SANDIA REPORT
SAND2020-XXXX
Printed November, 2020

Sandia National Laboratories

Novel Geometric Operations for Linear Programming

Mohamed Ebeida, Ahmed Abdelkader, Nina Amenta, Drew P. Kouri, Ojas Parekh, Cynthia A. Phillips, Nickolas Winovich

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

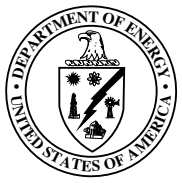
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods



Novel Geometric Operations for Linear Programming

Mohamed Ebeida[1], Ahmed Abdelkader [2], Nina Amenta[3], Drew P. Kouri[1], Ojas Parekh[1], Cynthia A. Phillips[1],Nickolas Winovich[4]

ABSTRACT

This report summarizes the work performed under the project "Linear Programming in Strongly Polynomial Time." Linear programming (LP) is a classic combinatorial optimization problem heavily used directly and as an enabling subroutine in integer programming (IP). Specifically IP is the same as LP except that some solution variables must take integer values (e.g. to represent yes/no decisions). Together LP and IP have many applications in resource allocation including general logistics, and infrastructure design and vulnerability analysis.

The project was motivated by the PI's recent success developing methods to efficiently sample Voronoi vertices (essentially finding nearest neighbors in high-dimensional point sets) in arbitrary dimension. His method seems applicable to exploring the high-dimensional convex feasible space of an LP problem.

Although the project did not provably find a strongly-polynomial algorithm, it explored multiple algorithm classes. The new medial simplex algorithms may still lead to solvers with improved provable complexity. We describe medial simplex algorithms and some relevant structural/complexity results.

We also designed a novel parallel LP algorithm based on our geometric insights and implemented it in the Spoke-LP code. A major part of the computational step is many independent vector dot products. Our parallel algorithm distributes the problem constraints across processors. Current commercial and high-quality free LP solvers require all problem details to fit onto a single processor or multicore. Our new algorithm might enable the solution of problems too large for any current LP solvers. We describe our new algorithm, give preliminary proof-of-concept experiments, and describe a new generator for arbitrarily large LP instances.

[1]Sandia National Laboratories
[2]University of Maryland
[3]University of California, Davis
[4]Purdue University

ACKNOWLEDGMENT

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

CONTENTS

1. Introduction **9**

2. Linear programming methods **11**

3. Theoretical contributions **14**
3.1. Theory 14
3.1.1. Medial simplex algorithms 14

4. Serial and Parallel linear programming solver implementations **16**
4.1. Random spokes 16
4.1.1. Spoke trimming 16
4.1.2. Spoke projection 17
4.2. Blocker constraints 18
4.2.1. Search space reduction 18
4.2.2. Informed shuffling 19
4.3. Lifting procedure 20
4.3.1. Violated constraints 20
4.3.2. Descent to feasible region 20

5. Benchmarking and empirical results **22**
5.1. Experimental Results on Netlib 22
5.2. Generating arbitrarily large LP instances 22

6. Conclusion **25**

References **26**

LIST OF FIGURES

Figure 2-1. The feasible region of an LP is a convex polytope. The optimal solution is always on the outside. LP algorithms search through d-dimensional space for an optimal solution, differing in their search paths. 12

Figure 4-1. Spoke trimming operation using constraint residuals for improved numerical stability. When computing the trim distance, we include the scaling factor λ to account for scenarios where the constraint hyperplane and spoke direction are approximately parallel. 17

Figure 4-2. We project the randomly sampled spoke directions onto the blocker subspace to refine the current search space and avoid running into nearby constraints. We also check to see whether the projected spoke directions are oriented in the positive objective direction, and flip the spoke directions if necessary. 19

Figure 4-3. Artificial lifting procedure used to obtain feasible starting positions. The initial position in the unlifted problem (left) satisfies the black constraint while violating the red constraint. The lifted problem (right) introduces a new dimension to the problem, tilts the violated constraint, and lifts the original starting position to ensure feasibility. 21

LIST OF TABLES

Table 5-1. Experimental results of Spoke LP Solver on Netlib using 1 node. 23
Table 5-2. Time results of Spoke LP Solver on Netlib with 2, 4, and 8 nodes. 23

1. INTRODUCTION

Linear Programming (LP) is the optimization of a linear function subject to linear inequality constraints. One example is multicommodity flow: movement of material through a network sharing link capacity. This applies to telecommunication network design, transportation and manufacturing. Linear programming is also a fundamental subroutine in algorithms to find exact and approximate solutions for Integer Programs (IPs). These are LPs with additional (non-linear) integerality constraints on some variables, representing decisions. IPs model a wide range of problems including mission-relevant applications in scheduling, NW logistics, infrastructure analysis, cybersecurity, and sensor placement and management.

LPs, and especially the integer programs LP enables, are workhorse optimization solvers. Those familiar with the method can quickly express practical applications that are within the class using an intuitive set of standard constructs. One can use IPs to express a linear optimization problem much the way a programmer uses a code to express an algorithm. Furthermore, commercial solvers (e.g. CPLEX, Gurobi) and free solvers (CLP or CBC) are good enough that they can solve many useful instances. This brings LP/IP methods to the "masses" of optimization consumers. It is worth taking the (short) time to code an application using a mathematical programming language (such as AMPL or pyomo) and seeing if the solvers can handle the problem. If so, the search gives a provably correct answer for the instance. This is true even for theoretically-intractable IP problems, because the search proved optimality for that instance.

Still, there are problems that are too big or too difficult for these solvers. This is a clear issue for problems directly expressed as LPs. Futhermore, even though many of the LPs solved as subroutines in an IP search are closely related to the last search, all IP solvers must solve the original full LP from scratch. If that that takes too long, then there is no hope for solving the IP. As an example of difficult LPs, Gearhart et. al. [6] did a comparitive study of free vs. commercial LP solvers because many government customers require delivered applications to depend only on free software. The military-personnel-management problem they were considering was an LP. They found that on challenging LP benchmark instances 7 years ago, even the best commercial solver, CPLEX timed out after attempting to solve the problem for 8 hours. The free solvers failed at this level on even more instances. We have experienced similar problems more recently. Our research problems tend to have different structure than the industrial problems commercial solvers are tuned for.

The most heavily used LP solvers also assume the whole constraint matrix fits into a single core. Extremely large problems have to be distributed across many processors and solved in parallel. No LP solver does that effectively now.

Our goal in this project was to explore new LP algorithms based on geometric insights from implicit Voronoi meshing and recursive hyperplane sampling research. This earlier work used

combinatorial geometric operations including, but not limited to, the pivot operation which is the basis of the simplex algorithm for LP, and used randomization in new ways. Since no previous LP researchers have come from this perspective, we hoped to succeed where previous researchers have failed, by combining our insights with the best aspects of previous approaches to solving LPs.

We envisioned making progress in two ways. On the theoretical end, our “moon shot" goal was to find an LP algorithms with *strongly* polynomial complexity. That is, we wanted an algorithm for which we could prove that the running time was bounded by a polynomial function of size only (number of variables and constraints) and does not depend on the numerical values in the input. Finding a strongly polynomial LP algorithm is 9th on Smale’s list of greatest unsolved math problems of the 21st century [12]. While we did not succeed in this ambitious goal, we did get some interesting results about polytope combinatorics, and advanced our understanding in this challenging area. We hope to do more work in this direction in the future.

Our practical goal was to develop parallel algorithms based on our collection of randomized combinatorial geometry operations that would allow larger LP problems to be solved on distributed systems. Current commercial and free LP solvers cannot scale to use more than a couple dozen processors despite multiple efforts. A new faster distributed LP solver will change the way the international optimization community computes. In this direction, we developed a new algorithm that does many fewer sparse linear system solves, instead relying on independent, easily distributed vector operations. We have a prototype implementation of this algorithm, and we report some results on distributed instances.

2. LINEAR PROGRAMMING METHODS

A linear program is the optimization (we will use minimization) of a linear objective function subject to linear constraints. A canonical form is min $c^T x$ subject to $Ax \leq b$ and $x \geq 0$. The input is an $n \times d$ matrix A, an d-vector c of costs, and an n-vector b. A linear equality $a^T x = 0$ forms a hyperplane in d dimensions. Each inequality restricts variables to a halfspace on one side of the hyperplane. The intersection of all the inequalities forms a convex polytope (we will assume the region is bounded) which is called the *feasible region*. The set of points x with a given objective cost γ (i.e. $c^T x = \gamma$) is a hyperplane. Figure 2-1(a) illustrates this hyperplane for several values of γ. The minimum possible cost is the smallest γ whose corresponding hyperplane still touches the feasible region. Thus the optimal solution is on the exterior of the polytope, and there is always an optimal vertex (point).

There are two practical classes of LP search-based solvers. The simplex method moves from vertex to vertex along edges on the polytope exterior (Figure 2-1(b)). Interior-point methods move exclusively through the polytope interior (Figure 2-1(c)). Both methods require sparse linear systems solves.

LP is solvable in polynomial time, but all previous provably-efficient methods have running times that depend on the sizes of numerical values in the input. A *strongly* polynomial-time algorithm for LP is one whose running time is bounded by a polynomial in n and d, independent of the sizes of the entries of A, b, and c. Arithmetic operations are treated as constant-time black-box operations. In particular, the number of high-level iterations of strongly polynomial-time algorithms do not depend on numerical properties such as condition numbers. Strongly polynomial-time algorithms are sometimes called *combinatorial* algorithms, since they are described in terms of high-level arithmetic operations on numbers. Such algorithms produce *exact* solutions as opposed to numerically approximate solutions, where the exact solution to a linear program would be represented by the set of input constraints that meet at an optimal vertex. Thus an implementation of a strongly polynomial-time algorithm would guarantee the exact solutions as long as numbers with sufficient precision are used. In contrast numerical optimization algorithms are only able to approach exact solutions as a function of the precision. Exact solutions can be critical in mission-driven scenarios where outputs of optimization algorithms drive decisions and precise verifiability is essential. Strongly polynomial-time algorithms are known for special classes of LPs [15].

The most relevant related works include ongoing work on efficient interior point methods, both randomized [8] and deterministic [16]. There is continued interest in exact algorithms whose running times depend on instance-specific parameters, e.g., the constraint matrix [3, 4]. Perhaps more related to the original goal of an exact algorithm with strongly-polynomial time, a randomized variant of simplex algorithm, leveraging ideas from smoothed analysis [2, 13], was shown to run in expected polynomial time [7] albeit with dependence on the encoding length.

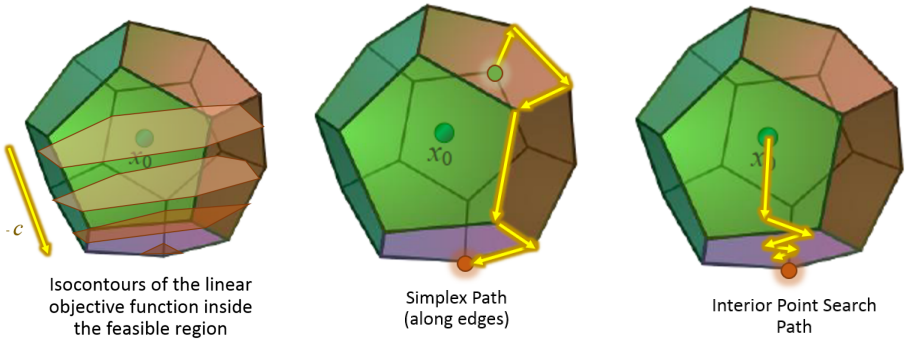


Figure 2-1. The feasible region of an LP is a convex polytope. The optimal solution is always on the outside. LP algorithms search through d-dimensional space for an optimal solution, differing in their search paths.

Novel combinatorial geometry operations The classic simplex algorithm for LP is based on the pivot operation. The input to the pivot operation is a set of n input constraints which meet to form a vertex of the feasible polytope P. The pivot operation swaps out one of those constraints for a new one, defining a new vertex of the feasible polytope with an improved value of the objective function. A sequence of pivots forms a path along the boundary of the feasible polytope, ending at the optimal vertex.

We explored the use of other combinatorial operations, which describe motions in the interior of the feasible polytope, hoping to achieve a combination of the advantages of the simplex algorithms and interior-point algorithms.

The first novel operation is *random spoke shooting*. The PI developed the Recursive Spoke Darts method in 2016 for tractable sampling of Voronoi vertices in arbitrary dimensions [5]. In a spoke shooting operation, we throw a ray, or *spoke*, from an initial point x_0 in the interior of P. We proceed along the spoke until we hit a facet (i.e., a polytope "side") f_1 of P. The facet f_1 can be determined by computing a dot product $\hat{c} \cdot a_i$ for each normalized row a_i of A. We select as f_1 an a_i that achieves the minimum nonnegative value of $\hat{c} \cdot a_i$. Computing these intersections of spokes with facets is computationally efficient and naturally paralleizable.

Once we have found f_1, we can recursively shoot spokes contained in the lower-dimensional feasible region of f_1; we say that f_1 has become a *tight constraint*, in which any extension to the path must be contained. While shooting spokes with a non-zero set of tight constraints does produce a path on the boundary of P, these spokes can be in the interior of a larger-dimensional face. Thus the set of paths we consider is much larger than the family of classic simplex paths, which are restricted to the one-dimensional edges of P.

When we shoot a spoke inside f_1, we end up at an $(n-2)$-dimensional face of P, formed by the intersection of f_1 and some additional facet f_2. At this point we have two options. First, we can *reduce dimension* by shooting spokes into the $(n-2)$-dimensional face, making both f_1 and f_2 tight constraints. If we reduce dimension at every step, we eventually end up with a set of n facets determining a feasible vertex of P. Second, we can *shuffle*, by swapping f_2 for f_1, if shooting

spokes in f_2 would allow us to improve the objective function. In general, a shuffle operation swaps out one tight constraint for a new constraint, giving a higher-dimensional analog for the pivot operation.

Whatever the dimension of the face formed by the current set of tight constraints, when shooting spokes into its interior, it is not clear what the best spoke direction is. Shooting spokes in the direction that best optimizes the objective function in the space (formed by projecting the objective function $-c$ into the linear space containing the face) is an obvious choice and, because it is easy to analyze, might lead to theoretical advances (although we have not been successful so far in proving that it does). Shooting several random spokes, on the other hand, allows much more parallelism, and gives us multiple chances to find a direction that gives us a significant improvement in the objective function. The use of random spokes is another idea that we borrow from the Recursive Spoke Darts algorithm.

In addition to shooting spokes, which gives us a wide range of possible paths on the polytope surface, we considered a class of paths which are forced to lie in the polytope interior. This was inspired by the successful weakly polynomial time algorithms, in which a path is constructed in the interior of P, ending very near the optimal vertex. We considered both pivot and spoke-shooting steps on the *medial axis* of the polytope, which is a combinatorial structure defining paths in the polytope interior. The medial axis is the locus of points that have more than one closest point on the boundary of P. Its edges are the locus of points with n or more closest points on the polytope boundary; that is, each edge point is the center of a ball with empty interior but at least n points of contact on boundary of P. The *medial simplex* algorithms extend simplex algorithms to consider walks on the edges of the medial axis, as well as the edges of the feasible polyhedron. Spoke shooting provides paths that traverse faces of the medial axis in all dimensions.

Both spoke shooting and using the medial axis open up new opportunities to develop combinatorial LP algorithms with better theoretical properties, practical advantages, or both.

3. THEORETICAL CONTRIBUTIONS

3.1. Theory

The theoretical goal of this project was to search for a strongly polynomial time algorithm for linear programming. A strongly polynomial time algorithm is one whose running time is a function of n, the dimension, and m, the number of constraints, while the running time of a a weakly polynomial time algorithm may depend also on L, the size of the input in bits (that is, the resolution of the numbers representing the input). Finding a strongly polynomial LP algorithm is one of the longest-standing open problems in computer science, and we did not succeed in solving it. We did, however, develop an interesting way of looking at the problem, which we hope will be useful.

3.1.1. Medial simplex algorithms

We focused on a particular path within the graph formed by the medial axis edges, which we call the *spine* of the polytope. Let's assume the LP is maximizing the objective function $c \cdot x$. At each level z, let $s(z)$ be the center of the largest ball with empty interior contained in the feasible polytope and centered at a feasible point on the plane $c \cdot x = z$. Any such point lies on an edge of the medial axis, and their union forms a path in the medial axis, the spine, containing both the minimum and maximum feasible points with respect to c, and the largest ball enclosed in the polytope. The spine is a natural combinatorial example of an interior path. Using LP duality, we can show that

Theorem 1. *The spine corresponds to a shadow-vertex path on the feasible polytope of the dual LP.*

Since we can construct an arbitrary dual polytope and place any shadow-vertex path in that position, and there are examples of exponential-length shadow-vertex paths, this theorem implies that the length of the spine can be exponential in n and m.

The fact that the determinisitic medial simplex algorithm has an exponential lower bound is not surprising; progress on combinatorial LP algorithms in recent years has mostly been based instead on creative applications of randomization. Specifically, the breakthrough 2004 "smoothed analysis" paper of Spielman and Teng [14] considered perturbing both the constraints and objective function of an LP, and showed that the perturbed LP could be solved in polynomial time using the shadow vertex algorithm. Of course, the solution of the perturbed LP is not the same as the solution of the original problem. This was followed by a series of refinements and simplifications. In 2006, Kelner and Spielman showed that a less consequential perturbation only of the right-hand side of an LP could be used as a subroutine in an algorithm that uses iterative rounding of the polytope (eg. like the ellipsoid method) to give a weakly polynomial LP algorithm. We attempted to build on both of these approaches.

We considered randomizing the medial simplex algorithm by assigning a multiplicative weight w_i to each constraint, so that the medial axis, and the spine, is computed with respect to weighted distance $w_i a_i \cdot x$ instead of the usual Euclidean distance given by $a_i \cdot x$. This is similar to, but not exactly the same as, the one-dimensional perturbation used by Kelner and Spielman. We hoped that in the geometric framework of the medial simplex algorithm we could get a better bound on the length of the spine, but unfortunately we found that:

Theorem 2. *The expected length of an edge on the dual shadow can be exponentially small in n, m and L.*

This implies that even after randomization the spine might contain an exponential number edges.

This hardly exhausts the possibilities of the medial simplex algorithm. One question we are interested in pursing is the relationship between the complexity of the spine and the central curve, the smooth path followed by interior point algorithms.

4. SERIAL AND PARALLEL LINEAR PROGRAMMING SOLVER IMPLEMENTATIONS

In this section, we will describe our parallel LP code. In contrast to our efforts in the theory section, the goal here is to design an algorithm that can scale on a large number of processors in a distributed memory system.

We begin by describing the randomized techniques used to explore the feasible region in Section 4.1 and Section 4.2. These techniques assume that we already have an initial feasible starting position, however, and in Section 4.3 we introduce a procedure for obtaining a feasible position that can be used to initiate the algorithm.

4.1. Random spokes

At an intuitive level, the algorithm consists of an iterative procedure designed to answer the following question at each step: given the current position, what is the direction and distance we should travel in order to improve our chances of finding the optimal vertex? The concept of *random spokes*, first introduced by the PI in [9] and [5], provides a natural framework for addressing this question. To construct a random spoke centered at a feasible position $x \in \mathbb{R}^d$, we first sample a direction θ from the unit-sphere $S^{d-1} = \{u \in \mathbb{R}^d : \langle u,u \rangle = 1\}$. Once the spoke direction has been selected, we probe the polytope along the ray $\{x + t \cdot \theta : t > 0\}$ to obtain geometric information used to compute the length of the spoke.

In order to fully specify how spokes are constructed, we make use of two fundamental spoke operations: spoke trimming and spoke projection. We use the trimming operation to determine the appropriate length of each spoke relative the problem geometry, and we use the spoke projection operation to restrict the direction of each spoke to a particular region of interest. Importantly, we perform both of these operations using only vector inner-products and matrix-vector multiplication with the constraint rows so that the resulting algorithm generalizes naturally to the distributed setting (i.e. by partitioning the constraints and distributing them across the available processors).

4.1.1. Spoke trimming

The main goal of the trimming operation is to compute how far we should travel along the ray $\{x + t \cdot \theta : t > 0\}$ in order to make the most progress possible without leaving the feasible region. In the absence of numerical considerations, we can perform this task quite easily since the distance $T_i(x;\theta)$ that can be traveled before violating constraint i is given by:

$$T_i(x;\theta) \;=\; \max\left((b_i - \langle a_i, x \rangle) / \langle a_i, \theta \rangle \;,\; 0 \right) \tag{4.1}$$

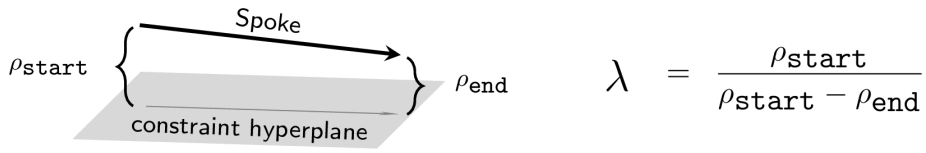


Figure 4-1. Spoke trimming operation using constraint residuals for improved numerical stability. When computing the trim distance, we include the scaling factor λ to account for scenarios where the constraint hyperplane and spoke direction are approximately parallel.

Since leaving the feasible region is equivalent to violating one of the constraints, the maximum distance $T(x;\theta)$ that can be traveled along the ray inside of the feasible region is given by:

$$T(x;\theta) \;=\; \max\{t \geq 0 : x + t\cdot\theta \sim \text{feasible}\} \;=\; \min_i T_i(x;\theta) \tag{4.2}$$

In practice, traveling the full distance $T(x;\theta)$ at each step can be problematic due to numerical issues. To improve the stability of the algorithm, we can repose the trimming operation in terms of the *constraint residuals*, $\rho^{(i)}_{\texttt{start}}$ and $\rho^{(i)}_{\texttt{end}}$, associated with the beginning and end of the spoke:

$$\rho^{(i)}_{\texttt{start}} \;=\; b_i - \langle a_i, x\rangle \quad \text{and} \quad \rho^{(i)}_{\texttt{end}} \;=\; b_i - \langle a_i, x + T(x;\theta)\cdot\theta\rangle \tag{4.3}$$

To begin, we identify and remove constraints with $\rho^{(i)}_{\texttt{start}} \leq \rho^{(i)}_{\texttt{end}}$ from the trimming procedure. For the remaining constraints, we use the constraint residuals to compute scaling factors $\lambda^{(i)}$:

$$\lambda^{(i)} \;=\; \rho^{(i)}_{\texttt{start}} \Big/ \left(\rho^{(i)}_{\texttt{start}} - \rho^{(i)}_{\texttt{end}}\right) \tag{4.4}$$

and define the final trimming distance by setting $\tau = \lambda^* \cdot T(x;\theta)$ where $\lambda^* = \min \lambda^{(i)}$. We also record the constraint index $i^* = \operatorname{argmin} \lambda^{(i)}$ corresponding to the constraint that ultimately trimmed the spoke (i.e. the constraint which first blocks progress in the selected spoke direction). We use the computed trim length τ to update the current position: $x \mapsto x + \tau\cdot\theta$. We also use the constraint information i^* to inform how we navigate through the polytope in subsequent steps.

4.1.2. Spoke projection

We control the algorithm's search space by placing constraints on the random directions used to construct spokes. This is particularly helpful when dealing with "nearby" constraints; these constraints effectively block movement in an entire half-space, and when the current position is located near several constraints the probability of making progress is reduced dramatically. To avoid being immediately blocked by nearby constraints, we restrict the random spoke directions to a subspace that is parallel to the (currently identified) nearby constraints. We enforce this restriction by projecting directions that have been sampled uniformly from the unit sphere onto the subspace consisting of directions that are orthogonal to the normal vectors of all nearby constraints.

Given a collection of nearby row constraints, we assemble the rows into a submatrix A_p and form the associated right-hand-side vector b_p. We then compute the projection π of the current position x by solving the linear system:

$$\begin{bmatrix} I & A_p^T \\ A_p & 0 \end{bmatrix} \begin{bmatrix} \pi \\ v \end{bmatrix} = \begin{bmatrix} x \\ b_p \end{bmatrix} \quad \Longrightarrow \quad \begin{cases} \pi + A_p^T v = x \\ A_p \pi = b_p \end{cases} \tag{4.5}$$

We then isolate the unknown vector v by multiplying the top equation by A_p and making the substitution $A_p\pi = b_p$ from the bottom equation to obtain:

$$A_p A_p^T v = A_p x - b_p \tag{4.6}$$

After solving this system for v using the conjugate gradient method, we obtain the desired projection π by recalling that $\pi = x - A_p^T v$.

4.2. Blocker constraints

To control how the algorithm explores the feasible region, we actively track which constraint trims the best spoke at each step. We refer to these constraints as *blockers* and use the term *blocker subspace* to refer to the space of vectors which are orthogonal to all of the current blockers.

We use the blocker subspace to restrict the randomized spokes to a search space which is parallel to each of the blocker constraint hyperplanes. This allows us to navigate through the feasible region of the polytope by exploring a series of *feasible slices* of the polytope of varying dimensions.

Since the feasible slice we are exploring is determined by the list of active blockers, we can perform transitions between feasible slices by selectively adding and removing blockers. We do this based on information computed in the spoke trimming operations, and introduce two fundamental operations for updating the blocker list: search space reduction and informed shuffling.

4.2.1. Search space reduction

When a new blocker constraint is encountered, we first attempt to reduce the search space by appending the new constraint to the current blocker list. By including the constraint as a blocker, we ensure that the spoke directions sampled in subsequent steps will be parallel to the new constraint's hyperplane. This is achieved by projecting the spokes to the current blocker subspace at each step by an application of the conjugate gradient method described in Section 4.1.2. As shown in Figure 4-2, this results in a lower-dimensional search space which is calibrated to the geometry of the constraint hyperplanes close to the current position. We also flip the spoke directions, if necessary, to ensure that each spoke is oriented in the positive objective direction.

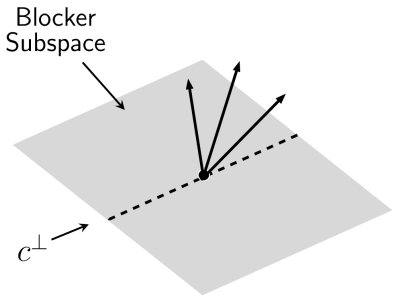


Figure 4-2. We project the randomly sampled spoke directions onto the blocker subspace to refine the current search space and avoid running into nearby constraints. We also check to see whether the projected spoke directions are oriented in the positive objective direction, and flip the spoke directions if necessary.

4.2.2. Informed shuffling

In some scenarios, the algorithm will fail to make progress after reducing the search space. This can be caused by having too many blockers relative to the problem dimension, but it can also occur in non-trivial scenarios due to the geometry of the current constraint hyperplanes and their orientation with respect to the objective direction.

When reducing the dimension of the search space fails, we perform a *shuffling* operation which consists of removing one of the existing blockers and replacing it with the newly encountered blocker. Geometrically, this corresponds to moving from a face of dimension k (where the number of current blockers is $n-k$), through an adjacent face of dimension $k-1$ (in which the newly encountered blocker is added), into an adjacent face of dimension k (where one of the old blockers has been removed). This is a higher-dimensional generalization of a simplex pivot, which moves from a one dimensional face of the feasible polytope, through a zero-dimensional face, to an adjacent one-dimensional face.

Just as with the simplex pivot operation, where not all edges make progress, not all adjacent k faces allow the shuffle step; the projection of the objective function direction c onto the adjacent face might be infeasible. If none of the current blockers can be removed in a shuffle that allows improvement of the objective function, the only option is to reduce the dimension. When $k = 1$, the shuffle operation is the simplex pivot, and unless we are at the global optimum, one of the adjacent edges will be in a direction improving c.

To efficiently check which constraints can be shuffled out, we compute the *duality coefficients* d associated with a matrix B, whose rows are the current blockers plus the newly encountered blocker, by solving the system $B^T B d = B^T c$.

Observation 1. *A row a_i allows feasible improvement of the objective function c by projecting it into the k-face formed by replacing a_i with the newly encountered blocker if and only if its duality coefficient is negative.*

The expression Bd gives the projection of c into the normal space of the $(k-1)$-face f formed by the intersection of all the blockers in B. If d_i is negative, it implies that c is directed into the negative half-space of a_i, that is, the improving direction is infeasible; projecting c into the subspace spanned by the remaining a_j (that is, the adjacent k-face formed by removing a_i) would remain in the negative half-space.

4.3. Lifting procedure

To initiate the procedure described in the previous section, we first need to identify a feasible starting position inside of the polytope. To accomplish this, we begin by sampling a random initial position from a unit-sphere and projecting the sampled point onto the space of equality constraints (i.e. constraints of the form $\langle a_i, x\rangle = b_i$ which must be satisfied at all times). Once the point is projected, we check whether or not the resulting position is feasible.

In most cases, the randomized starting position will not be feasible since it is unlikely to satisfy all of the LP constraints by chance. To handle scenarios where the initial position is infeasible, we introduce an artificial lifting procedure to account for the unsatisfied, or *violated*, constraints.

4.3.1. Violated constraints

When the initial position is infeasible, we introduce an additional *lifting* dimension to the problem. This allows us to view the original polytope as the zero-lifting-height cross-section of an artificially constructed "lifted" polytope in a higher dimensional space. In particular, we construct the lifted polytope by identifying the violated constraints associated with the initial infeasible position and *tilt* these constraints so that the initial position becomes feasible after increasing the lifted component sufficiently.

$$\rho_i = \max\left(\langle a_i, x\rangle - b_i\,,\,0\right) \quad , \quad \widehat{a}_i = [a_i, -\rho_i/R] \quad , \quad \widehat{x} = [x, R] \quad \text{where} \quad R = \max_i \rho_i \tag{4.7}$$

This ensures that $\langle \widehat{a}_i, \widehat{x}\rangle = \langle a_i, x\rangle - \rho_i \leq b_i$ for all constraints so that the lifted position $\widehat{x}$ is in fact feasible in the lifted polytope. In Figure 4-3, we show how this procedure can be visualized for a simple two-dimensional problem.

4.3.2. Descent to feasible region

Once we have constructed a feasible starting position in the lifted problem space, we initiate a descent procedure designed to find a feasible point with a zero-valued lifted component. This is motivated by the fact that the collection of feasible points in the lifted space with zero-valued lifted components coincides exactly with the feasible region of the original, unlifted problem. To perform the required descent in the lifted space, we simply apply the LP solver to the lifted problem and optimize in the negative direction of the lifted component.

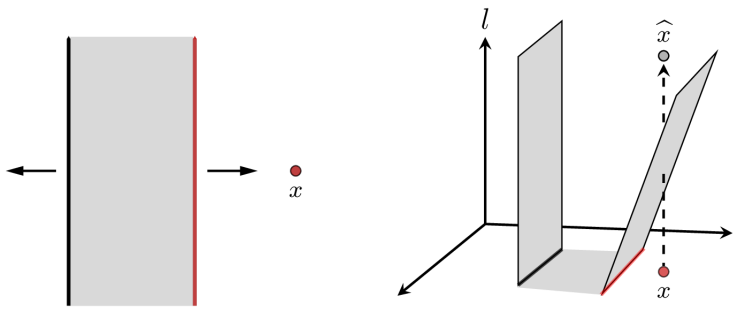


Figure 4-3. Artificial lifting procedure used to obtain feasible starting positions. The initial position in the unlifted problem (left) satisfies the black constraint while violating the red constraint. The lifted problem (right) introduces a new dimension to the problem, tilts the violated constraint, and lifts the original starting position to ensure feasibility.

5. BENCHMARKING AND EMPIRICAL RESULTS

In this section we describe our initial computational results. Since our code has a distributed matrix, we also needed test problems that were too large to fit in the memory of a single processor. We describe the problem generator we designed and implemented to make arbitrarily large LP instances.

(Cindy: There was a TODO around adding a description of the Netlib problems. I'm not sure what people wanted, but this is probably minimally OK.)

5.1. Experimental Results on Netlib

We debugged our code, Spoke-LP, on some small examples from the standard netlib dataset [10]. Table 5-1 gives some runtime statistics for a subset of the NETLIB problems. The solution matches those given in Netlib for all but E226, where the Netlib solution is -18.751929066. Table 5-2 shows runtimes for these problems when run on 2, 4, and 8 nodes. The runtimes actually increase with the number of nodes. This is because these problems are tiny by LP standards. They fit comfortably in the memory of a single processor. So the overhead for distributing the matrix is not justified. Nevertheless, these experiments debugged the parallel distributed-matrix code.

(Cindy: It would be good to give a reason why we selected this subset.)

5.2. Generating arbitrarily large LP instances

Since we believe there is no maintained, actively used distributed-matrix LP code, there are no public LP test sets that are so large they overwhelm the memory of a standard processor. Thus we had to generate our own test problems. Our generator is the linear-programming formulation for a graph problem. Therefore, by choosing a sufficiently large graph instance, we can create an arbitrarily large LP instance.

We used the maximum-density-subgraph LP formulation from Charikar [1]. Given an input graph $G = (V,E)$, and a subset of vertices $V' \subset V$, the *induced subgraph* is $G' = (V',E')$ where $E' = \{(u,v) \in E | u \in V' \wedge v \in V'\}$. That is, the induced subgraph includes all the original edges joining two of the selected vertices. The density of a graph $G = (V,E)$ is $\frac{|E|}{|V|}$. This half the average degree over the nodes in the graph, so is a measure of how dense the graph is. The *maximum density subgraph problem* takes a graph as input and finds its induced subgraph of maximum density.

Charikar [1] gives an elegant LP to solve this problem exactly. It has $O(|E|)$ variables and $O(|E|)$ constraints. The straightforward generalization of this LP to find the induced subgraph with the

Problem	Time	Steps	Vars	Rows	Eq Rows	Objective
ADLITTLE	0.09	160	97	138	15	225494.9632
AFIRO	0.04	48	32	51	8	-464.7531
AGG2	0.87	387	302	758	60	-20239252.3560
BANDM	16.58	602	472	472	305	-158.6280
BEACONFD	0.42	188	262	295	140	33592.4858
BLEND	0.13	151	83	114	43	-30.8121
BORE3D	3.59	321	315	344	215	1373.0804
BRANDY	4.74	482	249	292	139	1518.5099
CAPRI	22.12	852	353	583	158	2690.0129
DEGEN2	9.72	710	534	757	221	-1435.1780
E226	7.89	839	282	472	33	-25.8649
GROW7	1.73	361	301	581	140	-47787811.8147
SCTAP1	1.85	583	480	660	120	1412.2500
SHARE1B	2.84	428	225	253	89	-76589.3186
SHARE2B	0.12	173	79	162	13	-415.7322
STANDMPS	40.84	1949	1075	1362	284	1406.0175
STOCFOR1	0.13	119	111	165	63	-41131.9762

Table 5-1. Experimental results of Spoke LP Solver on Netlib using 1 node.

	2 nodes		4 nodes		8 nodes	
Problem	Time	Steps	Time	Steps	Time	Steps
ADLITTLE	0.23	153	0.38	160	0.77	151
AFIRO	0.04	44	0.08	47	0.22	48
AGG2	1.80	370	3.40	370	4.86	374
BANDM	24.07	575	32.16	558	50.65	577
BEACONFD	0.52	193	0.77	188	1.18	192
BLEND	0.36	164	0.65	148	1.32	163
BORE3D	6.05	297	10.01	307	17.40	303
BRANDY	8.68	499	13.87	480	19.42	500
CAPRI	42.07	816	77.59	879	119.81	856
DEGEN2	13.56	700	17.28	621	115.84	1373
E226	14.35	958	23.19	919	29.57	897
GROW7	1.89	350	4.04	360	4.09	363
SCTAP1	4.41	572	7.37	575	10.20	546
SHARE1B	5.62	428	11.41	428	20.53	472
SHARE2B	0.32	157	1.30	207	1.35	158
STANDMPS	63.13	1905	98.16	1966	161.59	1967
STOCFOR1	0.33	117	0.67	101	1.28	109

Table 5-2. Time results of Spoke LP Solver on Netlib with 2, 4, and 8 nodes.

maximum triangle density (most triangles per vertex) is even larger. It's size is determined by the number of triangles (3-cycles) in the graph, which can be quite large, especially for social networks.

We then needed a graph generator. Although there are many choices, we began with just the standard Erdös-Rényi (ER) random graphs. An ER random graph has two parameters: the number of nodes n and a probability p. Each edge occurs with probability p. We simply used the ER generator from the python Networkx package [11]. For a given n and p we can calculate the number of expected edges, hence the number of constraints and variables in the LP instance. Thus for a given fixed probability (graph density) p, we can choose values of n to give a suite of instances that increase in size by a desired factor. This is nice for testing scalability.

For problems that are too large to run on any current commercial LP solver, we must generate both the primal and the dual of an instance, solve both, and ensure the optimal objective values are (sufficiently) equal to verify we have correctly solved the instance.

6. CONCLUSION

Although we did not yet reach the “holy grail” of a strongly-polynomial LP algorithm, we believe that medial simplex algorithms still hold promise. Also, with additional engineering, the parallel distributed-matrix Spoke-LP code could solve problems too large to solve by any other means.

REFERENCES

[1] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In *Proceedings of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization*, APPROX '00, pages 84–95, Berlin, Heidelberg, 2000. Springer-Verlag.

[2] Daniel Dadush and Sophie Huiberts. A friendly smoothed analysis of the simplex method. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 390–403, 2018.

[3] Daniel Dadush, Sophie Huiberts, Bento Natura, and László A. Végh. A scaling-invariant algorithm for linear programming whose running time depends only on the constraint matrix. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, pages 761–774, 2020.

[4] Daniel Dadush, Bento Natura, and László A Végh. Revisiting tardos's framework for linear programming: faster exact solutions using approximate solvers. *arXiv preprint arXiv:2009.04942*, 2020.

[5] Mohamed Salah Ebeida and Ahmad Rushdi. Recursive spoke darts: Local hyperplane sampling for delaunay and voronoi meshing in arbitrary dimensions. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[6] Jared L. Gearhart, Kristin L. Adair, J. Detry, Justin D. Durfee, Katherine A. Jones, and Nataniel Martin. Comparison of open-source linear programming solvers. Technical Report SAND2013-8847, Sandia National Laboratories, 2013.

[7] Jonathan A. Kelner and Daniel A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, pages 51–60, 2006.

[8] Y. T. Lee and A. Sidford. Path finding methods for linear programming: Solving linear programs in $\tilde{O}$(vrank) iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 424–433, 2014.

[9] Scott A Mitchell, Mohamed S Ebeida, Muhammad A Awad, Chonhyon Park, Anjul Patney, Ahmad A Rushdi, Laura P Swiler, Dinesh Manocha, and Li-Yi Wei. Spoke-darts for high-dimensional blue-noise sampling. *ACM Transactions on Graphics (TOG)*, 37(2):1–20, 2018.

[10] Netlib linear programming problem set. `https://www.netlib.org/lp`.

[11] NetworkX network analysis in python. `https://networkx.org`.

[12] Steve Smale. Mathematical problems for the next century. *The Mathematical Intelligencer*, 20(2):7–15, March 1998.

[13] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004.

[14] Daniel A. Spielman and Shang-Hua Teng. Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time. *J. ACM*, 51(3):385–463, May 2004.

[15] Éva Tardos. A Strongly Polynomial Algorithm to Solve Combinatorial Linear Programs. *Operations Research*, 34(2):250–256, 1986.

[16] Jan van den Brand. *A Deterministic Linear Program Solver in Current Matrix Multiplication Time*, pages 259–278.

DISTRIBUTION

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop

Email—Internal [redacted]

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

Sandia
National
Laboratories