 *If* $\mathbf{X}$ *contains no lattice points then* $\mathbf{X}'$ *is empty.*

Proof: Let $\mathbf{U}''$ be the system of UTVPI constraints obtained by adding to $\mathbf{U}$, all the constraints obtained by repeated applications of the transitive and tightening inference rules of UTVPI constraints to the constraints in $\mathbf{U}$. The process of adding constraints stops, when no more constraints can be added to $\mathbf{U}''$. Let $\mathbf{X}''$ be the set of all solutions to $\mathbf{U}''$. Thus by construction if $\mathbf{X}$ has no lattice points $\mathbf{X}''$ is empty.

As stated previously, applications of the transitive inference rule correspond to edge reductions and so the constraints added in this fashion do not affect the linear feasibility of the system. Thus, we are only concerned with the constraints added through application of the tightening inference rule. We will now show that applications of the tightening rule correspond to the addition of absolute constraints, as described just before Theorem 3.2.1.

Observe that if the constraint $x_i \leq c$ is added in this way, then either $x_i + x_i \leq 2 \cdot c_i$ or $x_i + x_i \leq 2 \cdot c_i + 1$ is derivable from the original constraints. In both of these cases adding the constraint $-x_i \leq -c_i - 1$ would create an inconsistency in the system. Thus the constraint $x_i \leq c_i$ is a constraint of type (2) and is therefore added to $\mathbf{U}'$.

Thus every constraint added to $\mathbf{U}''$ through application of the tightening inference rule is added to $\mathbf{U}'$. This means that if a point $\mathbf{x}$ satisfies $\mathbf{U}'$ it also satisfies $\mathbf{U}''$. Thus $\mathbf{X}' \subseteq \mathbf{X}''$ and so if $\mathbf{X}$ contains no lattice points then $\mathbf{X}'$ is empty.

□

Lemma 3.2.5 *If* $\mathbf{X}$ *contains no lattice points then* $\mathbf{G}'$ *contains a path from a vertex* x_i *to itself that can be reduced to a single gray edge of negative weight.*

Proof: By Lemma 3.2.4, if $\mathbf{X}$ contains no lattice points then $\mathbf{X}'$ is empty. Thus, by Theorem 3.1.1, $\mathbf{G}'$ contains a path of type (a). This is exactly the type of path required. □

With the preceding lemmata proved, we now return to Theorem 2.

Theorem 2 1 *Either the constraint system* $\mathbf{U}$ *encloses a lattice point or (mutually exclusively)* $\mathbf{G}'$ *contains a path from a vertex* x_i *to itself that can be reduced to a single gray edge of negative weight.*

Proof: As shown in Lemma 3.2.3, if $\mathbf{G}'$ contains such a path then $\mathbf{X}$ contains no lattice points. Also we have that, by Lemma 3.2.5, if $\mathbf{X}$ contains no lattice points then $\mathbf{G}'$ contains precisely such a path. □

Theorem 3.2.1 is used in sections 7 and 8.

Chapter 4

Motivation and Related Work

Existing algorithms for deciding UTVPI systems require additional time and space (asymptotically), if they are also required to produce certificates which validate their output. For instance, the best known algorithm to date, which decides UTVPI systems ([1]), runs in $O(m \cdot n)$ time and $O(m + n)$ space, if all that is asked is whether a given UTVPI system is feasible. However, if it is also required to produce a model for a system, then the time and space complexities of their algorithm increase to $O(m \cdot n + n^2 \cdot \log n)$ and $O(n^2)$ respectively.

The first known decision procedure for UTVPI constraints is detailed in [13]. Their algorithm processed a set of UTVPI constraints with the goal of finding its transitive and tightening closure. Such a closure essentially is a finite representation of all possible UTVPI constraints that can be inferred from the input set of constraints (also see[14]). In other words, it found all deductions possible from any of its initial constraints, including rounded constraints intended to force integral solutions, and checked to see if the system of constraints thus generated was feasible by virtue of having no contradictions. This algorithm, which is not certifying, runs in time $O(m \cdot n^2)$ using $O(n^2)$ space. The procedure in [13] was improved in [15] from an ease-of-implementation standpoint by combining the transitive and tightening closures into a single step. However, the asymptotic complexity did not improve and the new algorithm did not provide provide certificates either.

A rather different approach was used in [11] to decide UTVPI systems while

also producing a model. Their algorithm uses Fourier-Motzkin elimination [16] to project the polyhedron representation of a system of UTVPI constraints down to a single variable in a solution-preserving manner, thereby determining bounds for that variable. The algorithm then works in reverse order to assign values to the rest of the variables. While producing a model for the system, this algorithm takes $O(n^3)$ time and $O(n^2)$ space.

The algorithm in [1] as mentioned earlier is the best known algorithm to date for deciding UTVPI systems. We will elaborate on their method, in order to provide the proper background to contrast our procedures.

Their algorithm begins by converting each constraint to a pair of difference constraints with positive and negative versions of each involved variable. For instance, a sum constraint $x_i + x_j \leq c_{ij}$ is converted into the following difference constraint pair: $x_i^+ - x_j^+ \leq c_{ij}$ and $x_j^- - x_i^- \leq c_{ij}$. Once all constraints are thus converted, we represent the converted constraint system by a constraint network as detailed in [8]. For instance, the constraint $x_j^- - x_i^- \leq c_{ij}$ results in an edge $x_j^- \overset{c_{ij}}{\leftarrow} x_i^-$. The resulting edges are then tightened by converting edges of the form $x_i \overset{c_{ii}}{\leftarrow} x_i$ where c_{ij} is odd to $x_i \overset{c_{ij}-1}{\leftarrow} x_i$ in order to ensure integral solutions. A negative cycle detection subroutine (such as the Bellman-Ford algorithm) then determines whether the system is satisfiable.

We note that in order for the algorithm in [1] to produce a model, it must compute the transitive and tightening closure of the original constraint system, *even* when such a set of constraints is known to be satisfiable. Indeed, it uses a procedure similar to the one in [13] and [15] to find bounds for all variables and assign values to them. A naive implementation of this algorithm runs in $O(n^3)$ time and uses $O(n^2)$ space. Utilizing Johnson's algorithm [8] for the transitive closure, resource complexity can be improved only to $O(m \cdot n + n^2 \cdot \log n)$ time and $O(n^2)$ space. However, even the improved algorithm is more expensive (asymptotically) to the ideal $O(m \cdot n)$ time and $O(m + n)$ space complexity of the non-certifying decision algorithm.

Recently, there has been some work on *incremental* satisfiability of UTVPI constraints. For instance, [17] describes an algorithm for incremental satisfiability check-

ing that runs in $O(m+n\cdot\log n)$ time. Incremental algorithms are extremely important from the perspective of SAT Modulo Theories [18].

Chapter 5

Linear feasibility Algortitm

Algorithm 5.0.1 represents our approach for checking linear feasibility in a UTVPI constraint system. This algorithm is a relaxation-based approach for traversing the constraint network corresponding to the constraint system. It returns either a set of of valid distance labels (which is a feasible solution), or a certificate of infeasibility of the system.

UTVPI-LINEAR-FEAS (system **U** of UTVPI constraints)

1: $G \leftarrow$ CONSTRUCT-NETWORK(**U**) [as described in Section 2]
2: $R \leftarrow$ RELAX-NETWORK(G)
3: **if** (R is a set of distance labels) **then**
4: Construct assignment R' where where each $x_i = \frac{\overset{\square}{d_i} - \overset{\blacksquare}{d_i}}{2}$.
5: **return** R' as "yes" certificate of valid linear solution.
6: **else**
7: **return** R as "no" certificate of linear infeasibility.
8: **end if**

Algorithm 5.0.1: Algorithm for checking linear feasibility

The algorithm maintains four distance labels for each vertex, x_i, as follows:

1. $\overrightarrow{d_i}$ - This label corresponds to a path, which reduces to an edge of type $x_0 \xrightarrow{c} x_i$, i.e., the shortest gray path from x_0 to x_i.

2. $\overleftarrow{d_i}$ - This label corresponds to a path, which reduces to an edge of type $x_i \xrightarrow{c} x_0$,

RELAX-NETWORK (network G as adjacency list)

1: Let $L[t, x]$ denote the last edge of the current shortest path of type t from node x_0 to node x. {There are 4 $L[\ ]$ values associated with each vertex, for a total space requirement of $O(n)$.}
2: Let $D[t, x]$ denote the set of distance labels corresponding to node x and edge type t. {As an example, $D[\ \square\ , x_i] = \overset{\square}{d}_i$. Observe that there are 4 $D[\ ]$ values associated with each vertex, for a total space requirement of $O(n)$.}
3: **for** (each node x_i in G) **do**
4: $\overset{\square}{d}_i\ , \overset{\blacksquare}{d}_i\ , \overset{\leftarrow}{d}_i, \vec{d}_i = n \cdot C$
5: **for** $t \in \{\ \square\ ,\ \blacksquare\ , \leftarrow, \rightarrow\}$ **do**
6: $L[t, x_i] = x_0\ t\ x_i$
7: **end for**
8: **end for**
9: $\overset{\square}{d}_0\ , \overset{\blacksquare}{d}_0\ , \overset{\leftarrow}{d}_0, \vec{d}_0 = 0$
10: **for** $(r = 1$ **to** $(2 \cdot n + 1))$ **do**
11: **for** each edge e in G **do**
12: RELAX-EDGE(e, D, L)
13: **end for**
14: **end for**
15: **for** (every edge $x_i \overset{c_{ij}}{\square} x_j$) **do**
16: **if** $(\overset{\square}{d}_i > \overset{\leftarrow}{d}_j + c_{ij})$ **then**
17: Backtrack along $L[\ \square\ , x_i]$ to **return** negative cycle
18: **end if**
19: **end for**
20: [the other edge cases are analogous according to the relaxation rules]
21: **return** (set of distance labels, D).

Algorithm 5.0.2: RELAX-NETWORK

RELAX-EDGE (edge e, distance labels D, and predecessor structure L)

1: **if** (e is an edge of type $x_i \overset{c_{ij}}{\square} x_j$) **then**
2: $\quad \overset{\square}{d}_i \leftarrow \min(\overset{\square}{d}_i, \overset{\leftarrow}{d}_j + c_{ij})$
3: $\quad$ **if** ($\overset{\square}{d}_i$ changed) **then**
4: $\quad\quad L[\square, x_i] = x_i \overset{c_{ij}}{\square} x_j$
5: $\quad$ **end if**
6: $\quad \vec{d}_i \leftarrow \min(\vec{d}_i, \overset{\blacksquare}{d}_j + c_{ij})$
7: $\quad$ **if** ($\vec{d}_i$ changed) **then**
8: $\quad\quad L[\rightarrow, x_i] = x_i \overset{c_{ij}}{\square} x_j$
9: $\quad$ **end if**
10: **end if**
11: [the other edge cases are analogous according to the relaxation rules in Figure 5]

Algorithm 5.0.3: RELAX-EDGE

i.e., the shortest gray path from x_i to x_0.

3. $\overset{\square}{d}_i$ - This label corresponds to a path, which reduces to an edge of type $x_0 \overset{c}{\square} x_i$, i.e., the shortest white path from x_0 to x_i.

4. $\overset{\blacksquare}{d}_i$ - This label corresponds to a path, which reduces to an edge of type $x_0 \overset{c}{\blacksquare} x_i$, i.e., the shortest black path from x_0 to x_i.

Three of these labels represent shortest path distances (of different types) from vertex x_0 to vertex x_i. The fourth label represents a type of shortest path distance from vertex x_i to vertex x_0.

These distance labels will be maintained so that the following four relationships will always hold.

1. $x_0 + x_i \leq \overset{\square}{d}_i$
2. $x_0 - x_i \leq \overset{\leftarrow}{d}_i$
3. $-x_0 + x_i \leq \vec{d}_i$
4. $-x_0 - x_i \leq \overset{\blacksquare}{d}_i$

This will be done by ensuring that after the k^{th} iteration of the main **for** loop of Algorithm 5.0.1 (Lines 7 through 15), three of the labels represent the lengths of the shortest valid k-paths from x_0 to x_i and one of the labels represents the length of the shortest valid k-path from x_i to x_0.

The label $\overset{\square}{d}_i$ is the length of the shortest white k-path from x_0 to x_i. This path reduces to the edge $x_0 \overset{\overset{\square}{d}_i}{\square} x_i$, which corresponds to the constraint $x_0 + x_i \leq \overset{\square}{d}_i$. Therefore the relation $x_0 + x_i \leq \overset{\square}{d}_i$ holds. *Example (11):*

The assignment of 0 to x_0, permits us to construct ever-tightening bounds on x_i.

Algorithm 5.0.1 needs to account for the four valid edge reductions described previously. These reductions correspond to the additions of pairs of UTVPI constraints that produce UTVPI constraints. The relaxation procedure runs $(2 \cdot n + 1)$ times. The different edge types also necessitate a more complex backtracking method once a negative cycle is found. For each vertex, we store four predecessor nodes, one node for each path type. For each path type it is also necessary to store the edge type used to get to the current vertex. This ensures that the backtracking procedure knows which path type to follow from the current vertex.

When we relax an edge in the network, we need to adjust the four distance labels as per the following relaxation rules:

edge	Changes to distance labels.			
$x_i \overset{c_{ij}}{\rightarrow} x_j$	$\overset{\leftarrow}{d}_i = \overset{\leftarrow}{d}_j + c_{ij}$	$\overset{\blacksquare}{d}_i = \overset{\blacksquare}{d}_j + c_{ij}$	$\vec{d}_j = \vec{d}_i + c_{ij}$	$\overset{\square}{d}_j = \overset{\square}{d}_i + c_{ij}$
$x_i \overset{c_{ij}}{\square} x_j$	$\overset{\square}{d}_i = \overset{\leftarrow}{d}_j + c_{ij}$	$\vec{d}_i = \overset{\blacksquare}{d}_j + c_{ij}$	$\overset{\square}{d}_j = \overset{\leftarrow}{d}_i + c_{ij}$	$\vec{d}_j = \overset{\blacksquare}{d}_i + c_{ij}$
$x_i \overset{c_{ij}}{\blacksquare} x_j$	$\overset{\blacksquare}{d}_i = \vec{d}_j + c_{ij}$	$\overset{\leftarrow}{d}_i = \overset{\square}{d}_j + c_{ij}$	$\overset{\blacksquare}{d}_j = \vec{d}_i + c_{ij}$	$\overset{\leftarrow}{d}_j = \overset{\square}{d}_i + c_{ij}$

Table 5.1: Relaxation Rules

The above table has to be interpreted in the following manner: Consider the edge type $x_i \overset{c_{ij}}{\rightarrow} x_j$. The corresponding row of the relaxation table indicates the following:

1. if $\overset{\leftarrow}{d}_i > \overset{\leftarrow}{d}_j + c_{ij}$, then $\overset{\leftarrow}{d}_i$ is assigned a value of $\overset{\leftarrow}{d}_j + c_{ij}$,

2. if $\overset{\blacksquare}{d}_i > \overset{\blacksquare}{d}_j + c_{ij}$, then $\overset{\blacksquare}{d}_i$ is assigned a value of $\overset{\blacksquare}{d}_j + c_{ij}$,

3. if $\vec{d}_j > \vec{d}_i + c_{ij}$, then $\vec{d}_j$ is assigned a value of $\overset{\leftarrow}{d}_i + c_{ij}$,

4. if $\overset{\square}{d}_j > \overset{\square}{d}_i + c_{ij}$, then $\overset{\square}{d}_j$ is assigned a value of $\overset{\square}{d}_i + c_{ij}$.

We have that the values $\overset{\leftarrow}{d}_j$, $\overset{\blacksquare}{d}_j$, $\vec{d}_i$, $\overset{\square}{d}_i$ do not change as a result of relaxing the edge $x_i \overset{c_{ij}}{\rightarrow} x_j$. Thus, the order the distance labels are updated does not matter as changing one does not affect the changes made to the others. The remaining rows should be interpreted in a similar fashion.

The relaxation rules ensure that the distance labels never increase in value.

Remark 5.0.1 *The relaxation rules maintain valid bounds for the constraints. For example, consider what happens when we relax the edge* $x_i \overset{c_{ij}}{\square} x_j$*, which corresponds to the constraint* $x_i + x_j \leq c_{ij}$*. We know from the definition of the four distance labels that* $x_0 - x_i \leq \overset{\leftarrow}{d}_i$*. Thus by adding these two constraints we get* $x_0 + x_j \leq \overset{\leftarrow}{d}_i + c_{ij}$*, which is a valid constraint. Hence,* $\overset{\leftarrow}{d}_i + c_{ij}$ *is a valid value for* $\overset{\square}{d}_j$*. Thus, if* $\overset{\leftarrow}{d}_i + c_{ij} < \overset{\square}{d}_j$ *we can set* $\overset{\square}{d}_j = \overset{\leftarrow}{d}_i + c_{ij}$*. The cases for the other distance labels and edge types can be explained in similar fashion.*

5.1 Resource Analysis

5.1.1 Initialization

This stage consists of converting the constraints into a constraint network. Since the network is stored as an adjacency list, adding each constraint as an edge takes $\mathcal{O}(1)$ time. Thus this entire conversion procedure takes $\mathcal{O}(m+n)$ time and $\mathcal{O}(m+n)$ space.

Finding C, the largest absolute edge weight, requires a search through all the edges and thus takes $\mathcal{O}(m)$ time and runs in $\mathcal{O}(1)$ space.

Adding the vertex x_0 takes constant time.

Adding the $4 \cdot n$ appropriate edges, 4 to every other vertex, takes $\mathcal{O}(n)$ time and space. Adding the absolute constraints takes $\mathcal{O}(m)$ time to locate all of the absolute constraints and then $\mathcal{O}(n)$ time to change the appropriate edge weights. Thus, this part of the initialization stage takes $\mathcal{O}(m+n)$ time and $\mathcal{O}(1)$ space.

Considering the dominant resources in this stage, the initialization process as a whole takes $\mathcal{O}(m+n)$ time and $\mathcal{O}(m+n)$ space.

5.1.2 Checking for Linear Feasibility

This stage consists of $(2 \cdot n + 1)$ rounds of edge relaxations. In each round $\mathcal{O}(m)$ edges are relaxed. Thus this stage runs in $\mathcal{O}(m \cdot n)$ time and $\mathcal{O}(m+n)$ space.

5.1.3 Producing a Rational Solution

From the set of distance labels we compute $\frac{(\overset{\square}{d_i} - \overset{\blacksquare}{d_i})}{2}$ for each x_i, thus taking $\mathcal{O}(n)$ time and space.

5.1.4 Producing a certificate of infeasibility

Once a contradiction is found we backtrack along the structure L to obtain a negative gray cycle. The details of this backtracking are described in the next subsection. This step takes $\mathcal{O}(n)$ time and space.

5.1.5 Overall Analysis

It is thus clear that Algorithm 5.0.1 runs in $\mathcal{O}(m \cdot n)$ time and $\mathcal{O}(m+n)$ space.

Chapter 6

Correctness of the Linear Algorithm

In Lemma 3.1.1, we showed that edge reductions are associative. This property allows paths which can each be reduced to a single edge to be combined. Algorithm 5.0.1 invokes Algorithm 5.0.2, which in turn makes multiple calls to a relaxation procedure (Algorithm 5.0.3) to detect cycles which can be reduced to a single gray edge of negative weight. The existence of such a cycle means that there exist constraints that can be added to produce a constraint of the form $x_i - x_i \leq c_{ii}$ where $c_{ii} < 0$, i.e., a contradiction.

We first show that restricting the traversal to $(2 \cdot n + 1)$ iterations of the edge relaxation procedure is sufficient to establish the presence of a negative gray cycle, if one exists.

Lemma 6.0.1 *At the end of the relaxation procedure the distance labels represent the lengths of the appropriate shortest paths, with* $(2 \cdot n + 1)$ *edges or fewer.*

Proof: We will show that after each relaxation involving x_i, $\overset{\square}{d}_i$ represents the length of a white path from x_0 to x_i, $\overset{\blacksquare}{d}_i$ represents the length of a black path from x_0 to x_i, $\vec{d}_i$ represents the length of a gray path from x_0 to x_i, and $\overset{\leftarrow}{d}_i$ represents the length of a gray path from x_i to x_0.

At the beginning of the traversal, we have that $\overset{\square}{d}_i$ is set to the weight of the white

edge from x_0 to x_i and is thus the length of a white path. Similarly $\overset{\blacksquare}{d}_i$ is the weight of the black edge from x_0 to x_i, $\overset{\leftarrow}{d}_i$ starts as the length of the gray edge from x_i to x_0, and $\vec{d}_i$ is the length of the gray edge from x_0 to x_i.

Thus at the start of the relaxation procedure, these four values represent the length of the shortest 1-path (path having at most one edge) of the appropriate type.

Now we assume that this holds at the beginning of any relaxation. Furthermore, assume that the edge being relaxed is of the form $x_i + x_j \leq c_{ij}$ (white edge). Observe that if no distance labels were modified, then the values still represent the weights of the appropriate paths. If $\overset{\square}{d}_i$ is modified, then we have that $\overset{\square}{d}_i$ becomes $c_{ij} + \overset{\leftarrow}{d}_j$ (see Figure 5). Since $\overset{\leftarrow}{d}_j$ represents a gray path from x_j to x_0, and since edge reductions are associative, that path, combined with the white edge from x_j to x_i, can be reduced to a single white edge from x_0 to x_i with weight $\overset{\square}{d}_i = c_{ij} + \overset{\leftarrow}{d}_j$. Thus this new path from x_0 to x_i is a white path.

An analogous argument holds for each type of edge relaxed and for each of the values $\overset{\blacksquare}{d}_i$, $\overset{\leftarrow}{d}_i$, and $\vec{d}_i$ using the various reduction and relaxation rules.

Assume that at the start of the k^{th} round of relaxation, each distance label represents the length of the shortest path of the appropriate type with k or fewer edges. During the k^{th} round of relaxation, every edge in the network is relaxed. Thus, at the end of the k^{th} round all possible $(k+1)^{th}$ edges have been considered. Since the appropriate distance decrease each time a shorter path is found, we have that at the end of the k^{th} round of relaxation the distance labels all represent the length of the shortest path of the appropriate type with $(k+1)$ or fewer edges.

Thus at the end of the relaxation procedure, after $(2 \cdot n + 1)$ rounds we have that the distance labels represent the length of the shortest paths (of the appropriate type) in the network with $(2 \cdot n + 1)$ edges or fewer. □

Lemma 6.0.2 *$(2 \cdot n + 1)$ invocations of Algorithm 5.0.3 (the edge relaxation procedure) are sufficient to establish the presence or absence of negative gray cycles.*

Proof:

We prove the contrapositive, i.e., we will show that if there are no negative gray cycles, then at most $(2 \cdot n + 1)$ invocations of Algorithm 5.0.3 will cause the distance labels to converge to their final values.

Note that if there are no negative gray cycles in the constraint network, then the shortest path of any type from x_0 to any given vertex x_i cannot use any vertex, more than twice. Thus this path cannot consist of more than $(2 \cdot n + 1)$ edges.

Therefore, by the preceding lemma, after $(2 \cdot n + 1)$ iterations of the relaxation procedure the distance labels do in fact correspond to the lengths of the actual shortest paths and so will not decrease with subsequent relaxations. □

Thus $(2 \cdot n + 1)$ runs of the relaxation step are sufficient to find negative gray cycles. We also need to show that adding the point x_0 and the corresponding edges does not cause the system to become infeasible. We will do this by showing that adding these edges does not cause the introduction of negative cycles into the constraint network.

Lemma 6.0.3 *The introduction of edges from x_0 of weight $n \cdot C$ does not introduce any negative cycles, if none were already present.*

Proof: Assume that adding these edges causes the creation of a new negative cycle. Since no negative cycles existed previously in the network, all the created negative cycles must use the point x_0. From Lemma 3.1.6, we know that at least one newly added negative cycle uses each vertex no more than twice. Thus it uses no more than $2 \cdot n$ of the original edges. Thus the weight of the part of the cycle in the original network has a weight no less than $-2 \cdot n \cdot C$, as no single edge has weight less than $-C$. However, since the cycle uses the vertex x_0, it must use at least two of the newly added edges. Thus the total weight of the cycle is no less than $n \cdot C + n \cdot C - 2 \cdot n \cdot C = 0$, contradicting the assumption that any negative cycles were added. □

Lemma 6.0.4 *The invariants $\overset{\blacksquare}{d}_i = \overleftarrow{d}_i$ and $\overset{\square}{d}_i = \overrightarrow{d}_i$ for each vertex x_i, are maintained through each invocation of Algorithm 5.0.3.*

Proof: This will be shown through induction on the number of relaxations.

Before the first relaxation, we have that for each x_i either $\overset{\square}{d}_i = \vec{d}_i = n \cdot C$ or, if the list of constraints included the absolute constraint $x_i \leq c_i$, $\overset{\square}{d}_i = \vec{d}_i = c_i$. Similarly, for each x_i either $\overset{\blacksquare}{d}_i = \overset{\leftarrow}{d}_i = n \cdot C$ or, if the list of constraints included the absolute constraint $-x_i \leq c_i$, then $\overset{\blacksquare}{d}_i = \overset{\leftarrow}{d}_i = c_i$.

Assume that the invariants are maintained at the beginning of the k^{th} relaxation. If the edge being relaxed is of the form $x_i + x_j \leq c_{ij}$ then we have that, either the distance labels are unchanged or that some of them are changed. If no values are changed then the invariants still hold. If $\overset{\square}{d}_i$ is changed, then from Figure 5 we have that, before the distance labels are updated, $\vec{d}_i = \overset{\square}{d}_i > c_{ij} + \overset{\leftarrow}{d}_j = c_{ij} + \overset{\blacksquare}{d}_j$.

From Figure 5 we know that relaxing the edge $x_i \overset{c_{ij}}{\square} x_j$ does not change the values of $\overset{\leftarrow}{d}_j$ or $\overset{\blacksquare}{d}_j$. Thus after the distance labels are updated we have that $\overset{\square}{d}_i = c_{ij} + \overset{\leftarrow}{d}_j = c_{ij} + \overset{\blacksquare}{d}_j = \vec{d}_i$, and so the invariant still holds.

An analogous argument shows that these invariants hold, regardless of the edge being relaxed. Similar arguments also show that the invariant $\overset{\blacksquare}{d}_i = \overset{\leftarrow}{d}_i$ is maintained over all edge relaxations. □

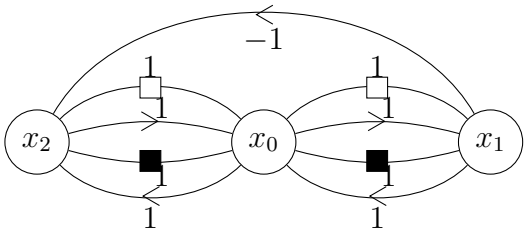

Figure 6.1: Example Constraint Network

In Figure 6.1, we have that initially $\vec{d}_2 = \overset{\square}{d}_2 = 1$ and that $\vec{d}_1 = \overset{\square}{d}_1 = 1$. After relaxing the edge $x_1 \overset{-1}{\rightarrow} x_2$ we have that, from Figure 5, $\overset{\square}{d}_2 = \overset{\square}{d}_1 + (-1) = 0$ and $\vec{d}_2 = \vec{d}_1 + (-1) = 0$. Thus after the relaxation we still have that $\overset{\square}{d}_2 = \vec{d}_2$. *Example (12):*

From this point onwards, we assume that no negative gray cycles were discovered

by Algorithm 5.0.2 and that it returns a set of distance labels. We will first show that there exists a rational solution; we will then show that our traversal determines the bounds for such a solution.

Lemma 6.0.5 *If Algorithm 5.0.2 returns a set of distance labels, then we have that for each each* x_i, $\overset{\square}{d}_i \geq -\overset{\blacksquare}{d}_i$.

Proof: We will show that after each relaxation involving x_i, $\overset{\square}{d}_i$ represents the length of a white path from x_0 to x_i and that $\overset{\blacksquare}{d}_i$ represents the length of a black path from x_0 to x_i.

At the beginning of the traversal, we have that $\overset{\square}{d}_i$ is set to the weight of the white edge from x_0 to x_i and is thus the length of a white path. Similarly $\overset{\blacksquare}{d}_i$ is the weight of the black edge from x_0 to x_i, $\overset{\leftarrow}{d}_i$ starts as the length of the gray edge from x_i to x_0, and $\vec{d}_i$ is the length of the gray edge from x_0 to x_i.

Now we assume that this holds at the beginning of any relaxation. If the edge being relaxed is of the form $x_i + x_j \leq c_{ij}$, then we have that if no distance labels were modified then the values still represent the weights of the appropriate paths. If $\overset{\square}{d}_i$ is modified then we have that $\overset{\square}{d}_i$ becomes $c_{ij} + \overset{\leftarrow}{d}_j$. Since $\overset{\leftarrow}{d}_j$ represents a gray path from x_j to x_0, and since edge reductions are associative, that path combined with the white edge from x_j to x_i, can be reduced to a single white edge from x_0 to x_i with weight $\overset{\square}{d}_i = c_{ij} + \overset{\leftarrow}{d}_j$. Thus this new path from x_0 to x_i is a white path.

An analogous argument holds for each type of edge relaxed and for each of the values $\overset{\blacksquare}{d}_i$, $\overset{\leftarrow}{d}_i$, and $\vec{d}_i$ using the various reduction and relaxation rules.

Thus at each stage of the relaxation process we have that $\overset{\square}{d}_i$ and $\overset{\blacksquare}{d}_i$ are the weights of paths of the appropriate type.

If $\overset{\square}{d}_i < -\overset{\blacksquare}{d}_i$ then there would be a gray cycle though x_0 and x_i, specifically the one formed by the white path of weight $\overset{\square}{d}_i$ from x_0 to x_i and the black path of weight $\overset{\blacksquare}{d}_i$ from x_0 to x_i. This is a proper cycle because each of the previously described parts can be reduced to a white edge of weight $\overset{\square}{d}_i$ from x_0 to x_i and a black edge of

weight $\blacksquare{d_i}$ from x_0 to x_i. These two edges, and thus the whole cycle, can be reduced to a single gray edge from x_0 to itself of weight $\square{d_i} + \blacksquare{d_i} = \square{d_i} - (-\blacksquare{d_i}) < 0$.

This, however, is a negative cycle and so, as shown previously, there must be a cycle which uses no vertex more than twice. Such a negative cycle would have been detected and returned by the relaxation method and so this situation cannot occur. Thus after running Algorithm 5.0.2 (the relaxation procedure), we have that for each x_i, $\square{d_i} \geq -\blacksquare{d_i}$. □

This means that, after running the relaxation procedure, for each x_i, the interval $[-\blacksquare{d_i}, \square{d_i}]$ is non-empty. We will now show that if each x_i is taken to be the center point of this interval then a valid rational solution to the system of constraints is produced. We will do this by showing that for variable x_i all absolute constraints are satisfied. Likewise, for each pair of variables x_i and x_j all two variable constraints are satisfied.

Lemma 6.0.6 *If Algorithm 5.0.2 returns a set of distance labels, then for each constraint of the form $x_i + x_j \leq c_{ij}$ we have that $\square{d_i} \leq c_{ij} + \blacksquare{d_j}$.*

Proof: Observe that prior to returning the distance labels, Algorithm 5.0.2 checks to see if any additional relaxations decrease any values in our table. We thus know at this point that no relaxation modifies the value of $\square{d_i}$. Relaxing the edge corresponding to the constraint $x_i + x_j \leq c_{ij}$, a white edge, would change the value of $\square{d_i}$ to $c_{ij} + \blacksquare{d_j}$ if the current value of $\square{d_i}$ is larger that $c_{ij} + \blacksquare{d_j}$. However, as the value does not change we must have that $\square{d_i} \leq c_{ij} + \blacksquare{d_j}$. □

Analogous proofs can be used to show that for each constraint $-x_i - x_j \leq c_{ij}$ we have that $\blacksquare{d_i} \leq c_{ij} + \square{d_j}$, for each constraint $x_i - x_j \leq c_{ij}$ we have that $\square{d_i} \leq c_{ij} + \square{d_j}$, and for each constraint $-x_i + x_j \leq c_{ij}$ we have that $\blacksquare{d_i} \leq c_{ij} + \blacksquare{d_j}$.

We can use these properties to show that the previously described solution is indeed valid.

Lemma 6.0.7 *If Algorithm 5.0.2 returns a set of distance labels, then assigning each x_i a value of $\frac{\square{d_i} - \blacksquare{d_i}}{2}$ results in a satisfying assignment.*

Proof: There are two types of constraints we must consider, absolute constraints and two variable constraints.

In the case of an absolute constraint, it is either of the form $x_i \leq c_i$ or of the form $-x_i \leq c_i$.

In the first case, i.e., $x_i \leq c_i$, there is, by construction, a white edge from x_0 to x_i of weight c_i. Thus after the relaxation procedure, we must have that $\overset{\square}{d}_i \leq 0 + c_i$. As shown before we also have that $-\overset{\blacksquare}{d}_i \leq \overset{\square}{d}_i$, so we have that $\frac{\overset{\square}{d}_i - \overset{\blacksquare}{d}_i}{2} \leq \overset{\square}{d}_i \leq c_i$ which satisfies the constraint.

In the second case, i.e., $-x_i \leq c_i$, there is, by construction, a black edge from x_0 to x_i of weight c_i. Thus after the relaxation procedure ,we must have that $\overset{\blacksquare}{d}_i \leq 0 + c_i$. As shown before we also have that $-\overset{\square}{d}_i \leq \overset{\blacksquare}{d}_i$, so we have that $(-\frac{\overset{\square}{d}_i - \overset{\blacksquare}{d}_i}{2}) \leq (\frac{\overset{\blacksquare}{d}_i - \overset{\square}{d}_i}{2}) \leq (\frac{\overset{\blacksquare}{d}_i + \overset{\blacksquare}{d}_i}{2}) \leq \overset{\blacksquare}{d}_i \leq c_i$ which satisfies the constraint.

There are four forms of two variable constraints and they are all satisfied as follows:

1. Constraints of the form $x_i + x_j \leq c_{ij}$ - From the previous lemma we have that $\overset{\square}{d}_i \leq c_{ij} + \overset{\blacksquare}{d}_j$ and $\overset{\square}{d}_j \leq c_{ij} + \overset{\blacksquare}{d}_i$. Thus, $\overset{\square}{d}_i + (-\overset{\blacksquare}{d}_j) \leq c_{ij}$ and $(-\overset{\blacksquare}{d}_i) + \overset{\square}{d}_j \leq c_{ij}$. Hence,

$$\frac{\overset{\square}{d}_i - \overset{\blacksquare}{d}_i}{2} + \frac{\overset{\square}{d}_j - \overset{\blacksquare}{d}_j}{2} = \frac{\overset{\square}{d}_i + (-\overset{\blacksquare}{d}_j)}{2} + \frac{(-\overset{\blacksquare}{d}_i) + \overset{\square}{d}_j}{2} \leq c_{ij}$$

2. Constraints of the form $x_i - x_j \leq c_{ij}$ - From the previous lemma, we have that $\overset{\square}{d}_i \leq c_{ij} + \overset{\square}{d}_j$ and $\overset{\blacksquare}{d}_j \leq c_{ij} + \overset{\blacksquare}{d}_i$. Thus, $\overset{\square}{d}_i + (-\overset{\square}{d}_j) \leq c_{ij}$ and $(-\overset{\blacksquare}{d}_i) + \overset{\blacksquare}{d}_j \leq c_{ij}$. Hence,

$$\frac{\overset{\square}{d}_i - \overset{\blacksquare}{d}_i}{2} - \frac{\overset{\square}{d}_j - \overset{\blacksquare}{d}_j}{2} = \frac{\overset{\square}{d}_i + (-\overset{\square}{d}_j)}{2} + \frac{(-\overset{\blacksquare}{d}_i) + \overset{\blacksquare}{d}_j}{2} \leq c_{ij}$$

3. Constraints of the form $-x_i + x_j \leq c_{ij}$ - From the previous lemma, we have that $\overset{\blacksquare}{d}_i \leq c_{ij} + \overset{\blacksquare}{d}_j$ and $\overset{\square}{d}_j \leq c_{ij} + \overset{\square}{d}_i$. Thus, $\overset{\blacksquare}{d}_i + (-\overset{\blacksquare}{d}_j) \leq c_{ij}$ and $(-\overset{\square}{d}_i) + \overset{\square}{d}_j \leq c_{ij}$. Hence,

$$-\frac{\overset{\square}{d}_i - \overset{\blacksquare}{d}_i}{2} + \frac{\overset{\square}{d}_j - \overset{\blacksquare}{d}_j}{2} = \frac{\overset{\blacksquare}{d}_i + (-\overset{\blacksquare}{d}_j)}{2} + \frac{(-\overset{\square}{d}_i) + \overset{\square}{d}_j}{2} \leq c_{ij}$$

4. Constraints of the form $-x_i - x_j \leq c_{ij}$ - From the previous lemma, we have that $\blacksquare{d}_i \leq c_{ij} + \square{d}_j$ and $\blacksquare{d}_j \leq c_{ij} + \square{d}_i$. Thus, $\blacksquare{d}_i + (-\square{d}_j) \leq c_{ij}$ and $(-\square{d}_i) + \blacksquare{d}_j \leq c_{ij}$. Hence,

$$-\frac{\square{d}_i - \blacksquare{d}_i}{2} - \frac{\square{d}_j - \blacksquare{d}_j}{2} = \frac{\blacksquare{d}_i + (-\square{d}_j)}{2} + \frac{(-\square{d}_i) + \blacksquare{d}_j}{2} \leq c_{ij}$$

Thus setting each $x_i = \frac{\square{d}_i - \blacksquare{d}_i}{2}$ satisfies all two variable constraints. □ Since $\square{d}_i$ and $\blacksquare{d}_i$ are integers, setting each $x_i = \frac{\square{d}_i - \blacksquare{d}_i}{2}$ results in a half-integral solution. It is to be noted that the half-integral solution returned by Algorithm 5.0.1 is the starting point for the integer feasibility algorithm discussed in Section 7.

Chapter 7

Integer Feasibility Algortitm

We use Algorithm 7.1.1 to determine whether a system of UTVPI constraints, $\mathbf{U}$, encloses a lattice point. The principal idea underlying our approach is the following: If the half-integral solution $\mathbf{a}$, returned by Algorithm 5.0.1 is not integral, then there exists a rounding procedure, which finds a lattice point within a $\frac{1}{2}$-neighborhood $\mathbf{a}$. (A $\frac{1}{2}$-neighborhood of a point $\mathbf{a}$, is the set of all points $\mathbf{b}$, such that $a_i - \frac{1}{2} \leq b_i \leq a_i + \frac{1}{2}$, $\forall i = 1, 2, \ldots, n$.) Furthermore, if no such lattice point exists, then $\mathbf{U}$ does not enclose a lattice point.

Rounding a variable x_i, corresponds to adding an absolute constraint involving x_i to the system. For instance, rounding down a variable x_i, is equivalent to assigning x_i the value $\lfloor a_i \rfloor$, and adding the constraint $x_i \leq \lfloor a_i \rfloor$ to $\mathbf{U}$. Similarly, rounding up a variable, x_i, is equivalent to assigning x_i the value $\lceil a_i \rceil$, and adding the constraint $-x_i \leq -\lceil a_i \rceil$.

If we start with a rational solution in which $a_1 = \frac{5}{2}$, then rounding x_1 down is equivalent to setting $x_1 = 2$ and adding the constraint $x_1 \leq 2$ to the system. *Example (13):*

There are two types of roundings used by Algorithm 7.1.1, viz.,

1. *Optional* roundings - These are roundings in which a variable x_i can be set to either $\lceil a_i \rceil$ or $\lfloor a_i \rfloor$, without causing an immediate contradiction.

2. *Forced* roundings - These are roundings in which one of the possible roundings

of a variable, x_i, causes an immediate contradiction.

A rounding causes an immediate contradiction, if the added constraint contradicts a constraint of type (1) from Section 3.2 (See page 17).

Consider the following system of UTVPI constraints: $l_1 : x_1 + x_2 \leq 0$ and $l_2 : x_1 - x_2 \leq 1$. Assume that $a_1 = \frac{1}{2}$. Rounding x_1 up would cause us to set a_1 to 1 and to add the constraint $l_3 : -x_1 \leq -1$. Note that the set $\mathbf{U}'$ (see page 17) contains the constraint $l_4 : x_1 \leq 0$, obtained by adding l_1 and l_2. Clearly, l_3 and l_4 contradict each other, i.e., we have an immediate contradiction. This means that x_1 is *forced* to be rounded down. *Example (14):*

After rounding a variable x_i, Algorithm 7.1.4 checks to see if any of the variables sharing a constraint with x_i needs to be rounded in order to satisfy all the constraints involved.

If rounding x_i in one direction eventually causes a contradiction (such a contradiction will be discovered in Line 18 of Algorithm 7.1.1 or Line 12 or Line 24 of Algorithm 7.1.3), then x_i is rounded in the other direction. If that rounding also results in a contradiction, then the system is declared infeasible.

After a variable has been successfully rounded and all the *resultant* roundings are performed, no future roundings will violate any constraint containing any of these variables. Thus x_i will not be rounded again. This is true on account of the structure of UTVPI constraint systems; observe that a general integer program does not have such a structure.

7.1 Algorithms

7.1.1 The Algorithm Produce-Solution()

Algorithm 7.1.1 finds an integral solution to the system of UTVPI constraints $\mathbf{U}$, or demonstrates that none exists. It starts with a half-integral solution $\mathbf{a}$, and proceeds to round the variables until a solution is found, or a contradiction is established.

The algorithm creates Z to store the integer solution being constructed. In the

Function PRODUCE-SOLUTION (set **U** of UTVPI constraints, and linear solution **a** of **U**)

1: {This is the main function that calls all the other functions. It returns either a feasible integral solution or a proof that none exists.}
2: **for** (each variable x_i) **do**
3: $Z_i = M$
4: **end for**
5: create tree T of constraints with node x_0 at the root
6: **for** (each variable x_i) **do**
7: **if** (a_i is an integer) **then**
8: $Z_i = a_i$
9: **else**
10: FORCED-ROUNDING(x_i,Z,**a**,T,**U**)
11: **end if**
12: **end for**
13: **while** (any Z_i is updated) **do**
14: **for** (each Z_i newly assigned) **do**
15: CHECK-DEPENDENCIES (x_i, Z, **a**, T,**U**)
16: **end for**
17: **end while**
18: **for** (every constraint in **U**) **do**
19: **if** (constraint is violated by current assignments to some Z_i and Z_j) **then**
20: **return** (violated constraint and constraints obtained by backtracking in T from x_i to x_0 and x_j to x_0)
21: **end if**
22: **end for**
23: $S \leftarrow$ Subset of **U** restricted to constraints consisting of only variables with $Z_i = M$
24: $O \leftarrow$ OPTIONAL-ROUNDINGS(S, Z, T, **a**) {These variables were not affected by either the forced roundings or the resultant roundings.}
25: **return** O

Algorithm 7.1.1: PRODUCE-SOLUTION

algorithm the variable M simple represents an arbitrary value that is much larger than any value of $\mathbf{a}$. This lets us easily see which variables have been rounded and which haven't. result of any rounding performed. This is necessary, because we need to check whether or not a particular variable has been rounded. It also creates a tree structure T, which will be used to return the constraints that demonstrate the integer infeasibility of the system. Each node of T is a variable x_i of the original system that has been rounded and the set of one or three constraints that were used to round x_i. Observe that the node corresponding to x_i in T contains three constraints if x_i was rounded because of a forced rounding, and one constraint if it was rounded because of a resultant rounding.

The parent of a node x_i, represents the rounding that necessitated the rounding of x_i. The children of the node represent all of the resultant roundings which stem from rounding x_i. Since each variable is rounded at most once, each node will occur at most once in the tree.

Consider the system of constraints $\mathbf{U}$, with the constraints $l_1 : x_1 + x_2 \leq 1$, $l_2 : x_1 - x_2 \leq 0$, $l_3 : -x_1 - x_3 \leq 2$ and $l_4 : x_3 + x_4 \leq 2$. Let $\mathbf{a}$ be the valid linear solution $a_1 = \frac{1}{2}$, $a_2 = \frac{1}{2}$, $a_3 = -\frac{5}{2}$ and $a_4 = \frac{9}{2}$.

From the constraints l_1 and l_2, and the tightening inference rule, we can deduce the constraint $l_5 : x_1 \leq 0$. Thus x_1 is rounded down and $Z_1 = \lfloor a_1 \rfloor = 0$. Accordingly, Algorithm 7.1.2 will create the node x_1 as a child of x_0 and that node will contain the constraints l_1, l_2 and l_5.

Since x_1 is rounded down, the constraint l_3 will be violated, unless x_3 is rounded up to $Z_3 = \lceil a_3 \rceil = -2$. Thus Algorithm 7.1.4 will create the node x_3 as a child of x_1 and that node will contain the constraint l_3.

Since x_3 is rounded up, the constraint l_4 will be violated, unless x_4 is rounded down to $Z_4 = \lfloor a_4 \rfloor = 4$. Thus Algorithm 7.1.4 will create the node x_4 as a child of x_3 and that node will contain the constraint l_4.

Therefore, after these steps the tree T will have the following structure:

Example (15):

Algorithm 7.1.1 does not alter the integer values of the linear solution $\mathbf{a}$, since

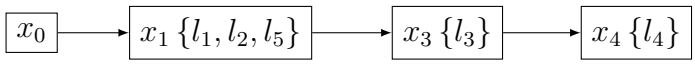

Figure 7.1: Tree T

they will also be part of the rounded solution. On the fractional values of **a**, it calls Algorithm 7.1.2, to perform forced roundings, as needed.

Algorithm 7.1.4 checks to see if other variables need to be rounded as a consequence of variables being rounded by Algorithm 7.1.2. These roundings are called resultant roundings.

Once the forced and resultant roundings are performed, Algorithm 7.1.1 checks to see if any constraint is violated. If a constraint involving the variables x_i and x_j is violated, then that constraint and all the constraints that caused x_i and x_j to be rounded are returned as proof of integer infeasibility. To determine which constraints caused variable x_i to be rounded, the algorithm starts with node x_i in the tree T and proceeds to traverse up the tree until the root node is reached returning all of the constraints stored in the nodes traversed. This is then repeated for variable x_j.

If no constraint is violated, then Algorithm 7.1.3 is called to perform optional roundings.

7.1.2 The Algorithm Forced-Rounding()

Algorithm 7.1.2 checks to see if a variable takes part in a forced rounding. If the variable if forced to be rounded, then that rounding is performed and the appropriate constraints are added to the tree T.

7.1.3 The Algorithm Optional-Roundings()

Algorithm 7.1.3 handles the rounding of variables that were left unaffected by the forced roundings and the subsequent resultant roundings. It first rounds a variable (say x_i) down and then calls Algorithm 7.1.4 to evaluate all of the resultant roundings. It then stores all of the new values in a temporary version of Z called Z^T.

Function FORCED-ROUNDING (variable x_i, variable weights Z, linear solution $\mathbf{a}$, constraint tree T, system $\mathbf{U}$)

1: **for** (each x_j that shares constraints with x_i) **do**
2: Define R as the set of constraints in $\mathbf{U}$ involving both x_i and x_j
3: **if** $(\{x_i + x_j \leq a_i + a_j,\ x_i - x_j \leq a_i - a_j\} \subseteq R)$ **then**
4: $Z_i = \lfloor a_i \rfloor$
5: create branch x_i from x_0 in T
6: add $\{x_i + x_j \leq a_i + a_j$, $x_i - x_j \leq a_i - a_j$, and $x_i \leq \lfloor a_i \rfloor$ to T under $x_i\}$
7: **end if**
8: **if** $(\{-x_i - x_j \leq -a_i - a_j,\ x_j - x_i \leq a_j - a_i\} \subseteq R)$ **then**
9: $Z_i = \lceil a_i \rceil$
10: create branch x_i from x_0 in T
11: add $\{-x_i - x_j \leq -a_i - a_j$, $x_j - x_i \leq a_j - a_i$, and $-x_i \leq -\lceil a_i \rceil$ to T under $x_i\}$
12: **end if**
13: **end for**

Algorithm 7.1.2: FORCED-ROUNDING

Function OPTIONAL-ROUNDINGS (set S of constraints, vector Z, tree T, linear solution $\mathbf{a}$)

1: create tree T^T of constraints with node x_0 at the root
2: create array Z^T of temporary variable assignments
3: create list Q of constraints to be returned in case of infeasibility.
4: **for** (each variable x_i) **do**
5: **if** ($Z_i = M$) **then**
6: $Z_i^T = \lfloor a_i \rfloor$
7: create branch x_i from x_0 in T
8: add $x_i \leq \lfloor a_i \rfloor$ to T under x_i
9: **while** (any Z_i^T is updated) **do**
10: **for** (each Z_i^T newly assigned) **do**
11: CHECK-DEPENDENCIES (x_i, Z^T, $\mathbf{a}$, T, S)
12: **end for**
13: **end while**
14: **for** (each constraint in S) **do**
15: **if** (constraint is violated by current assignments to some Z_j^T and Z_k^T) **then**
16: Add the violated constraint and the constraints in T along paths from x_j to x_i and x_k to x_i to Q
17: **for** ($j = 1$ **to** n) **do**
18: $Z_j^T = M$
19: **end for**
20: $Z_i^T = \lceil a_i \rceil$, create tree T^T of constraints with node x_0 at the root
21: create branch x_i from x_0 in T^T
22: add $-x_i \leq -\lceil a_i \rceil$ to T^T under x_i
23: **while** (any Z_i^T is updated) **do**
24: **for** (each Z_i^T newly assigned) **do**
25: CHECK-DEPENDENCIES (x_i, Z^T, $\mathbf{a}$, T^T, S)
26: **end for**
27: **end while**

28: **for** (each constraint in S) **do**
29: **if** (constraint is violated by current assignments to some Z_j^T and Z_k^T) **then**
30: Add the violated constraint and the constraints in T^T along paths from x_j to x_i and x_k to x_i to Q
31: **return** set Q of constraints
32: **end if**
33: **end for**
34: **for** ($i = 1$ **to** n) **do**
35: **if** ($Z_i^T \neq M$) **then**
36: $Z_i \leftarrow Z_i^T$
37: **end if**
38: **end for**
39: **break** out of enclosing **for** loop
40: **end if**
41: **end for**
42: **if** (no constraints violated as a result of rounding x_i down) **then**
43: **for** ($i = 1$ **to** n) **do**
44: **if** ($Z_i^T \neq M$) **then**
45: $Z_i \leftarrow Z_i^T$
46: **end if**
47: **end for**
48: **end if**
49: destroy Z^T, T^T, and Q, and $T \leftarrow x_0$
50: **end if**
51: **end for**
52: **return** Z as a valid integer solution.

Algorithm 7.1.3: OPTIONAL-ROUNDINGS

If this rounding of x_i succeeds, then the temporary values are made permanent and the algorithm proceeds onto the next unrounded variable. If rounding x_i down fails, then the algorithm stores the constraints that cause a contradiction when $x_i \leq \lfloor a_i \rfloor$ to Q and clears the temporary assignments.

Algorithm 7.1.3 then attempts to round x_i up, again evaluating all of the resultant roundings. This time, T^T, a temporary version of the tree T is used in addition to Z^T. If this rounding of x_i succeeds, then all temporary assignments are made permanent and the algorithm proceeds onto the next unrounded variable. If this rounding also fails, then the constraints that cause a contradiction when $-x_i \leq -\lceil a_i \rceil$ are added to Q, and Q is returned as a certificate of integer infeasibility.

The list of constraints Q can be divided into two parts. The constraints in the first part of Q add together to the constraint $x_i \leq c < 0$. Thus showing that the system is inconsistent when the constraint $x_i \leq \lfloor a_i \rfloor$ is added to the system. Similarly the constraints in the second part show that the system is inconsistent when $x_i \geq \lceil a_i \rceil$ is added.

7.1.4 The Algorithm Check-Dependencies()

Algorithm 7.1.4 checks to see if rounding x_i results in a constraint being violated; if a violation occurs, other variables undergo a *resultant* rounding.

Consider a UTVPI system with $l_1 : x_1 + x_2 \leq 1$ as one of its constituent constraints. Let $a_1 = \frac{1}{2}$ and $a_2 = \frac{1}{2}$ denote a valid linear solution. If x_1 is rounded up to $\lceil a_1 \rceil = 1$, l_1 is violated. In order to ensure that this constraint is not violated, x_2 needs to be rounded down to $\lfloor a_2 \rfloor = 0$. The rounding of x_2 is a resultant rounding. *Example (16):*

Function CHECK-DEPENDENCIES (variable x_i, vector Z^T of assignments, **a** linear solution, tree T of UTVPI constraints, system **U** of constraints)

1: **for** (each variable x_j sharing constraints with x_i) **do**
2: Set R to be the set of constraints involving both x_i and x_j
3: **if** $(Z_i^T = \lfloor a_i \rfloor)$ **then**
4: **if** $(-x_i + x_j \leq -a_i + a_j \in R$ and $Z_j^T = M)$ **then**
5: $Z_j \leftarrow \lfloor a_j \rfloor$
6: create branch x_j from x_i in T
7: add $-x_i + x_j \leq -a_i + a_j$ to T under x_j
8: **end if**
9: **if** $(-x_i - x_j \leq -a_i - a_j \in R$ and $Z_j^T = M)$ **then**
10: $Z_j \leftarrow \lceil a_j \rceil$
11: create branch x_j from x_i in T
12: add $-x_i - x_j \leq -a_i - a_j$ to T under x_j
13: **end if**
14: **end if**
15: **if** $(Z_i^T = \lceil a_i \rceil)$ **then**
16: **if** $(x_i + x_j \leq a_i + a_j \in R$ and $Z_j^T = M)$ **then**
17: $Z_j \leftarrow \lfloor a_j \rfloor$
18: create branch x_j from x_i in T
19: add $x_i + x_j \leq a_i + a_j$ to T under x_j
20: **end if**
21: **if** $(x_i - x_j \leq a_i - a_j \in R$ and $Z_j^T = M)$ **then**
22: $Z_j \leftarrow \lceil a_j \rceil$
23: create branch x_j from x_i in T
24: add $x_i - x_j \leq a_i - a_j$ to T under x_j
25: **end if**
26: **end if**
27: **end for**

Algorithm 7.1.4: CHECK-DEPENDENCIES

7.2 Resource Analysis

7.2.1 Forced roundings

We argue that the FORCED-ROUNDINGS() function and the subsequent call to CHECK-DEPENDENCIES() can be accomplished in $\mathcal{O}(m+n)$ space and $\mathcal{O}(m \cdot n)$ time.

Checking each x_i to see if it is forced to be rounded involves looking at all constraints involving x_i. This takes $\mathcal{O}(m)$ time total, since each constraint is considered at most twice, once for each variable involved in defining that constraint.

A forced rounding of a variable adds a constraint that can be deduced from existing constraints using the tightening inference rule, (see System (3.3)). As each variable is rounded, the resultant roundings of the new assignment are deduced, using CHECK-DEPENDENCIES(). During these deductions each variable is assigned a value at most once. Thus, as before, each edge (constraint) is processed at most twice, using a total of $\mathcal{O}(m)$ time. Likewise, each time a variable is rounded, a constraint can be deduced from the transitive inference rule.

The values given to these variables are now substituted back into the original constraint system and the consistency of this assignment is checked. This takes $\mathcal{O}(m)$ time. If an inconsistency is obtained, then the last constraint that was checked was violated. We then backtrack along the tree T of constraints to produce a series of constraints which produce a contradiction. Since, by construction, each such path is of length at most n, and we traverse 2 such paths, the backtracking process takes $\mathcal{O}(n)$ time.

7.2.2 Optional roundings

When there is a choice on the manner in which a variable can be rounded, performing all resultant roundings of making a particular choice takes $\mathcal{O}(m)$ time, for precisely the same reason that performing the resultant roundings of a forced rounding takes $\mathcal{O}(m)$ time. As before, checking for consistency takes $\mathcal{O}(m)$ time.

Each variable is rounded this way at most twice, once up and once down. Thus evaluating the roundings of all variables takes $\mathcal{O}(m \cdot n)$ time. Space is reused among variables, thus this process runs in $\mathcal{O}(m)$ space.

If an inconsistency is obtained after both of the possible roundings of a variable, x_i, then for both roundings, we backtrack along the tree T. This creates two sequences of constraints which produce contradictions. One contradiction assumes that $x_i \leq \lfloor a_i \rfloor$ and the the other assumes that $-x_i \leq -\lceil a_i \rceil$. By construction, each of the four backtracked paths is of length at most n, so this process takes $\mathcal{O}(n)$ time. This only occurs once, since only one refutation is necessary.

7.2.3 Overall analysis

All parts of the Algorithm 7.1.1 run in $\mathcal{O}(m \cdot n)$ time and $\mathcal{O}(m + n)$ space and therefore these are the resource bounds for this algorithm.

Chapter 8

Correctness of the Integer Algorithm

In this section, we argue the correctness of Algorithm 7.1.1. Note that Algorithm 7.1.1 starts with a feasible half-integral solution and performs a forced rounding (and if needed, a resultant rounding) or an optional rounding (and if needed, a resultant rounding) on variables which are not integral. In proof, we shall demonstrate that every rounding (forced, optional or resultant) corresponds to a deducible constraint.

Lemma 8.0.1 *Each forced rounding corresponds to a new constraint that can deduced from previously existing constraints by the tightening inference rule.*

Proof: Let x_i be the variable, with initial value a_i, that undergoes a forced rounding.

If x_i is rounded down then by Algorithm 7.1.2, then there must exist a variable x_j, with initial value a_j, such that there exist constraints $x_i - x_j \leq a_i - a_j$ and $x_i + x_j \leq a_i + a_j$. Using the tightening inference rule we can deduce the constraint $x_i \leq \lfloor \frac{a_i - a_j + a_i + a_j}{2} \rfloor = \lfloor a_i \rfloor$. This is the constraint that causes x_i to be rounded down.

Likewise, if x_j is rounded up by Algorithm 7.1.2, then there must exist a variable x_j, with initial value a_j, such that there exist constraints $-x_i - x_j \leq -a_i - a_j$ and $-x_i + x_j \leq -a_i + a_j$. Using the tightening inference rule we can deduce the constraint $-x_i \leq \lfloor \frac{-a_i - a_j - a_i + a_j}{2} \rfloor = -\lceil a_i \rceil$. This is the constraint that causes x_i to be rounded

up. □

Consider a UTVPI system with the following two constraints: $l_1 : x_1 + x_2 \leq 3$ and $l_2 : x_1 - x_2 \leq 0$. Assume that we have a fractional solution $a_1 = a_2 = \frac{3}{2}$. As per Algorithm 7.1.2, x_1 should be rounded down. As stated previously this results in adding the constraint $l_3 : x_1 \leq 1$ to the UTVPI system. However, l_3 can deduced from l_1 and l_2 using the tightening inference rule. *Example (17):*

Lemma 8.0.2 *Each resultant rounding that results from either a forced rounding or an optional rounding, corresponds to a new constraint that can deduced from previously existing and deduced constraints by the transitive inference rule.*

Proof: Let x_i be the variable, with initial value a_i, that is rounded due to a resultant rounding, caused by rounding the variable x_j, with initial value a_j. Since only non-integral values are rounded, in the algorithms discussed in Section 7, we can assume without loss of generality that both a_i and a_j are odd multiples of $\frac{1}{2}$.

We need to consider the following four cases:

1. x_i is rounded down as a result of x_j being rounded down - From Algorithm 7.1.4, there must be a constraint of type $x_i - x_j \leq a_i - a_j$ in $\mathbf{U}$. Since x_j was rounded down, the constraint $x_j \leq \lfloor a_j \rfloor = a_j - \frac{1}{2}$ is deducible from the original system. Using the transitive inference rule, we get the constraint $x_i \leq a_i - a_j + a_j - \frac{1}{2} = a_i - \frac{1}{2} = \lfloor a_i \rfloor$. This is the constraint that caused x_i to be rounded down.

2. x_i is rounded down as a result of x_j being rounded up - From Algorithm 7.1.4, there must be a constraint of type $x_i + x_j \leq a_i + a_j$ in $\mathbf{U}$. Since x_j was rounded up, the constraint $-x_j \leq -\lceil a_j \rceil = -a_j - \frac{1}{2}$ is deducible from the original system. Using the transitive inference rule, we get the constraint $x_i \leq a_i + a_j - a_j - \frac{1}{2} = a_i - \frac{1}{2} = \lfloor a_i \rfloor$. This is the constraint that causes x_i to be rounded down.

3. x_i is rounded up as a result of x_j being rounded down - From Algorithm 7.1.4, there must be a constraint of type $-x_i - x_j \leq -a_i - a_j$ in $\mathbf{U}$. Since x_j was rounded down, the constraint $x_j \leq \lfloor a_j \rfloor = a_j - \frac{1}{2}$ is deducible from the original

system. Using the transitive inference rule, we get the constraint $-x_i \leq -a_i - a_j + a_j - \frac{1}{2} = -a_i - \frac{1}{2} = -\lceil a_i \rceil$. This is the constraint that causes x_i to be rounded up.

4. x_i is rounded up as a result of x_j being rounded up - From Algorithm 7.1.4, there must be a constraint of type $-x_i + x_j \leq -a_i + a_j$ in $\mathbf{U}$. Since x_j was rounded up, the constraint $-x_j \leq -\lceil a_j \rceil = -a_j - \frac{1}{2}$ is deducible from the original system. Using the transitive inference rule, we get the constraint $-x_i \leq -a_i + a_j - a_j - \frac{1}{2} = -a_i - \frac{1}{2} = -\lceil a_i \rceil$. This is the constraint that causes x_i to be rounded up.

□

Let x_j be a variable rounded as a result of a forced or optional rounding. Let $\mathcal{V}$ be the set containing x_j and all of the variables rounded as a result of x_j being rounded.

Let x_i be any unrounded variable. We will show that x_i can be rounded up or down without violating a constraint involving variables in $\mathcal{V}$. Thus, if no constraint is violated as a result of rounding x_j, and performing all subsequent resultant roundings, the values of the variables in $\mathcal{V}$ can be considered permanent. This follows, since no subsequent roundings will violate a constraint involving any of these variables.

Lemma 8.0.3 *Any unrounded variable (after rounding x_j and performing any subsequent resultant roundings) can be rounded up or down, without violating any constraints shared with a variable in $\mathcal{V}$.*

Proof: Let x_i be an unrounded variable, where $x_i \notin \mathcal{V}$. Clearly, we are concerned *only* with constraints of the form: $\pm x_i \pm x_k \leq c_{ik}$, where $x_k \in \mathcal{V}$, since all other constraints are satisfied with the current assignment to the variables.

We assume the contrary, i.e, we assume that a constraint involving x_i and x_k, where $x_k \in \mathcal{V}$, is violated, when x_i is rounded in a certain direction.

The following four cases need to be considered:

1. x_k was rounded down and rounding x_i down results in a violation -
   In this case, there is a constraint that was satisfied when $x_i = a_i$ and $x_k = a_k$, but violated when $x_i = a_i - \frac{1}{2}$ and $x_k = a_k - \frac{1}{2}$. Thus, the violated constraint *must* be $-x_i - x_k \leq -a_i - a_k$. However, this constraint would cause Algorithm 7.1.4 to round x_i up as a result of rounding x_k. But this contradicts our assumption that x_i was unrounded to begin with.

2. x_k was rounded down and rounding x_i up results in a violation -
   In this case, there is a constraint that was satisfied when $x_i = a_i$ and $x_k = a_k$, but violated by $x_i = a_i + \frac{1}{2}$ and $x_k = a_k - \frac{1}{2}$. Thus, the violated constraint *must* be $x_i - x_k \leq a_i - a_k$. However, this constraint would cause Algorithm 7.1.4 to round x_i down as a result of rounding x_k. But this contradicts our assumption that x_i was unrounded to begin with.

3. x_k was rounded up and rounding x_i down results in a violation -
   In this case, there is a constraint that was satisfied when $x_i = a_i$ and $x_k = a_k$, but violated by $x_i = a_i - \frac{1}{2}$ and $x_k = a_k + \frac{1}{2}$. Thus, the violated constraint *must* be $-x_i + x_k \leq -a_i + a_k$. However, this constraint would cause Algorithm 7.1.4 to round x_i up as a result of rounding x_k. But this contradicts our assumption that x_i was unrounded to begin with.

4. x_k was rounded up and rounding x_i up results in a violation -
   In this case, there is a constraint that was satisfied when $x_i = a_i$ and $x_k = a_k$, but violated by $x_i = a_i + \frac{1}{2}$ and $x_k = a_k + \frac{1}{2}$. Thus, the violated constraint *must* be $x_i + x_k \leq a_i + a_k$. However, this constraint would cause Algorithm 7.1.4 to round x_i down as a result of rounding x_k. But this contradicts our assumption that x_i was unrounded to begin with.

Since all four cases result in a contradiction, it follows that no constraint involving x_k can be violated when x_i is rounded. □

The above lemma ensures that once the resultant roundings of rounding a variable are fully computed, and they do not produce any inconsistency, then those variables

do not need to be revisited. This stops the run time of the algorithm from exploding exponentially.

Theorem 8.0.1 *If Algorithm 7.1.1 declares a UTPVI system* **U** *to be feasible, then the system has integral solutions.*

Proof: Algorithm 7.1.1 declares **U** to be feasible, only if a valid integer solution has been computed. Indeed, the valid integral solution is returned by Algorithm 7.1.3. □

Theorem 8.0.2 *If Algorithm 7.1.1 declares a UTVPI system* **U** *to be infeasible, then the system* **U** *has no integral solutions.*

Proof: The algorithms can declare the system infeasible as a result of a forced rounding, and the subsequent resultant roundings, or as the result of an optional rounding and the subsequent resultant roundings.

If the system is declared infeasible as a result of a forced rounding, then there is a constraint between some x_i and x_j that is violated when all resultant roundings are computed. Both x_i and x_j must already have been rounded. Thus, there are four cases which need to be considered, depending on the type of constraint violated:

1. The violated constraint is of the form $l_1 : x_i + x_j \leq c_{ij}$ -

   Since the initial (linear) solution, **a**, was valid we have that $a_i + a_j \leq c_{ij}$. Thus, for l_1 to be violated, x_i and x_j must both have been rounded up. So we have that $c_{ij} < a_i + a_j + 1$.

   Since x_i and x_j were rounded up, from Lemmas 8.0.1 and 8.0.2, we know that the constraints $l_2 : -x_i \leq -\lceil a_i \rceil = -a_i - \frac{1}{2}$ and $l_3 : -x_j \leq -\lceil a_j \rceil = -a_j - \frac{1}{2}$, are deducible from the existing constraints in the system. When these constraints are added to the violated constraint, we get that $0 \leq c_{ij} - a_i - a_j - 1 < 0$, which is a contradiction that establishes the integer infeasibility of **U**.

2. The violated constraint is of the form $l_1 : x_i - x_j \leq c_{ij}$ -

   Since the initial (linear) solution, **a**, was valid we have that $a_i - a_j \leq c_{ij}$. Thus,

for l_1 to be violated, x_i must have been rounded up and x_j must have been rounded down. So we have that $c_{ij} < a_i - a_j + 1$.

From Lemmas 8.0.1 and 8.0.2, we know that the constraints $l_2 : -x_i \leq -\lceil a_i \rceil = -a_i - \frac{1}{2}$ and $l_3 : x_j \leq \lfloor a_j \rfloor = a_j - \frac{1}{2}$ are deducible from existing constraints in the system. When these constraints are added to the violated constraint, we get that $0 \leq c_{ij} - a_i + a_j - 1 < 0$ which is a contradiction that establishes the integer infeasibility of $\mathbf{U}$.

3. The violated constraint is of the form $l_1 : x_j - x_i \leq c_{ji}$ -
   Since the initial (linear) solution, $\mathbf{a}$, was valid we have that $a_j - a_i \leq c_{ji}$. Thus, for l_1 to be violated, x_j must have been rounded up and x_i must have been rounded down. So we have that $c_{ji} < a_j - a_i + 1$.

   From Lemmas 8.0.1 and 8.0.2, we know that the constraints $l_2 : -x_j \leq -\lceil a_j \rceil = -a_j - \frac{1}{2}$ and $l_3 : x_i \leq \lfloor a_i \rfloor = a_i - \frac{1}{2}$ are deducible from existing constraints in the system. When these constraints are added to the violated constraint, we get that $0 \leq c_{ji} - a_j + a_i - 1 < 0$ which is a contradiction that establishes the integer infeasibility of $\mathbf{U}$.

4. The violated constraint is of the form $l_1 : -x_i - x_j \leq c_{ij}$ -
   Since the initial (linear) solution, $\mathbf{a}$ was valid we have that $-a_i - a_j \leq c_{ij}$. Thus, for the constraint to be violated x_i and x_j must both have been rounded down. So we have that $c_{ij} < -a_i - a_j + 1$.

   Since x_i and x_j were rounded down, from Lemmas 8.0.1 and 8.0.2, we know that the constraints $l_2 : x_i \leq \lfloor a_i \rfloor = a_i - \frac{1}{2}$ and $l_3 : x_j \leq \lfloor a_j \rfloor = a_j - \frac{1}{2}$ are deducible from existing constraints in the system. When these constraints are added to the violated constraint, we get that $0 \leq c_{ij} + a_i + a_j - 1 < 0$, which is a contradiction that establishes the integer infeasibility of $\mathbf{U}$.

If the system is declared infeasible as a result of an optional rounding, then for some variable x_k, it is clear that that the systems $\mathbf{U} \cup \{x_k \leq \lfloor a_k \rfloor\}$ and $\mathbf{U} \cup \{-x_k \leq$

$-\lceil a_k \rceil\}$ are infeasible. Since all possible integer values of x_k are covered by one of the two systems, we can conclude that $\mathbf{U}$ has no integer solutions. $\square$

As discussed above, Algorithm 7.1.1 starts with an arbitrary half integral solution and always maintains $\lfloor a_i \rfloor \leq Z_i \leq \lceil a_i \rceil$, for each $Z_i \neq M$. Thus, we have the following corollary.

Corollary 8.0.1 *If a system* $\mathbf{U}$ *of UTVPI constraints is integer feasible, and* $\mathbf{a}$ *is a valid half-integral solution to* $\mathbf{U}$*, then there exists an integral solution* $\mathbf{Z}$ *such that for each* $i = 1 \ldots n$*,* $\lfloor a_i \rfloor \leq Z_i \leq \lceil a_i \rceil$.

Chapter 9

An Illustrative Example

In this section, we apply the algorithms developed in previous sections to an illustrative sample UTPVI system.

Consider the UTVPI system defined by System (9.1).

$$\begin{aligned} l_1 : x_1 + x_2 &\leq 2 \\ l_2 : x_1 - x_2 &\leq 1 \\ l_3 : x_3 - x_2 &\leq 1 \\ l_4 : x_4 - x_2 &\leq 0 \\ l_5 : -x_3 - x_4 &\leq -2 \\ l_6 : -x_1 &\leq -1 \end{aligned} \tag{9.1}$$

The constraint network corresponding to System (9.1) is provided in Figure 9. The edges of weight 8 from x_0 are not displayed.

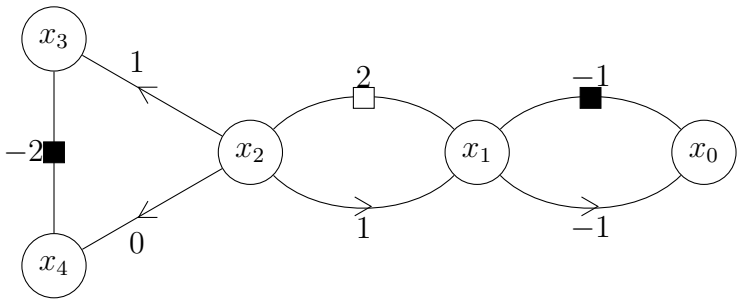

Figure 9.1: Constraint network for example constraints.

Our initial distance table is as follows:

x_i	$\square$ d_i	$\blacksquare$ d_i	$\rightarrow$ d_i	$\leftarrow$ d_i
x_0	0	0	0	0
x_1	8	8	8	8
x_2	8	8	8	8
x_3	8	8	8	8
x_4	8	8	8	8

Table 9.1: Initial Distance Values

We relax each edge of the constraint network, as discussed in Section 5. The relaxations change the distance labels as indicated in the table below:

edge			new distance values							
x_i		x_j	$\square$ d_i	$\blacksquare$ d_i	$\rightarrow$ d_i	$\leftarrow$ d_i	$\square$ d_j	$\blacksquare$ d_j	$\rightarrow$ d_j	$\leftarrow$ d_j
x_0	$\overset{1}{\blacksquare}$	x_1	0	0	0	0	8	-1	8	-1
x_1	$\overset{1}{\rightarrow}$	x_0	8	-1	8	-1	0	0	0	0
x_1	$\overset{2}{\square}$	x_2	8	-1	8	-1	1	8	1	8
x_2	$\overset{1}{\rightarrow}$	x_1	1	0	1	0	2	-1	2	-1
x_2	$\overset{1}{\rightarrow}$	x_3	1	0	1	0	2	8	2	8
x_2	$\overset{0}{\rightarrow}$	x_4	1	0	1	0	1	8	1	8
x_3	$\overset{-2}{\blacksquare}$	x_4	2	-1	2	-1	1	0	1	0

Table 9.2: First Round of Relaxations

The distance table after the first round of relaxations is given below:

x_i	$\overset{\square}{d_i}$	$\overset{\blacksquare}{d_i}$	$\overset{\rightarrow}{d_i}$	$\overset{\leftarrow}{d_i}$
x_0	0	0	0	0
x_1	2	-1	2	-1
x_2	1	0	1	0
x_3	2	-1	2	-1
x_4	1	0	1	0

Table 9.3: Distance Values After First Round

The second round of relaxations alters the distance labels as recorded in the table below:

edge			new distance values							
x_i		x_j	$\overset{\square}{d_i}$	$\overset{\blacksquare}{d_i}$	$\overset{\rightarrow}{d_i}$	$\overset{\leftarrow}{d_i}$	$\overset{\square}{d_j}$	$\overset{\blacksquare}{d_j}$	$\overset{\rightarrow}{d_j}$	$\overset{\leftarrow}{d_j}$
x_0	$\overset{1}{\blacksquare}$	x_1	0	0	0	0	2	-1	2	-1
x_1	$\overset{1}{\rightarrow}$	x_0	2	-1	2	-1	0	0	0	0
x_1	$\overset{2}{\square}$	x_2	2	-1	2	-1	1	0	1	0
x_2	$\overset{1}{\rightarrow}$	x_1	1	0	1	0	2	-1	2	-1
x_2	$\overset{1}{\rightarrow}$	x_3	1	0	1	0	2	-1	2	-1
x_2	$\overset{0}{\rightarrow}$	x_4	1	0	1	0	1	0	1	0
x_3	$\overset{-2}{\blacksquare}$	x_4	2	-1	2	-1	1	0	1	0

Table 9.4: Second Round of Relaxations

The distance table after the second round of relaxations is given below:

x_i	$\overset{\square}{d_i}$	$\overset{\blacksquare}{d_i}$	$\overset{\rightarrow}{d_i}$	$\overset{\leftarrow}{d_i}$
x_0	0	0	0	0
x_1	2	-1	2	-1
x_2	1	0	1	0
x_3	2	-1	2	-1
x_4	1	0	1	0

Table 9.5: Distance Values After Second Round

We observe that the distance table at the end of the second round of relaxations is *identical* to the distance table at the end of the first round of relaxations. This means that additional relaxations will not affect the distance table and hence we will not show the relaxations from subsequent rounds. Indeed, this table is the final distance table.

Thus the resultant linear solution to the system of equations is $x_0 = \frac{0-0}{2} = 0$, $x_1 = \frac{2-(-1)}{2} = 1.5$, $x_2 = \frac{1-0}{2} = .5$, $x_3 = \frac{2-(-1)}{2} = 1.5$, and $x_4 = \frac{1-0}{2} = .5$, which is a valid solution.

Our next task is to compute an integer solution from the half-integral linear solution obtained above. As discussed in Section 7, this requires the rounding of the half-integral values in a consistency-preserving manner.

We execute the following steps:

1. Check x_1 for forced roundings.
   (a) The constraints involving x_1 are l_1, l_2 and l_6.
   (b) The contraints l_1 and l_2 force x_1 to be rounded down to $x_1 = 1$.
2. Check x_2 for forced roundings.
   (a) The constraints involving x_2 are l_2, l_3 and l_4.

(b) None of them forces x_2 to be rounded; it follows that x_2 does not undergo a forced rounding.

3. Check x_3 for forced roundings.

   (a) The constraints involving x_3 (and another variable) are l_1 and l_5.

   (b) Neither of them forces x_3 to be rounded; it follows that x_3 does not undergo a forced rounding.

4. Check x_4 for forced roundings.

   (a) The constraints involving x_4 are l_4 and l_5.

   (b) Neither of them forces x_4 to be rounded; it follows that x_4 does not undergo a forced rounding.

Thus the only forced rounding is to round x_1 down to 1. Now we need to check if results in any other roundings.

1. Examine the two variable constraints involving x_1; these constraints are l_1, l_2 and l_6.

2. Neither l_1 nor l_2 force x_2 to be rounded.

3. Thus rounding x_1 does not force any other variables to be rounded; therefore no contradictory roundings are obtained.

4. Thus $x_1 = 1$ is a valid assignment for x_1.

We now check optional roundings, always attempting to round down before attempting to round up.

1. Round x_2 down to 0.

   (a) Examine the two variable constraints involving x_2; these constraints are l_1, l_2, l_3, and l_4.

(b) Since x_1 has already been rounded, we can ignore l_1 and l_2.

(c) l_3 forces x_3 to be rounded down to $x_3 = 1$.

(d) l_4 forces x_4 to be rounded down to $x_4 = 0$.

(e) We now check if rounding x_3 and x_4 requires any additional roundings, however there are no remaining unrounded variables so no additional roundings are performed.

(f) Observe that the constraint l_5 is violated by the current assignments to x_3 and x_4.

(g) Thus, x_2 cannot be rounded down.

2. Round x_2 up to 1.

(a) Examine the two variable constraints involving x_2; these constraints are l_1, l_2, l_3, and l_4.

(b) Since x_1 has already been rounded, we can ignore l_1 and l_2.

(c) l_3 does not force x_3 to be rounded.

(d) l_4 does not force x_4 to be rounded.

(e) Since no additonal roudings we performed, no constranits are violated by rounding x_2 up.

(f) Thus, $x_2 = 1$ is a valid assignment to x_2, given that $x_1 = 1$

3. Round x_3 down to 1.

(a) Examine the two variable constraints involving x_3; these constraints are l_3 and l_5.

(b) Since x_2 has already been rounded, l_3 can be ignored.

(c) Observe that l_5 forces x_4 to be rounded up to $x_4 = 1$.

(d) Checking for the roundings that result from rounding x_4, we find that there are no resultant roundings.

(e) No constraint is violated by the current assignments to x_3 and x_4 so they are valid.

(f) Thus $x_3 = 1$ and $x_4 = 1$ are valid assignments for x_3 and x_4, given the previous assignments $x_1 = 1$ and $x_2 = 1$.

4. We do not have to round x_4, since it has been made integral.

Now all variable have been given valid integer assignments. Thus the generated, valid, integer solution to the given system of equations is $x_1 = 1$, $x_2 = 1$, $x_3 = 1$ and $x_4 = 1$. This concludes the example.

Chapter 10

Conclusion

This paper introduced new algorithms for checking the linear and integer feasibility of a conjunction of UTVPI constraints. Our algorithms run in $O(m \cdot n)$ time and $O(m + n)$ space and are therefore optimal from the perspective of these resources. The claim of optimality follows from the fact that UTVPI constraints subsume difference constraints and that all known algorithms for difference constraint systems run in $O(m \cdot n)$ time and $O(m \cdot n)$ space. Additionally, our algorithms are certifying in that they produce a model, when the input instance is feasible and a refutation in the event that the input instance is infeasible. An important contribution of this paper is the characterization of linear and integer infeasibility in terms of the existence of certain paths and cycles in the appropriately constructed constraint network.

References

[1] S. K. Lahiri and M. Musuvathi, "An Efficient Decision Procedure for UTVPI Constraints," in *Proceedings of the 5^{th} International Workshop on the Frontiers of Combining Systems, September 19-21, Vienna, Austria*, New York, 2005, pp. 168–183, Springer.

[2] Robert Nieuwenhuis and Albert Oliveras, "Dpll(t) with exhaustive theory propagation and its application to difference logic.," in *CAV*, 2005, pp. 321–334.

[3] Antoine Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006.

[4] Patrick Cousot and Radhia Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.

[5] R. Gerber, W. Pugh, and M. Saksena, "Parametric dispatching of hard real-time tasks.," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 471–479, 1995.

[6] Inga Sitzmann and Peter J. Stuckey, "O-trees: A constraint-based index structure," in *Australasian Database Conference*, 2000, pp. 127–134.

[7] Dorit S. Hochbaum and Joseph (Seffi) Naor, "Simple and fast algorithms for linear and integer programs with two variables per inequality," *SIAM Journal on Computing*, vol. 23, no. 6, pp. 1179–1192, Dec. 1994.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.*, MIT Press, 2001.

[9] R. Rubinfield, *A Mathematical Theory of Self-checking, Self-testing and self-correcting Programs*, Ph.D. thesis, Computer Science Division, University of California, Berkeley, 1990.

[10] K. Mehlhorn and S. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press, Cambridge, 1999.

[11] K. Subramani, "On deciding the non-emptiness of 2SAT polytopes with respect to first order queries," *Mathematical Logic Quarterly*, vol. 50, no. 3, pp. 281–292, 2004.

[12] Peter Z. Revesz, "Tightened transitive closure of integer addition constraints," in *SARA*, 2009.

[13] J. Jaffar, M. J. Maher, P. J. Stuckey, and H. C. Yap, "Beyond Finite Domains," in *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, 1994.

[14] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella, "An improved tight closure algorithm for integer octagonal constraints," in *VMCAI*, 2008, pp. 8–21.

[15] W. Harvey and P. J. Stuckey, "A unit two variable per inequality integer constraint solver for constraint logic programming," in *Proceedings of the 20th Australasian Computer Science Conference*, 1997, pp. 102–111.

[16] G. B. Dantzig and B. C. Eaves, "Fourier-Motzkin Elimination and its Dual," *Journal of Combinatorial Theory (A)*, vol. 14, pp. 288–297, 1973.

[17] Andreas Schutt and Peter J. Stuckey, "Incremental satisfiability and implication for utvpi constraints," *INFORMS Journal on Computing*, vol. 22, no. 4, pp. 514–527, 2010.

[18] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli, "Abstract dpll and abstract dpll modulo theories," in *LPAR*, 2004, pp. 36–50.