POLYNOMIAL ALGORITHM FOR THE k-CUT PROBLEM

Olivier Goldschmidt *
IEOR Department

Dorit S. Hochbaum †
IEOR Department and School of Business Administration

University of California
Berkeley, California 94720

Abstract

The k-cut problem is to find a partition of an edge weighted graph into k nonempty components, such that the total edge weight between components is minimum. This problem is NP-complete for arbitrary k and its version involving fixing a vertex in each component is NP hard even for $k = 3$. We present a polynomial algorithm for k fixed that runs in $O(n^{k^2})$steps.

Key words: clustering, min-cut, max-flow, complexity, polynomial algorithm, k-cut.

1 Introduction

Given a graph $G = (V, E)$ with nonnegative edge weights. A k-cut in G is a set of edges E', which when deleted, separate the graph into exactly k nonempty components. The k-cut problem is to find such set E' of minimum total weight. This problem is an extension of the 2-cut problem which is solvable via repeated applications of a max-flow min-cut algorithm ($|V| - 1$ applications suffice). A related extension of the min-cut problem is the k-cut problem with specified vertices. Here a set of k vertices is prescribed. The aim is to find a minimum weight k-cut that contains exactly one vertex of the given k, in each of the k components.

Though the k-cut problem with specified vertices is similar to the polynomially solvable min-cut problem, it is NP-hard even for $k = 3$. This result is due to [DJP83] (D. Johnson, private communication, 83). Our k-cut problem is however easier, since it is polynomially reducible to the problem with specified vertices; simply enumerate all subsets of k vertices and use an oracle solving the specified vertices version. The optimal k-cut will be the smallest of the ones enumerated. Therefore the NP-completeness of the specified vertices problem

*supported in part Intercollegiate Center for Management (Belgium) and by I.B.E.R. School of Business Administration - University of California Berkeley

†Supported in part by the National Science Foundation under grant ECS-85-01988 and by the Office of Naval Research under grant N00014-88-K-0377

0272-5428/88/0000/0444$01.00 © 1988 IEEE

does not imply NP-completeness for the k-cut problem.

Still, for arbitrary k, the k-cut problem is NP-complete. This follows from a reduction from the maximum clique problem (for details, see section 5). It is also known that the problem is polynomial for k fixed on planar graphs [DJP83]. A particularly fast algorithm for the planar 3-cut is given in [HS85]. The question of the complexity of the k-cut problem for k fixed has been open for several years. The problem was formulated in connection to generating travelling salesperson cutting planes (Pulleyblank, 82). It does have numerous applications otherwise, particularly in clustering-related setups, such as VLSI design. The k-cut problem is easily interpreted as a clustering problem. The edge weights represent the similarity between each pair of objects (represented by nodes). The k clusters that maximize the similarity between objects in same clusters or alternatively maximize the dissimilarity between objects in separate clusters is identical to the k-cut problem. Another area of applications is network reliability where the value of the minimum k-cut is important factor in assessing the reliability (or its lack thereof) of the network.

Other variants of the k-cut problem, for which polynomial algorithms have been developed, are ratio problems in which the value of a k-cut appears in the numerator. One such variant is the problem of finding the value of k, and the k-cut such that the ratio of the k-cut to k is minimum [Chv73]. The easy solution to this problem is attained for $k = 2$, thus requiring only the solution for the known polynomial 2-cut problem. Another variant was studied by Gusfield [Gus83], and by Cunningham [Cun85]. This problem is called the strength problem and it seeks to minimize the ratio of the k-cut to the number of additional components, i.e. the denominator is $k - 1$. This problem does not possess a straight forward solution such as the closely related problem in [Chv73], and a strongly polynomial algorithm for its solution is given in [Cun85]. Recently, Gusfield [Gus87] has devised a faster algorithm for the strength problem.

In this paper we present a polynomial time algorithm for the k-cut problem. The running time of the algorithm is $O(n^{k^2})$, where n is the number of vertices in the graph. One cannot expect to get rid of the exponential dependence of the running time on k unless NP=P. In that sense this algorithm is the best one can get for this problem.

Perhaps, the most natural greedy approach to consider for solving the k-cut problem is to apply the minimum 2-cut polynomial algorithm recursively. Such approach entails selecting k vertices and asserting that the k-cut consists of a minimum 2-cut separating one of these vertices from the others. The algorithm will then proceed with a recursive application of the minimum 2-cut routine. Unfortunately, such an approach does not work, even for the unweighted planar case, as illustrated in [HS85]. The underlying false assertion behind the greedy is, that one of the cuts separating one component from the others is a minimum (s,t)- cut. This assertion is "almost" true as we prove in the main theorem of this paper. In fact the assertion is true with the qualification that the one component separated by the minimum (s,t)-cut is large enough.

Another approach for solving the minimum k-cut is by a straight forward enumeration procedure. The entire enumeration of all possible k-partitions and the evaluation of the weight of the cut for each one, will certainly yield the optimal solution. However the number of such possible partitions is exponential in n. Our algorithm is based on enumeration as well, but it limits the size of subsets to be enumerated by $O(k)$. It is otherwise similar to the greedy approach in that it separates recursively, using

minimum (s,t)-cut one set of vertices from another set of vertices.

The essence of the idea of the polynomial algorithm, that works optimally as opposed to the greedy approach, is that more than one vertex is needed in the source set of the (s,t)-cut. More precisely, for k even, one of the cuts separating one component from the others in the optimal k-cut (or $(k-1)$-cut), is a minimum (s,t)-cut separating $k-2$ vertices in that component from $k-1$ vertices outside that component. This theorem is the main result of this paper.

We first present the general theorem followed by a description of the polynomial algorithm and its running time.

2 Main theorem

Let C be a minimum k-cut in G separating V into k components $V_1, V_2, \ldots, V_k$. Let C_i be the cut separating V_i from $V - V_i, i = 1, \ldots, k$. Let $w(A)$ be the sum of weights of edges in A. Without loss of generality, we assume that

$$w(C_1) \leq w(C_2) \leq \ldots \leq w(C_k) \qquad (1)$$

In case there is more than one optimal k-cut, we choose the one in which the component V_1 is maximal.(i.e. there is no other optimal k-cut with a component strictly containing V_1).

We will consider minimum (s,t)-cuts separating two sets of vertices such that one set is contained in the source set of the cut, and the other is contained in the sink set of the cut. The one set of vertices that is to be contained in the source set will be called the core of the cut. If there is more than one minimum cut we shall always choose the unique one which is **maximal**, i.e. its source set is not contained in any of the other minimum (s,t)-cuts' source sets.

Let $k \geq 4$ be even. The theorem is stated for the k-cut and the $(k-1)$-cut.

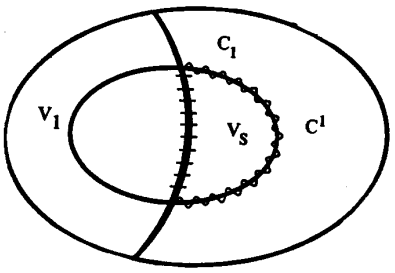

Figure 1: *The wiggly line is C_{1ext}, the section of C_1 within V_s. The crossed line is the section C^1_{ext}.*

Theorem 1 *If $|V_1| \geq k-2$, then there exists $(k-2)$ vertices in V_1 and $(k-1)$ vertices in $V - V_1$ ($(k-2)$ vertices in $V - V_1$ for the odd cut, $(k-1)$-cut) such that the minimum (s,t)-cut separating the vertices from V_1 from the other ones is equal to C_1.*

Proof: Let V_s be the source set of a maximal minimum cut C^1 generated by a core of $(k-2)$ vertices in V_1 and $k-1$ vertices, one from each component $V_2, \ldots, V_k$, which is of largest weight among all such minimum cuts. Since C_1 is a cut separating the same two sets of vertices, $w(C_1) \geq w(C^1)$. We assume that $C^1 \neq C_1$ for all such cuts C^1 and will prove a contradiction. **case(i)**: C^1's source set V_s contains vertices of $(V - V_1)$. Consider the set of edges of C^1 that have both endpoints in $(V - V_1)$, C^1_{ext}, and the edges of C_1 that have both endpoints in V_s, C_{1ext}. Now, both contradictory inequalities (1) and (2) must apply:

1. $w(C_{1ext}) < w(C^1_{ext})$, since else, exchange C_{1ext} by C^1_{ext} and get still a k-cut of

smaller or equal weight with one component strictly containing V_1. Note that all edges of the optimal k-cut are kept, except perhaps for the ones that have both endpoints in V_s). The inequality must be strict by the maximality of V_1.

2. $w(C^1_{ext}) \leq w(C_{1ext})$. This follows from the fact that C^1 is a minimum cut and, if $C_{1ext} < C^1_{ext}$, an exchange will result in a smaller cut separating the same two sets of vertices.

It follows that case(i) is impossible. In particular, it follows that it is impossible to have $w(C_1) = w(C^1)$ where $C_1 \neq C^1$. This is ruled out since, on the one hand, from case (i), V_s cannot contain vertices of $V - V_1$. On the other hand, V_s cannot be strictly contained in V_1 where $w(C_1) = w(C^1)$ due to its being a maximal source set of any minimum cut, while C_1 is also a cut of equal weight with a source set containing V_s. Therefore, it suffices to assume henceforth, that $w(C^1) < w(C_1)$. This will lead to a contradiction shown in case (ii) thus proving that $C^1 = C_1$.

case(ii): For all cuts with $(k-2)$ core in V_1, $V_s \subset V_1$. We shall show that a smaller k-cut than the optimum can be generated, thus leading to a contradiction.

For the sake of clarity, the remainder of the proof is first illustrated for $k = 4$.

Let $\{s_1, s_2\} \in V_1$ be the core of the cut C^1. Since the source set V_s is strictly contained in V_1, there exists a vertex v_1 in $V_1 - V_s$. Consider now the core $\{s_2, v_1\}$ and the corresponding minimum cut C'' with its maximal source set V_s. By the maximality rule, V_s does not contain s_1. We claim that $C^1 \bigcup C''$ creates a 4-cut in the graph $G = (V, E)$: C^1 separates V into two sets, V_s, and $V - V_s$, whereas C'', on one hand, separates V_s into one set containing s_1 and another containing s_2, and, on the other hand, separates $V - V_s$ into one set containing v_1, and another containing $V - V_1$ (by case(i)).

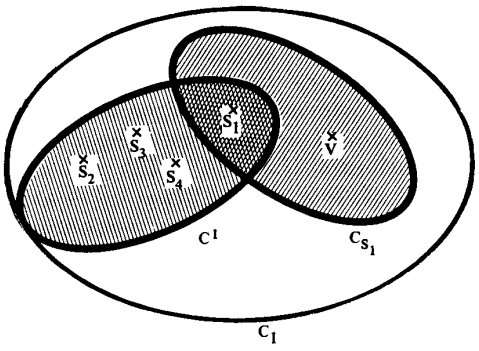

Figure 2

Therefore, we obtain a 4-cut, $C^1 \bigcup C''$, whose weight is strictly less than $2w(C_1)$. The original 4-cut, C, has weight

$$w(C) = \frac{1}{2}\sum_{i=1}^{4} w(C_i) \geq 2w(C_1),$$

which leads to a contradiction.

In the general case, the construction hereby described creates k non-empty components within V.

Let $\{s_1, \ldots, s_{k-2}\} \in V_1$ be the core vertices of the cut C^1. Since its source-set V_s is strictly contained in V_1, there exists a vertex v in $V_1 - V_s$. Consider now the core $\{s_2, \ldots, s_{k-2}, v\}$ and the source set V_{s_1}, of the minimum cut, C_{s_1}, generated with that core. C_{s_1} creates two new components, one containing s_1, and the other containing v. (Else V_s was not maximal, contradiction) Hence we now have four components in V, the two generated by C^1 and the two generated by C_{s_1}.

The following set of $k-1$ vertices is selected as the core candidates set: $\{v, s_1, \ldots, s_{k-2}\}$. Each core subsequently selected will be gen-

erated by excluding one of the vertices $\{s_2, \ldots, s_{k-2}\}$.

Let C_{s_i} be the minimum cut generated by the core in which s_i is excluded. Each of these cuts, C_{s_i}, separates at least one new component containing s_i. Some of them may separate more than one new component. Let q be the number of such cuts generating at least two components. If $q \geq (k-4)/2$, then any $(k-4)/2$ of them create at least $k-4$ new components, which added to four components, separated by $C^1 \bigcup C_{s_1}$, yield a (k+1)-cut of total weight strictly less than

$$\frac{k}{2} w(C_1) \leq \frac{k}{2} \frac{2}{k} w(C) = w(C),$$

hence a contradiction. Note that some cuts might generate more than two components in which case these components will not include any vertex of the core candidate set. This will result in a separation of more than $k-1$ components. We can though always omit segments of such cuts that create these additional components while only reducing the total weight. Up to this point, the proof is a straightforward generalization of the 4-cut construction.

Suppose that there is not a sufficient number of cuts, C_{s_i}, separating more than one component each , say, $q < (k-4)/2$ (actually it is possible that $q = 0$). Since we need an additional even number of components, there will be at least two cuts separating a single component each (to which s_j, the excluded vertex from the core candidate set, belongs). We call such cuts single-cuts. Consider the subset of its edges which is interior to V_s (both endpoints in V_s). Since all other single-cuts will contain this component (else they would be creating more than one single component) , we can uniquely represent each such cut by the section of edges of C^1 not overlapping with the cut. Also, because it is a minimum cut, the weight of the interior edges (with both endpoints in V_s) is not larger than the weight of the representing section of C^1. Since these components are disjoint, also the corresponding sections of C^1 are disjoint. The total added weight of all the interior edges of the single-component cuts is therefore not larger than $w(C^1)$ and there are at least two of them. The total weight of the k-cut obtained is hence strictly less than

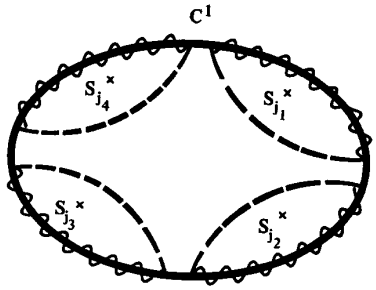

Figure 3: *The total weight represented by the dashed lines is strictly less than the total weight of the edges represented by the wiggly lines.*

$$\begin{aligned} 2w(C_1) \quad + \quad & qw(C_1) + w(C_1) \\ \leq \quad & 2w(C_1) + \frac{k-6}{2} w(C_1) + w(C_1) \\ \leq \quad & \frac{k}{2} w(C_1) \leq \frac{k}{2} \frac{2}{k} w(C) \\ = \quad & w(C) \end{aligned}$$

which is a contradiction.

For the $(k-1)$-cut one component is redundant and should be removed. This is done by considering the cut C_{s_1} with the source set V_{s_1}. The two sections of the cut, one section within V_s and the other outside V_s are considered. The larger of the two sections in terms of total edge weight is removed. The

number of components will be decreased by one as a result (either the component containing s_1 or the component containing v_1 will be eliminated). Since this cut section that now contributes only one component has weight strictly less than $w(C_1)/2$, the contradiction follows as before.

3 The algorithm and its complexity

We first describe the special case of the 3-cut and the 4-cut algorithms. Assume $|V| \geq 4$, else the partition is trivial.

Initialization: Let $w^* = \infty$

Phase1: Consider all possible partitions in which $|V_1| = 1$. For each $v \in V$, let w_1 be the sum of weights of the edges adjacent to v. Let w_2 be the minimum 2-cut in $V - \{v\}$. If $w_1 + w_2 < w^*$, then $w^* \leftarrow w_1 + w_2$ and record the resulting 3-partition (V_1, V_2, V_3).

Phase2: Consider all possible quadruple of vertices $(s_1, s_2; t_1, t_2)$. Contract s_1 and s_2 to a single vertex s and t_1 and t_2 to a single vertex t. Find a maximal source set minimum (s,t)-cut, with a source set V_s and a sink set V_t. Find a minimum (t_1, t_2)-cut in the set V_t induced in the original graph (t is expanded back to t_1 and t_2). Let the sum of these cuts be w. If $w < w^*$, then $w^* \leftarrow w$ and record the resulting 3-partition.

Output: w^* and the optimal partition (V_1, V_2, V_3).

The running time of the 3-cut algorithm is n^4 times the running time of the minimum (s,t)-cut. The fastest such algorithm currently known has complexity $O(nm\log(n^2/m))$ [GT87]. This running time is bounded by $O(n^3)$ which will be used in computing the complexity of the k-cut algorithm.

The 4-cut algorithm is essentially identical to the 3-cut algorithm except that a quintuple $(s_1, s_2; t_1, t_2, t_3)$ is selected, rather than a quadruple, and in the sink set in phase1 or phase2, we search for the optimal 3-cut using the 3-cut algorithm.

As for the general k-cut algorithm; let $k' = \begin{cases} k-2 & \text{if } k \text{ is even} \\ k-1 & \text{if } k \text{ is odd} \end{cases}$. The k-cut algorithm works recursively. For each pair of subsets, one of k' vertices and the other of $k-1$ vertices, a maximal source set minimum cut with k' vertices as a source and $k-1$ vertices as a sink is identified. The sink set is then optimally partitioned into $(k-1)$ minimum cut using a recursive application of the algorithm. For each such subset selection the weight of the resulting k-cut is evaluated. Also all k-cuts in which the first component has less than k' vertices are enumerated; here too, the remaining vertices are optimally partitioned into $(k-1)$-cut. The smallest valued k-cut and the corresponding k- partition are the output of the algorithm.

The complexity of the k-cut algorithm is $O(n^{k'+k-1})$ times n^3 and the $(k-1)$-cut complexity. For k even, the complexity is equal to $O(n^{2k-3}(n^3 + n^{2k-4}(n^3 + ...(n^5(n^3 + n^4(n^3 + n^3))))))) = O(n^{k^2})$ A precise evaluation of the exponent of n yields the running time for the k-cut algorithm:

$$\begin{array}{ll} O(n^{k^2-3k/2+2}) & \text{for } k \text{ even} \\ O(n^{k^2-3k/2+5/2}) & \text{for } k \text{ odd.} \end{array}$$

The careful reader may have already observed that rather than recurse the algorithm in order to identify a $(k-1)$-cut in the sink set for all possible subsets of vertices, it suffices to find a minimum $(k-1)$-cut that contains in each component, one of the $(k-1)$ vertices. This however, as was noted in the introduction, is an NP-hard problem so using it in the algorithm is not a wise approach. Still, a modification of this idea of reducing the num-

ber of enumerated subsets works. As it turns out, "many" choices of $(k-1)$ vertices will yield precisely the same sink set.(These are all $(k-1)$-subsets of the set of vertices constituting the sink set). Still, according to the algorithm, we are going to enumerate all possible subsets, of the required size, in order to generate the minimum $(k-1)$-cut in the sink set. This duplication can be avoided by enumerating for a given collection of $(k-1)$ "sink" vertices all possible additions of $(k-1)'-1$ vertices to one of them, where the others constitute the "sink" vertices. Note that such procedure does not deliver an optimal $(k-1)$-cut containing such vertices, one in each component. It merely ends up enumerating all possible subsets of the sink set since all $(k-1)$ subsets of the sink set will eventually be selected in the enumeration phase required for generating the first component of the k-cut. This improvement is similar to the one done for the 3-cut problem where the (t_1, t_2)- cut is identified rather than the 2-cut in the sink set. More precisely, rather than enumerating all $(k-1)'+(k-2)$-subsets in the sink set for the purpose of identifying a $(k-1)$-cut, it is sufficient to consider the subset of $(k-1)$ vertices generating the sink set and produce all subsets of $(k-1)'$ vertices including each one of those with $(k-1)'-1$ additional vertices as a core, and all the remaining $(k-2)$ vertices as sinks. The number of subsets thus enumerated is only $(k-1)\binom{n}{(k-1)'-1}$. This leads to a bound of $O(n^{k^2/2-k+11/2})$. Further improvements might still be possible, however there is no hope of removing the exponential dependence on k, unless P=NP.

4 Summary

In this paper the k-cut problem is proved to be polynomial. There is still, ample improvement possible in the running time of an algorithm solving the k-cut problem. It is conceivable for instance, that using a different approach one might be able to reduce the running time to an expression that is still exponential in k, yet the base of the exponent is a constant rather than the number of nodes, n. (2^k times a polynomial in n alone is an example of what such nice expression might look like). The exponential dependence of k must though remain, unless NP=P.

Though the algorithm for the k-cut problem is not quite what one might call practical, it nevertheless provides some valuable insight to possible approaches to implementable solutions. For one thing, one might consider an intelligent and implicit enumeration of the subsets of vertices generating the cuts, and embed such enumeration in a branch-and-bound procedure. Moreover, the algorithm supports the use of heuristic of greedy type that generate successively minimum cuts such as the 4/3 - approximation in [HT83], or the mentioned general heuristic which provides a $(2-2/k)$ - approximation for the k-cut problem in [DJP83]. These heuristics can in fact be viewed as a single application of our algorithm for one selection of a subset of vertices.

Another direction of research worth pursuing, is the generalization of the polynomiality of the k-cut problem to matroids. Let $M=(E,f)$ be a graphic matroid where E is the set of its elements (i.e. the edges of the corresponding graph) and where $f: 2^{|E|} \rightarrow \{0,\ldots,|V|-1\}$ is the rank function of the matroid. It is known [Law76] that, if $A \subseteq E$ is a subset of the elements of a graphic matroid, then $f(A)=|V|-k$ where k is the number of components in the corresponding subgraph $G_A=(V,A)$. Hence the k-cut problem can be interpreted as: find a maximum weighted subset of the elements of a graphic matroid such that the value of the rank function on this subset is equal to $|V|-k$. An interesting question is the polynomiality of this problem for other

types of matroids.

5 For arbitrary k, the k-cut problem is NP-complete

The maximum clique decision problem is the following: given a graph G and a positive integer M, does the largest complete subgraph in G contain exactly M vertices.

The (0,1)-weighted k-cut problem is equivalent to finding a partition of V into k non empty components such that the number of edges having both ends in the same component is maximum. A simple calculation shows that if G has a clique of size $M = |V| - (k-1)$, then any solution of the k-cut problem will exhibit such clique as one of its component. Hence the maximum clique problem for arbitrary k reduces to the k-cut problem with arbitrary k and, since the maximum clique problem is NP-complete for arbitrary k, so is the k-cut problem for arbitrary k.

Acknowledgement

The authors are grateful to Stuart G. Mentzer for pointing out a flaw in a previous version of the main proof.

References

[Chv73] V. Chvátal. Tough graphs and hamiltonian circuits. *Discrete Mathematics*, 5:215 – 228, 1973.

[Cun85] W.H. Cunningham. Optimal attack and reinforcement of a network. *JACM*, 32(3):549 – 561, 1985.

[DJP83] Dalhaus, D.S. Johnson, C.H. Papadimitriu, P. Seymour, and M. Yanakakis. The complexity of the multiway cuts. Extended Abstract, 1983.

[GT87] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. In *Proc. 18th Annual ACM Symposium on Theory of Computing*, pages 136 – 146, 1987.

[Gus83] D. Gusfield. Connectivity and edge disjoint spanning trees. *Information Processing Letters*, 16:87 – 89, 1983.

[Gus87] D. Gusfield. Computing the strength of a network in $O(|V|^3|E|)$ time. Technical Report CSE-87-2, U.C. Davis, 1987. Computer Science Division.

[HS85] D.S. Hochbaum and D.B. Shmoys. An $O(|V|^2)$ algorithm for the planar 3-cut problem. *SIAM J.on Algebraic and Discrete Methods*, 6:4:707 – 712, 1985.

[HT83] D.S. Hochbaum and L. Tsai. A greedy algorithm for the 3-cut problem and its worst case bound. Technical Report IP-318, U.C. Berkeley, 1983. Economic Theory and Econometrics.

[Law76] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Hort,Rinehart and Wilson, New-York, 1976.

A Las Vegas Algorithm for Linear Programming When the Dimension is Small

Kenneth L. Clarkson
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract

This paper gives an algorithm for solving linear programming problems. For a problem with n constraints and d variables, the algorithm requires an expected

$$O(d^2n) + (\lg(n/d^2))^{\lg d+2}O(d)^{d/2+O(1)},$$

arithmetic operations, as $n \to \infty$. The constant factors do not depend on d. The expectation is with respect to the random choices made by the algorithm, and the bound holds for any given input. The technique can be extended to other convex programming problems. For example, an algorithm for finding the smallest sphere enclosing a set of n points in E^d has the same time bound.

1 Introduction

In some applications of linear and quadratic programming, the number of variables will be small. Such applications include Chebyshev approximation, linear separability, and the smallest enclosing circle problem. Megiddo [Meg84] gave an algorithm for linear programming that requires $O(2^{2^d}n)$ time, where n is the number of constraints and d is the number of variables. (Unless otherwise indicated, we assume unit-cost arithmetic operations.) This time bound is optimal with respect to n, and acceptable when d is very small. Variant algorithms have been found with the slightly better time bound $O(3^{d^2}n)$ [Cla86, Dye86]. Unfortunately, Megiddo's approach must take $\Omega(d!n)$ time, since it recursively solves linear programming problems with fewer variables [Dye86]. Dyer and Frieze [DF87] used random sampling [Cla87, Cla88] to obtain a variant of Megiddo's algorithm, with an expected time bound no better than $O(d^{3d}n)$. This paper gives an algorithm requiring expected time $O(d^2n) + (\lg(n/d^2))^{\lg d+2}O(d)^{d/2+O(1)}$, as $n \to \infty$, where the constant factors do not depend on d. The leading term in the dependence on n is $O(d^2n)$, a considerable improvement in d. The second term arises from the solution by the algorithm of $O(\lg n/d^2)^{\lg d+2}$ "small" linear programs with $O(d^2)$ constraints and d variables. The solution of these linear programming problems with the simplex algorithm requires $O(d)^{d/2+O(1)}$ time.

The key idea of the algorithm is random sampling, applied as in [Cla87, Cla88], to quickly throw out redundant constraints.

The next section presents the algorithm and proves that it terminates and is correct. In Section 3, a time bound is given and proven. The last section contains some concluding remarks.

2 The LP algorithm

2.1 The problem

Suppose a system of linear inequality constraints $Ax \leq b$ is given, where A is a given $n \times d$ matrix, b is a given n-vector, and x is a d-vector $(x_1, \ldots, x_d)$. We assume for the moment that $b \geq 0$; this implies that 0 satisfies all the constraints, and so is feasible. Each inequality in this system defines a closed halfspace H of points that satisfy that inequality. The collection of these n halfspaces is a set S. The intersection $\cap_{H \in S} H$ is a polyhedral set $\mathcal{P}(S)$.

We consider the LP (Linear Programming) problem of determining the maximum x_1 coordinate of points x satisfying all the constraints, or

$$x_1^* = \max\{x_1 \mid Ax \leq b\},$$

We will use the minimum norm solution for this LP, that is, the point $x^*(S)$ with Euclidean norm $\|x^*(S)\|_2$ equal to

$$\min\{\|x\|_2 \mid Ax \leq b, x_1 = x_1^*\}.$$

It is readily shown that such a point is unique. (The requirement of a minimum norm is added to obtain

0272-5428/88/0000/0452$01.00 © 1988 IEEE

this uniqueness. For background on linear programming, see e.g. [Sch86].)

It may be that the given LP problem is unbounded, so that points with arbitrarily large x_1 coordinates are in $\mathcal{P}(S)$. It will be useful to define an answer even for such LP problems. Rather than returning a point, the algorithm will return the ray with an endpoint of 0, in the direction of $x^*(\hat{S})$, where $\hat{S}$ is the set of constraints $Ax \leq 0$, together with the constraint $x_1 = 1$.

Note that a ray $x^*(S) \subset \mathcal{P}(S)$, recalling the assumption $b \geq 0$. Because of this, we will say that $x^*(S)$ satisfies all the constraints of S. In general, a ray z will be said to satisfy a constraint halfspace H just when $z \subset H$; otherwise z violates H.

2.2 The algorithm

The optimum point (or ray) is unique; furthermore, the optimum is determined by some d or fewer constraints of S. That is, there is a set $S^* \subset S$ of size d or less such that $x^*(S^*) = x^*(S)$, so the optimum for S^* alone is the same as for S. The constraints in $S \setminus S^*$ are redundant, in the sense that their deletion from S does not affect the optimum.

The main idea of the new algorithm is the same as for Megiddo's algorithm: throw away redundant constraints quickly. The further development of this idea is very different, however. The new algorithm builds a set $V^* \subset S$ over several phases. In each phase, a set $V \subset S \setminus S^*$ is added to V^*. The set V has two important properties: its size is no more than $2\sqrt{n}$, and it contains a constraint in S^*. After $d+1$ phases, V^* contains S^*, and also V^* has $O(d\sqrt{n})$ elements. That is, the algorithm quickly throws away the large set of redundant constraints $S \setminus V^*$. The algorithm proceeds recursively with V^*, and the recursion terminates for "small" sets of constraints. For these constraints, the appropriate optima are found using the simplex algorithm.

The algorithm is given in pseudo-code in Figure 1. The optimum $x^*(S)$ is computed as follows. Let $C_d = 9d^2$. If $n \leq C_d$, then compute the optimum $x^*(S)$ using the simplex algorithm. (More precisely, compute the maximum x_1 coordinate using simplex, then use a simplex-like "active set" method [GMW81] to find the minimum norm solution.) If $n > C_d$, then repeat the following, with V^* initially empty: let $R \subset S \setminus V^*$ be a random subset of size $r = d\sqrt{n}$, with all subsets of that size equally likely. Let $x^* \leftarrow x^*(R \cup V^*)$, determined recursively, and let V be the set of constraints violated by x^*. If $|V| > 2\sqrt{n}$, then try again with a new subset R, repeating until $|V| \leq 2\sqrt{n}$. Include the final set V in V^*; if V is empty, then exit the algorithm, returning x^* as $x^*(S)$. Otherwise, stay in the loop.

```
function x*(S : set_of_halfspaces)
return x* : LP_optimum;

V* ← φ; C_d ← 9d²;
if n ≤ C_d then return simplex(S);
repeat
  repeat
    choose R ⊂ S at random, |R| = r = d√n;
    x* ← x*(R ∪ V*);
    V ← {H ∈ S | x* violates H}
  until |V| ≤ 2√n;
  V* ← V* ∪ V;
until V = φ;
return x* ;
end function x*;
```

Figure 1: The randomized function x^* for LP.

Termination of the inner loop with probability 1 follows from a bound of $\sqrt{n}$ on the expected size of V. This bound is proven in §3. By Markov's inequality, with probability at least 1/2 the set V will contain no more than $2\sqrt{n}$ constraints. This implies that the inner loop will stop after 2 iterations on average.

After the inner loop terminates, the set V has a key property: if nonempty, it contains at least one constraint of S^* that is not in V^*. Suppose x^* is a point, and that on the contrary, $V \neq \phi$ contains no constraints of S^*. Let point $x \succeq y$ if $(x_1, -\|x\|_2)$ is lexicographically greater than or equal to $(y_1, -\|y\|_2)$. We know that x^* satisfies all constraints in S^*, and so $x^*(S^*) \succeq x^*$. Since $R \cup V^* \subset S$, we know that $x^* \succeq x^*(S) = x^*(S^*)$, and so x^* has the same x_1 coordinate and norm as $x^*(S^*)$. There is only one such point in $\mathcal{P}(S^*)$, so $x^* = x^*(S^*) = x^*(S)$, and V must be empty. A similar argument holds if x^* is a ray.

Termination of the algorithm follows from this property of V. We know that after i executions of the outer loop, the set V^* contains at least i constraints of S^*. This means that by the d iterations or possibly before, we have $S^* \subset V^*$. With this true, the computed x^* will be $x^*(S)$ at the next iteration, and V will be empty, so that the algorithm stops.