Generalized Algorithms for Bounded Integer Programs
with Two Variables per Constraint

Reuven Bar-Yehuda and Dror Rawitz
Computer Science Department
Technion - IIT, Haifa 32000 Israel
e-mail: {reuven,rawitz}@cs.technion.ac.il
http://www.cs.technion.ac.il/~{reuven,rawitz}

May 11, 1998

Abstract

We discuss the problem of a bounded integer programs with \( n \) variables and \( m \) constraints, for which there are up to two variables per constraint. We present an \( O(mU) \) time feasibility algorithm for such integer programs (where \( U \) is the range size of the variables). We also present an \( O(mnlU) \) time 2-approximation algorithm, the correctness of which is proven using the local-ratio technique. Another result is an \( O(mU) \) time algorithm, which finds an optimal solution for a special case of integer programs with two variables per constraint, called monotone systems.

Our algorithms extend the concept of forced assignments to boolean variables, which is used in algorithms for 2SAT and min-2SAT, to forced ranges for bounded variables. This concept yields fast and simple algorithms.

We prove that all three algorithms work with generalizations of linear constraints called \( xy \)-convex constraints. We also show that the 2-approximation algorithm and the optimization algorithm for monotone systems work with monotone weight functions.

1 Introduction

1.1 Definitions for the Linear Integer Program

We discuss the problem of an integer program with bounded variables, over constraints with no more than two variables per constraint. We formulate the problem as:

\[
\text{(2VIP) } \min \sum_{i=1}^{n} w_i x_i \\
\text{s.t.} \quad a_k x_{i+k} + b_k x_{j_k} \geq c_k \quad \forall k \in \{1, \ldots, m\} \\
x_i \in [\ell_i, u_i] \quad \forall i \in \{1, \ldots, n\}
\]

where \( 1 \leq i_k, j_k \leq n, w_i \geq 0, a_k, b_k, c_k \in \mathbb{Z} \) and \( [\ell_i, u_i] \) is the set \( \{x \in \mathbb{Z} : \ell_i \leq x \leq u_i\} \). We denote \( U = \max_{i} \{u_i - \ell_i\} \) and \( w(x) = \sum_{i=1}^{n} w_i x_i \).

A vector \( x \) is called a feasible solution, if it satisfies all the constraints.

A feasible solution \( x^* \) is called an optimal solution, if every feasible solution \( x \) satisfies \( w(x^*) \leq w(x) \).

A feasible solution \( x \) is called an \( r \)-approximation, if \( w(x) \leq r \cdot w(x^*) \), where \( x^* \) is an optimal solution.
An algorithm is called an \( r \)-approximation algorithm if it always returns a \( r \)-approximation, when the system is satisfiable.

This problem is a generalization of the min-2SAT problem; if we define \( u = 1 \) all constraints can be rewritten as boolean constraints. Moreover, if \( \forall k a_k = b_k = c_k = 1 \), as well, we get the vertex cover problem, which is known to be NP-hard (see [5]). This implies that an \( r \)-approximation algorithm for 2VIP is also an \( r \)-approximation algorithm for vertex cover, thus, an \( r \)-approximation algorithm for a constant \( r < 2 \) will be a major breakthrough.

1.2 Results for the Linear Integer Program

In [4] Even, Itai and Shamir present a linear time feasibility algorithm for 2SAT instances. Their algorithm chooses an arbitrary variable, then discovers the forced values of other variables caused by either one of the two possible assignments for the chosen variable. By finding the impact of both assignments in parallel they achieve linear time complexity. Our algorithm uses the same concept, however, we are not dealing with forced values, but with forced ranges, e.g. if we know that \( x_1 \leq 2 \) and one of the constraints in the integer program is the inequality \( x_1 + x_2 \geq 5 \), we get that \( x_2 \geq 3 \). Our approach yields time complexity of \( O(mU) \). Another \( O(mU) \) algorithm for the feasibility problem of linear integer programs, based on an idea of Feder, appears in [8]. This algorithm first transforms the integer program into a 2SAT instance, and then uses the feasibility algorithm for 2SAT from [4].

In [9] Hochbaum and Naor presented an \( O(mn^2 \log m + nmU) \) time feasibility algorithm for integer programs with monotone linear inequalities with two variables per constraint, and an \( O(nmU^2 \log(n^2 U/m)) \) algorithm for finding an optimal solution for such integer programs. An inequality with two variables is called monotone if it is of the form \( ax_i - bx_j \geq c \) where \( a, b > 0 \). Lagarias in [10] proved that checking whether a monotone linear integer program has a feasible solution is NP-complete. The algorithm from [9] includes the construction of a graph representing the monotone integer program in question and the use of a maximum flow algorithm (such as the one in [6]) to find an optimal solution. We present an optimization algorithm for linear monotone systems and other systems. Our optimization algorithm for monotone systems works only for non-negative weights, but doesn’t involve flow. We use, again, the concept of checking the impact of a single variable’s range on the ranges of the variables. Using this concept we present an \( O(mU) \) optimization algorithm for monotone systems.

By using the algorithm from [9] Hochbaum, Megido, Naor and Tamir present in [8] an \( O(nmU^2 \log(n^2 U/m)) \) 2-approximation algorithm for linear integer programs with two variables per inequality, for which \( \ell = 0 \). Their algorithm transforms the integer program into a monotone linear integer program, and finds an optimal solution for the monotone system. When transferring the solution back to the original problem it becomes a solution consisting of integer multiples of \( \frac{1}{m} \), so the algorithm uses a feasible solution, found by the above mentioned \( O(mU) \) feasibility algorithm, to build a 2-approximation solution.

Gusfield and Pitt in [7] present an \( O(nm) \) time 2-approximation algorithm for the min-2SAT problem. Their algorithm uses the same concept of forced assignments as in [1]. The main idea of their algorithm is to check the impact of the two possible assignments for a variable, and to calculate the cost of the two choices. Then the algorithm reduces the total cost of the cheaper option from the weights of the variables included in the impact of both trials. We present an \( O(nmU) \) generalization of their approach. We use the local-ratio technique (see [2], [3], [1]) to prove the correctness of our algorithm. Note that the running time of our \( O(nmU^2 \log(n^2 U/m)) \) time 2-approximation algorithm dominates that of the \( O(nmU^2 \log(n^2 U/m)) \) time 2-approximation
algorithm from [8].

1.3 Generalizations

All the algorithms we present in this paper can be generalized to some non-linear systems, as well. We generalize the optimization algorithm for monotone systems and the 2-approximation algorithm to objective functions of the form $\sum_{i=1}^{n} w_i(x_i)$, where all the $w_i$'s are monotone functions. We also define a generalization of a linear inequality, called a xy-monotone constraint, and show that all three algorithms can be generalized to work with such constraints too.

1.4 Overview

The rest of the paper is organized as follows. In section 2 we present the $O(mU)$ time feasibility algorithm for linear integer systems. The $O(mU)$ time optimality algorithm for monotone linear systems is given in section 3. Section 4 includes the $O(nmU)$ time 2-approximation algorithm for linear integer systems. In section 5 we show that the algorithms work for the generalizations of the objective function and of the constraints.

2 Feasibility Algorithm

In this section we present an $O(mU)$ time algorithm which finds a feasible solution for a 2VIP instance, if such a solution exists. This algorithm generalizes the linear time feasibility algorithm for instances of 2SAT of Even, Itai and Shamir from [4].

2.1 The Impact of a Range

Assume that we are given an instance of 2VIP with the bounds $\ell$, $u$ and there exists the following constraint: $ax_i + bx_j \geq c$ for $a, b > 0$. This inequality implies that $x_j \geq \left\lceil \frac{c-ax_i}{b} \right\rceil \geq \left\lceil \frac{c-u\ell}{b} \right\rceil$. Thus, we may get a new lower bound for the variable $x_j$ (see example in Figure 1).

If $\left\lceil \frac{c-u\ell}{b} \right\rceil > u_j$, there is no feasible solution within the given bounds $u, \ell$. However, if $u_j > \left\lceil \frac{c-u\ell}{b} \right\rceil > \ell_j$ we can update $\ell_j$ and then check if there are variables affected by the change of $\ell_j$. Other kind of inequalities influence upper bounds or lower bounds, as well. We call the process of updating the bounds, starting from the range of one variable, the impact of the
routine - impact \((\ell, u, i)\):

For each constraint involving \(x_i\) and another variable \(x_j\):

1. Use the constraint and the range of \(x_i\) to update the range of \(x_j\).

2. If \(u_j < \ell_j\)
   then return 'fail'.

3. If \(\ell_j\) or \(u_j\) changed
   then call \(\text{Impact} (\ell, u, j)\), if \(\text{Impact}\) fails return 'fail'.

Figure 2: Routine Impact

range. If we get \(u_j < \ell_j\) for a variable \(x_j\) during this process, then a feasible solution satisfying the original bounds does not exist.

The routine in Figure 2, called \(\text{Impact}\), is the basis of all the algorithms presented in this paper. This routine receives as input the two arrays \(\ell\) and \(u\) of size \(n\) and an index \(i\). \(\text{Impact}\) calculates the impact of \(u_i\) and \(\ell_i\) on the bounds \(\ell, u\) of all the variables.

We now prove that we do not lose feasible solutions after activating \(\text{Impact}\).

Lemma 1: If \(y\) is a feasible solution, which satisfies the bounds given to \(\text{Impact}\) as input, then it satisfies the updated bounds.

Proof: We enumerate the changes in the bounds \(u\) and \(\ell\): \(u^0, \ell^0\) are the bounds given as input to \(\text{Impact}\) and \(u^t, \ell^t\) are the bounds after \(t\) changes. Assuming the contrary, we examine the first change in the bounds, for which a feasible solution \(y\) was lost. In this change, denoted by \(k\), the range of a variable \(x_i\) had an impact on the range of a variable \(x_j\), due to a constraint.

In other words, the range of \(x_i\), which includes \(y_i\), forced a range for \(x_j\), which does not include \(y_j\). This implies that \(y_i\) and \(y_j\) don't satisfy the constraint, in contradiction to the feasibility of \(y\).

2.2 The Algorithm

Assume we are given the bounds \(\ell, u\) for the variables, and we want to reduce the \(i\)th range.

We can choose an integer \(a \in [\ell_i, u_i]\) and call \(\text{Impact}\) twice, once to calculate the impact of the range \([a, u_i]\) and then to calculate the impact of the range \([\ell_i, a - 1]\). If both calls yield a conflict, no feasible solution exists. Otherwise, if one of the calls succeeds, we can use \(\text{Impact}\) again on the resulting ranges to reduce another range. To perform this task we present a routine called \(\text{Bisect}\) in Figure 3.

The following Lemma proves the consistency of \(\text{Bisect}\):

Lemma 2: If there exists a feasible solution, which satisfies the bounds \(u\) and \(\ell\) given as input to \(\text{Bisect}\), and \(\text{Impact}\) succeeds when checking the impact of a reduced range, then there is at least one feasible solution which satisfies the resulting bounds of \(\text{Impact}\).

Proof: Assume that \(\text{Impact}\) succeeds. We denote the bounds given as input to \(\text{Bisect}\) by \(u\) and \(\ell\), and the resulting bounds from a successful execution of \(\text{Impact}\) by \(u'\) and \(\ell'\). We also denote by \(y\) a feasible solution for the instance of 2VIP, which satisfies \(u\) and \(\ell\). We define a
Routine - Bisect($\ell, u$):

1. Choose a variable $x_i$, for which $\ell_i < u_i$.
2. Choose a value $\alpha \in [\ell_i + 1, u_i]$.
3. $\ell^1 \leftarrow \ell, u^1 \leftarrow u$.
4. $u^1_i \leftarrow \alpha - 1$.
5. Call Impact ($\ell^1, u^1, i$).
6. $\ell^2 \leftarrow \ell, u^2 \leftarrow u$.
7. $u^2_i \leftarrow \alpha$.
8. Call Impact ($\ell^2, u^2, i$).
9. If Impact fails both times then return 'unsatisfiable'.
10. Choose a successful run of Impact and replace the bounds $\ell, u$ by the updated ranges of this run.

We will show that $y'$ is a feasible solution. First we examine a constraint on the variables $x_i, x_j$, for which $(y_i, y_j) \in [\ell_i', u_i'] \times [\ell_j', u_j']$. This constraint is satisfied, because $y$ is a feasible solution. Other constraints include at least one variable $x_i$, for which $y_i < \ell_i'$ or $y_i > u_i'$. In this case $y_i' = \ell_i'$ or $y_i' = u_i'$, so not satisfying this constraint implies that when Impact changed a lower or upper bound of $x_i$ to $y_i'$ it should have excluded $y_i'$ from the range available for $x_j$. By the definition of $y'$ this is not true, thus $y'$ is a feasible solution.

**Theorem 1:** A feasible solution for 2VIP can be found in time $O(mU)$.

**Proof:** Lemma 2 proves that if Bisect starts with the bounds $u$ and $\ell$, for which there exists a feasible solution, and adds a new bound, which excludes all feasible solutions, then Impact fails. This means that if Impact succeeds, we can use Impact on the reduced ranges again. This is the main idea of the feasibility algorithm. We start with the original bounds, and our goal is to reduce the ranges for the $x_i$’s. We do this by running Bisect repeatedly until Impact fails or until we get $\ell = u$. Achieving this when all constraints are satisfied gives us a solution $x = \ell = u$.

We first prove that this algorithm recognizes unsatisfiable instances. Every iteration of the algorithm shrinks one of the ranges by at least one. If, in routine Bisect, both runs of Impact fail, the algorithm outputs 'unsatisfiable'. Assuming the contrary, we get a solution $x = u = \ell$. As there is no feasible solution, at least one of the constraints is not satisfied, so when the
range of one of the variables in the constraint was updated, the range of the second variable should have been reduced not to include the value assigned to it by the algorithm.

To achieve time complexity of $O(mU)$, we run both calls to Impact (made in Bisect) in parallel, and prefer the faster option of the two, if a choice exists. This approach was used in [4] by Even, Itai and Shamir to construct a linear time feasibility algorithm for instances of 2SAT by finding the impact of the two possible assignments to a boolean variable.

After every change in the range for $x_i$, we have to check the $m_i$ constraints involving this variable, in order to discover the impact of the change. As $x_i$ can be changed up to $u_i$ times, we conclude that the total time complexity of the changes is $O(\sum_{i=1}^{n} m_i(u_i - \ell_i)) = O(mU)$ (the time wasted on unfinished trials is bounded by the time complexity of the chosen trials).

$\square$

3 Optimization Algorithm for Monotone Systems

In this section we study a special case of a bounded integer program with two variables per constraint called a monotone system. We start with the definition of the system:

**Definition 1:** A monotone inequality is an inequality on two variables with coefficients of opposite signs. A linear system with two variables per inequality is called monotone if all the constraints in the system are monotone.

We formulate the following monotone system:

\[
\begin{align*}
\text{(M2VIP)} \quad & \min \sum_{i=1}^{n} w_i x_i \\
\text{s.t.} \quad & a_k x_{i_k} - b_k x_{j_k} \geq c_k \quad \forall k \in \{1, \ldots, m\} \\
\quad & x_i \in [\ell_i, u_i] \quad \forall i \in \{1, \ldots, n\}
\end{align*}
\]

where $1 \leq i_k, j_k \leq n$, $w_i \geq 0$, $a_k, b_k \in \mathbb{N}$ and $c_k \in \mathbb{Z}$.

In the following theorem we present an algorithm, which outputs an optimal solution for a monotone system (M2VIP), if the system is satisfiable, otherwise it recognizes the system as unsatisfiable. Our algorithm uses the fact that in monotone systems, when calculating the impact of a bound on a variable, a lower bound induces lower bounds, and an upper bound induces upper bounds. For example, a constraint in a monotone 2SAT instance is of the form $x_i \geq x_j$, so if we don’t have forced assignments for some of the variables, we get that $x = 0$ satisfies all constraints of the above form.

**Theorem 2:** An optimal solution for M2VIP can be found in time $O(mU)$.

**Proof:** We describe an algorithm for finding an optimal solution for a monotone system. The algorithm starts with the initial bounds, and then it calls routine Impact with the input $u, \ell, i$ for every $i \in \{1, \ldots, n\}$. If one of the calls of Impact fails, the algorithm outputs 'unsatisfiable', otherwise it outputs the vector $x = \ell$.

From Lemma 1 we get that if one of the calls of Impact fails, then no feasible solution exists. The Lemma also implies that if the instance of M2VIP is satisfiable then the above algorithm terminates without a failure of Impact. We prove that $x = \ell$ is a feasible solution. Assuming the contrary means that one of the monotone constraints involving two variables $x_i$ and $x_j$ is not satisfied by $x = \ell$. In this case, Impact should have changed $\ell_i$, w.l.o.g., after
updating \( \ell_j \). Note that even though we implicitly made the assignment \( u \leftarrow \ell \), we don’t have to worry about changes in \( u \) influencing \( \ell \) because of the monotone nature of the constraint. From Lemma 1 we get that any feasible solution \( y \) for this instance of M2VIP has to satisfy \( y \geq \ell \), which means that \( \ell \) is an optimal solution (and the only optimal solution if \( \forall_i w_i > 0 \)).

As stated before, changing the range of \( x_i \) might cause changes in the ranges of other variables. The existence of a constraint on \( x_i \) and another variable \( x_j \) makes \( x_j \) a candidate for a range update. This means that we have to check the \( m_i \) constraints involving \( x_i \), to discover the consequences of changing its range each time this range changes. Because \( x_i \) can be changed up to \( u_i - \ell_i \) times, we get that the time complexity of finding an optimal solution for a monotone system is \( O(\sum_{i=1}^n m_i(u_i - \ell_i)) = O(mU) \).

4 2-approximation Algorithm

In this section we present an \( O(nmU) \) time 2-approximation algorithm for 2VIP, when \( \ell = 0 \). Our algorithm generalizes the \( O(nm) \) time 2-approximation algorithm for min-2SAT presented by Gusfield and Pitt in [7]. Our 2-approximation algorithm is, in fact, a special case of our feasibility algorithm, namely, the 2-approximation algorithm chooses variables and bounds in a specific order to get a 2-approximation.

The local-ratio technique (see [2],[3],[1]) uses weight reductions. Using weight reductions directly will be time consuming, so we first define an extension of the objective function \( w(x) \):

**Definition 2:** For \( x \in \mathbb{N}^n \) and \( \overline{\ell}, \overline{\pi} \in \mathbb{R}^n \) we define:

\[
W(x, \overline{\ell}, \overline{\pi}) = \sum_{i=1}^n \Delta(x_i, \overline{\ell}_i, \overline{\pi}_i)
\]

where

\[
\Delta(x_i, \overline{\ell}_i, \overline{\pi}_i) = \begin{cases} 
(x_i - \overline{\ell}_i) w_i, & \overline{\ell}_i \leq x_i \leq \overline{\pi}_i \\
(\overline{\pi}_i - \overline{\ell}_i) w_i, & x_i \geq \overline{\pi}_i \\
0, & x_i \leq \overline{\ell}_i
\end{cases}
\]

Intuitively \( W(x, \overline{\ell}, \overline{\pi}) \) is the extra cost of \( x \) after we have already paid for \( \overline{\ell} \), but we are willing to pay just up to \( \overline{\pi} \). Obviously \( w(x) = W(x, 0, u) \).

Using this extension of the weight function, we can also define an extension of the original problem, using the new weight function and the same constraints. Note that for every \( \overline{\ell}, \overline{\pi} \) and \( x \), we get:

\[
W(x, \overline{\ell}, u) = W(x, \overline{\ell}, \overline{\pi}) + W(x, \overline{\pi}, u)
\]

This is our way to get the weight decomposition needed for the local-ratio technique.

Our 2-approximation algorithm uses an array denoted by \( \overline{\ell} \), to store the values of \( x_i \)'s, paid for. As in the feasibility algorithm our aim is to reduce the ranges of the variables until we get \( \ell = u \), thus \( \overline{\ell} \) must satisfy \( \ell \leq \overline{\ell} \leq u \) in all stages of the algorithm. Note that in some stage of the algorithm, it may be that, for now, we paid only for a part of the cost required in order to get to the next integer level, thus \( \overline{\ell} \in \mathbb{R}^n \).

The heart of the algorithm is the routine \textit{Bisect2} (Figure 4). \textit{Bisect2} like \textit{Bisect} , first checks the impact of the two ranges \( [\ell_i, \alpha - 1] \) and \( [\alpha, u_i] \) for a variable \( x_i \) and a value \( \alpha \in [\ell_i + 1, u_i] \). Then \textit{Bisect2} calculates the cost of both options, and chooses the less costly option (if both possibilities are feasible). It updates \( \ell \) and \( u \) according to the cheaper option. Then, it raises \( \ell \) in order to pay for part of the impact to the lower bounds of the more costly option, and for the raise in \( \ell \), due to the cheaper option.
Routine - Bisect2(ℓ, u, ℓ):

1. Choose a variable \( x_i \), for which \( \ell_i < u_i \).
2. \( \alpha \leftarrow \left[ \frac{1}{2} (\ell + u) \right] \).
3. \( \ell_1 \leftarrow \ell, u_1 \leftarrow u \).
4. \( u_1 \leftarrow \alpha - 1 \).
5. Call Impact \((\ell_1, u_1, i)\).
6. If Impact succeeds
   then \( \text{Cost}_1 \leftarrow W(\ell_1, \ell, u) \)
   else \( \text{Cost}_1 = \infty \).
7. \( \ell_2 \leftarrow \ell, u_2 \leftarrow u \).
8. \( \ell_2 \leftarrow \alpha \).
9. Call Impact \((\ell_2, u_2, i)\).
10. If Impact succeeds
    then \( \text{Cost}_2 \leftarrow W(\ell_2, \ell, u) \)
    else \( \text{Cost}_2 = \infty \).
11. If \( \text{Cost}_1 < \text{Cost}_2 \)
    then \( C_{\text{min}} = \text{Cost}_1, \ell_{\text{min}} = \ell_1, u_{\text{min}} = u_1 \) and \( C_{\text{max}} = \text{Cost}_2, \ell_{\text{max}} = \ell_2, u_{\text{max}} = u_2 \).
    else \( C_{\text{min}} = \text{Cost}_2, \ell_{\text{min}} = \ell_2, u_{\text{min}} = u_2 \) and \( C_{\text{max}} = \text{Cost}_1, \ell_{\text{max}} = \ell_1, u_{\text{max}} = u_1 \).
12. If \( C_{\text{min}} = \infty \)
    then return 'unsatisfiable'.
13. If \( C_{\text{max}} < \infty \)
    then let \( \overline{\pi} \) be an array, for which \( \forall i \overline{\pi}_i \leq \overline{\pi}_i \leq \min \{ \ell_{\text{max}}, \ell_i \} \), and \( W(\overline{\pi}, \ell, u) = C \).
14. \( \forall i \overline{\pi}_i \leftarrow \max \{ \overline{\pi}_i, \ell_{\text{min}} \} \).
15. \( \overline{\pi} \leftarrow \overline{\pi} \).
16. \( \ell \leftarrow \ell_{\text{min}} \).
17. \( u \leftarrow u_{\text{min}} \).

Figure 4: Routine Bisect2
Lemma 3: Bisect2 is a specific implementation of Bisect.

Proof: As mentioned earlier, Bisect2 chooses a variable \( x_i \) and a value \( \alpha \in [\ell_i, u_i] \), and checks the consequences of the bounds \( [\alpha, u_i] \) and \( [\ell_i, \alpha - 1] \) in every iteration. It simply doesn’t make an arbitrary choice between the two possible new bounds. It tries both, and makes a decision based on those trials (if it has a choice). \( \square \)

Lemma 4: When using \( W(\cdot, \bar{r}, \bar{\pi}) \) as an objective function, for \( \bar{r} \) and \( \bar{\pi} \) just before step 15 in Bisect2, every feasible solution is a 2-approximation.

Proof: We consider the system with the objective function \( W(\cdot, \bar{r}, \bar{\pi}) \). In Bisect2, we first raised \( \bar{\pi} \) to get \( W(\bar{\pi}, \bar{r}, u) = C \). This means that we pay \( C \) for part of the impact of the more expensive choice. Then we assign \( \bar{u}_i \leftarrow \max \{\bar{u}_i, \ell_i^{\min}\} \) for all \( i \in \{1, \ldots, n\} \), which ensures that we pay \( C \) for the impact of the cheaper option, as well. Thus, all feasible solutions with regard to \( W(\cdot, \bar{r}, \bar{\pi}) \) cost at least \( C \), because all feasible solutions must fit into one of the two options. This implies that an optimal solution \( x^* \) satisfies:

\[
W(x^*, \bar{r}, \bar{\pi}) \geq C
\]

and on the other hand, for every feasible solution \( x \):

\[
W(x, \bar{r}, \bar{\pi}) \leq W(\bar{\pi}, \bar{r}, \bar{\pi}) \leq 2C
\]

which means that any feasible solution is a 2-approximation. \( \square \)

The next Lemma proves this paper’s version of the Decomposition Observation from [1]:

Lemma 5: (Decomposition Observation)

\[
Opt(W(\cdot, \bar{r}, \bar{\pi})) + Opt(W(\cdot, \bar{\pi}, u)) \leq Opt(W(\cdot, \bar{r}, u))
\]

Proof: As mentioned earlier, any feasible solution \( x \) satisfies:

\[
W(x, \bar{r}, u) = W(x, \bar{r}, \bar{\pi}) + W(x, \bar{\pi}, u)
\]

We denote by \( x^* \) an optimal solution for the system with regard to \( W(\cdot, \bar{r}, \bar{\pi}) \), by \( y^* \) an optimal solution with regard to \( W(\cdot, \bar{\pi}, u) \), and by \( z^* \) an optimal solution with regard to \( W(\cdot, \bar{r}, u) \).

\[
W(x^*, \bar{r}, \bar{\pi}) + W(y^*, \bar{\pi}, u)) \leq W(z^*, \bar{r}, \bar{\pi}) + W(z^*, \bar{\pi}, u)) \leq W(z^*, \bar{r}, u)
\]

the Lemma follows. \( \square \)

Theorem 3: A 2-approximation for 2VIP can be found in time \( O(nmU) \).

Proof: First we present the 2-approximation algorithm. We start the algorithm with the original bounds. Then (much like the optimization algorithm for monotone systems) it calls Impact with input \( \ell, u, i \) for every \( i \in \{1, \ldots, n\} \). If one of the calls of Impact fails, the algorithm outputs 'unsatisfiable'. If all calls of Impact succeed, the algorithm assigns \( \bar{r} \leftarrow \ell \). The rest of the algorithm consists of calls to Bisect2. Each call to Bisect2 involves two invocations of Impact, one for the range \( [\alpha, u_i] \) and the other for the range \( [\ell_i, \alpha - 1] \), where \( \alpha = \lfloor \frac{1}{2}(\ell_i + u_i) \rfloor \). As in the feasibility algorithm, our goal is to reduce the ranges for the variables. We do this
by running \textit{Bisect}2 repeatedly until \textit{Impact} fails or until we get \( \ell = u \). Achieving this when all constraints are satisfied gives us a solution \( x = \ell = u \).

We prove the correctness of the algorithm by first showing that it outputs a feasible solution, then by showing that the solution is actually a 2-approximation.

From Lemma 1 we get that the first stage of the algorithm doesn’t exclude any feasible solution, and from Lemma 3 we get that the rest of the algorithm is simply a specific implementation of the feasibility algorithm. Thus, we get that if no feasible solution exists the algorithm outputs ‘unsatisfiable’. On the other hand, if a feasible solution does exist, the algorithm outputs a feasible solution.

We prove by induction on the iterations of the algorithm (as we reduce the ranges in each iteration, there are a finite number of iterations). There are two possible scenarios for a run of \textit{Bisect}2. In the first, one of the bounds is found infeasible, and in the other, both bounds are feasible. If just one of the trials is feasible then the corresponding bounds are the only bounds possible, so, as in the feasibility algorithm, we continue with these bounds. In this case we assign \( \ell \leftarrow \ell \). Note that by Lemma 2 this is our only option. Thus, the changes in \( \ell \) and \( \ell \) are local optimizations.

On the other hand, if both trials are feasible, \textit{Bisect}2 calculates the total cost of the two choices. The total cost of a choice includes the cost of changing the lower bounds of the variables affected by the new bound. In such cases \textit{Bisect}2 does two things: updates \( \ell \) and updates \( \ell \). We prove that the first is a local 2-approximation and the second is a local optimization.

As in Lemma 4, we consider \( \ell \) and \( \pi \) just before step 15 in \textit{Bisect}2. We denote by \( x^* \) an optimal solution for the system with regard to \( W (\cdot, \ell, \pi) \), by \( y^* \) an optimal solution with regard to \( W (\cdot, \pi, u) \), and by \( z^* \) an optimal solution with regard to \( W (\cdot, \ell, u) \).

\[
W(x, \ell, u) = W(x, \ell, \pi) + W(x, \pi, u) \quad \text{[by definition]}
\leq W(\pi, \ell, \pi) + W(x, \pi, u) \quad \text{[by monotony of } W(\cdot, \ell, \pi)\text{]}
\leq 2W(x^*, \ell, \pi) + 2W(y^*, \pi, u) \quad \text{[Lemma 4 and the induction hypothesis]}
\leq 2W(z^*, \ell, \pi) + 2W(z^*, \pi, u) \quad \text{[optimality of } x^* \text{ and } y^*\text{]}
\leq 2W(z^*, \ell, u) \quad \text{[by definition]}
\]

The last two inequalities can be explained directly by the Decomposition Observation.

Thus, we conclude that raising \( \ell \) to \( \pi \) is a local 2-approximation. We pay up to \( 2C \) and lower the optimum by at least \( C \).

We now prove that the assignment \( \ell \leftarrow \ell_{\min} \) in routine \textit{Bisect}2 is a local optimization. We show that after the change in \( \ell \), an optimal solution for the chosen option costs no more than an optimal solution for the second option. We look at a feasible solution for the second option denoted by \( y \). This solution can be changed to:

\[
y'_i = \begin{cases} y_i, & \ell_{\min} \leq y_i \leq u_{\min} \\
\ell_{\min}, & \ell_{\min} \leq y_i < \ell_{\min} \\
u_{\min}, & y_i > u_{\min} 
\end{cases}
\]

By using the same arguments as in Lemma 2 we conclude that \( y' \) is a feasible solution. Moreover, \( y' \) is cheaper than \( y \). For \( y_i \geq \ell_{\min} \), \( y'_i \) costs at most as \( y_i \), because for these values \( y'_i \leq y_i \). If \( y_i < \ell_{\min} \), then \( y'_i = \ell_{\min} \), but by the assignment made in step 15 in \textit{Bisect}2, we get that \( y'_i = \ell_{\min} \) is free of charge.

In order to get the required time complexity, \textit{Bisect}2 must choose the \( x_i \)'s wisely. One possibility is to choose the variables in increasing order, i.e. \( x_1, x_2, \ldots \), then after invoking \textit{Bisect}2 on \( x_n \) to start from the beginning. We call \( n \) calls of \textit{Bisect}2 on all \( n \) variables a
pass. From the same arguments as in theorem 2, we get that the time complexity of a single invocation of 
\( \text{Bisect2} \) is \( O(mU + n) = O(mU) \). One pass may involve all \( n \) variables, so its time complexity of one pass is \( O(nmU) \). By choosing \( \alpha = \left[ \frac{1}{2}(\ell_i + u_i) \right] \) in \( \text{Bisect2} \), we reduce the range of \( x_i \) at least by half. This means that in a single pass we reduced the possible ranges for all the variable at least by half. So we get that the total time complexity is no greater than:

\[
\sum_{k=1}^{\log U} O(mU \frac{k}{2}) = O(mnU)
\]

### 5 Generalizations for Non-Linear Systems

#### 5.1 Definitions

The following definition generalizes a linear weight function for a variable \( x_i \):

**Definition 3**: A non-negative weight function \( w \) is called a monotone weight function if

\[
w(\alpha) \leq w(\alpha + 1) \quad \text{for} \quad \alpha \in [\ell_i, u_i - 1].
\]

For the rest of this paper we assume that monotone weight functions are continues over the reals.

**Example 1**: Here is an example of an objective function consisting of monotone weight functions:

\[
w(x) = \sum_{i=1}^{3} w_i(x_i)
\]

for

\[
w_1(x_1) = e^{x_1}
\]

\[
w_2(x_2) = \begin{cases} 4, & x_2 < 4 \\ x_2, & 4 \leq x_2 < 7 \\ 7, & 7 \leq x_2 \end{cases}
\]

\[
w_3(x_3) = \pi \cdot x_3
\]

The next definition generalizes a linear constraint:

**Definition 4**: A \( xy \)-monotone constraint \( p \) on two variables is a constraint which satisfies:

\[
\forall x, x_1, x_2 : x_1 \leq x \leq x_2, p(x, y) \geq p(x_1, y) \cdot p(x_2, y)
\]

\[
\forall y, y_1, y_2 : y_1 \leq y \leq y_2, p(x, y) \geq p(x, y_1) \cdot p(x, y_2)
\]

and that between every two points \((x_1, y_1), (x_2, y_2)\) for which the constraint is satisfied there is a connecting path of points for which the constraint is satisfied.

We assume that for a constraint \( p(x, y) \) we have an \( O(1) \) time oracle, which gives us a tight range on \( x \), given a range for \( y \), and vice versa. Note that an \( O(1) \) oracle can be constructed easily for linear constraints. In Figure 5 you can find examples of \( xy \)-monotone constraints.

We formulate the generalized integer programs as:

\[
\text{G2VIP} \quad \min W(x) = \sum_{i=1}^{n} w_i(x_i)
\]

s.t.

\[
p_k(x_i, x_{j_k}) \quad \forall k \in \{1, \ldots, m\}
\]

\[
x_i \in [\ell_i, u_i] \quad \forall i \in \{1, \ldots, n\}
\]
Figure 5: Examples of $xy$-monotone constraints

where $1 \leq i_k, j_k \leq n$, the $w_i$’s are monotone weight functions, and the $p_k$’s are $xy$-monotone constraints.

The next definition is a generalization of a monotone linear constraint:

**Definition 5**: A monotone constraint $p$ is a $xy$-monotone constraint on two variables $x, y$, for which a change in the lower bound of one of the variables may change only the lower bound of the second variable, and a change in the upper bound of one of the variables may change only the upper bound of the second variable (constraint no. 1 in Figure 5 is a monotone constraint). A system with only monotone constraints is called a generalized monotone system.

A monotone constraint generalizes the definition of a monotone inequality, as an inequality on two variables with coefficients of opposite signs, from [8].

We formulate the following generalized monotone system:

$$\text{(GM2VIP)} \quad \min W(x) = \sum_{i=1}^{n} w_i(x_i)$$

\[ \text{s.t.} \]

$$p_k(x_{i_k}, x_{j_k}) \quad \forall k \in \{1, \ldots, m\}$$

$$x_i \in [\ell_i, u_i] \quad \forall i \in \{1, \ldots, n\}$$

where $1 \leq i_k, j_k \leq n$, the $w_i$’s are monotone weight functions and the $p_k$’s are monotone constraints.

The definitions of a feasible solution, an optimal solution, a $r$-approximation and a $r$-approximation algorithm are generalized in a straightforward manner.

### 5.2 Generalizing the Algorithms

In this section we prove that the three algorithms for linear integer programs, presented in this paper, work for the generalized integer programs, as well. It is easy to see that Lemma 1 works for generalized instances too. On the other hand, in order to generalize the feasibility algorithm and the 2-approximation algorithm we have to prove that Lemma 2 works, as well:

**Lemma 6**: Lemma 2 can be generalized for instances of G2VIP.
Figure 6: The ranges for $x_i, x_j$, when $y_k > u'_i$ and $y_j > u'_j$.

**Proof:** Using the same notations of Lemma 1, we need to prove that the vector $y'$ is a feasible solution. The vector $y'$ was defined as:

$$y_i' = \begin{cases} y_i, & l'_i \leq y_i \leq u'_i \\ l'_i, & y_i < l'_i \\ u'_i, & y_i > u'_i \end{cases}$$

As in Lemma 1, a constraint on the variables $x_i, x_j$, for which $l'_i \leq y_i \leq u'_i$ and $l'_j \leq y_j \leq u'_j$ is satisfied, because $y$ is a feasible solution. Other constraints include at least one variable $x_i$, for which $y_k$ is not in the range $[l'_i, u'_i]$.

We examine a constraint on the variables $x_i, x_j$ for which $y_k > u'_i$ and $y_j > u'_j$. This means that $y'_k = u'_i$ and $y'_j = u'_j$ (see Figure 6). Due to the tightness of the bounds produced by the oracle there must exist a value $z_i$, for which the pair $(z_i, u'_i)$ satisfies the constraint. By the same argument there must exist a value $z_j$, for which the pair $(u'_i, z_j)$ satisfies the constraint.

The constraint is xy-monotone, so the points $(y_k, y_j)$ and $(z_i, u'_i)$ must be connected by a route of feasible points. This route starts in point $(y_k, y_j)$ and goes down or to the left, but never up or to the right. If the route traverses through $(y'_k, y'_j)$, we are done. Otherwise, this route must traverse above $(y'_k, y'_j)$. This puts $(y'_k, y'_j)$ between two points which satisfy the constraint.

Thus, because the constraint is xy-monotone, we get that $(y'_k, y'_j)$ satisfies it, as well. Other positions of the point $(y_k, y_j)$ can be resolved similarly.

In order to extend Theorem 3 to instances of G2VIP, for which $\forall_i u_i(\ell_i) \geq 0$, we have to prove:

**Lemma 7:** Bisect2 can be generalized to work with monotone weight functions.

**Proof:** Working with monotone weight functions means that the differences between integer values of a variable are not of uniform height. Thus, if we redefine $\Delta(x_i, \ell_i, \pi_i)$ as:

$$W(x, \ell, \pi) = \sum_{i=1}^{n} \Delta(x_i, \ell_i, \pi_i)$$
\[ \Delta(x_i, t_i, \bar{i}) = \begin{cases} 
 w_i(x_i) - w_i(t_i), & t_i \leq x_i \leq \bar{i} \\
 w_i(\bar{i}) - w_i(t_i), & x_i \geq \bar{i} \\
 0, & x_i \leq t_i 
\end{cases} \]

does not work as is.

From the above discussion we get:

**Theorem 4:** Theorems 1 and 3 can be generalized for instances of G2VIP and theorem 2 can be generalized for instances of GM2VIP.

**Remark 1:** If \( x_i \in \{s_{i,1}, \ldots, s_{i,n_i}\} \), we can define a new variable \( x'_i \in \{0, \ldots, n_i\} \) and a new monotone weight function \( w'_i(s_i, a) = w_i(s_i, a) \). And we get that the same results hold for \( t_i = 0 \) and \( u_i = n_i \).

**Acknowledgment**

We would like to thank Avigail Orni for her careful reading and suggestions.

**References**


