OUTPUT-SENSITIVE ALGORITHMS FOR TUKEY DEPTH AND RELATED PROBLEMS

David Bremner Dan Chen John Iacono Stefan Langerman Pat Morin

ABSTRACT. The *Tukey depth* (Tukey 1975) of a point p with respect to a finite set S of points is the minimum number of elements of S contained in any closed halfspace that contains p. Algorithms for computing the Tukey depth of a point in various dimensions are considered. The running times of these algorithms depend on the value of the output, making them suited to situations, such as outlier removal, where the value of the output is typically small.

1 Introduction

Let S be a set of n points in $\mathbb{R}^d$. The *Tukey depth*, or *halfspace depth* of a point $p \in \mathbb{R}^d$ with respect to S can be defined in several equivalent ways [24]:

$$\begin{aligned} \mathrm{depth}(p,S) &= \min\{|h \cap S| : h \text{ is a closed halfspace containing } p\} && (1)\\ &= \min\{|h \cap S| : h \text{ is a closed halfspace with } p \text{ on its boundary}\} && (2)\\ &= \min\{|S'| : p \text{ is outside the convex hull of } S \setminus S'\} && (3)\end{aligned}$$

Algorithms for computing the point $p \in \mathbb{R}^d$ of maximum Tukey depth have a rich history [11, 10, 3] that has recently culminated in Chan's $O(n \log n + n^{d-1})$ expected time algorithm. A point of maximum Tukey depth serves as a d-dimensional generalization of the (1-dimensional) median and performs well as a robust estimate of the "center" of S [19, 20, 23].

In this paper we consider the simpler problem of computing the Tukey depth of a given point p with respect to a set S. Our algorithms have running times that are dependent on the value, k, of the output. These algorithms are thus particularly well-suited to problems such as outlier-removal where the goal is to identify points of small depth since they run quickly when the depth of p is small. Specifically, we present the following results:

1. A simple $O(n + k \log k)$ time algorithm for points in $\mathbb{R}^2$ (Section 2). The most complicated data structure used in this algorithm is a binary heap.
2. An $O(n + (n-k)\log(n-k))$ time algorithm to find the largest clique in an interval graph, where k is the size of the clique found (Section 3). This problem is related to the Tukey depth problem in $\mathbb{R}^2$.
3. An $O(n \log n + k^2 \log n)$ time algorithm for points in $\mathbb{R}^3$ and an $O(n + k^{11/4} n^{1/4} \log^{O(1)} n)$ time algorithm for points in $\mathbb{R}^4$ (Section 4). These algorithms rely on results of Chan on linear programming with violated constraints [4] which in turn rely on sophisticated range searching data structures [12, 18] and/or dynamic convex hull data structures [2].

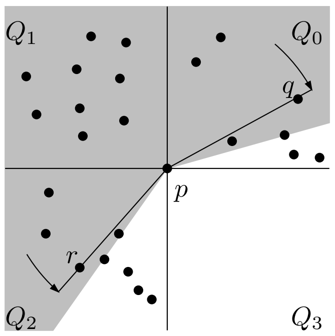

Figure 1: Computing the quantity $\mathrm{depth}_1(p, S)$.

4. A simple $O(d^k \,\mathrm{LP}(n, d-1))$ time algorithm for points in $\mathbb{R}^d$, where $\mathrm{LP}(n, d)$ denotes the time required to determine the feasibility of a linear program having n constraints and d variables (Section 5). Not surprisingly, this algorithm is also based on linear programming with violated constraints and is obtained by presenting a fixed-parameter tractable algorithm for a parameterization of the NP-hard MAXIMUMFEASIBLESUBSYSTEM problem.

For the remainder of this paper we use the following notations: For points $p, q \in \mathbb{R}^d$, p_i denotes the ith coordinate of p, $\|p\| = (\sum_{i=1}^{d} p_i^2)^{1/2}$, and $p \cdot q = \sum_{i=1}^{d} p_i q_i$. The unit sphere in $\mathbb{R}^{d+1}$ is denoted by $\mathbb{S}^d = \{p \in \mathbb{R}^{d+1} : \|p\| = 1\}$. The top side of this sphere is denoted by $\mathbb{S}^d_+ = \{p \in \mathbb{S}^d : p_{d+1} > 0\}$, the bottom side is denoted by $\mathbb{S}^d_- = \{p \in \mathbb{S}^d : p_{d+1} < 0\}$ and the equator is denoted by $\mathbb{S}^d_0 = \{p \in \mathbb{S}^d : p_{d+1} = 0\}$.

2 An Algorithm for Points in $\mathbb{R}^2$

In this section we give a simple $O(n + k \log k)$ time algorithm to compute the Tukey depth of a point $p \in \mathbb{R}^2$ with respect to a set S of n points in $\mathbb{R}^2$. We first note that an $O(n \log n)$ time *sort-and-scan* algorithm is easily obtained by sorting the points of S radially about p and then scanning the resulting sorted list using two pointers [11]. The main idea behind our algorithm to is to reduce the problem to a *kernel* of size $O(k)$ on which we can apply this sort-and-scan algorithm.

The algorithm begins by partitioning $\mathbb{R}^2$ into 4 quadrants around p that, in counterclockwise order, we denote by $Q_0, \ldots, Q_3$. The algorithm then simultaneously begins computing the 4 quantities $\mathrm{depth}_0(p, S), \ldots, \mathrm{depth}_3(p, S)$ where

$$\mathrm{depth}_i(p, S) = \min\{|h \cap S| : h \text{ is a closed halfspace containing } Q_i\} \ . \tag{4}$$

Clearly, $\mathrm{depth}(p, S) = \min\{\mathrm{depth}_i(p, S) : 0 \le i \le 3\}$ since any closed halfspace containing p contains at least one of the four quadrants. In the remainder of this section we will describe how to compute $k_i = \mathrm{depth}_i(p, S)$ in $O(n + k_i \log k_i)$ time. Since the computation can stop once $\mathrm{depth}_i(p, S)$ has been computed for the index i that minimizes (4), running the computation of $k_0, \ldots, k_3$ in parallel yields an $O(n + k \log k)$ time algorithm, where $k = \mathrm{depth}(p, S)$.

Let $S_i = S \cap Q_i$. To compute $\mathrm{depth}_i(p, S)$ we create two binary heaps H_{i-1} and H_{i+1} that store the elements of S_{i-1}, respectively S_{i+1}, in clockwise, respectively, counterclockwise, order around p.[1] Creating these two heaps takes $O(n)$ time using the standard bottom-up algorithm to construct a binary heap [7, Chapter 6]. Next we extract elements one at a time from each of H_{i-1} and H_{i+1} until either (a) one of the heaps is empty or (b) we extract two elements q from H_{i-1} and r from H_{i+1} such that the angle $\angle qpr > \pi$. Suppose we have extracted ℓ elements each from H_{i-1} and H_{i+1} when this occurs. Then it is easy to verify that

$$|S_i| + \ell - 1 \leq \mathrm{depth}_i(p, S) \leq |S_i| + 2\ell - 1 \enspace .$$

Next, we continue to extract as many elements as possible from each of H_{i-1} and H_{i+1} up to a maximum of an additional $\ell - 1$ elements each. The total time required to extract these at most $4\ell - 2$ elements from the two heaps is $O(\ell \log n)$. By sorting and scanning all the elements extracted from the heaps plus the elements of S_i we can then compute $\mathrm{depth}_i(p, S)$ in an additional

$$O((|S_i| + \ell) \log n) = O(k_i \log n)$$

time. This yields an a total running time of

$$O(n + k_i \log n) = O(n + k_i \log k_i) \enspace ,$$

as required. This completes the proof of:

Theorem 1. *The Tukey depth of a point p with respect to a set S of n points in $\mathbb{R}^2$ can be computed in $O(n + k \log k)$ time, where k is the value of the output.*

3 An Algorithm for Max-Clique in Interval Graphs

The problem of computing Tukey depth in $\mathbb{R}^2$ can be viewed as a problem on a set of circular arcs. By (2), computing the Tukey depth of p is equivalent to finding a unit normal vector v such that the halfspace with p on its boundary and having inner normal v contains as few points of S as possible. Note that the set of unit normals in $\mathbb{R}^2$ is homeomorphic to the unit circle $\mathbb{S}^1$ and that each point $q \in S$ defines an open circular arc of $\mathbb{S}^1$ such that all choices of v in this circular arc yield a halfspace that does not contain q. Thus, the Tukey depth problem reduces to the problem of finding a vector v that is contained in the largest number of circular arcs. The partitioning into 4 quadrants used in the algorithm of Theorem 1 works because all the circular arcs are actually half circles.

An obvious generalization of the Tukey depth problem is that of, given a set of n circular arcs of $\mathbb{S}^1$, finding a point $p \in \mathbb{S}^1$ contained in the largest number of arcs. This problem is easily solved in $O(n \log n)$ time by the sort-and-scan algorithm. Unfortunately, it is not possible to obtain an algorithm whose running time depends on the number k of arcs containing p or even on the number $(n-k)$ of arcs not containing p. This is because the decision problem of testing whether a set of n arcs covers $\mathbb{S}^1$ has an $\Omega(n \log n)$ lower-bound [1]. This problem is equivalent, by taking the complement of each arc, to the problem of finding the point contained in the maximum number of arcs. In particular, the original set of arcs do not cover $\mathbb{S}^1$ if and only if there is a point p contained in every complementary arc.

Since we can not hope to solve the problem for circular arcs of $\mathbb{S}^1$, we settle for the next best thing. Let I be a set of real intervals. Here we describe an $O(n+(n-k)\log(n-k))$ time algorithm to find

[1]Here and in the remainder of this section S_i is treated implictly as $S_{i \bmod 4}$.

a point $p \in \mathbb{R}$ that is contained in the largest number of intervals in I. Here k is the number of intervals in I that contain p. Let $p_1, \ldots, p_{2n}$ denote the $2n$ endpoints of the intervals in I, in increasing order. For convenience we use the convention that $p_i = -\infty$ for $i \leq 0$ and $p_i = +\infty$ for $i > 2n$. Together, the following two observations imply that all the points contained in many intervals are clustered together.

Lemma 1. *Let $q \in [p_i, p_{i+1}]$ be a point contained in k intervals of I. Then, for any $0 \leq r \leq n$, every point $q' \in [p_{i-r}, p_{i+r+1}]$ is contained in at least $k - r$ intervals of I.*

Proof. Without loss of generality, assume that $q' \in [q, p_{i+r+1}]$. There are at most r endpoints of intervals in I contained in the interval $[q, q']$. Therefore there are at most r intervals that contain q but not q'. □

Lemma 2. *Let $q \in [p_i, p_{i+1}]$ be a point contained in k intervals of I. Then, for any $n - k \leq r \leq n$, every point $q' \notin [p_{i-r}, p_{i+r+1}]$ is contained in at most $2n - k - r$ arcs of C.*

Proof. Without loss of generality, assume that $q' > p_{i+r+1}$. Then, as we walk from q to q' we encounter at least r endpoints of intervals in I. At most $n - k$ of these endpoints are left endpoints of intervals and at least $r - (n - k)$ of these are right endpoints. Thus, the number of intervals that contain q' is at most

$$k + (n - k) - (r - (n - k)) = 2n - k - r \ ,$$

as required. □

At a high level our algorithm is fairly simple. Suppose we are given a value k and only wish to find a value $p \in \mathbb{R}$ contained in at least k intervals of I. We begin by taking a regular sample $s_1, \ldots, s_{2t}$ of $p_1, \ldots, p_{2n}$ so that any interval $[s_i, s_{i+1}]$ between two consecutive sample points contains at most n/t points of $p_1, \ldots, p_{2n}$. We then compute, for each sample point s_i the number of intervals in I that contain s_i. By Lemma 1, if there exists any point $p \in \mathbb{R}$ contained in k intervals of I then the two sample points s_j and s_{j+1} on either side of p are *high depth samples* that are each contained in at least $k - n/t$ intervals of I. Furthermore, by Lemma 2, the only high depth samples are contained in the interval $[p_{i-r}, p_{i+r}]$ for $r = 2(n - k) + n/t$.

If we choose $t = \sqrt{n}$ then $r = O(n - k + \sqrt{n})$. Thus, by computing an interval $[p_a, p_b]$ that contains all high depth samples we can find the point p contained in the largest number of intervals of C by applying the standard sort-and-scan algorithm on the $O(n - k + \sqrt{n})$ endpoints of the intervals of C that fall in the interval $[p_a, p_b]$. The running time of the sort-and-scan algorithm is $O(m \log m)$ where m is the number of points to be scanned. In this case $m = O(n - k + \sqrt{n})$ for a running time of

$$O((n - k + \sqrt{n}) \log(n - k + \sqrt{n})) = O(n + (n - k) \log(n - k)) \ ,$$

as required.

In implementating the above ideas, several issues arise:

1. The value of k is not known in advance. However, we do not need the exactly value of k and the value of k can be estimated to within an additive error of $\sqrt{n}$ by computing, for each sample point s_i, the number of intervals of I that contain s_i (see Issue 3, below) and using the maximum of these values as an estimate for k.

2. We can not obtain a perfectly regular sample $s_1, \ldots, s_{2\sqrt{n}}$ of $p_1, \ldots, p_{2n}$ in $O(n)$ time. However, we do not require a perfectly regular sample. By taking a random sample of size $c\sqrt{n}\log n$ for an appropriate constant c we obtain a set of samples $s_1, \ldots, s_{c\sqrt{n}\log n}$ such that, with high probability, no interval $[s_i, s_{i+1}]$ contains more than $\sqrt{n}$ endpoints of intervals of I [16].

3. We can not compute, in $O(n)$ time, for each sample point s_i, the number of intervals of I that contain s_i. However, random sampling helps again here. Let $d(s_i)$ denote the number of elements of I that contain s_i. By taking a random sample $I' \subseteq I$, $|I'| = \sqrt{n}$ we can determine for each s_i a number d_i such that, with high probability,

$$d(s_i) - O(n^{4/5}) \leq d_i \leq d(s_i) + O(n^{4/5}) \enspace .$$

By storing the $\sqrt{n}$ elements of I' in an interval tree [17] and then querying this interval tree with the $c\sqrt{n}\log n$ sample elements the numbers $d_1, \ldots, d_{c\sqrt{n}\log n}$ can be computed in $O(\sqrt{n}\log^2 n)$ time.

None of the above issues have any significant effect on the running time of the overall algorithm, which is still dominated by the final sort-and-scan step on a problem of size $O(n - k + \sqrt{n})$. The correctness of the resulting output depends on the success of the random sampling steps described in points 2 and 3, above. However, Lemma 2 implies that this final sort-and-scan step allows us to check the correctness of the output and restart the algorithm from scratch if necessary. This yields:

Theorem 2. *There exists a randomized algorithm that, given a set I of n real intervals, finds a value $p \in \mathbb{R}$ contained in the largest number of intervals of I and that runs in $O(n + (n-k)\log(n-k))$ expected time.*

4 Algorithms for Points in $\mathbb{R}^3$ and $\mathbb{R}^4$

The previous section showed how the problem of computing the Tukey depth of a point in $\mathbb{R}^2$ is equivalent to the problem of finding a point contained in the largest number of halfcircles on the unit circle $\mathbb{S}^1$. A similar statement is true in $\mathbb{R}^d$: Each point $q \in S$ defines an open halfsphere $q^* = \{v \in \mathbb{S}^{d-1} : v \cdot q < 0\}$. That is, all vectors in q^* are the inner normals of hyperplanes that contain p but do not contain q. Thus, the problem of determining the Tukey depth of p reduces to the problem of finding the point contained in the largest number of halfspheres in $S^* = \{q^* : q \in S\}$.

We observe that this problem can be solved by solving three problems in $\mathbb{R}^{d-1}$. Each open halfsphere $q^* \in S^*$ is the intersection of an open halfspace $q^\#$ with $\mathbb{S}^{d-1}$. Consider the intersection of $q^\#$ with the hyperplane $H_+ = \{(x_1, \ldots, x_d) : x_d = 1\}$. By central projection, there is a 1-1 correspondence between points in $\mathbb{S}_+^{d-1}$ and H_+ and this projection has the property that $r \in \mathbb{S}_+^{d-1}$ is in q^* if and only if the projection of r is in $q^\# \cap H_+$. Thus, finding the point in $\mathbb{S}_+^{d-1}$ contained in the largest number of halfspheres is equivalent to finding a point in H_+ contained in the largest number of halfspaces. A similar statement holds regarding $\mathbb{S}_-^{d-1}$ using the hyperplane $H_- = \{(x_1, \ldots, x_d) : x_d = -1\}$. Finally, finding the point in $\mathbb{S}_0^{d-1}$ contained in the smallest number of halfspheres is a $(d-1)$-dimensional Tukey depth problem.

The above discussion shows that computing the Tukey depth of a point in $\mathbb{R}^d$ reduces to one Tukey depth problem in $\mathbb{R}^{d-1}$ and two instances of the problem MAXIMUMFEASIBLESUBSYSTEM in $\mathbb{R}^{d-1}$: Given set K of n halfspaces in $\mathbb{R}^{d-1}$, find the subset K' of K of minimum cardinality such that $\cap(K \setminus K')$ is non-empty. The current best results for MAXIMUMFEASIBLESUBSYSTEM in small dimensions are due to Chan [4]. Using two instances of his algorithm for MaximumFeasibleSubsystem in $\mathbb{R}^2$, respectively, $\mathbb{R}^3$, and running them in parallel gives:

Theorem 3. *The Tukey depth of a point p with respect to a set S of n points in $\mathbb{R}^3$ can be computed in $O(n \log n + k^2 \log n)$ time, where k is the value of the output.*

Theorem 4. *The Tukey depth of a point p with respect to a set S of n points in $\mathbb{R}^4$ can be computed in $O(n \log n + k^{11/4} n^{1/4} \log^{O(1)} n)$ time, where k is the value of the output.*

5 An Algorithm for Points in $\mathbb{R}^d$

Finally, we consider the general case of point sets in $\mathbb{R}^d$. In the previous section we showed that computing the Tukey depth of a point p with respect to a set S of n points in $\mathbb{R}^d$ can be reduced to two instances of MaximumFeasibleSubsystem in $\mathbb{R}^{d-1}$ and one Tukey depth computation in $\mathbb{R}^{d-1}$. In this section we give a fixed-parameter tractable [8] algorithm for MaximumFeasibleSubsystem.

The algorithm uses linear programming as a subroutine in the following way: Given a collection K of halfspaces in $\mathbb{R}^{d-1}$, an algorithm for linear programming can be used to either

1. Determine a point $p \in \cap K$ if such a point exists or,
2. report a subset $B \subseteq K$, $|B| \le d$, such that $\cap B = \emptyset$.

The set B reported in the latter case is called a *basic infeasible subsystem*. Standard combinatorial algorithms for linear programming, including algorithms for linear programming in small dimensions [6, 9, 13, 14, 21, 22] as well as the simplex method (c.f., [5]), can easily be made to report a basic infeasible subsystem. A method of finding basic infeasible subsystems from interior point linear-programming methods is described in Appendix A.

Let $\text{BIS}(K)$ denote a routine that outputs a basic infeasible subsystem of K if K is infeasible, and that outputs the empty set otherwise. The following algorithm solves the MaximumFeasibleSubsystem decision problem:

```
MFS($K, k$)
 1: {$\star$ determine if there exists $K' \subseteq K$, $|K'| \le k$, such that $\cap(K \setminus K') \neq \emptyset$ $\star$}
 2: $B \leftarrow \text{BIS}(K)$
 3: if $B = \emptyset$ then
 4:     return true
 5: if $k = 0$ then
 6:     return false
 7: for each $h \in B$ do
 8:     if MFS($K \setminus \{h\}, k-1) = \text{true}$ then
 9:         return true
10: return false
```

Correctness of the above algorithm is easily established by induction on the value of k. The running time of the algorithm is given by the recurrence

$$T(n,k) \le \text{LP}(n, d-1) + dT(n-1, k-1) \ ,$$

where $\mathrm{LP}(n, d)$ denotes the running time of an algorithm for solving a linear programming with n constraints and d variables. This recurrence readily resolves to $O(d^k \,\mathrm{LP}(n, d-1))$.

Using this as a subroutine for Tukey depth computation we obtain an algorithm whose running time is given by the recurrence

$$S(n, d, k) \leq O(d^k \,\mathrm{LP}(n, d-1)) + S(n, d-1, k)$$

which resolves to $O((d+1)^k \,\mathrm{LP}(n, d-1))$. Running this algorithm for $k = 0, 1, 2, \ldots$ completes the proof of:

Theorem 5. *The Tukey depth of a point p with respect to a set S of n points in $\mathbb{R}^d$ can be computed in $O((d+1)^k \,\mathrm{LP}(n, d-1))$ time, where k is the value of the output and $\mathrm{LP}(n, d)$ is the time to solve a linear program with n constraints and d variables.*

References

[1] M. Ben-Or. Lower bounds for algebraic computation trees (preliminary report). In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC83)*, pages 80–86, 1983.

[2] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pages 617–626, 2002.

[3] T. M. Chan. An optimal randomized algorithm for maximum Tukey depth. In *Proceedigs of the 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 423–429, 2004.

[4] T. M. Chan. Low-dimensional linear programming with violations. *SIAM Journal on Computing*, 34:879–893, 2005.

[5] Vašek Chvátal. *Linear programming*. A Series of Books in the Mathematical Sciences. W. H. Freeman and Company, New York, 1983.

[6] K. L. Clarkson. Las vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM*, 42:488–499, 1995.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, second edition, 2001.

[8] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1998.

[9] M. E. Dyer. Linear time algorithms for two- and three-variable linear programs. *SIAM Journal on Computing*, pages 31–45, 1984.

[10] S. Langerman and W. Steiger. Optimization in arrangements. In *Proceedings of the 20th Symposium on Theoretical Aspects of Computer Science*, volume 2607 of *Lecture Notes in Computer Science*, pages 50–61. Springer-Verlag, 2003.

[11] J. Matoušek. Computing the center of planar point sets. In J. E. Goodman, R. Pollack, and W. Steiger, editors, *Computational Geometry: Papers from the Special Year*, pages 221–230. AMS, Providence, 1991.

[12] J. Matoušek. Reporting points in halfspaces. *Computational Geometry: Theory and Applications*, 2:169–186, 1992.

[13] N. Megiddo. Linear time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM Journal on Computing*, 12:759–776, 1983.

[14] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM*, 31:114–127, 1984.

[15] Nimrod Megiddo. On finding primal- and dual-optimal bases. *ORSA J. Comput.*, 3(1):63–65, 1991.

[16] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, 1998.

[17] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New-York, 1985.

[18] E. Ramos. On range reporting, ray shooting, and k-level construction. In *Proceedings of the 15th ACM Symposium on Computational Geometry (SoCG 1999)*, pages 390–399, 1999.

[19] P. J. Rousseeuw and I. Ruts. Constructing the bivariate tukey median. *Statistica Sinica*, 8:827–839, 1998.

[20] I. Ruts and P. J. Rousseeuw. Computing depth contours of bivariate point clouds. *Computational Statistics and Data Analysis*, 23:153–168, 1996.

[21] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6:423–434, 1991.

[22] M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In *Proceedings of the 9th Symposium on Theoretical Aspects of Computer Science*, volume 5777 of *Lecture Notes in Computer Science*, pages 569–579, 1992.

[23] C. G. Small. A survey on multidimensional medians. *International Statistics Review*, 58:263–277, 1990.

[24] J. W. Tukey. Mathematics and the picturing of data. In *Proc. International Congress of Mathematicians*, volume 2, pages 523–531, 1975.

[25] Stephen A. Vavasis and Yinyu Ye. Identifying an optimal basis in linear programming. *Annals of Operations Research*, 62:565–572, 1996. Interior point methods in mathematical programming.

A Computing a Basic Infeasible Subsystem

For any matrix M, let M_J denote the set of rows indexed by J. Given a system of linear inequalities $Mx \geq b$, $M \in \mathbb{R}^{m \times d}$, a *basic infeasible subsystem* is a subset of $\{1 \dots m\}$ such that the system $M_I x \geq b_I$ is infeasible, and $|I| \leq d+1$. We consider the standard first stage simplex problem (see e.g. [5], p. 39). Let e denote the m-vector of all ones, c the length $d+1$ binary vector with exactly one one in the last position and let $A = [Me]$. We can write the first stage LP for our system as

$$\begin{aligned} &\min c^T x = x_{d+1} \\ &\text{subject to} \\ &Ax \geq b \end{aligned} \tag{P}$$

In the case of an infeasible system, the optimal value of this LP will be strictly positive. The dual LP of (P) is

$$\begin{aligned} &\max b^T y \\ &\text{subject to} \\ &yA = c \\ &y \geq 0 \end{aligned} \tag{D}$$

In what follows, we generally follow the notation of [15], except that we interchange the definitions of the primal and dual LPs. Define a *basic partition* (or just *basis*) (β, η) as a partition of the row indices of A such that A_β is nonsingular. For each basic partition, we define a *primal basic solution*

$$x^* = A_\beta^{-1} b_\beta$$

and a *dual basic solution*

$$y^* = cA_\beta^{-1}$$

We say that a basis is *primal feasible* (resp. *dual feasible*) if x^* is feasible for (P) (respectively y^* is feasible for (D)). It is a standard result of linear programming duality that a basis which is both primal and dual feasible defines a pair (x^*, y^*) of optimal solutions to the primal and dual LP's; such a partition is called an *optimal basis partition*.

In general LP algorithms (either directly in the case of Simplex type algorithms, or via postprocessing using e.g. [15, 25]) provide an optimal basis partition (β, η). Consider the relaxed LP

$$\begin{aligned} &\min c^T x \\ &\text{subject to} \\ &A_\beta x \leq b_\beta \end{aligned} \tag{R}$$

It is easy to verify that an optimal basis partition for (P) is also primal and dual feasible for (R). This implies that the system $M_\beta \geq b_\beta$ is infeasible, and provides a a basic infeasible system.