INTEGER PROGRAMMING WITH 2-VARIABLE EQUATIONS AND 1-VARIABLE INEQUALITIES

MANUEL BODIRSKY, GUSTAV NORDH, AND TIMO VON OERTZEN

Abstract. We present an efficient algorithm to find an optimal integer solution of a given system of 2-variable equalities and 1-variable inequalities with respect to a given linear objective function. Our algorithm has worst-case running time in $O(N^2)$ where N is the number of bits in the input.

1. Introduction

We present an efficient algorithm to find an *optimal integer solution* of a given system of 2-variable equalities and 1-variable inequalities with respect to a given linear objective function. More precisely, the input consists of

- a finite set of variables $x_1, \dots, x_n$,
- equations of the form $ax + by = c$ where x and y are variables, and a, b, c are rational numbers,
- inequalities of the form $x \leq u$ or $x \geq l$, where x is a variable and u, l are rational numbers, and
- a linear objective function $\sum_{i=1}^{n} w_i x_i$ where the the w_i's are rational numbers.

The task is to find an assignment of *integer values* to the variables $x_1, \dots, x_n$ such that all equations and inequalities are satisfied and the function $\sum_{i=1}^{n} w_i x_i$ is maximized.

If instead of 2-variable equalities we are given 2-variable *inequalities*, then the problem obviously becomes NP-hard (this can be seen by a reduction from the maximum independent set problem). If instead of 2-variable equalities we are given 3-variable equalities, then the problem again becomes NP-hard. This follows by a trivial reduction from the NP-complete problem 1-in-3-SAT [7], where each 1-in-3 clause 1-in-3(x, y, z) is reduced to $x + y + z = 1$, $x, y, z \geq 0$. Hence, 2-variable equalities and 1-variable inequalities is a maximal tractable class in the sense that allowing longer equalities *or* longer inequalities results in NP-hard problems.

We remark that finding integer solutions to linear equation systems *without inequalities* is tractable [4], and has a long history. Faster algorithms have been found in for example [2, 10]. Integer programming can also be solved in polynomial time if the *total* number of variables is two [6]; Lenstra [5] has generalized this result to any fixed finite number of variables. See [3] for one of the fastest known algorithms for 2-variable integer programming, and for more references about linear programming in two dimensions.

In our algorithm for a system of 2-variable equalities and 1-variable inequalities over the integers we use the fact that such equation systems reduce to systems that define a one-dimensional solution space. This idea was also used by Aspvall and Shiloach [1] in their algorithm for solving systems of 2-variable equations over the *rational numbers.* We use this fact to compute an *optimal* integer solution by solving a system of modular equations in polynomial time. One of our contributions is to show that the necessary computations can be performed in total quadratic time in the input size.

Since the problem to decide whether a single 2-variable equation $ax+by=c$ with $a,b,c \in \mathbb{Z}$ has an integer solution for x,y is equivalent to deciding whether the gcd of a and b is a divisor of c, we cannot expect an algorithm for our problem that is faster than gcd computations. There are sub-quadratic algorithms for computing the gcd of two N bit integers with running times in $O(log(N)M(N))$, where $M(N)$ is the bit-complexity of multiplying two N bit integers [8]. Using the classical Schönhage Strassen [9] integer multiplication algorithm, this gives a running time for gcd in $O(N(log(N))^2 log(log(N)))$.

We view it as an interesting open problem whether integer programming over 2-variable equalities and 1-variable inequalities can be shown to be no harder than gcd computations, i.e., with a running time of $O(G(N))$ where $G(N)$ is the bit complexity of computing gcd of two N bit integers. We also emphasize that the quadratic time algorithm we give does not rely on sub-quadratic algorithms for multiplication, division, or gcd.

2. Preliminaries

The (representation-) size of an integer a is the length of its binary encoding. The size of the input is denoted by N throughout this paper and is defined to be the sum of the sizes of all the numbers in the input. Addition and subtraction are computable in time $O(log(b))$ time on input $(a,b) \in \mathbb{Z}^2$ where $a \leq b$. Multiplication, division, and gcd are computable in $O(log(a)log(b))$ time on input $(a,b) \in \mathbb{Z}^2$.

The following simple lemma is useful in the complexity analysis of our algorithm.

Lemma 1. *Given a tuple $(a_1,\dots,a_n) \in \mathbb{Z}^n$ and binary functions $f_1,\dots,f_m$ from $\mathbb{Z}^2$ to $\mathbb{Z}$, where each f_i is computable in $O(log(a)log(a'))$ time on any input $(a,a') \in \mathbb{Z}^2$. Then, any n-ary function $h(x_1,\dots,x_n)$ from $\mathbb{Z}^n$ to $\mathbb{Z}$ expressible in terms of the functions $f_1,\dots,f_m$ with the restriction that no argument x_i appears more than once in this arithmetic expression can be computed in time $O((\sum_{i=1}^n log(a_i))^2)$ on input $(a_1,\dots,a_n)$.*

Proof. The proof is by induction on n, the arity of the function $h(x_1,\dots,x_n)$. The basis $n=2$ holds since each function f_i is computable in $O(log(a_1)log(a_2))$ time on input (a_1,a_2) and $log(a_1)log(a_2) \leq (log(a_1)+log(a_2))^2$. Let $T(a_1,\dots,a_n)$ denote the time it takes to compute h on input $(a_1,\dots,a_n)$ and use $T(a_1,\dots,a_n) \leq (\sum_{i=1}^n log(a_i))^2$ as induction hypothesis. Now, given $h(x_1,\dots,x_{n+1}) = f_i(h(x_1,\dots,x_j),h(x_{j+1},\dots,x_{n+1}))$, then

$$T(a_1,\dots,a_{n+1}) \leq T(a_1,\dots,a_j) + T(a_{j+1},\dots,a_{n+1}) + log(a_1\dots a_j)log(a_{j+1}\dots a_{n+1})$$

$$\leq \left(\sum_{i=1}^{j} log(a_i)\right)^2 + \left(\sum_{i=j+1}^{n+1} log(a_i)\right)^2 + 2\left(\sum_{i=1}^{j} log(a_i)\right)\left(\sum_{i=j+1}^{n+1} log(a_i)\right) = \left(\sum_{i=1}^{n+1} log(a_i)\right)^2$$

□

3. Reduction to an acyclic system

We show how to partition the system of equations into independent subsystems, each having a one-dimensional solution space (i.e., the solution space can be expressed using one free parameter). The *graph* of an instance of our problem is the graph that has a vertex for each variable and an edge for each equation (connecting the two vertices corresponding to the variables in the equation). A instance of our problem is called an *acyclic* (or *connected*) system if the graph of the system is acyclic (or *connected*, respectively), and a *connected component* of a system F is a subsystem of F whose graph is a connected component of the graph of F.

Proposition 2. *There is an $O(N^2)$ time algorithm that computes for a given system of two variable linear equations an equivalent acyclic subsystem.*

Proof. If the system is not connected, then it can be split into its connected components that can be treated independently. So we assume in the following that the system is connected.

If the system is acyclic, then we are done. Hence, assume that the graph contains at least one cycle. Note that a system of equations corresponding to a cycle on n vertices has n variables and n equations, which means that if no equation in the cycle is redundant, then the system has at most one solution. Arbitrarily choose a variable x (with the goal of expressing every other variable in terms of x) and express it as $x = 1x + 0$. The propagation to the rest of the graph is done as follows. If a variable y has been expressed as $y = a'x + c'$ and there is an equation $ay + bz = c$, then express z as $z = a''x + c''$ where $a'' = -aa'/b$ and $c'' = (c - ac')/b$. Since the graph contains at least one cycle, we will get two expressions for the same variable. For example, assume that z is already expressed as $z = a'''x + c'''$ (as a result of a previous propagation step) when the expression $z = a''x + c''$ is computed above, then subtracting these expressions results in $0 = (a''' - a'')x + (c''' - c'')$.

Now, three cases occur.

(1) If $a''' = a''$ and $c''' \neq c''$ then there is an inconsistency and the system has no solution.
(2) If $a''' \neq a''$, then $x = -(c''' - c'')/(a''' - a'')$. This implies that the system has at most one solution, because it is connected. The existence of a solution can be checked by propagating the value of x in the original system of equations.
(3) Otherwise, $a''' = a''$ and $c''' = c''$, and hence the equation $ay + bz = c$ is redundant. In this case the equation is removed from the system, and we continue expressing variables in terms of x.

Hence, the process above will either terminate with (1) the conclusion that the system has no solution satisfying the unary inequalities, (2) the unique solution satisfying the unary inequalities, or (3) an equivalent acyclic subsystem.

To see that the algorithm above runs in quadratic time, observe that all multiplication and division operations involved, with the exception of the division in case (2) above, take one of its two arguments, say a_i, directly from the input. Hence, leaving out the division in case (2) above, the cost of all these operations is bounded by $O(\sum_{i=1}^{n}(log(a_1 \dots a_{i-1}a_{i+1} \dots a_n)log(a_i))$ $\leq O(N \sum_{i=1}^{n} log(a_i)) = O(N^2)$, where $a_1, \dots, a_n$ are the numbers in the input (i.e., $N = \sum_{i=1}^{n} log(a_i)$). Now, the division in case (2) above is in $O(N^2)$ and need only be performed at most once for each independent system of equations since in this case we know that the system has at most one solution. In this case, by a similar argument as above, the propagation step to determine the solution is also performed in time $O(N^2)$. □

The final step in this section is to translate the upper and lower bounds on the variables into an upper and lower bound on the parameter x. If y has the upper bound u and the expression for y is $y = ax + c$, then in case that a is positive we get the bound $\lfloor(\lfloor u \rfloor - c)/a\rfloor \geq x$, and in case a is negative we get the bound $\lceil(\lfloor u \rfloor - c)/a\rceil \leq x$. Lower bounds l on y are treated analogously. After translating all bounds we can obtain the strongest upper bound and lower bound on x, denoted u^* and l^* respectively (obviously, if $u^* < l^*$, then there is no solution).

4. Expression for the solution space in terms of one free parameter

In this section we start with an acyclic connected system of equations and compute a one parameter expression for the solution space over $\mathbb{Z}$. Assume for the sake of presentation that the coefficients a, b, c in all equations $ax+by=c$ are integer. This is without loss of generality since every equation can be brought into this form by multiplying both sides of the equation by the product of the denominators of a, b, and c. This can clearly be done in quadratic time and increases the bit size of the input by at most a constant factor.

Check that each individual equation $ax+by=c$ has integer solutions. Recall that a Diophantine equation of the form $ax+by=c$ has integer solutions if and only if $gcd(a,b)|c$. Simplify the equations by dividing a, b, and c by $gcd(a,b)$. In the resulting system we now have $gcd(a,b)=1$ for each equation $ax+by=c$.

Proposition 3. *There is an $O(N^2)$ time algorithm for solving acyclic connected systems of two variable equations over the integers.*

Proof. We perform a depth-first search on the graph of the system, starting with any variable x from the system. The goal is to find an expression for the solution space of the form $x \equiv s \pmod{t}$. That is, the assignment $x := i$ can be extended to an integer solution to the entire system if and only if $i \equiv s \pmod{t}$.

If we enter a variable y in the DFS and y has an unexplored child z, then continue recursively with z. If z is a leaf in the tree, meaning that there is a unique equation $ay+bz=c$ where z occurs, then rewrite the equation as $ay \equiv c \pmod{b}$. Note that an assignment $y := i$ can be extended to a solution of $ay+bz=c$ if and only if $ai \equiv c \pmod{b}$. Compute the multiplicative inverse a^{-1} of $a \pmod{b}$ (which exists since $gcd(a,b)=1$ and can be retrieved from the gcd computation). The congruence above can now be rewritten as $y \equiv c' \pmod{b}$ where $c' = ca^{-1}$. If all children of y have been explored, then y is explored and we backtrack.

If v is the parent of y through the equation $dv+ey=f$, then rewrite the equation using the congruence $y \equiv c' \pmod{b}$, into $dv+e(c'+kb)=f$ which is equivalent to $dv+ebk=f-ec'$. Check that $gcd(d,eb)|(f-ec')$ (otherwise there is no solution and we reject) and divide d, eb, and $f-ec'$ by $gcd(d,eb)$ giving the equation $d'v+e'k=f'$ with $gcd(d',e')=1$. Rewrite this equation as the congruence $d'v \equiv f' \pmod{e'}$ which in turn is rewritten as $v \equiv f'' \pmod{e'}$ by multiplying both sides with the multiplicative inverse of $d' \pmod{e'}$ (which again exists since $gcd(d',e')=1$).

Suppose that v already has an explored child (with congruence $v \equiv c \pmod{b}$) when we have finished exploring another child of v, giving rise to the congruence $v \equiv c' \pmod{b'}$. We then combine these congruences in a similar fashion as discussed above already twice (by computing greatest common divisors and multiplicative inverses).

The result (if a solution exists) is a congruence $v \equiv c'' \pmod{b''}$, which replaces the two old congruences, and we continue the depth first search.

To see that the algorithm runs in quadratic time, note that the depth first search in the tree described above is nothing else than an evaluation of an arithmetic expression involving addition, subtraction, multiplication, division, and gcd computations. The fact that the arithmetic expression in our graph representation is acyclic guarantees that no argument appears more than once in the arithmetic expression (in the sense of Lemma 1). Hence, since all the operations involved are computable in $O(log(a)log(b))$ time on arguments $(a,b) \in \mathbb{Z}^2$, we can apply Lemma 1 to get the desired bound of $O(N^2)$. □

5. Computing an optimal solution

We combine the results from the previous sections and obtain the desired result.

Theorem 4. *There is an algorithm that computes the optimal solution of a given integer program with 2-variable equalities and 1-variable inequalities in $O(N^2)$ time, where N is the number of bits in the input.*

Proof. Note that we trivially get an optimal solution to the original problem by computing an optimal solution to each connected component of the system (if it exists; if there exists a component that has no feasible solution, we return 'no feasible solution' and terminate).

We are given the expression $x \equiv s \pmod{t}$ for the solution space from the previous section and the upper and lower bounds on x, u^* and l^* respectively, from Section 3. Since $x \equiv s \pmod{t}$ is a one parameter expression for the solution space, all solutions lie on a line in $\mathbb{R}^n$. Since the goal function is linear and all solutions lie on a line, the goal function is increasing along one direction of this line and decreasing in the other direction, unless the goal function is orthogonal to the solution space, in which case all solutions are equally good and we return any solution.

We evaluate the goal function on the solutions we get by assigning u^* and l^* to x, respectively. Assume u^* gives the best value for the goal function, meaning that the goal function is increasing for increasing x. The largest value of $x \leq u^*$ satisfying $x \equiv s \pmod{t}$ is $s + t\lfloor (u-s)/t \rfloor$. We propagate this assignment to the rest of the system to get the optimal solution to our original problem. Again it should be clear (by similar reasoning as in the previous sections) that the necessary computations can be done in $O(N^2)$ time. $\square$

References

[1] B. Aspvall and Y. Shiloach. A fast algorithm for solving systems of linear equations with two variables per equation. *Linear Algebra and its Applications*, 34:117–124, 1980.

[2] T. Chou and G. Collins. Algorithms for the solution of systems of linear diophantine equations. *SIAM J. Comput.*, 11(4):687–708, 1982.

[3] F. Eisenbrand and S. Laue. A linear algorithm for integer programming in the plane. *Math. Program.*, 102(2):249–259, 2005.

[4] R. Kannan and A. Bachem. Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix. *SIAM J. Comput.*, 8(4):499–507, 1979.

[5] H.W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[6] H. E. Scarf. Production sets with indivisibilities. part ii: The case of two activities. *Econometrica*, 49:395–423, 1981.

[7] T. Schaefer. The complexity of satisfiability problems. In *STOC*, pages 216–226, 1978.

[8] A. Schönhage. Schnelle berechnung von kettenbruchentwicklungen. *Acta Inf.*, 1:139–144, 1971.

[9] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7:281–292, 1971.

[10] A. Storjohann and G. Labahn. Asymptotically fast computation of hermite normal forms of integer matrices. In *ISSAC*, pages 259–266, 1996.

École Polytechnique, LIX (CNRS UMR 7161), Palaiseau, France
E-mail address: bodirsky@lix.polytechnique.fr

École Polytechnique, LIX (CNRS UMR 7161), Palaiseau, France
E-mail address: nordh@lix.polytechnique.fr

Max Planck Institute for Human Development, Berlin, Germany
E-mail address: vonoertzen@mpib-berlin.mpg.de