Solving Linear Programs Using Multiparty Computation

Tomas Toft[1,2] [⋆]

[1] CWI Amsterdam, The Netherlands
[2] Dept. of Mathematics and Computer Science, TU/e, The Netherlands

Abstract. Solving linear programming (LP) problems can be used to solve many different types of problems. Immediate examples include certain types of auctions as well as benchmarking. However, the input data may originate from different, mistrusting sources, which implies the need for a privacy preserving solution.
We present a protocol solving this problem using black-box access to secure modulo arithmetic. The solution can be instantiated in various settings: Adversaries may be both active and adaptive, but passive and/or static ones can be employed, e.g. for efficiency reasons. Perfect security can be obtained in the information theoretic setting (up to 1/3 corruptions), while corruption-of-all-but-one is possible in the cryptographic setting. The latter allows a two-party protocol.
The solution is based on the well known simplex method. Letting n denote the number of initial variables and m the number of constraints, each pivot requires only $\mathcal{O}(\text{loglog}(m))$ rounds in which $\mathcal{O}(m(m+n))$ multiplication protocols and $\mathcal{O}(m+n)$ comparison protocols are invoked; this is equivalent to the base-algorithm. A constant-rounds variation is also possible, this increases the number of comparisons to $\mathcal{O}(m^2+n)$.
Key words: Multiparty computation, Secure collaboration, Linear programming

1 Introduction

Multiparty computation (MPC) allows mutually mistrusting parties to jointly perform a computation without revealing their data. It is natural to consider applying such techniques to solving linear programming (LP) problems. The applications are not limited to simply managing resources, though this is of course possible. They span diverse topics, including benchmarking and auctions.

One application of an MPC LP-solver is the relative performance payscheme of Nielsen and Toft [NT07], where employers wish to motivate employees through bonus schemes based on perfomance compared to the competition, i.e. through benchmarking. But no company is willing to share production costs and other

[⋆] Work partially performed at Aarhus University. Supported by the research program Sentinels (`http://www.sentinels.nl`). Sentinels is being financed by Technology Foundation STW, the Netherlands Organization for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs.

trade secrets with their competitors. This paper provides an efficient instantiation of the primitives needed in [NT07].

Another example is multi-attribute auctions that allow bidding to not only reflect price, but also to take quality into account. In reverse auctions, for instance, one party offers a contract to perform some task, say to construct a road with some minimal specification. Bidders supply their price for doing so, and the cheapest offer is chosen. However, the parties also have private information regarding the cost of, and willingness to pay for, higher quality. Bogetoft and Nielsen have constructed such auctions using LP problems [BN08].

It seems highly likely that economics may provide many other interesting applications. Informally, the *revelation principle* states that for games of incomplete information, there is a revelation mechanism, where it optimal for the players to truthfully supply their preferences (to a "trusted third party" who computes the optimal solution.) With the present work it is simply a question of designing mechanisms as LP problems.

Related Work: There are multiple algorithms for solving LP problems. The ellipsoid and interior point methods due to Khachiyan and Karmarkar respectively [Kha79,Kar84] allow LP problems to be solved in polynomial time. Formulating one of these as a Boolean circuit and applying one of the classic results of MPC, [Yao86,GMW87,BGW88,CCD88], provides a solution. Applying a general solution is expensive, though. The evaluation of every Boolean gate consists of executing some cryptographic primitive, i.e. *every* basic arithmetic operation performed consists of *many* invocations of those primitives. Moreover, constructing a round-efficient solution in the information theoretic setting is not immediate.

Li and Atallah have proposed a solution to solving LP problems [LA06]. They consider two honest-but-curious parties who learn a maximizing assignment to the variables. In addition, they provide ad-hoc checks against malicious behaviour, e.g. verifying that the assignment does not violate any constraints. This may limit the damage possible, but does not rule out malicious behaviour in general.

The solution is based on integer computation (additive sharing and public key cryptography) and simplex, where pivots are performed on a permuted (masked) tableau. Termination is tested publicly implying that the number of iterations is leaked. Complexity is $\mathcal{O}(k(m^2 + mn + \ell(m + n)) + \ell m(n + m))$[3] modular exponentiations, where k is the number of iterations, ℓ the bit-length of the numbers involved, m the number of constraints, and n the number of initial variables. There are, however, a number of problems with Li and Atallah's solution.

The index of the final variable entering the basis is leaked. This is easily fixed by randomizing the tableau at the end. A second problem occurs when the output is determined. Li and Atallah first output the basis, then set the basis variables to their respective values. Seeing only a maximizing solution, however, may reveal less information, e.g. it should not be possible to distinguish between

[3] [LA06] considers n to be the overall number of variables, this is $m + n$ here.

basic variables set to zero and co-basic variables. It is highly questionable that this leaks any useful information in practice. Yet, there is no guarantee that this is the case, so the issue must at least be considered before applying the solution.

A more significant problem is that the protocol may provide an incorrect result. The secure computation may also leak information in this case. The problem occurs at the end of the computation. Basis variables are sought out by determining columns with only one non-zero position – doing this blindly does not work. Consider maximizing $f(x_1, x_2) = x_1 + x_2$ under the constraints that $x_1, x_2 \geq 0$ and $c_1x_1 + c_2x_2 + x_S = 1$, where x_S is a slack-variable (larger examples are easily constructed.) If $c = c_1 = c_2$, then the final state is

$$\begin{array}{cccc} c & c & 1 & 1 \\ 0 & 0 & c & 1 \end{array}$$

which assigns $1/c$ to both x_1 and x_2, which violates the constraint. With some work it should be possible to determine a maximizing assignment, which doesn't violate constraints. However, leakage is a problem: in the above example both parties learn that $c_1 = c_2$.

A final issue is that it is infeasible to perform more than a few pivots. Each pivot potentially doubles the bit-lengths needed to represent the values of the tableau. As the computation is secure, we must not learn the actual bit-lengths, implying that we must always work with full-size numbers. Starting with 32-bit values, after ten iterations these have grown to 32 thousand bits. After twenty they have increased to 32 million. Thus, even for small inputs their basic operations soon become modular exponentiations with a million-bit modulus.

Two related problems are those of Distributed Constraint Satisfaction Problems (DisCSP) and Distributed Constraint Optimization Problems (DCOP). Multiple parties wish to find some optimizing assignment to variables but have private constraints/priorities regarding the solution space. Though seemingly similar, these problems are quite different. First, we consider arbitrarily distributed data. Second and more importantly, their constraints are not linear. Moreover, though DisCSP and DCOP literature state privacy as important, most work focuses solely on efficiency. Full scale MPC solutions for DisCSPs and DCOPs has been considered, e.g. by Silaghi et al. [Sil04,SFP06]. The basic idea in that work is to test the entire search space (with improvements for certain types of problems) and pick a random solution. This exhaustive approach *could* be applied to LP problems, however, with large, rational-valued search spaces this is clearly infeasible.

Contribution: A solution for solving LP problems is constructed based on black-box, secure modular arithmetic along with additional sub-protocols, e.g. for comparison. The notion can be formalized, e.g. using the arithmetic black-box of Damgård and Nielsen [DN03]. Using abstract primitives allows different instantiations. Information theoretic security is possible, e.g. based on Shamir sharing and the protocols of Ben-Or et al. [Sha79,BGW88]. So is computational security, say through threshold Paillier encryption [Pai99,DJ01,CDN01]. Both provide solutions in the multiparty setting and ensure security against active adversaries. The cryptographic solution also allows a two-party version.[4]

As with [LA06], the solution is based on simplex, however, rather than masking the tableau (which is difficult in the present setting), the pivots are performed obliviously. A second difference is that the possibility of cycling – never terminating – is considered here. Though it is rarely encountered in practice, it ensures that termination can be guaranteed. Cryptographic protocols may be applied as building blocks in many settings, and it should not be assumed that the input is "real-world data." Similar to Li and Atallah, information on the number of pivots performed is leaked. Means for reducing the leak (hiding the actual number of pivots) are considered, but as simplex may require an exponential number of pivots, without exponential work there can be no 100% guarantees.

The problems of the protocol of Li and Atallah do not occur in this work. Moreover, the solutions presented in this paper may also be applied there. Finally, the present protocol has slightly better complexity – $\mathcal{O}(m(n+m))$ secure multiplications and $\mathcal{O}(n+m)$ secure comparisons are needed per pivot. With a Paillier based solution, one multiplication is comparable to a constant number of exponentiations, while comparisons are equivalent to $\mathcal{O}(\ell)$ exponentiations; the $\ell m(n+m)$ term is eliminated. With secret sharing, computational complexity improves further, as we are working directly on (shares of) the actual values rather than encryptions – secure multiplications are implemented using modular multiplication rather than exponentiation.

Round complexity takes a hit in the present solution, though. Where [LA06] provides a constant-rounds pivot, here $\mathcal{O}(\text{loglog}(m))$ rounds are needed. A constant round version is possible, but this increases complexity to $\mathcal{O}(n+m^2)$ secure comparisons.

Finally, rather than providing the output, here the result is shared in the end, implying that it may be used in further computation. Moreover, in difference to Li and Atallah, the present solution provides full precision, i.e. it is guaranteed that there are no rounding errors. Note that computing an approximation rather than the exact result may also introduce security problems as demonstrated by Feigenbaum et al., [FIM$^+$06]. It seems unlikely, that this will pose a problem for "real-world input," though.

Overview: Section 2 of this paper describes the cryptographic primitives and introduces notation. Section 3 gives a brief overview of the simplex method. Section 4 introduces notation and primitives for shared arrays. In the remaining sections, the details of the simplex protocol are presented. Section 5 specifies the overall task and contains general remarks. Following this, Sect. 6 describes the implementation of the body of the simplex algorithm using MPC primitives. Sections 7 and 8 considers iteration, termination, and obtaining the result in the end. Finally Sect. 9 considers the required size of the modulus, while Sect. 10 provides concluding remarks.

[4] [CDN01] can be modified to provide security against malicious behaviour from all but one parties, though naturally termination cannot be guaranteed in this case.

2 Cryptographic Preliminaries and Notation

Integer arithmetic can be simulated with arithmetic modulo some M, as long as M is chosen sufficiently large, i.e. such that no overflow occurs. Secure integer computation can therefore be simulated using linear, cryptographic primitives also allowing multiplication (through a protocol). This can be instantiated with secret sharing (e.g. Shamir sharing over $\mathbb{Z}_q$) or (threshold) homomorphic public key cryptography (e.g. Paillier encryption.) Both examples imply that M only has large prime-factors, which is required below.

Negative values such as $-a$ will be represented in the natural way, i.e. as $M - a \in \mathbb{Z}_M$. This ensures that both addition and multiplication involving negative values work as desired. We specify no further details regarding the scheme, though we do require that the multiplication protocol is constant-rounds[5] as well as a some form of composition theorem allowing both sequential and parallel execution. This implies that information leaks do not occur in sub-protocols – it can only occur when information is intentionally revealed.

We use terminology from secret sharing, writing secret values in square brackets, $[a]$. Secure computation will be described using an infix notation with

$$[c] \leftarrow [a] \cdot [b]$$

denoting a run of the multiplication protocol on shared inputs a and b; the resulting shares are stored in c. This eases readability and is easily translated to protocol executions and local computation.

The basic measure for communication complexity will be the number of secure multiplications performed. As the underlying primitives are linear, additions and multiplications by public values do not require interaction, and are considered costless. The number of rounds refer to the number of messages transmitted during the entire protocol. This can only be specified through invocations of primitives. However, by assuming that sharing, multiplication, and reconstruction is constant-rounds, the difference will only be off by a constant.

In order to present the protocol a few additional primitives are needed; these can be constructed from the above. First off, shared bits, $[b] \in \{0, 1\}$, are needed. Such values will be used repeatedly, e.g. for conditional selection.

$$[b] \mathbin{?} [a] : [a']$$

selects either $[a]$ (if $[b]$ is one) or $[a']$ (if $[b]$ is zero) and is a shorthand of

$$[b] \cdot ([a] - [a']) + [a'].$$

Protocols for comparison of shared values are required, with $[a] \overset{?}{<} [a']$ denoting a protocol run resulting in a shared bit, $[a < a']$. This can be realised in constant-rounds, [DFK+06,NO07].[6] An equality test is defined analogously; in both cases the number of multiplications is linear in the bit-lengths of the inputs.

[5] This restiction is for complexity analysis alone.

[6] [DFK+06,NO07] are easily modified to consider negative inputs.

3 Simplex

The simplex method is a well-known strategy for solving linear programs. Despite an exponential worst-case complexity, it is efficient in practice and used in a large range of applications. This section will contain a brief overview of (the steps of) the algorithm. For a full description as well as explanation of the details, the reader is referred to [Chv83]. As the basic primitives described in the previous section involve only integer computation, a variation of simplex using integer pivoting is considered. This technique, generally attributed to Edmonds, is described in detail by Rosenberg in [Ros05].

A linear program consists of n variables, $x_1, \dots, x_n \geq 0$, and m constraints:

$$\sum_{i=1}^{n} c_{j,i} \cdot x_i \leq b_j \text{ for } j \in \{1, \dots, m\}.$$

The goal is to maximise an objective function, f:

$$f(x_1, \dots, x_n) = \sum_{i=1}^{n} f_i \cdot x_i,$$

where the $c_{j,i}$ and f_i are integers in this work. In order to maximize f, slack-variables, $x_{n+1}, \dots x_{n+m} \geq 0$ are introduced, resulting in equalities in the constraints:

$$x_{n+j} + \sum_{i=1}^{n} c_{j,i} x_i = b_j \text{ for } j \in \{1, \dots, m\}.$$

A solution will be an assignment to the x_i such that the constraints hold. In simplex, solutions allowing only m variables to be positive are constructed, this is known as the basis. Each of these variables are associated with a constraint. All other variables (the co-basis) take the value 0. The execution of the algorithm starts by considering an initial solution with the basis consisting of the slack-variables (taking the values b_j).[7] By repeatedly moving variables in and out of the basis, the problem is rephrased and the solution improved (the value obtained when evaluating f increased) with every iteration until one maximising f is found.

Consider the tableau form of a linear programming problem:

$$\begin{array}{ccc|c|c}
c_{1,1} & \dots & c_{1,n} & 1\,0\,0 \dots 0\,0 & b_1 \\
c_{2,1} & \dots & c_{2,n} & 0\,1\,0 \dots 0\,0 & b_2 \\
 & \dots & & \dots & \dots \\
c_{m,1} & \dots & c_{m,n} & 0\,0\,0 \dots 0\,1 & b_m \\
\hline
-f_1 & \dots & -f_n & 0\,0\,0 \dots 0\,0 & z = 0
\end{array}$$

where column i, $1 \leq i \leq m + n$, is associated with variable x_i, and row j, $1 \leq j \leq m$ is associated with constraint j.

[7] It is assumed that the linear program is origin-feasible, i.e. setting $x_i = 0$ for all i does not violate any constraint. This can be avoided with standard techniques.

For the tableau, the right-most column (except the bottom row), will be known as the b-vector, similarly the bottom row (except the right-most element), will be called the f-vector; note $f_{n+1}, \dots, f_{n+m}$ initially set to zero. The sub-matrix of the constraints consisting of the columns of the slack-variables is initialised to the identity matrix. The columns of the tableau associated with the basis variables will always be the columns of I multiplied by a positive integer, p' (the previous pivot element, initially 1, see below). The basis variable of the current solution associated with the j'th constraint (row) takes the value b_j/p' in the solution. The value z in the bottom right-hand corner is the objective function, f, evaluated at the current solution and multiplied by p', initially $f(0, \dots, 0) = 0$.

The integer pivoting variation of the simplex algorithm repeatedly updates the tableau through the following steps:

I. Determine a column, C, with a negative value in the f-vector. The (guaranteed co-basic) variable associated with C is chosen to enter the basis. C will be referred to as the *pivot column.*
II. Determine a row, R, such that its intersection with C, C_R, is positive (constraining) and b_R/C_R is minimal. This row is called the *pivot row*, the element C_R is called the *pivot element.* The basic variable associated with R is the one selected for leaving the basis; after the current iteration the basic variable associated with row R will be the one associated with C.
III. Multiply all entries in the non-pivot rows by the pivot element.
IV. Subtract a multiple of the pivot row from all non-pivot rows such that the updated pivot column will consist entirely of zeros except for the pivot row.
V. Divide all non-pivot rows by the *previous* pivot element (which is initialised to 1)
VI. If one or more negative values in the f-vector exist, go to step I.

One problem with simplex is that it may cycle indefinitely. Though rarely encountered in practice, the issue must be considered. Fortunately it is easily handled using Bland's rule: when confronted with a choice of entering or leaving variable, pick the one with the lowest index, see e.g. [Chv83]. This implies that the index of the variable associated with a row (constraint) must be stored along with it and updated once the entering and leaving variables have been determined.

4 Secret Shared Arrays

Secret shared arrays will be referred to using boldface and capital letters,

$$[\mathbf{A}] = \big([a_1], [a_2], \dots, [a_k]\big),$$

this is convenient for denoting multiple related, sharings. Indexing is written $[\mathbf{A}]\,(i)$ meaning $[a_i]$. Finally, the length of a shared array will be written $[\mathbf{A}]$.len.

Expressions of the form $[\mathbf{A}]\,([i])$ are needed. Shared indexes will be stored as unary counters, arrays consisting of all 0's except for the i'th position, which is

1. These will be written in boldface to denote that in essence, they are arrays. Indexing is now simply the computation of a dot product:

$$[\mathbf{A}]([\mathbf{i}]) = \sum_{j=1}^{[\mathbf{A}].\text{len}} [\mathbf{A}](j) \cdot [\mathbf{i}](j),$$

with assignments, $[\mathbf{A}]([\mathbf{i}]) \leftarrow [x]$, translating to updating *every* entry of $[\mathbf{A}]$,

$$[\mathbf{A}](j) \leftarrow [\mathbf{i}](j) \mathbin{?} [x] : [\mathbf{A}](j).$$

Both require $[\mathbf{A}]$.len secure multiplications, which may be performed in parallel.

The integer value of an index, val($[\mathbf{i}]$), may be computed as $\sum_{j=1}^{[\mathbf{i}].\text{len}} [\mathbf{i}](j) \cdot j$. General indexing – computing $[\mathbf{A}]([i])$ where $[i]$ is a field element – is possible by transforming $[i]$ to $[\mathbf{i}]$, [RT07]. Complexity remains linear and constant-rounds.

The notation introduced for arrays will also be used for *shared matrices*. Simplex continuously updates a matrix, $[\mathbf{T}]$, representing the tableau-form of the problem. Indexing is done using two variables; $[\mathbf{T}](r, c)$ denotes the entry in the r'th row, c'th column. The r'th row (c'th column) of $[\mathbf{T}]$ is written $[\mathbf{T}](r, \cdot)$ ($[\mathbf{T}](\cdot, c)$). Viewing columns (rows) as separate arrays allows shared indexes.

Two high-level protocols are also required, first a prefix-or computation on arrays of shared bits, $[\mathbf{B}]$. The goal is $[\mathbf{B}']$, with $[\mathbf{B}'](j) = \vee_{i=1}^{j} [\mathbf{B}](i)$ for $1 \leq j \leq [\mathbf{B}]$.len. A constant-rounds solution using $\mathcal{O}([\mathbf{B}].\text{len})$ multiplications is described in [DFK$^+$06], this is denoted $\text{pre}_\vee(\cdot)$. The second requirement is a computation of the minimal element of an array of length k, along with its index. This is possible using $\mathcal{O}(k)$ comparisons in $\mathcal{O}(\text{loglog}(k))$ rounds, as well as $\mathcal{O}(k^2)$ comparisons in $\mathcal{O}(1)$ rounds. Details are available in Appendix A. Note that the result is valid for *any* comparison operator on *any* data.

5 Privacy Preserving Simplex

Assume that a LP problem with n variables $x_1, \ldots, x_n \geq 0$, m constraints $\sum_{i=1}^{n} c_{j,i} \cdot x_i \leq b_j$, and objective function $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} f_i \cdot x_i$ is provided in the form of sharings of the values of the constraints and terms of f,

$$[c_{1,1}], \ldots, [c_{m,n}], [b_1], \ldots, [b_m], [f_1], \ldots, [f_n].$$

The problem is assumed to be bounded, i.e. there exists an optimal solution. If this is not the case, it can be detected underway. The goal is an assignment to the x_i maximizing f without violating any constraint. This information can be extracted from the b-vector of the final tableau and the final pivot element.

In the following, m will always denote the number of constraints with n signifying the number of initial variables; overall there are $m + n$ variables. The tableau matrix, $[\mathbf{T}]$, has $m + 1$ rows and $m + n + 1$ columns. The f-vector and b-vector will be denoted $[\mathbf{F}]$ and $[\mathbf{B}]$ respectively, while $[\mathbf{S}]$ of length m stores the indexes of variables associated with the constraints, i.e. the basis. In the current solution $x_{[\mathbf{S}](j)}$ takes the value $\frac{[\mathbf{B}](j)}{[p']}$, where $[p']$ is the most recent pivot element.

6 Translating the Body of Simplex

Translation of the simplex-iteration will be done by considering each of the five steps individually. It is assumed that output from previous steps is available.

Step I, Determining the variable to enter the basis. This step consists of determining the pivot column by computing the minimal index, $[\mathbf{c}]$, such that $[\mathbf{F}]([\mathbf{c}])$ is negative. This is clearly a candidate selected using Bland's rule. The required computation is seen as Protocol 1.

Protocol 1 Selecting the pivot column

Input: The tableau, $[\mathbf{T}]$ (including $[\mathbf{F}]$ by definition).

Output: $[\mathbf{c}]$ and $[\mathbf{C}]$, such that $[\mathbf{F}]([\mathbf{c}])$ is negative, $[\mathbf{c}]$ is minimal, and $[\mathbf{C}]$ is a copy of the $[\mathbf{c}]$'th column of $[\mathbf{T}]$.

```
   for $i \leftarrow 1, \dots, n+m$ do
       $[\mathbf{D}](i) \leftarrow [\mathbf{F}](i) \overset{?}{<} 0$
   end for
   $[\mathbf{D}'] \leftarrow \mathrm{pre}_{\vee}([\mathbf{D}])$
5: $[\mathbf{c}](1) \leftarrow [\mathbf{D}'](1)$
   for $i \leftarrow 2, \dots, n+m$ do
       $[\mathbf{c}](i) \leftarrow [\mathbf{D}'](i) - [\mathbf{D}'](i-1)$
   end for
   $[\mathbf{C}] \leftarrow [\mathbf{T}](\cdot, [\mathbf{c}])$
```

For correctness, note that $[\mathbf{D}](i)$ is 1 if x_i associated with the i'th column is a candidate for entering the basis, otherwise it is 0. $[\mathbf{D}'](i)$ is 1 iff $[\mathbf{D}](i') = 1$ for some $i' \leq i$. Hence, $[\mathbf{c}](i) = 1$ exactly for the smallest i with $[\mathbf{D}](i) = 1$ and 0 otherwise, i.e. of the form required. $[\mathbf{C}]$ is correct by construction.

Regarding complexity, the initial loop performs $n + m$ comparisons. This is followed by a prefix-or of length $m + n$ and a costless loop. The indexing needed to compute $[\mathbf{C}]$ is equivalent to $m + 1$ indexings into arrays of length $n + m$; overall this is $\mathcal{O}(m \cdot (n + m))$ multiplications. For round complexity, note that the body of the initial loop does not depend on previous iterations, thus they may be executed concurrently. This implies $\mathcal{O}(1)$ rounds overall.

Step II, Determining the variable to leave the basis. The goal is to determine the tightest constraint on the variable, $x_{[\mathbf{c}]}$, i.e. find $[\mathbf{r}] \in \{1, 2, \dots, m\}$ such that

$$[\mathbf{C}]([\mathbf{r}]) \cdot x_{[\mathbf{c}]} = [\mathbf{B}]([\mathbf{r}])$$

implies the smallest, non-negative value of $x_{[\mathbf{c}]}$. In addition to this, (copies of) the pivot row and the pivot element, $[\mathbf{R}]$ and $[p] = [\mathbf{C}]([\mathbf{r}])$, must also be obtained.

$[\mathbf{C}](j) \leq 0$ implies that constraint j does not limit $x_{[\mathbf{c}]}$, thus, only constraints with $[\mathbf{C}](j) > 0$ are relevant; these will be called *applicable*, the rest *non-applicable.* The primary goal is the index, $[\mathbf{r}]$, of an applicable constraint with the *rational* value $\frac{[\mathbf{B}]([\mathbf{r}])}{[\mathbf{C}]([\mathbf{r}])}$ minimal; ties are broken using Bland's rule.

Overall, this step consists of defining a comparison operator on constraints. The protocol for computing a minimal entry of an array noted in Sect. 4 does the rest. Three values of each constraint are needed: For the j'th constraint $[\mathbf{B}](j)$ and $[\mathbf{C}](j)$ define the constraint and whether it is applicable, while $[\mathbf{S}](j)$ must be used when Bland's rule is applied.

In order to simplify the construction of the comparison operator, the problem is transformed such that all non-applicable constraints become applicable. The fractional values of non-applicable constraints is simply replaced by one which is larger than the maximal possible, $\frac{\infty}{1}$. Here ∞ simply represents some value larger than the largest possible value of the $[\mathbf{B}](j)$'s, e.g. $\infty = 2^k$ if these values are k-bit. The updated arrays $[\mathbf{C}']$ and $[\mathbf{B}']$ are computed as

$$([\mathbf{C}'](j), [\mathbf{B}'](j)) \leftarrow \left([\mathbf{C}](j) \overset{?}{>} 0 \,?\, [\mathbf{C}](j) : 1; \quad [\mathbf{C}](j) \overset{?}{>} 0 \,?\, [\mathbf{B}](j) : \infty \right)$$

for $1 \le j \le m$. This ensures that all constraints are applicable, using only $\mathcal{O}(m)$ comparisons and multiplications in $\mathcal{O}(1)$ rounds. The solution is unchanged as the LP was assumed bounded, i.e. some initial constraint was applicable.

The next task is to define the comparison operator, $\overset{?}{\sqsubset}$, for constraints – triples $([\mathbf{C}'](j), [\mathbf{B}'](j), [\mathbf{S}](j))$. It is in essence just a comparison of non-negative fractions, $\frac{B'}{C'}$, except that in the case of equality, the S values must be compared. Noting that for non-negative integers a_n and b_n, and positive integers a_d and b_d,

$$\frac{a_n}{a_d} < \frac{b_n}{b_d} \Leftrightarrow a_n \cdot b_d < b_n \cdot a_d$$

the desired output may be computed using integer comparison. An analogous equation holds for equality, thus, Bland's rule applies exactly when the two products are equal. Details are seen in Protocol 2. Correctness follows by the above, while complexity is equivalent to the standard comparison under big-$\mathcal{O}$.

Protocol 2 Constraint comparison, $\overset{?}{\sqsubset}$

Input: Triples $([C_i], [B_i], [S_i])$ and $([C_j], [B_j], [S_j])$ representing applicable constraints to be compared – $[C_i]$ and $[C_j]$ represent the relevant entries of the pivot column while $[B_i]$ and $[B_j]$ represent the values of the b-vector. Finally $[S_i]$ and $[S_j]$ are the index-values of the current basis variables associated with the two constraints, i.e. the candidates to leave the basis.

Output: $[b] \in \{0, 1\}$ – one if the left argument, $([C_i], [B_i], [S_i])$, constrains less than the right, $([C_j], [B_j], [S_j])$, with ties broken using Bland's rule.

```
$[l] \leftarrow [B_i] \cdot [C_j]$
$[r] \leftarrow [B_j] \cdot [C_i]$
$[b] \leftarrow ([l] \overset{?}{=} [r]) \,?\, ([S_i] \overset{?}{<} [S_j]) : ([l] \overset{?}{<} [r])$
```

$[\mathbf{r}]$ is determined by applying one of the protocols of Appendix A on the array

$$\big(([\mathbf{C}'](1), [\mathbf{B}'](1), [\mathbf{S}](1)), \ldots, ([\mathbf{C}'](m), [\mathbf{B}'](m), [\mathbf{S}](m))\big)$$

using Protocol 2 as comparison operator. This determines the desired index, $[\mathbf{r}]$, along with the triple, $([C'_r], [B'_r], [S_r])$.[8] Obtaining a copy of the pivot row, $[\mathbf{R}]$, is simply an indexing operation, $[\mathbf{R}] \leftarrow [\mathbf{T}]([\mathbf{r}], \cdot)$. Note that $[C'_r] = [\mathbf{C}']([\mathbf{r}]) = [\mathbf{C}]([\mathbf{r}])$ is the pivot element. Finally, $[\mathbf{S}]$ must reflect the updated basis, i.e. $[\mathbf{S}]([\mathbf{r}])$ must be set to val$([\mathbf{c}])$.

The initial transformation and computing the minimal using Protocol 2 requires $\mathcal{O}(m)$ comparisons and multiplications. Determining $[\mathbf{R}]$ requires $m+n+1$ indexing operations in arrays of length m, and the update of $[\mathbf{S}]$ a single indexing operation of size m. Overall this amounts to $\mathcal{O}(m \cdot (m+n))$ multiplications and $\mathcal{O}(m)$ comparisons in $\mathcal{O}(\text{loglog}(m))$ rounds.

Step III, Multiply all non-pivot rows by the pivot element. The previous steps determined the variables to enter and leave the basis in the form of indexes, $[\mathbf{c}]$ and $[\mathbf{r}]$, as well as the pivot element, $[p]$. The third step consists of multiplying all entries of the tableau – *except* the ones in the pivot row – by the latter. This is accomplished by Protocol 3. For correctness, note that after step 4, all entries in $[\mathbf{M}]$ are $[p]$ except the $[\mathbf{r}]$'th, which is $[1]$. Thus, step 7 multiplies the entries of non-pivot rows by the pivot element, while leaving the pivot row implicitly untouched. Only the indexing with $[\mathbf{r}]$ and the updating of the tableau require multiplication, implying $\mathcal{O}(m \cdot (n+m))$ multiplications in all. Round complexity is constant as all entries of the tableau may be updated concurrently.

Protocol 3 Multiplication of all non-pivot rows by the pivot element

Input: The tableau, $[\mathbf{T}]$, the index of the pivot row, $[\mathbf{r}]$, and the pivot element, $[p]$.
Output: The updated tableau, $[\mathbf{T}']$, with all non-pivot rows multiplied by $[p]$.

```
    for $j \leftarrow 1, \ldots, m+1$ do
        $[\mathbf{M}](j) \leftarrow [p]$
    end for
    $[\mathbf{M}]([\mathbf{r}]) \leftarrow 1$
 5: for $j \leftarrow 1, \ldots, m+1$ do
        for $i \leftarrow 1, \ldots, n+m+1$ do
            $[\mathbf{T}'](j,i) \leftarrow [\mathbf{M}](j) \cdot [\mathbf{T}](j,i)$
        end for
    end for
```

Step IV, Subtract a multiple of the pivot row from all non-pivot rows. Note first that as all non-pivot rows have been multiplied by the pivot element, the multiple to subtract from non-pivot row j is the j'th entry of the original pivot

[8] An unbounded LP implies that $[B'_r] = \infty$ will occur, allowing this to be detected.

column, [**C**]. The computation is analogous to that of step III; set the multiple to be subtracted from the pivot row to zero and subtract the relevant multiple for each entry in the tableau, Protocol 4. Correctness is equivalent to step III, as is the complexity obtained: $\mathcal{O}(m \cdot (m+n))$ multiplications in $\mathcal{O}(1)$ rounds.

Protocol 4 Subtraction of a multiple of the pivot row from all non-pivot rows

Input: The tableau, [**T**], the index of the pivot row, [**r**], the pivot row, [**R**], and the original pivot column, [**C**].

Output: The updated tableau, [**T′**], such that all non-pivot rows are zero in the pivot column.

```
    $[\mathbf{C}]([\mathbf{r}]) \leftarrow 0$
    for $j \leftarrow 1, \ldots, m+1$ do
        for $i \leftarrow 1, \ldots, n+m+1$ do
            $[\mathbf{T}'](j,i) \leftarrow [\mathbf{T}](j,i) - [\mathbf{C}](j) \cdot [\mathbf{R}](i)$
 5:     end for
    end for
```

Step V, Divide non-pivot rows by the previous pivot element. The final step requires an integer division for all non-pivot row entries. General constant-rounds integer division is possible but expensive. However, this is not a general case: $[p']$ divides $[\mathbf{T}](j,i)$ for all entries not in the pivot row (see [Ros05] for a proof). This simplifies the problem, as division can be implemented as multiplication by the multiplicative inverse (in the field) of the divisor.[9] Element inversion is possible using the well-known protocol of Bar-Ilan and Beaver, [BB89].

Having obtained $\left[(p')^{-1}\right]$, this must be multiplied onto all non-pivot rows. This problem is equivalent to step III; the tableau may be updated with an invocation of Protocol 3. Inversion is efficient, implying that complexity is dominated by Protocol 3, $\mathcal{O}(m \cdot (n+m))$ multiplications in $\mathcal{O}(1)$ rounds.

Overall complexity. Combining the above analyses, the overall complexity of steps I through V is found to be $\mathcal{O}(m \cdot (m+n))$ multiplications and $\mathcal{O}(m+n)$ comparisons. Each step performs at most a constant number of multiplications per entry in the tableau – sometimes hidden in indexing operations – in addition to the comparisons in the first two steps. Round complexity is dominated by step II, $\mathcal{O}(\mathrm{loglog}(m))$. Reducing round complexity to $\mathcal{O}(1)$ increases the number of comparisons to $\mathcal{O}(n+m^2)$.

7 Iteration and Termination

Evaluating the termination condition – determining whether to continue – is costless except for the final (failing) test. Consider an optimistic approach where

[9] For Paillier encryption, a negligible number of elements are not invertible; this does not pose a problem.

step I is performed with no knowledge of whether or not $[\mathbf{F}]$ contains a negative entry. If no such entry exists, then the array of test results – and therefore also the "index" determined – will contain all 0's. The sum of entries of the index is therefore 0 in this case, while a proper index sums to 1.

Simple reconstruction at every iteration allows the parties to determine if they are done; this leaks the number of pivots but no more. If this is not acceptable, the point of termination can be hidden using dummy pivots. The termination condition remains secret and computation continues. After each iteration, conditional selections are used to ensure that the final tableau is not changed by choosing the previous one if termination had already occured. This does not alter the big-$\mathcal{O}$ compexity.

Unless exponentially many iterations are performed Some information will be leaked. Needing exponentially many pivots is unlikely, though. It is sometimes stated that in practice, runtime is linear in the number of constraints and logarithmic in the number of variables. Performing U pivots, where U is a *likely* upper bound, reveals only that – as expected – fewer than U pivots were performed.

Other compromises and variations are also possible. Public testing of the termination condition can be restricted to every k'th pivot. The exact number of iterations remains secret, but roughly the right amount are performed. Alternatively, given multiple LP problems of the same size, it is possible to only reveal the overall number of pivots performed. Conditional selection can be used to obliviously replace the "working-tableau" by a fresh problem upon termination, it will not be known what tableau a given pivot considered. The drawback is that every LP problem is touched at every iteration resulting in worse complexity. Naturally, it is also possible to construct hybrids between all these variations.

8 Determining the Solution from the Tableau

Recall that the goal of securely solving an LP is sharings of the assignments to the variables as well as a sharing of the evaluation of the objective function at that point. By explicitly obtaining sharings, the protocol can be used not only for solving the LP and providing (parts of) the solution directly to parties, it is also possible to use the output as input for additional MPC, allowing the present work to be used as a building block.

The solution consists of rational values given as the b-vector and the final pivot element, $[p']$. These values must be assigned to the basis variables represented by $[\mathbf{S}]$: the j'th basis variable, $x_{[\mathbf{S}](j)}$, takes the value $\frac{[\mathbf{B}](j)}{[p']}$; co-basic variables are simply 0. The result is stored uniquely as $[\mathbf{S}]$, $[\mathbf{B}]$, and $[p']$, but seeing these reveals information. Examples include which slack-variables are in the basis and which basis variable is associated with which constraint.

Computing an array containing the values assigned to the numerator of each x_i in the solution eliminates this problem; this is accomplished by Protocol 5.

All variables are initialised to 0, after which the values of $[\mathbf{S}]$ are used to index. m indexing operations on an array of length n[10] provides the desired result.

Protocol 5 Assigning the solution to the variables, x_i

Input: The solution stored as the b-vector, $[\mathbf{B}]$ and the list of basis variables, $[\mathbf{S}]$.
Output: An array $[\mathbf{X}]$, such that $[\mathbf{X}]\,(i)$ takes the value of the numerator of the value assigned to x_i.

```
    for $i \leftarrow 1, \ldots, n$ do
        $[\mathbf{X}]\,(i) \leftarrow 0$
    end for
    for $i \leftarrow 1, \ldots, m$ do
5:      $[\mathbf{X}]\,([\mathbf{S}]\,(j)) \leftarrow [\mathbf{B}]\,(j)$
    end for
```

Naïvely, the updates of the $[\mathbf{X}]\,(i)$ must occur sequentially implying $\mathcal{O}(m)$ rounds. However, no entry of $[\mathbf{X}]$ is updated more than once, as the basis variables are distinct. It is therefore possible to parallelize the conditional selections. "Non-triggering" selections result in a 0 which is to be added to the original value; performing all these in parallel first and *then* performing the additions provides a constant-rounds solution with the same complexity, $\mathcal{O}(n \cdot m)$.

The rational value $[\mathbf{X}]\,(i)\,/[p']$ is assigned to x_i and can be provided to any party or used in subsequent computation. The same is true for $f(x_1, \ldots, x_n) = [z]/[p']$. However, seeing one of these values leaks the final pivot element. Values must be represented in a canonical way to prevent leakage, i.e. the fractions must be reduced.

This can be done using the technique of Fouque et al. [FSW02]: rational values may be encoded as numerator times inverse of denominator in the prime field.[11] The conversion is efficient and this provides a canonical representation. Naturally the technique only works for values of a bounded size. However, this issue was already encountered when simulating integer computation; it is merely a question of selecting an appropriate modulus initially.

9 Choosing a Modulus

A final point to consider is the choice of modulus, q. How large numbers must be representable? [Goe94] provides upper bounds on the maximal bit-length of the values of the tableau through Hadamard's inequality. At most

$$L = \log(det_{max}) + B + F + m + n + 1$$

bits are required per value, where B and F are the bit-lengths of the original b_i's and f_i's respectively, while det_{max} is the maximal determinant of an $m \times m$

[10] Slack-variables are ignored, for $[\mathbf{S}]\,(j) \geq n$ the update does implicitly nothing.
[11] This also works Paillier encryption, though not all elements are invertible.

sub-matrix of constraint-rows. Letting C denote the bit-length of the $c_{j,i}$, by Hadamard's inequality,

$$\log(det_{max}) \leq (m \cdot (2C + \log(m)))/2.$$

Note that q must be of twice this bit-length: comparing fractions require double precision and so does the reduction of fractions at the very end.

As an example, consider the case of 32-bit inputs, 32 variables, and 16 constraints. Approximately 650-bits are required for tableau-values implying that a 1300-bit prime must be used. While relatively large, this cannot be avoided – at least not with perfect precission – as there are LP problems, which require this.

10 Concluding Remarks

The privacy-preserving LP-solver presented is a good solution in the sense that the most used operations – the arithmetic – are cheap. The $\mathcal{O}(m(m \cdot n))$ secure multiplications are simply invocations of a basic primitive. Though this makes the $\mathcal{O}(m + n)$ secure comparisons slightly more difficult, performing bit-wise addition or multiplication is very costly and a lot more difficult.

Regarding privacy, how can we be certain that there are no unintentional information leaks? As noted above, information is only disclosed when a value is *intentionally* revealed; privacy follows from the privacy of the primitives. This only occurs for the termination condition and the output, and neither carry any additional information. The former is either 0 or 1, while the latter is stored in a canonical way.

Note that considering an abstract comparison protocol implies that if a new and improved protocol is constructed, overall complexity of this work immediately improves. Moreover, for an actual implementation of the LP-solver, it could be preferable to use a non-constant-rounds solution. This approach can reduce actual runtime, as expensive tricks to obtain constant-rounds can be avoided. Big-$\mathcal{O}$-complexity remains the same – at least with the present knowledge – but the hidden constants are reduced. Moreover, as those constants are small, the actual number of rounds may increase only slightly if indeed at all.

11 Acknowledgements

This work could not have been performed without the help of others. Troels Bjerre Sørensen is thanked for introducing me to integer pivoting as well as explaining linear programming and the simplex method. Peter Bro Miltersen is thanked for discussions of the simplex method. Ivan Bjerre Damgård, Jesper Buus Nielsen, and Berry Schoenmakers are thanked for the many discussions, suggestions and comments. Finally, Otto Johnston is thanked for commenting on presentation.

References

[BB89] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds of interaction. In Piotr Rudnicki, editor, *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 201–209, New York, 1989. ACM Press.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10. ACM Press, 1988.

[BN08] P. Bogetoft and K. Nielsen. Dea based auctions. *European Journal of Operational Research*, 184(2):685–700, 2008.

[CCD88] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19. ACM Press, 1988.

[CDN01] R. Cramer, I. Damgård, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. *Lecture Notes in Computer Science*, 2045:280–300, 2001.

[Chv83] V. Chvátal. *Linear Programming.* W. H. FREEMAN, 1983.

[DFK$^+$06] I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, volume 3876 of *Lecture Notes in Computer Science (LNCS)*, pages 285–304. Springer, 2006.

[DJ01] I. Damgård and M. Jurik. A generalization, a simplification and some applications of Paillier's probabilistic public-key system. In *Public Key Cryptography*, pages 110–136, Berlin, 2001. Springer-Verlag. Lecture Notes in Computer Science Volume 1992.

[DN03] I. Damgård and J. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer Berlin / Heidelberg, 2003.

[FIM$^+$06] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. Strauss, and R. Wright. Secure multiparty computation of approximations. *ACM Transactions on Algorithms*, 2(3):435–472, 2006.

[FSW02] P. Fouque, J. Stern, and G. Wackers. CryptoComputing with rationals. In *Financial Cryptography 2002*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2002.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM Press.

[Goe94] M. Goemans. Linear programming. Course notes, `http://www-math.mit.edu/~goemans/notes-lp.ps`, October 1994.

[Jáj92] J. Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[Kha79] L. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20, 1979.

[LA06] J. Li and M. Atallah. Secure and private collaborative linear programming. In *Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006.*, 2006.

[NO07] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC 2007: 10th International Workshop on Theory and Practice in Public Key Cryptography*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2007.

[NT07] K. Nielsen and T. Toft. Secure relative performance scheme. In Xiaotie Deng and Fan Chung Graham, editors, *Internet and Network Economics, Third International Workshop*, volume 4858 of *Lecture Notes in Computer Science*. Springer, 2007.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in cryptology – EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin / Heidelberg, 1999.

[Ros05] G. Rosenberg. Enumeration of all extreme equlibria of bimatrix games with integer pivoting and improved degeneracy check, 2005. CDAM Research Report LSE-CDAM-2005-18, `http://www.cdam.lse.ac.uk/Reports/Abstracts/cdam-2005-18.html`.

[RT07] T. Reistad and T. Toft. Secret sharing comparison by transformation and rotation. In *Proceedings of the International Conference on Information Theoretic Security (ICITS) 2007*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007. To appear.

[SFP06] M. Silaghi, B. Faltings, and A. Petcu. Secure combinatorial optimization simulating dfs tree-based variable elimination. In *AI and Math 2006 Proceedings*, 2006. Available at `http://anytime.cs.umass.edu/aimath06/proceedings.html`.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[Sil04] M. Silaghi. A suite of secure multi-party computation algorithms for solving distributed constraint satisfaction and optimization problems. Technical Report CS-2004-04, Florida Institute of Technology, 2004.

[Yao86] A. Yao. How to generate and exchange secrets. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

A Computing the Minimal of Multiple Values

An efficient means of obtaining the minimal of multiple shared values is required in simplex. Formalising the problem, let an array $[\mathbf{A}]$ of length k be given, the goal will be to compute a sharing of the minimal value, $[min] = \min_{1\leq i\leq k} [\mathbf{A}](i)$, along with an sharing of its index, $[\mathbf{i}]$. All of what is described in this appendix is simple adaptation of results from parallel algorithms, see any text-book e.g. [Jáj92].

For simplicity assume that k is a power of two. The obvious solution is to construct a binary tree, Protocol 6. It is easily verified that $\mathcal{O}(k)$ comparisons and multiplications are used. Moreover, as all computation in the loops parallelize, the overall round complexity is $\mathcal{O}(\log(k))$. Correctness is immediate.

It is possible to improve on the round complexity of the above. Let $\wedge(\cdot)$ denote a protocol taking an array of shared bits as input and returning their logical `AND`

Protocol 6 $\min_{\mathcal{O}(\log(k))}(\cdot)$ – Computing the minimal element in $\mathcal{O}(\log(k))$ rounds.

```
Input: $[\mathbf{A}]$, such that $k = [\mathbf{A}]$.len is a two-power.
Output: $[\mathbf{i}]$ and $[min]$, such that $[min] = [\mathbf{A}]([\mathbf{i}])$ is minimal.
  if $[\mathbf{A}]$.len $= 1$ then
      $[min] \leftarrow [\mathbf{A}](1)$
      $[\mathbf{i}] \leftarrow (1)$
  else
5:    for $j \leftarrow 1, \ldots, k/2$ do
          $[\mathbf{B}](j) \leftarrow [\mathbf{A}](2 \cdot j - 1) \overset{?}{<} [\mathbf{A}](2 \cdot j)$
          $[\mathbf{A}'](j) \leftarrow [\mathbf{B}](j)\ ?\ [\mathbf{A}](2 \cdot j - 1) : [\mathbf{A}](2 \cdot j)$
      end for
      $([\mathbf{i}'], [min]) \leftarrow \min_{\mathcal{O}(\log(k))}([\mathbf{A}'])$
10:   for $j \leftarrow 1, \ldots, k/2$ do
          $[\mathbf{i}](2 \cdot j - 1) \leftarrow [\mathbf{B}](j) \cdot [\mathbf{i}'](j)$
          $[\mathbf{i}](2 \cdot j) \leftarrow (1 - [\mathbf{B}](j)) \cdot [\mathbf{i}'](j)$
      end for
  end if
```

Protocol 7 $\min_{\mathcal{O}(1)}(\cdot)$ – Computing the minimal element in $\mathcal{O}(1)$ rounds.

```
Input: $[\mathbf{A}]$ with $[\mathbf{A}]$.len $= k$.
Output: $[\mathbf{i}]$ and $[min]$, such that $[min] = [\mathbf{A}]([\mathbf{i}])$ is minimal.
  for $j \leftarrow 1, \ldots, k$ do
      $[\mathbf{B}](j, j) \leftarrow 1$
  end for
  for $j \leftarrow 1, \ldots, k$ do
5:    for $j' \leftarrow j + 1, \ldots, k$ do
          $[\mathbf{B}](j, j') \leftarrow [\mathbf{A}](j) \overset{?}{>} [\mathbf{A}](j')$
          $[\mathbf{B}](j', j) \leftarrow 1 - [\mathbf{B}](j, j')$
      end for
  end for
10: for $j \leftarrow 1, \ldots, k$ do
      $[\mathbf{i}](j) \leftarrow \wedge([\mathbf{B}](\cdot, j))$
  end for
  $[min] \leftarrow [\mathbf{A}]([\mathbf{i}])$
```

and consider Protocol 7. Each column in the [$\mathbf{B}$]-matrix is associated with an entry of [$\mathbf{A}$]. For the minimal entry, the column will contain all ones, however, for a non-minimal entry, at least one zero will exist. Thus, computing the `AND` of the bits of every column results in an index for the minimal element. Note that the protocol provides the correct result even when the minimal entry is not unique, in this case the minimal index of the minimal value is returned. Concerning complexity, $\mathcal{O}(k^2)$ comparisons and $\mathcal{O}(k)$ multiplications are required, however, everything parallelizes and can therefore be performed in $\mathcal{O}(1)$ rounds. The only non-trivial detail is unbounded fan-in `AND`. Assuming that $k < q$, where q is defines the field, this can be done by adding all bits, and computing whether the sum equals the number of input-bits, i.e. using an equality.

Protocol 7 can be used to construct an $\mathcal{O}(\text{loglog}(k))$ rounds protocol using only $\mathcal{O}(k)$ comparisons and multiplications. This is done in two steps, first an $\mathcal{O}(\text{loglog}(k))$ rounds solution using $\mathcal{O}(k \cdot \text{loglog}(k))$ comparisons and multiplications is constructed, this is then used in conjunction with Protocol 6 to construct a protocol requiring only $\mathcal{O}(k)$. For the full details see [Jáj92] Sect. 2.6.2 and 2.6.3.

Divide [$\mathbf{A}$] into $\sqrt{k}$ sub-arrays of length $\sqrt{k}$ each, apply recursion, and use Protocol 7 to compute the minimal from the $\sqrt{k}$ candidates returned. The index is obtained similar to Protocol 6; this protocol has the stated complexity. The linear version is obtained through *accelerated cascading*: Perform $\lceil \text{logloglog} k \rceil$ iterations of Protocol 6 reducing the problem by a factor of $\text{loglog}(k)$. From there the $\mathcal{O}(k \cdot \text{loglog}(k))$ protocol is applied, the overall result is the desired $\mathcal{O}(\text{loglog}(k))$ rounds, $\mathcal{O}(k)$ comparisons and multiplications.

A final observation is that the protocols above can be used with *any* multiparty comparison on *arbitrary* data, as long as there exists some notion of "minimal." Naturally, the overall complexity depends on the complexity of the comparison operator.