Experimental Evaluation of Randmozed Pivoting Rules for the Simplex Method

Goktug Saatcioglu

May 16, 2019

Abstract

This project explores randomized pivoting strategies for the simplex method and experimentally evaluates their effectiveness against deterministic strategies. The study shows that randomization has the potential to be an effective alternative to current pivoting rules but numerical stability must also be considered. Possible directions for future work are also highlighted.

1 Introduction

The simplex method for solving linear programming problems is often considered as one of the most important algorithms developed in the 20th century [20]. A simplex or simplex-like method for finding the solution to a linear program is incomplete without a pivoting rule or strategy being specified. The primary goal of pivoting is to define a new search direction for the simplex method which in turn defines a new vertex with a lower value of the objective function. In addition to finding an optimal vertex, we would also like our pivoting rule to get there as fast as possible and not be effected by stalling and/or cycling in the case of degenaracy. These goals can sometimes be in conflict with each other as fast rules such as Dantzig's least index rule can find a solution quickly for most problems but does not guarantee termination due to cycling. On the other hand, Bland's least index rules as introduced in [4] are guaranteed to terminate but are considerably slower in practice. Thus, a choice must be made which greatly influences the observed performance when solving a linear program using the simplex method. Interestingly, the choice of a pivot strategy also plays an important role in the theoretical run-time properties of simplex and simplex-like methods.

The theoretical run-time of a simplex or simplex-like method is primarily determined by its pivoting rule. Since the other aspects of the simplex method, including solving a system of equations and checking for optimality, can be done in polynomial time, the pivoting rule will determine which vertices will be visited and in which order they will be visited in. This in turn determines the number of iterations needed before finding a solution to a feasible and bounded linear program. The process of selecting a pivot in general runs in polynomial time. However, the actual path a pivoting strategy leads to can cause the simplex method to run in exponential, and sometimes sub-exponential, time for a specific linear program. The question of whether the simplex method is a strongly polynomial algorithm is still an open question and this is because all currently known pivoting rules lead to at worst exponential time complexity [30]. However, we also do not currently have a proof showing that the simplex method cannot be strongly-polynomial for all pivoting methods. Beginning with the construction of the Klee-Minty cube to show that Dantzig's rule leads to exponential time performance of the simplex method [2], for each new proposed pivoting algorithm it has either been proven that the rule has exponential time performance in the worst case or its worst-case complexity is not yet known. Most of these proofs are done by creating a specific polytope or problem where the pivoting rule exhibits the desired worst-case behavior.

This search for a pivoting rule that leads to a polynomial time simplex instance has sparked an interest in randomized pivoting strategies. Recent advances include the development of randomized simplex-like algorithms that have sub-exponential worst-case upper bounds. A nice survey of these rules can be found in [16]. Due to the analysis of Spielman and Teng we also know that the simplex method for deterministic pivoting rules, specifically for the shadow vertex method, has polynomial smoothed complexity [24]. That is, this work provides a theoretical explanation to why the deterministic pivoting rules in practice exhibit polynomial time behavior for most problems. No such work exists for randomized simplex methods and furthermore, no known major experimental evaluations or implementations of randomization exist. Overall, research into randomized simplex methods shows certain benefical theoretical properties but experimental studies are hard to come by.

This project aims to explore and experimentally evaluate two randomized simplex algorithms and one simplex-like algorithm and compare them against three well known and widely used deterministic simplex pivoting rules. The main goal is to observe numerical issues and the behavior of the randomized methods in a practical setting.

We consider the random edge and random facet rules alongside a modified version of Clarkson's first algorithm from [14] as our randomized implementations. Our determinsitic strategies are Dantzig's rule, Bland's rules and the steepest edge rule. Section 2 briefly goes over linear programming basics and concepts. Then we move onto section 3 where we describe the three deterministic pivoting rules and the three random pivoting strategies that will be experimentally evaluated. Section 4 goes over the Matlab scripts implemented for the project. Then, section 5 presents the experiments run and the results obtained where 1 is a table that attempts to summarize the results. Here an analysis of the pivoting rules is also given. In section 6 the flaws in the analysis are outlined along. Finally, section 7 concludes the paper. A GitHub repository containing all the material used for this project can be found here: `https://github.com/goki0607/randomized-pivoting-project`.

2 Linear Programming Overview

We present a brief overview of linear programming definitions that will be used throughout the project. Furthermore, we provide an overall idea of the simplex method. For a more detailed treatment of the subject matter we suggest looking at [25], [9], [8] and [33] which are all excellent resources.

This project considers a linear program in all-inequality form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq B \end{array}$$

where c^T is the $1 \times d$ objective function vector which we wish to minimize, A is the $n \times d$ matrix of inequality constraints and B is the $n \times 1$ vector of lower bounds of the inequalities given in A. A point is $\bar{x}$ is said to be feasible if and only if

$$A\bar{x} \geq B$$

and a feasible set/region is the set of all points S such that

$$\forall \bar{x} \in S.\, A\bar{x} \geq B.$$

A feasible region is bounded if the feasible set S is finite and is unbounded if S is infinite. Otherwise, if S is empty then the problem is said to be infeasible meaning no solution exists.

A constraint $a^T \in A$ is a row of A. The active set $\mathbb{A}$ of A at $\bar{x}$ are the rows of A such that each constraint in the active set is satisfied with equality $a_i^T \bar{x} = b_i$ and any constraint that is not satisfied with equality is inactive such that $a_j^T \bar{x} > b_j$. Any subset $W \subseteq \mathbb{A}$ such that the normals of each $a_i \in W$ form an independent basis for $\mathbb{R}^m$ is referred to as a working-set of size m. Given the consistent constraint $A\tilde{x} \geq B$, a point $\tilde{x}$ is a vertex if and only if d linearly independent constraints are active at $\tilde{x}$. If exactly n linearly independent constraints are active then the vertex is non-degenerate and if more than d linearly independent constraints are active then the vertex is degenerate.

The simplex method finds an optimal solution to a feasible linear programming problem if it exists and otherwise reports that the problem is unbounded. It does so by obtaining what are referred to as Lagrange multipliers which are the solution to the system of equations

$$W^T \lambda = c$$

where W is the $n \times d$ non-singular (by definition) matrix of the working set constraints at some feasible point $\hat{x}$. If all entries of the vector λ are ≥ 0 then we say that $\hat{x}$ is a solution to the linear program. Otherwise, if there exists at least one component that is < 0 then we can define a feasible search direction $\hat{p}$ that moves off the constraint corresponding to that multiplier and towards another constraint which defines a new working set $\tilde{W}$. This new working set also defines a new point $\tilde{x}$ which corresponds to a strictly lower value of the objective function. Furthermore, if we can take a step of infinity, i.e. there is no constraint blocking us from decreasing the objective function arbitrarily, then we say that objective function is unbounded from below and the solution to the problem is $-\infty$. A pivoting rule in this context helps us define which constraint to move off from if there are multiple candidates for removal and which one to add into the new working set if there are more than one blocking constraints.

At a very high level, the simplex method moves from vertex to vertex of the polytope defined by the linear program until it finds an optimal solution or determines that the objective function is unbounded. As an additional note, degeneracy refers to when we have a degenerate vertex such that the working set of this vertex is ambiguously defined. The ambiguity arises from the fact that the working set must always be square to solve the system of equations for the search direction but the active set at a degenerate point is not square. Thus, there are many working sets we can choose from. This is problematic as it can lead to cycling and/or stalling where we simply cycle through all the active constraints at the degenerate point adding and removing them from the working set without making any progress.

An all-inequality simplex method requires an intial feasible point x_0 that has a working set W_0 to begin the search process. The task of finding an initial feasible point can also be done by linear programming and this is referred to as Phase 1. While we skip over the details here, the code given in appendix B demonstrates how we use a Phase 1 LP to find an initial feasible point. If the initial problem is infeasible then the Phase 1 LP will also be infeasible such that we can detect infeasibility at this stage. Then, if we find a feasible point, we continue onto Phase 2 to solve the actual problem using the x_0 obtained from Phase 1. An implementation of the simplex method is given in appendix C. As can be seen, we keep iterating until the optimality conditions are met and at each iteration use a pivot strategy to find the search direction. The variable `s` refers to the variable leaving the working set and `t` refers to the variable entering the working set.

Finally, for the sake of thoroughness, the dual of an all-inequality problem is given by the following linear program

$$\begin{aligned} &\text{minimize} \quad B^T\lambda \\ &\text{subject to} \quad A^T\lambda = c,\ \lambda \geq 0 \end{aligned}$$

where B^T is the $1 \times n$ objective function vector which we wish to minimize, A^T is the $d \times n$ matrix of equality constraints and C is the $d \times 1$ vector of lower bounds of the inequalities given in A. A good treatment of duality theory is given in [21].

This project focues on the pivoting strategies that define `s` and `t` and complete the specification of the simplex method. The next section discusses various rules that will be considered for the project.

3 Pivoting Rules

We consider three deterministic pivoting strategies, one randomized pivoting strategy and two simplex-like randomized algorithms. We begin by describing the deterministic pivoting rules.

3.1 Bland's Rules

Bland's rules were introduced by Robert G Bland in [4] to address the issue of cycling due to degeneracy and ensure the simplex method terminates in a finite number of steps. Rather than picking the entering and leaving variables with respect to some property of the linear program being solved, the indices of the constraints in the order they appear in the constraint matrix are considered. These rules are also referred to as "least-index" rules.

Given one step of a simplex iteration, let λ be the Lagrange multipliers obtained in that iteration and W the current working set. Then, after checking for the optimality conditions, define the set Λ as

$$\Lambda \triangleq \{i \mid \lambda_i < 0\}$$

and define the set ω as

$$\omega \triangleq \{j_i \mid i \in \Lambda \wedge w_j \in W\}.$$

Λ is the set of indices that corresponds to the negative components of the Lagrange multipliers. Similarly, ω is the set of indices of the constraints in the working set W which corresponds to the negative multipliers. If $\Lambda = \emptyset$ then we know that we are at an optimal point. Otherwise, we define the constraint s to be removed as

$$s = \min \omega$$

so that we remove the smallest index in ω. This corresponds to deleting the constraint with a negative multiplier and the smallest index in the original constraint numbering with respect to A. Next, we need to define how to add a constraint after finding the search direction using s. We let S be the set of indices that are blocking at the current iteration so that we choose

$$t = \min S.$$

Note that if $S = \emptyset$ then the problem is unbounded. So we now add the smallest index in S which is also the smallest index in the original constraint numbering with respect to A. We keep picking s_k and t_k for the simplex iterates k until we either find an optimal vertex or discover that the program is unbounded.

Overall, Bland's rules are a simple strategy for selecting entering and leaving constraints by using a lexicographic ordering of the indices of A. It has the added benefit of resolving cycling and ensuring termination. However, the strategy is seldom used in practice as it does not perform as fast as some of the other rules which will be described. The worst-case complexity of this pivoting strategy is exponential and is shown with a slight modification to the Klee-Minty cube [2] by [10].

3.2 Dantzig's Rule

Dantzig's greedy rule is considered to be the original strategy for the simplex method proposed by George B Dantzig [33]. It is also sometimes called the textbook rule. Given the Lagrange multiplers λ at a single iteration of the simplex method, we define the set Λ as the set of negative components of λ

$$\Lambda \triangleq \{\lambda_i \mid \lambda_i < 0\}.$$

Then the leaving constraint s is defined as

$$s = i \quad \text{where} \quad \lambda_i = \min \Lambda.$$

This means that we pick the most negative multiplier which in turn maximizes the rate of decrease of objective value once the corresponding constraint is removed. Let p be the search direction obtained from s. Then the entering variable is defined from the set of blocking constraints S where

$$S \triangleq \{a_i^T p \mid a_i \text{ is blocking at } x \text{ with direction } p\}$$

such that t is the index

$$t = i \quad \text{where} \quad a_i p = \min S.$$

If there is a tie then we pick the smallest index out of the tying constraints.

Dantzig's rule is simple to implement and known to be efficient in practice but can cycle in the case of degeneracy. Some good examples of linear programs that cycle are given in [23] where it is also shown that for cycling to happen there must be at least four variables and six constraints in the linear program being solved. The rule's worst case complexity is exponential which was shown with the construction of the famous Klee-Minty hypercube [2]. Yet, the method also works remarkably fast in practice with [24] attempting to explain this behavior using smoothed analysis. Note that the analysis in [24] is not for Dantzig's rule but the same principles are applicable (but not proven).

3.3 Steepest Edge

Rather than picking the constraints that lead to the greatest absolute decrease in the objective value we can also consider picking the constraints that lead greatest decrease in the objective value with respect to the length of the edge the simplex method traverses. Dantzig's rule measures the rate of change of the objective function with respect to the old vertex x and the new vertex $\bar{x}$. On the other hand, steepest edge measures the rate of change of the objective function with respect to the distance between the old vertex x and the new vertex $\bar{x}$. So the leaving constraint in the case of steepest edge is defined using the set Σ where

$$\Sigma \triangleq \{\frac{\lambda_i}{\|p_i\|} \mid Wp_i = e_i \wedge \lambda_i < 0\},$$

W is the current working set matrix and e_i is the vector of zeros with the i-th component set to i. The leaving constraint s is

$$s = i \quad \text{where} \quad \frac{\lambda_i}{\|p_i\|} = \min \Sigma$$

such that we end up picking the index that leads to the highest normalized reduction in the objective function. This is because if

$$\frac{\lambda_s}{\|p_s\|} \leq \frac{\lambda_j}{\|p_j\|}$$

where $s \neq j$ and $\lambda_s, \lambda_j < 0$, then

$$\frac{c^T p_s}{\|p_s\|} \leq \frac{c^T p_j}{\|p_j\|}.$$

While it may seem that we need to compute the solution to $k = |\Sigma|$ amount of systems, i.e. find p_i for each $Wp_i = e_i$, algorithms given by [5] and [11] present various methods to efficiently compute the $\|p_i\|$'s from one iteration to another. However, since we do not actually implement one of these schemes in appendix C and stick to the naive solution of solving k systems we will not be presenting the details of these methods here. After finding a search direction, the entering constraint t is defined to be the same as in the case of Dantzig's rule.

Steepest edge is a popular pivoting rule as it is not effected by re-scaling any one of the variables in the linear program. [23] shows that steepest edge may also cycle for certain examples and [7] gives a construction for a linear program that causes the steepest edge pivoting rule to exhibit exponential time complexity.

We conclude our discussion of the deterministic pivoting strategies by directing the reader to the implementation of the simplex method given in appendix C. The functions `bland_out`, `dantzig_out` and `steepest_out` compute the leaving constraint index for the respective methods at each iteration. Similarly, the functions `bland_in`, `dantzig_in` and `steepest_in` compute the entering constraint for the respective methods at each iteration. Further discussion on the implementation details is given in section 4. We now turn our attention to randomized strategies.

3.4 Random Edge

The simplest and most intuitive randomized simplex method is to pick the leaving and incoming constraints randomly so as to randomly pick an edge to traverse. The only requirement is that we decrease the value of the objective value so that we are always making progress randomly. While this strategy is not attributed to any specific paper, good descriptions of it can be found in [15] and [19]. We proceed as follows. Given the Lagrange multipliers λ at a single iteration of the simplex method, we define

$$\Lambda = \{\lambda_i \mid \lambda_i < 0\}$$

as the set of the negative components of λ. The leaving constraint s is defined by first picking a random number j uniformly from the integral range $[1, k]$ where $k = |\Lambda|$, i.e. k is the size of Λ. Then s is

$$s = j \quad \text{where} \quad \lambda_j \in \Lambda$$

such that the randomly obtained index s corresponds to a negative component of the multiplier λ. Similarly, the entering constraint is defined from the set of blocking constraints S where

$$S \triangleq \{a_i^T p \mid a_i^T \text{ is blocking at } x \text{ with direction } p\}$$

and l is picked uniformly at random from the range $[1, m]$ where $m = |\Lambda|$. So t becomes

$$t = l \quad \text{where} \quad a_l^T p \in \Lambda$$

meaning we randomly pick one of the blocking constraints. Note that if $S = \emptyset$ then we have the same result as in the deterministic cases as we know that the objective function is unbounded from below. It is easy to see that the rule ends up randomly picking an edge that leads to a lower value of the objective function.

We point out that the description given here is not standard when compared to descriptions given in the literature. Gartner in [15] states that the random edge strategy should consider the set of all edges and then pick one randomly. However, this is inefficient as considering all possible edges would mean that we get a combinatorial explosion in terms of the set of search directions considered. Rather than computing all edges and then randomly picking one we choose to randomly generate one edge which is equivalent in terms of the end goal but more efficient as we only solve one system of linear equations to obtain a search direction. While we realize that this is probably not what Gartner meant when describing the strategy, it is still important to point out the distinction.

In terms of theoretical complexity, only exponential upper bounds are known for the random edge strategy [31] [26]. The reason for this is that proofs on worst-case complexity for random edge are difficult to do. However, there is plenty of work on showing lower bounds which can at times be quadratic for certain linear programs [31]. It is believed that random edge does indeed behave better than deterministic strategies but definitive evidence for this claim does not exist as of now.

On a more practical level, we would like to point out that the literature on random edge almost always assumes non-degeneracy such that these results rely on the fact there will be no cycling. While this is fine for proving worst-case time bounds we discuss what may happen in a practical setting if a cycle exists in a linear program. We attempt to show that the random edge strategy may cycle in the case of degenaracy but will do so with smaller probability each time it takes a cycling path. Let δ be a sequence of working sets such that if we start at δ_1 and move to δ_2 then we cycle with some length n and end up back at δ_1 again. So δ is the sequence

$$\delta = \delta_1 \delta_2 \ldots \delta_n \delta_1 = W_1 W_2 \ldots W_n W_1.$$

It is clear that to get from W_1 to W_2 we must move off of a specific constraint in W_1 and add a specific constraint to get W_2. The same argument applies for the rest of the sequence. For the sake of simplicity assume we start the simplex method at W_1 with the degenerate non-optimal vertex x_1. Let the active set at x_1 be of size m and the size of the working set W_1 be of size d where $d < m$ by the assumption of degeneracy. Then with at most probability $\frac{1}{d}$ we remove the specific constraint that leads to a cycle. By assuming that there is only one entering constraint, we will then move to W_2 with probability 1. We then assume once we enter W_2 we will definitely cycle. Once we get back to W_1 the probability of cycling twice becomes

$$(\frac{1}{d})^2$$

so the probability of cycling n times is

$$(\frac{1}{d})^n$$

which approaches 0 as n approaches ∞. This reasoning generalizes to the case where there may be many constraints following W_2 that may lead to the cycle or escape the cycle. In this case, the probability of entering the cycle continually will get even smaller. Note that we know there always exists a sequence of constraints that allow us to resolve cycling due to the proof of termination of Bland's rule [4]. Thus, we conclude that a random edge simplex strategy may cycle but will do with smaller probability at each cycling iteration, i.e. when the sequence of the cycle ends. This is clearly better than Dantzig's rule and the steepest edge rule as the probability of cycling in those cases are 1 once we enter the cycle. The random edge rule may cycle but the cycling should degenerate over time.

3.5 Random Facet

In contrast to random edge, there are also a group of randomized algorithms referred to as random facet. All such random facet methods have provable sub-exponential worst-case run-times. Kalai in [12] introduces a randomized simplex algorithm that runs in sub-exponential time. Similarly, Matousek, Sharir and Welzl introduce a different algorithm in [18] which is also shown to be sub-exponential. Following these two advancements, Goldwasser in [16] shows the surprising result that these algorithms are the dual of each other despite the fact that they are derived in very different ways. For this project we will present a version of random facet given by [32]. Note that [32] also gives an improved version of random facet but for the sake of simplicity we stick to the basic random facet and only make a slight improvement to it.

A facet is a constraint that is included in the working set constraint matrix W at some vertex x. So the active set constraint matrix W is the set of facets of x. It is a well known fact that at least one solution to a linear programming problem lies on a vertex if an optimal objective function value exists [32]. An edge from vertex x to x' can be defined by removing a facet from W and adding a new constraint $f' \notin W$ such that we get the new working set matrix

$$W' = W \setminus \{f\} \cup \{f'\}$$

where f' becomes a facet of x'. If the problem is unbounded then f' may define a ray but we assume for the sake of simplicity that the problems we are dealing with are bounded. If the objective function value at x' with W' is less than the objective function value at x with W then we take a pivoting step from x to x'. Furthermore, we say that W^* is optimal if and only if no pivot steps can be taken at the point x^* defined by W^*. The pair (F, B) is a pair of sets of constraints where F is the set of all inequality constraints in our problem and B is the set of active constraints when we start the random facet method. So only constraints in $F \cap B$ are candidates for a pivoting step such that

$$|F \cap B| = d'$$

defines a smaller linear program of size d' variables with constraints $B \setminus F$ which is equivalent to the intial linear program [32].

The central idea of the random facet algorithm is as follows. Given a starting working set B we choose randomly a facet $f \in F \cap B_0$ and recursively find the optimal vertex from all the vertices that have f as a constraint. This means we reduce the size of the linear program and the inequality of f is replaced by equality. If the vertex we find is optimal then we exit the program. However, if it is not optimal then it must be that f should not be active so f must leave the working set. In this case f must be exchanged with some new f' such that we get a new working set matrix B'. This reasoning leads to us obtaining the algorithm given in [32]. The presentation below also makes a slight modification.

```
function RandomFacet($F$,$B_0$)
    if $F \cap B_0 = \emptyset$ then return $B_0$
    else
        $f \leftarrow$Random($F \cap B_0 = \emptyset$)
        $B_1 \leftarrow$RandomFacet($F \setminus \{f\}$,$B_0$)
        $B_2 \leftarrow$Pivot($F$,$B_1$,$f$)
        if $B_1$ is optimal then return $B_1$
        else if $B_1 \neq B_2$ then return RandomFacet($F \setminus \{f\}$,$B_2$)
        else return $B_1$
        end if
    end if
end function
```

We see that RandomFacet is a recursive function that takes the set of inequality constraints F and the set of active contraints B_0 at x_0. If $F \cap B_0 = \emptyset$ then B_0 is the optimal solution of that problem. Otherwise, we select a random facet from $f = F \cap B_0$ and recursively call RandomFacet with f removed from F. This will give us the solution to the subproblem defined by the removal which will be assigned to B_1. Next, we try to perform a simplex pivot by removing f from B_1 and getting a new working set matrix B_2. If we cannot remove f then B_1 must be equal to B_2 such that B_1 is an optimal solution to the larger problem. Otherwise, f cannot be active so we remove it from F and solve the problem with the working set matrix B_2 that has a lower objective function value. We know B_2 has a lower objective function value as a pivot must have been successful such that $B_1 \neq B_2$. The function Pivot does a simplex pivot from B_1 by removing f if f corresponds to a negative multiplier. While the leaving constraint is well defined, i.e. it has to be f if f is not optimal, the choice entering constraint is left to the user. We decide to uniformly randomly pick one blocking constraint from the set of blocking constraints. The Random function removes a facet from the intersection uniformly at random.

As an improvement, we add the check for whether B_1 is optimal as it is not specified whether the Pivot function actually checks for the optimality of the whole solution or just the optimality of f. If it does the latter then we would require more recursive calls to terminate but since the pivot already obtains the multipliers λ_1 associated with B_1, we can actually check whether B_1 is globally optimal rather than the local optimality property of f. If B_1 is optimal then we terminate the recursion early leading to some gains in efficiency.

This version of the algorithm is proven to run in sub-exponential time in the worst-case by Kalai in [12] and a simpler proof is available in [32]. We skip the details here but note that the proof involves defining a recurrence relation which is a function over the number of variables and constraints of the linear program. This relation models the expected number of recursive calls the algorithm makes such that we get an upper bound on the number of pivoting steps performed which turns out to be sub-exponential.

Overall, random facet can be considered as a specialization of the generic simplex method [32]. This is because the algorithm proceeds just like the simplex method by considering multiple vertices and finding an optimal one but rather than obtaining a search direction from the Lagrange multipliers at a vertex, it eliminates the constraints that are not active at an optimal point. If a good random choice is made then it seems that the algorithm should terminate fairly quickly.

Finally, just like for the case of random edge, we reason about what happens if a vertex is degenerate. In this case we actually do not get any cycling as random facet eliminates constraints while also attempting to pivot rather than finding a specific search direction from the Lagrange mutlipliers. Thus, the random facet algorithm will discard the constraints at a degenerate vertex rather quickly as long as that vertex is not optimal. If we sample the active constraints at an non-optimal degenerate vertex one after the other then we will throw them away one by one. If the degenerate vertex is optimal then the algorithm will simply return that point and terminate. Therefore, the random facet algorithm does not cycle.

The implementation for random edge can be found in appendix C where the functions `random_out` and `random_in` carry out the random edge constraint deletion and addition, respectively. For random facet we implement a new function given in appendix D which is the instantiation of the algorithm given in section 3.5. More details on the implementation can be found in section 4. We move onto describing an unique subexponential randomized algorithm given by Ken Clarkson in [14] as our final pivoting strategy.

3.6 Clarkson's Algorithm

Clarkson presents 4 algorithms in [14] "for linear programming and integer programming when the dimension is small." That is, if the number of constraints is much greater than the number of variables then Clarkson's algorithm works in sub-exponential time in the worst case. The first 2 of Clarkson's algorithms are for linear programming while the next 2 are for integer programming. We focus only on the first one.

Clarkson begins with a description of linear programming similar to the one given in section 2. The main assumption is that the feasible region is non-empty and there is a unique solution to the linear program. Even if the objective function is unbounded, a unique solution can be defined where one possible value is $-\infty$.

The first algorithm depends on the fact that the optimal value of a linear program is always unique and is only determined by d or fewer constraints of A [14]. That is, there must always be d active constraints at an optimal solution but some of these constrains can be trivially active. All other constraints are either strictly satisfied in the case of non-degeneracy and otherwise there is a mix of active and non-active constraints. Inactive constraints are considered to be redundant in the sense that if they were to be removed from A the optimal value for the linear program would not change. Thus, Clarkson's first algorithm attempts to build a set V^* which is a subset of A that collects only the constraints that are important for solving the problem. It does this by checking which constraints are violated at a point obtained by solving a smaller problem and building the set V of violated constraints. If some constraints are violated then they must be important so we proceed to add them into V^*. If the constraints are not violated then we know they are not important and can discard them. Both the size V^* and V are bounded such that the algorithm can quickly throw away unimportant constraints. If the problem is small enough then the algorithm calls $x_s^*(S)$ which solves a linear program of size $\leq 9d^2$ and this is the base case for the recursion. We finally terminate when V is empty such that no constraints are violated. The pseudocode for the algorithm is given below. Clarkson refers to the set of constraints as S rather than A.

```
function CLARKSON1(S)
    $V^* \leftarrow \emptyset$
    $C_d \leftarrow 9d^2$
    if $n \leq C_d$ then return $x_s^*(S)$
    end if
    repeat
        choose $R \subset S \setminus V^*$ at random
        $r \leftarrow |R| = d\sqrt{n}$
        $x^* \leftarrow$ CLARKSON1$(R \cup V^*)$
        $V \leftarrow \{H \in S \mid x^* \text{ violates } H\}$
        if $|V| \leq 2\sqrt{n}$ then
            $V^* \leftarrow V^* \cup V$
        end if
        until $V = \emptyset$
        return $x^*$
    end function
```

Note how the bounds on both the size of R and V work to build V^* such that V^*contains no more than $O(d\sqrt{n})$ elements [14] so that we throw away most, but not all, of the constraints that are redundant. For his second algorithm, Clarkson uses a similar reasoning but instead of throwing away redundant constraints, everytime a constraint is found to be important, i.e. violated, the probability of sampling that constraint from S is increased. This way the algorithm attempts to pick random constraints from S until it finds the optimal point and as the probability of picking the important constraints increases, the algorithm will converge to picking them rather than the unimportant ones. We skip the implementation details of the second algorithm as it its not pertinent to this project.

Both of Clarkson's linear programming algorithms are proven to exhibit in the worst case sub-exponential time complexity. In the proofs given in [14] the function x^*_S is assumed to be the simplex method. So in the implementation given in appendix E we use the simplex method with Bland's anti-cycling rules. Thus, the theoretical worst-case run-time of the implemented algorithm is also sub-exponential. Our main goal is to test whether this probabilistic approach of constraint elimination is effective in solving linear programs. Since we skip over the details of the proofs given by Clarkson, we refer the reader to [14].

As a final note, Clarkson's proof of the sub-exponential run-time also considers the case of degeneracy. In the case of degenaracy either a small pertubation can be introduced or a rule that relies on lexicographic ordering such as Bland's rules can be used. Since we opt for the latter our algorithm is gauranteed not to cycle while in Clarkson's proof pertubations are chosen to obtain a non-degenerate linear program [1] [14]. Overall, degeneracy is not an issue in Clarkson's algorithms.

We have seen three deterministic algorithms and three randomized algorithms that either specialize the simplex method or complete it by specifying the pivoting strategy. Their strengths and weaknesses have been discussed with the primary strengths of randomized algorithms being that they are less effected by degeneracy and could theoretically perform better than their deterministic counterparts with a good random pivot choice. On the other hand, the deterministic algorithms provide a clear framework to construct proofs and reason about the simplex iterates, and are known to work well in linear programming solvers. The deterministic pivots can be considered classical in the sense that they rely on the mechanism of the simplex algorithm to find an optimal vertex. On the other hand, random facet and Clarkson's algorithm rely on more general facts about a linear program to obtain simplex-like algorithms that achieve the same end goal. Randomized methods are seldom-used in practice so we wish to test whether they live up to their theoretical potential.

4 Implementation Details

We test each pivoting strategy on a select subset of the Netlib dataset [29]. The reason for not testing on the whole dataset is that the programs implemented in appendices A through E are not efficient enough to solve most of the problems in a reasonable amount of time. We observe that we can solve problems that are of size < 1000 constraints and < 1000 variables so we select all such problems from the dataset. The function given in appendix A kicks off the evaluation process and for each test file runs all of the pivoting methods. We store the optimal objective value found, the number of iterations a method took, the CPU time taken and an exit flag. If the exit flag is -1 then the method reached the maximum number of iterations, if the exit flag is 0 then the method found an optimal vertex, if the exit flag is 1 then the problem is infeasible and if the exit flag is 2 then the problem is unbounded. Since the problems we consider are all feasible and bounded we should never observe the exit flags of 1 and 2 [29]. The test function also checks for singular matrix warnings given by Matlab and records whether such a warning was given. If a warning is detected then that run of a pivoting strategy is marked with a 1 and otherwise it is 0. In the case of a warning, the exit code could be 1 and 2 due to numeriacal instability caused by using singular matrices. The reason for why this is done is discussed in section 6. We also enclose everything in a try-catch block so if Matlab fails due to either possible ill-conditioning of a problem or a specific method's inability to handle the problem's size, we can continue processing the next dataset and ignore the problematic dataset. However, we have no way of recovering the problem message for now.

The next testing function is the one in appendix A that performs a 2 phase LP. It begins by reading from a MPS file that defines a problem in the dataset and then starts phase 1. We have found that the increase in constraints and variables caused by setting up a phase 1 LP makes our pivoting methods perform especially slow so we let Matlab's internal `linprog` solver find an initial feasible vertex. We do, however, actually set up the problem before passing it onto `linprog`. Then a phase 2 LP is done which calls a function depending on the `option` parameter where each function employs a specific pivoting strategy. There is also some boilerplate code that records the number of iterations, the exit flags and the CPU time of the calls using Matlab's `tic` and `toc`. The reason for not using `timeit` is because `timeit` runs a function 10 times which ends up slowing down our testing process by a lot. So rather than testing the function multiple times we only take the CPU time of one run as its runtime. This can be somewhat problematic for evaluating the efficiency of randomized algorithms but we leave the discussion of the limitations of the experiment to section 6.

The programs given in appendices C, D and E are implementations of the simplex method, the random facet method and Clarkson's first algorithm, respectively. Explanations for the theory behind each strategy is given in section 3 and the implementations stick to the theoretical presentation. The only extra piece of code that might stick out is the use of boiler plate to record statistics such as iterations and exit flags. For the random facet algorithm the number of iterations is measured as the number of pivot steps made rather than the number of recursive calls. The reason for this is that the stack size of the random facet is always bounded by the size of the working set of a linear program so counting the number of recursive calls can be misleading. Instead, since we know that the pivoting step is always performed to check for optimality we count the number of pivot steps. For Clarkson's algorithm the number of iterations are given by the sum of the number of iterations the algorithm calls everytime it reaches a base case. Finally, for the classical simplex method we say that the number of iterations taken are equivalent to the number of times the algorithm stays in the while loop. If we ever reach the maximum number of iterations then we make sure to end the execution and keep whatever result was last computed.

Although the implementations are considered inefficient for solving larger problems, they do still take advantage of sparsity and use sparse representations of the problem matrices. We rely on Matlab's internals to pick the best algorithm for a given context. Furthermore, a utiliy function is given in appendix F that estimates the rank of a matrix using the QR decomposition and is used throughout the other programs. It can be called with a specific tolerance level or it will use the default tolerance of `1e-10`. Additionally, our simplex and simplex-like methods use the default tolerance and maximum iteration values Matlab's `linprog` solver uses [35]. The default constraint tolerance for the dual-simplex implementation in `linprog` is `1e-7` and the default maximum number of iterations is given by

```
10*(numberOfEqualities + numberOfInequalities + numberOfVariables).
```

In our case we use the default tolerance and simplify the maximum number of iterations to

```
10*(numberOfInequalities + numberOfVariables)
```

as we have no equalitiy constraints. All implementations and files can be found in the following GitHub repository: `https://github.com/goki0607/randomized-pivoting-project`.

Finally, we run all test's on NYU Courant's Snappy 1 server. This system is equipped with a Two Intel Xeon E5-2680 (2.80 GHz) (20 cores) CPUs, has 128 GB of memory and runs CentOs 7. The author's MacBook Air proved to be inadequate for testing so we use a machine whose purpose is for numerical computation.

5 Experiments and Analysis

5.1 Result of Experiments

We give the results of running each pivoting strategy on each dataset considered. The tables for when Matlab throws an exception are omitted. However, if Matlab gives a singular matrix warning for a specific method this is recorded in the `Failure` column of the table. The other columns are as explained in section 4 and we also note that the CPU Time measurement is given in seconds. Due to lack of time and the methods taking too long to complete, only a subset of the planned datasets have been tested. However, we have gathered plenty of data to make certain observational judgements and in total we test 31 datasets. Table 1 also provides a summary of certain results. For each method we record the number of times it lead to a singular matrix, the number of times an optimal solution was found, the number of times it reached its max iteration threshold and an optimal point was not found, the number of times it recorded the lowest objective value compared to the other methods and the average CPU running time (again measured in seconds). If a method fails for a dataset then we do not take into account any of the other measures for the summary table. However, if a method reaches the maximum number of iterations then we still consider all the measures as such runs are not failures. All tables except the summary table are given in the appendix section G and the summary table is given below.

5.2 Summary Table

Pivot Choice	Average CPU Time	Occurrences of Optimal Solution	Occurrences of Max Iterations	Occurrences of Smallest Objective Value	Ocurrences of Failure
Bland	8185.170475	7	**23**	6	1
Danztig	6238.237211	**13**	17	9	1
Steepest Edge	11527.942982	10	<u>20</u>	<u>10</u>	1
Random Edge	<u>982.143201</u>	<u>12</u>	10	**18**	**8**
Random Facet	**401.503758**	9	19	8	<u>2</u>
Clarkson	9060.511167	7	**23**	6	1

Table 1: Summary of all test runs.

In the table above we indicate the smallest Average CPU Time in bold and underline the second smallest Average CPU Time. For all other columns we bold the highest value and underline the second highest value.

5.3 Analysis of Results

From the summary table we see that on average, when the method does not cause a singular Matrix warning, the random facet method followed by the random edge strategy performs the best. It seems that random facet performs much more poorly compared to the other algorithms when the problem size is small. However, as the problem gets larger the CPU time of random facet is smaller compared to the others and for some cases such as in table 5 and table 10 the difference is remarkable. A possible reason why this is happening could be that the random facet algorithm is randomly choosing optimal facets more often than those that are non-optimal which allows it to eliminate a lot of non-optimal facets quickly. Another reason why random facet could be better is that the algorithm builds an optimal working set bottom-up, i.e. from a smaller working set we get larger working sets, and only performs a pivot step fully if a facet is not optimal. In contrast, the simplex method with deterministic pivoting will complete a full k iterations of pivoting until the optimal point is found but the random facet will only complete one full iteration if a facet is non-optimal. The second fastest strategy is random edge and this is also surprising. Here we know that the random edge algorithm is less computationally intensive compared to Dantzig's rule or the steepest edge rule and slightly more expensive compared to Bland's rules. This is because we only need to sample from an uniform distribution and there already exists efficient algorithms to do this. Beyond this reasoning there isn't much indication as to why random edge is performing so well. We theorize that it may have to do with the fact that random edge at any given point will behave like any of the three deterministic pivoting rules meaning that at any given point we are taking an optimal path with some probability. However, quantifying this probability is hard. We also see that Clarkson's algorithm performs the second worst which makes sense as most of the problems in the Netlib dataset [29] does not satisfy the small dimensionality requirement. Recall that the small dimensionality requirement is the assumption that the dimensions of the problem, i.e. the number of variables, is greatly smaller than the number of constraints. From the deterministic pivoting rules the steepest edge rule performs the worst. This can be attributed to the fact that the implementation of steepest edge in appendix C is naive. A more efficient algorithm as given in [5] and [11] should certainly decrease the run-time. Finally, Dantzig's rule performs better than Bland's rule and this is not surprising as this has been observed in practice many times. Overall, two of the presented randomized algorithms definitely show promise in terms of CPU run-time performance.

We now turn our attention to the other columns of the summary table 1. We see that in terms of failing due to obtaining a singular working set the random edge method followed by the random facet method perform the poorest. In fact, the random edge method performs much worse in this scenario compared to the other methods. This high rate of failure can be primarily attributed to numerical instability caused by the inefficient implementations of the simplex method. However, this is only half of the story as we see that the deterministic strategies only fail once for some dataset. It seems that the random edge rule is causing numerical issues to become more pronounced and this phenemenon could be attributed to the randomness of the pivot rule. That is, if we obtain a set of possible constraints the deterministic methods will either choose the one that is most negative or has the least index which makes it much less likely to pick a "faux" index. On the other hand, if a Lagrange multiplier is a value such as `-1e-5` then the random edge rule will pick this one with the same probability that it will pick a Lagrange multiplier that is a value such as `-1e10`. Clearly `-1e10` is the better choice in terms of both decreasing the value of the objective function and numerical stability as the number `-1e10` definitely corresponds to a non-optimal constraint (unless there is some catastrophic issue relating to floating-point arithmetic or ill-conditioning). On the other hand, the value `-1e-5` is ambigious as we can't with certainty tell that this is a multiplier of a non-optimal constraint. Thus, the random nature of random edge also makes it more unpredictable in terms of numerical stability. The same can be said for random facet but here the issue is not as dangerous as we do not always do a full pivot step. Clarkson's algorithm is stable and this is not surprising as it solves linear programs using Bland's rules which is also stable for these tests. Furthermore, the deterministic algorithms are also stable and the reasons for this have been discussed in the context of random edge's stability. Now we see that there might be a trade-off between stability and speed with randomized strategies.

The three columns that measure the amount of times each strategy finds an optimal point, the number of times each strategy times out and the number of times each strategy finds the smallest objective value compared against the other methods attempt to compare each method with the others. Note that there are slightest disrepancies in the table due to floating point error. That is, two methods that satisfy the optimality conditions may report two different optimal objective function values due to accrued round-off error. We of course know that in theory the optimal objective function value is unique but in practice this may not be the case. In terms of finding an optimal solution both random edge and random facet work well and they are close to the deterministic stratgies Dantzig's rule and steepest edge rule. This observation shows us that the randomized rules can sometimes be as effective as their deterministic counterparts. In terms of timing out we see that both Bland's rules and Clarkson's method times out the most. It is known that Bland's rules in practice are slow and since we use the simplex method with Bland's rules as a subroutine for Clarkson's method we observe the same poor performance. It seems that Clarkson's method's performance not only depends on dimensionality of the problem but also the simplex implementation picked for the base case. Finally, the smallest objective value column attempts measures how much progress has been made by each method regardless of whether it has timed out or found an optimal objective function value. We see that random edge is the best in this case by a large margin but this difference can also be attributed to possible numerical instability. Since random edge appears to be unstable, it could also be accruing a lot of round-off errors making this measure an anomaly. The other methods behave the same with steepest edge being the second best in terms of making progress and is followed by Dantzig's rule. This is probably due to the fact that both algorithms actively try to reduce the objective value as much as possible from one iteration to another. We see that the random edge and random facet methods behave similarly to their deterministic counterparts while Clarkson's method's behavior closely follows the underlying solver being used for the base case.

The experimental results show the promise of random edge and random facet for a general class of linear programming problems and backup their theoretical claims in terms of runtime. They can be potentially much faster than deterministic methods but seem to be also more susceptible to numerical instability. On the other hand, Clarkson's method has been dissapointing in the sense that it does not perform as well as the other two randomized algorithms. This suggests that a more specialized testing dataset with the "right" dimensions might be necessary to discover any potential benefits of the method. The deterministic algorithms mostly match our expectations of them and our results demonstrate why Dantzig's rule is a popular choice in the simplex method. The steepest edge rule could definitely be made with a more efficient implementation which would improve its metrics in table 1. Our results definitely call for more experimental evaluation where the experimental method is improved by addressing the shortcomings explain in the next section (6).

6 Evaluation

6.1 Flaws in Implementation and Analysis

The biggest flaw in our experiments is that our implementations are not efficient. Most of the problems considered for this project can be solved very quickly using Matlab's `linprog` function. `linprog` uses the primal-dual simplex method whereas we use the primal simplex method. The result of the randomized simplex algorithms being faster needs to be verified with much more efficient implementations of all the pivot strategies so that we can rule out implementation flaws as the cause of the performance differences. The second important flaw is that our implementations are not necessarily numerically stable. We rely on Matlab to solve linear equations and use tolerances at certain points but do not take advantage of the linear algebra methods modern solvers use [22]. A numerical method can only be as good as its implementation regardless of any promising theoretical properties it may have. Thus, we should also test our randomized methods with more stable methods so that we can determine whether randomized methods can always lead to instability or whether the instability is tied to the instability of the implementation. The issue of efficiency and stability are the main points that should be addressed in a future study. Additionally, improving efficiency will allow us to use Matlab's `timeit` function which would be a better way of measuring performance as we would average the run times of the algorithms over 10 attempts. Furthermore, improving efficiency will allow us to tackle larger problems and possibly also include a phase 1 linear program to our tests. This would give us a more hollistic overview of each strategy.

Another flaw in the analysis is that the problems being considered are not necessarily suitable to be used with Clarkson's algorithm. We end up observing interesting behavior for the random edge and random facet algorithms but nothing significant happens with Clarkson's algorithm. In fact, Clarkson's algorithm behaves very similarly to the simplex method with Bland's rules but a little slower due to more overhead caused by the other parts of the algorithm. Either linear programs that have low dimensionality and a large number of constraints need to be generated or such problems ought to be found. One idea is to consider the problem given in the Netlib dataset and take their duals which in effect will flip the dimensionality. Since the problems in the Netlib set are bounded, by duality theory, the dual problems will also be bounded and have an unique minimal objective function value. Identifying such potential problems that satisfy the dimensionality requirements and running experiments on them could provide better insight into Clarkson's algorithm. Furthermore, Clarkson's other algorithm could also be considered along with Hansen's improved random facet [32] and other algorithms presented in Goldwasser's survey of randomzied algorithms [16]. Additionally, [17] presents an interesting algorithm that combines Clarkson's algorithm with Kalai's. In fact, expanding the evaluation to other deterministic pivoting rules such as Zadeh's least-entered rule [27], Cunningham's least recently considered rule [6] and Harris' devex rule [3] could also be interesting. An exponential run-time lower bound was proven for Cunningham's rule recently in [34] while [28] has proven a sub-exponential lower for Zadeh's rule. Both algorithms were proposed to defeat the problems caused by Klee-Minty cubes and comparing their performance against the randomized algorithms could be interesting.

Overall, the flaws in our implementation and experimental evaluation define new directions for research. A more industrial implementation of the simplex method with randomization is the natural next step for this work.

6.2 Numerical Instability

We expand slightly on the issue of the numerical instability caused by the randomized methods. The issue of failure occurred more often during the initial phases of testing due to a lower tolerance level being used. Rather than relaxing this tolerance we proceeded with a different strategy. Before removing a constraint from the working set and adding a new blocking constraint in we checked whether the resulting matrix would be singular. If singularity/near-singularity was observed then the pivot was rejected and the simplex method moved to the next iteration without making progress. The idea here is that if there are multiple leaving constraints and multiple entering constraints then we should be able to repeat a randomized algorithm until a numerically non-singular new working set is found. This however works very poorly in practice. In fact for a small subset of the problems tested for this project we tried this strategy. The efficiency of both random facet and random edge were esentially lost as both methods started cycling. While we claim that that both methods cannot cycle under degeneracy a different kind of cycling was observed. With some probability both methods would perform a sequence of working set updates until reaching a point where any further exchanges would cause a singular matrix to be obtained so that the method would reject all updates and cycle until timing out. What is even worse is that the method would sometimes cycle at a point with an objective function value very near to the optimal value. This suggests that such a check cannot be useful most of the time as we cannot predict this behavior. Our observation is akin to the fact that it is hard to predict cycling under degeneracy for deterministic pivoting strategies. Thus, we proceeded to relax the tolerance levels which seems to have alleviated the issue somewhat. But, as can be seen in table 1, the issue has not been completely resolved and alternative strategies should be explored.

7 Conclusion

Before concluding we would like to have a brief discussion regarding complexity theory and the simplex method. Over the years the discussion on whether there exists a pivoting strategy such that the simplex method becomes a strongly polynomial time algorithm has been significant. Before the Klee-Minty cube [2] the worst case complexity of the simplex method was not known. Following the demonstration that the simplex method with Dantzig's rule runs at worst in exponential time considerable research has been done into alternative pivoting strategies. As outlined in the introduction (section 1) and section 3 each new proposed pivoting strategy has been proven to either have an exponential worst-case run-time, a sub-exponential lower bound or an unknown time bound which means that they cannot be used to obtain a strongly polynomial time algorithm. Even the randomized schemes that have been harder to reason about have been proven to have exponential [13] or sub-exponential [16] running times. The search continues to this day and a proof for whether there exists a pivoting rule such that simplex algorithm is definitely a strongly polynomial algorithm is an open question. One interesting development is [30] where the authors show for some of the popular simplex pivoting rules, if a pivoting rule is PSPACE-complete then that pivoting rule cannot be used to obtain a strongly polynomial simplex method unless P = PSPACE. Of course, the question of $P \overset{?}{=}$ PSPACE is also an important open problem. This work seems like a right step in defining an intractability proof for a general class of simplex pivoting rules [30].

On the other hand, we know from the smoothed analysis introduced in [24] that the simplex method runs in polynomial time for most real-world problems. What is interesting is that it does not seem like most industrial solvers or algorithms included numerical computing software utilize randomization for pivoting. The most popular choice appears to be the primal-dual simplex algorithm with some deterministic pivoting choice that is known to work well due to years of testing and knowledge. Our experiments show that randomized strategies could be promising in obtaining significant speed-ups in solving a linear program but there are numerical instability issues that need to be addressed. Furthermore, more specialized algorithms such as Clarkson's first algorithm could prove to be effective for problems with a specific dimensionality (i.e. where the number of contraints are much greater than the number of variables). We propose that more testing needs to be done where the tests address the issues raised in section 6 and the scope of the comparisons should be widened to primal-dual methods. Yet, it is clear from this project that randomized algorithms can be potentially more effective alternatives to the current deterministic strategies used to solve linear programs and further experimental work should be done.

8 References

[1] Abraham Charnes. “Optimality and degeneracy in linear programming”. In: *Econometrica (pre-1986)* 20.2 (1952), p. 160.

[2] Victor Klee and GI Minty. *How good is the simplex algorithm?, Inequalities. III (Proc. Third Sympos., Univ. California, Los Angeles, Calif., 1969)*. 1972.

[3] Paula MJ Harris. “Pivot selection methods of the Devex LP code”. In: *Mathematical programming* 5.1 (1973), pp. 1–28.

[4] Robert G Bland. “New finite pivoting rules for the simplex method”. In: *Mathematics of operations Research* 2.2 (1977), pp. 103–107.

[5] Donald Goldfarb and John Ker Reid. “A practicable steepest-edge simplex algorithm”. In: *Mathematical Programming* 12.1 (1977), pp. 361–371.

[6] William H Cunningham. “Theoretical properties of the network simplex method”. In: *Mathematics of Operations Research* 4.2 (1979), pp. 196–208.

[7] Donald Goldfarb and William Y Sit. “Worst case behavior of the steepest edge simplex method”. In: *Discrete Applied Mathematics* 1.4 (1979), pp. 277–285.

[8] H Papadimitriou Christos and Kenneth Steiglitz. “Combinatorial optimization: algorithms and complexity”. In: *Prentice Hall Inc.* (1982).

[9] Vasek Chvatal, Vaclav Chvatal, et al. *Linear programming*. Macmillan, 1983.

[10] Katta G Murty. *Linear programming*. Springer, 1983.

[11] John J Forrest and Donald Goldfarb. “Steepest-edge simplex algorithms for linear programming”. In: *Mathematical programming* 57.1-3 (1992), pp. 341–374.

[12] Gil Kalai. “A subexponential randomized simplex algorithm”. In: *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. ACM. 1992, pp. 475–482.

[13] Andrei Z Broder et al. “The worst-case running time of the random simplex algorithm is exponential in the height”. In: *Information Processing Letters* 56.2 (1995), pp. 79–81.

[14] Kenneth L Clarkson. “Las Vegas algorithms for linear and integer programming when the dimension is small”. In: *Journal of the ACM (JACM)* 42.2 (1995), pp. 488–499.

[15] Bernd Gärtner. “Randomized Optimization by Simplex Type Methods”. PhD thesis. Freie Universität Berlin, 1995.

[16] Michael Goldwasser. “A survey of linear programming in randomized subexponential time”. In: *ACM SIGACT News* 26.2 (1995), pp. 96–104.

[17] Bernd Gärtner and Emo Welzl. “Linear programming—randomization and abstract frameworks”. In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 1996, pp. 667–687.

[18] Jiřı Matoušek, Micha Sharir, and Emo Welzl. "A subexponential bound for linear programming". In: *Algorithmica* 16.4-5 (1996), pp. 498–516.

[19] Bernd Gärtner, Martin Henk, and Günter M Ziegler. "Randomized simplex algorithms on Klee-Minty cubes". In: *Combinatorica* 18.3 (1998), pp. 349–372.

[20] Barry A Cipra. "The best of the 20th century: Editors name top 10 algorithms". In: *SIAM news* 33.4 (2000), pp. 1–2.

[21] Gerard Sierksma. *Linear and integer programming: theory and practice*. CRC Press, 2001.

[22] Robert E Bixby. "Solving real-world linear programs: A decade and more of progress". In: *Operations research* 50.1 (2002), pp. 3–15.

[23] Julian AJ Hall and Ken IM McKinnon. "The simplest examples where the simplex method cycles and conditions where EXPAND fails to prevent cycling". In: *Mathematical Programming* 100.1 (2004), pp. 133–150.

[24] Daniel A Spielman and Shang-Hua Teng. "Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time". In: *Journal of the ACM (JACM)* 51.3 (2004), pp. 385–463.

[25] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[26] Bernd Gärtner and Volker Kaibel. "Two new bounds for the random-edge simplex-algorithm". In: *SIAM journal on discrete mathematics* 21.1 (2007), pp. 178–190.

[27] Norman Zadeh. "What is the worst case behavior of the simplex algorithm". In: *Polyhedral computation* 48 (2009), pp. 131–143.

[28] Oliver Friedmann. "A subexponential lower bound for Zadeh's pivoting rule for solving linear programs and games". In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer. 2011, pp. 192–206.

[29] David M. Gay. *netlib/lp*. Aug. 2013. URL: `https://www.netlib.org/lp/`.

[30] Ilan Adler, Christos Papadimitriou, and Aviad Rubinstein. "On simplex pivoting rules and complexity theory". In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer. 2014, pp. 13–24.

[31] Thomas Dueholm Hansen, Mike Paterson, and Uri Zwick. "Improved upper bounds for random-edge and random-jump on abstract cubes". In: *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2014, pp. 874–881.

[32] Thomas Dueholm Hansen and Uri Zwick. "An improved version of the Random-Facet pivoting rule for the simplex algorithm". In: *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. ACM. 2015, pp. 209–218.

[33] George Dantzig. *Linear programming and extensions*. Princeton university press, 2016.

[34] David Avis and Oliver Friedmann. “An exponential lower bound for Cunningham’s rule”. In: *Mathematical Programming* 161.1-2 (2017), pp. 271–305.

[35] MathWorks. URL: `https://www.mathworks.com/help/optim/ug/linprog.html`.

Appendices

A Main Function for Testing

```
function test()
    files = dir('../tests/*.mps');
    in = files(1).folder;
    rows = {'danzig'; 'bland'; 'steepest'; 'randedge'; ...
        'randfacet'; 'clarkson'};
    cols = {'t'; 'f'; 'its'; 'flag'; 'fails'};
    csv = '.csv';
    out = '../results/';
    success = {};
    failure = {};
    for i=1:length(files)
        name = files(i).name;
        file = strcat(in,strcat('/',name));
        p = mpsread(file);
        arr = zeros(length(rows),length(cols));
        flag = 0;
        j = 1;
        while j <= length(rows)
            disp(j)
            try
                [~,t,f,its,flag] = evallp(p,j-1);
                arr(j,1:4) = [t f its flag];
                [warnMsg, ~] = lastwarn;
                if ~isempty(warnMsg)
                    arr(5) = 1;
                end
                lastwarn('');
                j = j + 1;
            catch
                flag = 1;
                break
            end
        end
        if flag == 1
            failure{end+1} = name; %#ok<AGROW>
            continue
        end
        success{end+1} = name; %#ok<AGROW>
        table = array2table(arr,'VariableNames',cols,'RowNames',rows);
        name = name(1:end-4);
        outfile = char(fullfile(out,strcat(name,csv)));
        disp(outfile)
        writetable(table, outfile, ...
            'Delimiter',',','QuoteStrings',true)
    end
    table = cell2table(success);
    outfile = char(fullfile(out,strcat('success',csv)));
    writetable(table, outfile, ...
         'Delimiter',',','QuoteStrings',true)
    table = cell2table(failure);
    outfile = char(fullfile(out,strcat('failure',csv)));
    writetable(table, outfile, ...
         'Delimiter',',','QuoteStrings',true)
end
```

B Function to Solve a Two-Phase LP

```
function [x,t,f,its,flag] = evallp(p,option)
% Two phase linear programming solver. P is a mps object which describes
% the problem we wish to solve while option triggers what kind of solution
% strategy we would like to proceed with.
% Inputs:       p -- mps object that describes the problem,
%          option -- which solving approach we would like, between 0 and 3
%                    is the pivoting strategies described in simplex.m,
%                    4 is random facet and 5 is clarkson's algorithm.
% Outputs:    x -- an (possibly) optimal solution to the given problem,
%             t -- time statistics using timeit,
%             f -- the optimal solution value or -Inf/Inf if unbounded/
%                  infeasible,
%           its -- number of iterations performed by the strategy taken.
%          flag -- exit flag: 2 if unbounded, 1 if infeasible, 0 if
%                  solution found and -1 if maximum number of iterations.
    f = p.f;
    Aineq = -p.Aineq;
    bineq = -p.bineq;
    Aeq = [-p.Aeq; p.Aeq];
    beq = [-p.beq; p.beq];
    d = length(f);
    idx1 = find(p.lb > -Inf);
    idx2 = find(p.ub < Inf);
    I = eye(d,d);
    Alub = [I(idx1,:); -I(idx2,:)];
    blub = [p.lb(idx1,:); -p.ub(idx2,:)];
    A = [Aineq; Aeq; Alub];
    b = [bineq; beq; blub];
    c = f;
    p1 = @() phase11(A,b);
    tic
    [x,w,flag,i] = p1();
    disp('phase 1 finished')
    t = toc;
    if flag == 2
        x = (1:d)' + -Inf;
        f = -Inf;
        its = i;
        return
    elseif flag == 1
        x = (1:d)' + Inf;
        f = Inf;
        its = i;
        return
    else
        i1 = i;
    end
    if option >= 0 && option <= 3
        g = @() simplex(w,A,b,x,c,option);
    elseif option == 4
        g = @() simplex_random_facet(w,A,b,x,c);
    elseif option == 5
        g = @() clarkson(A,b,x,c);
    end
    if option == 5
        tic
        [x,f,its,flag] = g();
        t1 = toc;
    else
        tic
        [x,f,~,its,flag] = g();
        t1 = toc;
    end
    disp('phase 2 finished')
    t = t1;
    its = its + i1;
end

function [x,w,flag,its] = phase11(A,b)
% Phase 1 LP. Attempts to find a feasible point for the problem defined by
% A (ineqaulity constraints) and b (the right side of the inequality) by
% minimizing the sum of the constraints that are violated. If the problem
% is infeasible the program will have exit flag 1, if the maximum number of
% iterations is reached by the solver the program will have exit flag -1
% and otherwise if a solution is found the program exits with 0. The
% program also returns the working set w to continue onto a possible phase
% 2, the possibly feasible points xnew and the iterations performed by the
% solving strategy. The solving strategy is defined by the option variable
% and described in evallp.m.
    [n,d] = size(A);
    [W0,idx] = licols(A');
    x0 = W0' \ b(idx,:);
    S = A*x0-b;
    V = find(S < 0);
    nv = length(V);
    P = zeros(n,nv);
    P(sub2ind(size(P),V,(1:nv)')) = 1;
    Ap = [A P; zeros(nv,d) eye(nv,nv)];
    bp = [b; zeros(nv,1)];
    cp = sparse([zeros(d,1); ones(nv,1)]);
    [xf,f,e,o] = linprog(cp,-Ap,-bp);
    its = o.iterations;
    if e == 0
        flag = -1;
    elseif f ~=0
       flag = 1;
     else
       flag = 0;
     end
    x = xf(1:d);
    wc = A*x==b;
    [B,~] = licols(A(wc,:)');
    [~,w] = intersect(A,B','rows');
end
```

C The Simplex Method with Various Pivoting Rules

```
function [x,f,w,its,flag] = simplex(w, A, b, x, c, pivot)
% Simplex method function that performs the simplex method using a pivoting
% code which is an integer between 0 and 3. 0 is the textbook rule, 1 is
% Bland's rule, 2 is steepest edge and 3 is random edge. The maximum number
% of iteratons is determined by the Matlab linprog rule 10*(# of equality
% constraints + # of inequality constraints + # of variables) which in our
% case becomes 10*(#ineq+#vars).
% working set is increased.
% Inputs:      w -- working set at x, must be vector of indices,
%              A -- set of constraints, must be a matrix of constraints,
%              b -- set of inequalities corresponding to set of
%                   constraints, assumed to be in all-inequality form
%                   (... >= b_i),
%              x -- the initial starting point, must be a vector and a
%                   vertex of the problem,
%              c -- coefficients on the objective function, must be a
%                   vector, i.e. min c^Tx,
%          pivot -- must be in the range [0,3] as explained.
% Outputs:    x -- an optimal solution to the given problem,
%             f -- the optimal solution value or -Inf if unbounded,
%             w -- the working set upon exit of the program,
%           its -- number of iterations performed,
%          flag -- exit flag: 1 if unbounded, 0 if solution found and -1 if
%                  maximum number of iterations reached and solution is not
%                  optimal.
        format compact
        format long e
        W = A(w,:);
        lambda = W' \ c;
        k = 1;
    max_its = 10*(size(A,1)+size(A,2));
        while any(lambda<-1e-7) && k <= max_its  %#ok<ALIGN>
        if pivot==0
            s = bland_out(w, lambda);
        elseif pivot==1
            s = dantzig_out(lambda);
        elseif pivot==2
            s = steepest_out(lambda, W);
        elseif pivot==3
            s = random_out(lambda);
        else
            error('please give a correct pivot code')
        end
        es = zeros(length(lambda),1);
        es(s) = 1;
        p = W \ es;
        [Ap,idx] = setdiff(A,W,'rows');
        D = idx(Ap*p<-1e-7);
        if isempty(D)
            flag = 1;
            f = -Inf;
            its = k;
            return
        end
        gamma = (A(D,:)*x-b(D)) ./ (-A(D,:)*p);
        alpha_f = min(gamma);
        alpha = alpha_f;
        x = x + alpha*p;
        S = D(gamma==alpha_f,:);
        if pivot==0
            t = bland_in(S);
            if t == -1
                flag = 1;
                f = -Inf;
                its = k;
                return
            end
        elseif pivot==1
            t = dantzig_in(A, S, p);
        elseif pivot==2
            t = steepest_in(A, S, p);
        elseif pivot==3
            t = random_in(S);
            if t == -1
                flag = 1;
                f = -Inf;
                its = k;
                return
            end
        else
            error('please give a correct pivot code')
        end
        w(s) = t;
        W = A(w,:);
        lambda = W' \ c;
        k = k + 1;
    end
    if k > max_its
        if any(lambda<-1e-7)
            flag = -1;
        else
            flag = 0;
        end
    else
        flag = 0;
    end
    f = c' * x;
    its = k;
    w = sort(w);
end

function s = bland_out(w, lambda)
% Bland's rule for outgoing constraint.
    choices = lambda<0;
    elem = min(w(choices));
    s = find(w==elem);
end

function t = bland_in(S)
% Bland's rule for incoming constraint.
    if isempty(S)
        t = -1;
        return
    end
    t = min(S);
end

function s = dantzig_out(lambda)
% Danzig's rule for outgoing constraint.
    [~, s] = min(lambda);
end

function t = dantzig_in(A, S, p)
% Danzig's rule for incoming constraint.
```

```
    [~, idx2] = min(A(S,:)*p);
    t = S(idx2);
end

function s = steepest_out(lambda, W)
% Steepest edge rule for outgoing constraint.
    idx = find(lambda<0);
    lambdas = zeros(length(idx),1);
    for i=1:length(idx)
        es = zeros(length(lambda),1);
        es(idx(i)) = 1;
        p = W \ es;
        lambdas(i) = lambda(i)/norm(p);
    end
    [~,idx2] = min(lambdas);
    s = idx(idx2);
end

function t = steepest_in(A, S, p)
% Steepest edge rule for incoming constraint (nothing fancy).
    [~, idx2] = min(A(S,:)*p);
    t = S(idx2);
end

function s = random_out(lambda)
% Random edge rule for outgoing constraint.
    choices = find(lambda<0);
    idx = randi([1 size(choices,1)], 1);
    s = choices(idx);
end

function t = random_in(S)
% Random edge rule for incoming constraint.
    if isempty(S)
        t = -1;
        return
    end
    idx = randi([1 size(S,1)], 1);
    t = S(idx);
end
```

D The Random Facet Method

```
function [x,f,w,its,flag] = simplex_random_facet(wp, Ap, bp, xp, cp)
% Simplex random facet function that performs the simplex method using
% the random facet pivoting rule. The maximum number of iterations is
% defined to be the maximum number of pivot moves made after the first
% recursive call and is given by 10*(#ineq+#vars).
% working set is increased.
% Inputs:      wp -- working set at x, must be vector of indices,
%              Ap -- set of constraints, must be a matrix of constraints,
%              bp -- set of inequalities corresponding to set of
%                    constraints, assumed to be in all-inequality form
%                    (... >= b_i),
%              xp -- the initial starting point, must be a vector and a
%                    vertex of the problem,
%              cp -- coefficients on the objective function, must be a
%                    vector, i.e. min c^Tx,
% Outputs:     x -- an optimal solution to the given problem,
%              f -- the optimal solution value or -Inf if unbounded,
%              w -- the working set upon exit of the program,
%            its -- number of iterations performed,
%           flag -- exit flag: 1 if unbounded, 0 if solution found and -1 if
%                   maximum number of iterations reached and solution is not
%                   optimal.
    global wg Ag bg cg iters max_its;
    wg = wp;
    Ag = Ap;
    bg = bp;
    cg = cp;
    iters = 0;
    max_its = 10*(size(Ap,1)+size(Ap,2));
    nums = (1:length(Ap))';
    [w,x,flag] = random_facet(nums,wp,xp);
    its = iters;
    if flag == 1
        f = Inf;
    else
        f = cp'*x;
    end
    w = sort(w);
end

```

```
function [B,x,flag] = random_facet(F,B0,x0)
% The random facet algorithm.
    global iters max_its ;
    if iters > max_its
        B = B0;
        x = x0;
        flag = -1;
    elseif isempty(intersect(F,B0))
        B = B0;
        x = x0;
        flag = 0;
    else
        f = random_f(intersect(F,B0));
        [B1,x1,flag1] = random_facet(setdiff(F,f),B0,x0);
        [B2,x2,flag2,optimal] = pivot(F,B1,f,x1);
        if flag1 == 1 || flag2 == 1
            flag = 1;
            B = B0;
            x = x0;
        elseif flag1 == -1 || flag2 == -1
            flag = -1;
            B = B0;
            x = x0;
        elseif optimal == 1
            B = B1;
            flag = 0;
            x = x1;
        elseif ~isequal(B1,B2)
            [B,x,flag] = random_facet(setdiff(F,f),B2,x2);
        else
            B = B1;
            x = x1;
            flag = 0;
        end
    end
end

function f = random_f(F)
% Auxilarry function to obtain a random facet from a set of facets F.
    idx = randi([1 length(F)], 1);
    f = F(idx);
end
```

```

function [B,x,flag,optimal] = pivot(~,B1,f,x0)
% Function to pivot away from constraint f. We randomly pivot to another
% vertex.
    global Ag bg cg iters;
    optimal = 0;
    iters = iters + 1;
    W = Ag(B1,:);
    lambda = W' \ cg;
    s = find(B1==f);
    if (lambda(s) >= -1e-7)
        B = B1;
        x = x0;
        flag = 0;
        optimal = 1;
        return
    end
    es = zeros(length(lambda),1);
    es(s) = 1;
    p = W \ es;
    [Ap,idx] = setdiff(Ag,W,'rows');
    D = idx(Ap*p<-1e-7);
    if isempty(D)
        B = B1;
        x = x0;
        flag = 1;
        return
    end
    gamma = (Ag(D,:)*x0-bg(D)) ./ (-Ag(D,:)*p);
    alpha_f = min(gamma);
    x = x0 + alpha_f*p;
    S = D(gamma==alpha_f,:);
    if isempty(S)
        B = B1;
        x = x0;
        flag = 1;
        return
    end
    t = random_in(S);
    if t == -1
        B = B1;
        x = x0;
```

```
            flag = 1;
            return
        end
        B = B1;
        B(s) = t;
        flag = 0;
end

function t = random_in(S)
% Random edge rule for incoming constraint.
    if isempty(S)
        t = -1;
        return
    end
    idx = randi([1 size(S,1)], 1);
    t = S(idx);
end
```

E Clarkson's First Algorithm

```
function [x,f,its,flag] = clarkson(Ap, bp, xp, cp)
    global Ag xg bg cg iter max_its;
    Ag = Ap;
    xg = xp;
    bg = bp;
    cg = cp;
    iter = 0;
    max_its = 10*(size(Ap,1)+size(Ap,2));
    S = (1:length(Ap))';
    [x,f,~,flag] = xr(S);
    its = iter;
end

function [x,f,its,flag] = xr(S)
    global Ag bg iter max_its;
    n = length(S);
    d = size(Ag,2);
    Cd = 9*d*d;
    if iter > max_its
        flag = -1;
        return
    elseif n <= Cd
```

```
        [x,f,its,flag] = solve(S);
        iter = iter + its;
    else
        Vstar = [];
        while 1
            r = floor(d*sqrt(n));
            R = randsample(setdiff(S,Vstar),r);
            [x,f,its,flag] = xr(union(R,Vstar));
            disp(iter)
            disp(f)
            if flag ~= 0
                return
            end
            V = S(Ag(S,:)*xstar-bg(S,:)<0);
            if length(V) <= 2*sqrt(n)
                Vstar = union(Vstar,V);
            end
            if isempty(V)
                break;
            end
        end
        flag = 0;
    end
end

function [x,f,its,flag] = solve(S)
    global Ag xg bg cg;
    [x,f,its,flag] = evallp(Ag(S,:),bg(S,:),cg,xg,0);
end

function [x,f,its,flag] = evallp(A,b,c,~,option)
% Two phase linear programming solver. This time we wish to solve the
% problem given an A b and c rather than using a mps file. Pption triggers
% what kind of solution strategy we would like to proceed with.
% Inputs:       p -- mps object that describes the problem,
%          option -- which solving approach we would like, between 0 and 3
%                    is the pivoting strategies described in simplex.m,
%                    4 is random facet and 5 is clarkson's algorithm.
% Outputs:    x -- an (possibly) optimal solution to the given problem,
%             t -- time statistics using timeit,
%             f -- the optimal solution value or -Inf/Inf if unbounded/
%                  infeasible,
%            its -- number of iterations performed by the strategy taken.
%           flag -- exit flag: 2 if unbounded, 1 if infeasible, 0 if
%                   solution found and -1 if maximum number of iterations.
    p1 = @() phase11(A,b);
    [x,w,flag,i] = p1();
    if flag == 2
        x = (1:d)' + -Inf;
        f = -Inf;
        its = i;
        return
    elseif flag == 1
        its = i;
        x = (1:d)' + Inf;
        f = Inf;
        return
    else
        i1 = i;
    end
    if option >= 0 && option <= 3
        g = @() simplex(w,A,b,x,c,option);
    elseif option == 4
        g = @() simplex_random_facet(w,A,b,x,c);
    elseif option == 5
        error("can't call clarkson within clarkson")
    end
    [x,f,~,its,flag] = g();
    its = its + i1;
end

function [x,w,flag,its] = phase11(A,b)
% Phase 1 LP. Attempts to find a feasible point for the problem defined by
% A (ineqaulity constraints) and b (the right side of the inequality) by
% minimizing the sum of the constraints that are violated. If the problem
% is infeasible the program will have exit flag 1, if the maximum number of
% iterations is reached by the solver the program will have exit flag -1
% and otherwise if a solution is found the program exits with 0. The
% program also returns the working set w to continue onto a possible phase
% 2, the possibly feasible points xnew and the iterations performed by the
% solving strategy. The solving strategy is defined by the option variable
% and described in evallp.m.
    [n,d] = size(A);
    [W0,idx] = licols(A');
    x0 = W0' \ b(idx,:);
    S = A*x0-b;
    V = find(S < 0);
    nv = length(V);
    P = zeros(n,nv);
    P(sub2ind(size(P),V,(1:nv)')) = 1;
    Ap = [A P; zeros(nv,d) eye(nv,nv)];
    bp = [b; zeros(nv,1)];
    cp = sparse([zeros(d,1); ones(nv,1)]);
    [xf,f,e,o] = linprog(cp,-Ap,-bp);
    its = o.iterations;
    if e == 0
        flag = -1;
    elseif f ~=0
       flag = 1;
     else
       flag = 0;
     end
    x = xf(1:d);
    wc = A*x==b;
    [B,~] = licols(A(wc,:)');
    [~,w] = intersect(A,B','rows');
end
```

F Rank Estimation of a Matrix

```
function [Ap,idx] = licols(A,tol)
% Function to retrieve the linearly independent columns of a given matrix A
% along with the set of indices the selected columns with respect to the
% original matrix A. The function uses the QR decomposition of A to find
% all such columns.
% Inputs:    A -- the matrix which we wish to find linearly independent
%                 columns for,
%          tol -- the tolerance level for the selections, default is 1e-10.
% Outputs:  Ap -- the matrix which only contains linearly independent
%                 columns where the columns of Ap are a subset of the
%                 columns of A,
%          idx -- list of indices of A which are linearly independent.
     if ~nnz(A) %% A is the zero matrix, cannot have independent columns
         Ap = [];
         idx = [];
     else
```

```
        if nargin<2, tol=1e-10; end
        [~,R,E] = qr(A,0);
        if ~isvector(R)
            dr = abs(diag(R));
        else
            dr = R(1);
        end
        r = find(dr >= tol*dr(1),1,'last'); %% estimate the rank
        %idx = sort(E(1:r));
        idx = E(1:r);
        Ap = A(:,idx);
        idx = idx';
    end
end
```

G Results

G.1 Adlittle

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	3.936156	1043390.83920138	2701	-1	0
Dantzig	3.729423	738366.178492967	2701	-1	0
Steepest Edge	9.236625	710766.082454923	2701	-1	0
Random Edge	3.865459	589081.436255597	2701	-1	0
Random Facet	1.607076	738366.178492958	2785	-1	0
Clarkson	3.942862	1043390.83920138	2751	-1	0

Table 2: Results for the adlittle dataset.

G.2 Afiro

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	0.404795	2.17368334004308e-14	1008	-1	0
Dantzig	0.007775	-455.961471428571	32	0	0
Steepest Edge	0.59002	-5.03349520404522e-31	1008	-1	0
Random Edge	0.146859	-455.961471428571	381	0	0
Random Facet	0.215206	-4.47001826341646e-28	534	0	0
Clarkson	0.403302	2.17368334004308e-14	1025	-1	0

Table 3: Results for the afiro dataset.

G.3 Agg

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	51.594445	323796940.649623	8988	-1	0
Dantzig	50.799613	323796940.649623	8988	-1	0
Steepest Edge	78.266193	323796940.649623	8988	-1	0
Random Edge	58.345327	292918892.404989	8988	-1	0
Random Facet	8.279151	323796940.649623	9081	-1	0
Clarkson	51.353875	323796940.649623	9475	-1	0

Table 4: Results for the agg dataset.

G.4 Agg2

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	167.723218	579187407.759519	12385	-1	0
Dantzig	168.181058	1159041196.50807	12385	-1	0
Steepest Edge	241.135652	579187407.759519	12385	-1	0
Random Edge	163.767261	263750464.889409	12385	-1	0
Random Facet	19.353062	579187407.759519	12657	-1	0
Clarkson	146.577901	579187407.759519	12969	-1	0

Table 5: Results for the agg2 dataset.

G.5 Agg3

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	147.379093	619945466.361699	12375	-1	0
Dantzig	148.533825	620025415.381237	12375	-1	0
Steepest Edge	233.781978	619945466.361699	12375	-1	0
Random Edge	154.851338	238775282.60111	12375	-1	0
Random Facet	21.372841	619945466.361699	12630	-1	0
Clarkson	153.088173	619945466.361699	12949	-1	0

Table 6: Results for the agg3 dataset.

G.6 Bandm

Pivot Choice	CPU Time	Function Value	Iterations	Flag	Failure
Bland	476.15341	51.8461494058539	16168	-1	0
Dantzig	507.130812	51.8461494058423	16168	-1	0
Steepest Edge	1.05909	-Inf	642	1	1
Random Edge	8.244884	-Inf	866	1	1
Random Facet	154.803401	-Inf	16393	1	1
Clarkson	484.531418	51.8461494058539	16795	-1	0

Table 7: Results for the bandm dataset.

G.7 Beaconfd

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	0.476425	34184.772295159	203	0	0
Dantzig	0.342401	34184.7722981787	186	0	0
Steepest Edge	0.636872	34184.2760229194	216	0	0
Random Edge	0.402223	34184.7719741586	194	0	0
Random Facet	2.02303	34184.7722983032	4128	0	0
Clarkson	0.504298	34184.772295159	344	0	0

Table 8: Results for the beaconfd dataset.

G.8 Blend

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	3.954742	0	2863	-1	0
Dantzig	0.153605	-30.8121498458282	149	0	0
Steepest Edge	7.267025	0	2863	-1	0
Random Edge	3.787269	0	2863	-1	0
Random Facet	1.276257	0	2925	-1	0
Clarkson	3.921273	0	2895	-1	0

Table 9: Results for the blend dataset.

G.9 Boeing1

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	314.992875	197.352975791393	14267	-1	0
Dantzig	314.114266	199.105505791393	14267	-1	0
Steepest Edge	409.134049	198.578945791393	14267	-1	0
Random Edge	343.44687	-268.030633517596	14267	-1	0
Random Facet	42.352087	199.105505791393	14571	-1	0
Clarkson	316.871231	197.352975791393	14803	-1	0

Table 10: Results for the boeing1 dataset.

G.10 Boeing2

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	18.085242	-157.892018703069	5535	-1	0
Dantzig	17.044042	-157.892018703069	5535	-1	0
Steepest Edge	23.411929	-168.017018703069	5535	-1	0
Random Edge	18.083524	-184.268296700969	5535	-1	0
Random Facet	3.438899	-157.892018703069	5635	-1	0
Clarkson	18.039013	-157.892018703069	5779	-1	0

Table 11: Results for the boeing2 dataset.

G.11 Bore3d

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	153.725752	3370.38375552941	10971	-1	0
Dantzig	128.734479	3370.38447479072	10971	-1	0
Steepest Edge	141.085359	3370.38375553658	10971	-1	0
Random Edge	130.76862	5.1781912903004e+51	10971	-1	1
Random Facet	13.54083	3370.38375552939	11151	-1	0
Clarkson	153.988247	3370.38375552941	11051	-1	0

Table 12: Results for the bore3d dataset.

G.12 Brandy

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	0.000749	5820.29850499479	437	0	0
Dantzig	0.000142	5820.29850499479	437	0	0
Steepest Edge	0.000153	5820.29850499479	437	0	0
Random Edge	0.000136	5820.29850499479	437	0	0
Random Facet	0.063135	5820.29850499479	594	0	0
Clarkson	0.036808	5820.29850499479	873	0	0

Table 13: Results for the brandy dataset.

G.13 Capri

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	201.127048	4589.72924292783	12958	-1	0
Dantzig	198.745903	4589.72924292783	12958	-1	0
Steepest Edge	214.392736	4590.57925052216	12958	-1	0
Random Edge	200.87837	4589.72922101656	12958	-1	0
Random Facet	11.058678	4590.57925052214	13119	-1	0
Clarkson	200.458515	4589.72924292783	13395	-1	0

Table 14: Results for the capri dataset.

G.14 Degen2

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	852.840857	-1322.17266666667	18796	-1	0
Dantzig	2191.575835	-14872.8842358104	18796	-1	0
Steepest Edge	3713.44579	-1322.17266666667	18796	-1	0
Random Edge	154.417961	-Inf	2501	1	1
Random Facet	68.363953	-1322.17266666667	19310	-1	0
Clarkson	1727.648622	-1322.17266666667	20261	-1	0

Table 15: Results for the degen2 dataset.

G.15 E226

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	133.737154	38.0310689159362	8624	-1	0
Dantzig	131.170821	1147.85163696209	8624	-1	0
Steepest Edge	175.200388	38.0310689159245	8624	-1	0
Random Edge	7.544327	-Inf	853	1	1
Random Facet	14.302554	38.0310689159225	8851	-1	0
Clarkson	111.164102	38.0310689159362	9047	-1	0

Table 16: Results for the e226 dataset.

G.16 Etamacro

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	1778.066747	-675.432775178026	23198	-1	0
Dantzig	1749.518247	-679.27778195418	23198	-1	0
Steepest Edge	1866.91961	-678.062526166985	23198	-1	0
Random Edge	1743.514164	-2.09683084197647e+119	23198	-1	1
Random Facet	55.494937	-677.385668465272	23547	-1	0
Clarkson	1767.646174	-675.432775178026	23745	-1	0

Table 17: Results for the etamacro dataset.

G.17 Farm

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	0.00056	39900	4	0	0
Dantzig	0.000366	39900	4	0	0
Steepest Edge	0.000388	39900	4	0	0
Random Edge	0.000377	39900	4	0	0
Random Facet	0.008042	39900	23	0	0
Clarkson	0.010046	39900	6	0	0

Table 18: Results for the farm dataset.

G.18 Grow15

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	3197.351485	-2096985.45666421	24901	-1	0
Dantzig	97.410704	-106870941.293575	891	0	0
Steepest Edge	2838.851915	0	24901	-1	0
Random Edge	97.969691	-Inf	932	1	1
Random Facet	67.124235	0	25526	-1	0
Clarkson	3243.78847	-2096985.45666421	24901	-1	0

Table 19: Results for the grow15 dataset.

G.19 Israel

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	24.470204	7615577.32939396	4769	-1	0
Dantzig	22.54268	7137671.02693493	4769	-1	0
Steepest Edge	30.235674	7615577.32939396	4769	-1	0
Random Edge	27.266246	7092401.09510754	4769	-1	0
Random Facet	6.544408	7137678.067149	4850	-1	0
Clarkson	24.209742	7615577.32939396	4957	-1	0

Table 20: Results for the israel dataset.

G.20 Kb2

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	0.473169	-1749.90012990621	229	0	0
Dantzig	0.28864	-1749.9001299062	132	0	0
Steepest Edge	3.594973	-1749.90012990621	932	0	0
Random Edge	3.495352	0	1501	-1	0
Random Facet	1.074955	0	1541	-1	0
Clarkson	0.554344	-1749.90012990621	229	0	0

Table 21: Results for the kb2 dataset.

G.21 Nsic1

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	1038.437963	-49580.0000000002	14355	-1	1
Dantzig	670.685615	-91622.9999999995	14355	-1	1
Steepest Edge	850.942389	-5542188	14355	-1	0
Random Edge	26.9856	-Inf	843	1	1
Random Facet	184.072833	0	14799	-1	1
Clarkson	632.397509	-49580.0000000002	14629	-1	1

Table 22: Results for nsic1 dataset.

G.22 Nsic2

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	579.569388	-49979	14647	-1	0
Danztig	479.689338	-486020	14647	-1	0
Steepest Edge	646.818398	-499341	14647	-1	0
Random Edge	50.397737	-Inf	1818	1	1
Random Facet	49.89871	-399	15074	-1	0
Clarkson	572.584861	-49979	15073	-1	0

Table 23: Results for nsic2 dataset.

G.23 Recipe

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	0.256658	-266.616	113	0	0
Danztig	0.147516	-266.616	88	0	0
Steepest Edge	0.218927	-266.616	102	0	0
Random Edge	0.208822	-266.616	102	0	0
Random Facet	1.690962	-266.616	4584	0	0
Clarkson	0.283113	-266.616	166	0	0

Table 24: Results for recipe dataset.

G.24 Sc50a

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	1.243183	-55.4518191591693	1661	-1	0
Danztig	0.032066	-64.5750770585645	47	0	0
Steepest Edge	0.042618	-64.5750770585645	51	0	0
Random Edge	0.034719	-64.5750770585645	51	0	0
Random Facet	0.493815	-64.5750770585645	1247	0	0
Clarkson	1.249279	-55.4518191591693	1661	-1	0

Table 25: Results for sc50a dataset.

G.25 Sc50b

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	0.041858	-70	60	0	0
Danztig	0.040101	-70	58	0	0
Steepest Edge	0.049244	-70	56	0	0
Random Edge	0.043537	-70	63	0	0
Random Facet	0.561939	-70	1433	0	0
Clarkson	0.053038	-70	60	0	0

Table 26: Results for the sc50b dataset.

G.26 Sc105

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	7.309537	-50.774739937875	3561	-1	0
Danztig	0.203841	-52.2020612117072	105	0	0
Steepest Edge	0.442597	-52.2020612117072	160	0	0
Random Edge	0.278079	-52.2020612117072	142	0	0
Random Facet	1.580061	0	3610	-1	0
Clarkson	7.236006	-50.774739937875	3561	-1	0

Table 27: Results for the sc105 dataset.

G.27 Sc205

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	43.573645	-10.846181525569	7021	-1	0
Danztig	1.582241	-52.2020612117072	248	0	0
Steepest Edge	3.186176	-52.2020612117073	343	0	0
Random Edge	2.348221	-52.2020612117072	368	0	0
Random Facet	3.221558	0	7221	-1	0
Clarkson	43.747345	-10.846181525569	7021	-1	0

Table 28: Results for the sc205 dataset.

G.28 Scagr7

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	14.706095	-1704551.21968213	5094	-1	0
Danztig	14.763084	-1704551.21968213	5094	-1	0
Steepest Edge	20.907758	-1704551.21968213	5094	-1	0
Random Edge	0.061348	-Inf	183	1	0
Random Facet	3.757515	Inf	5150	1	0
Clarkson	14.567853	-1704551.21968213	5257	-1	0

Table 29: Results for the scagr7 dataset.

G.29 Share2b

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	3.487865	-324.202865741403	2808	-1	0
Danztig	3.564655	-334.296682340298	2808	-1	0
Steepest Edge	6.830906	-324.202865741405	2808	-1	0
Random Edge	0.576046	-2.98747919928055e+42	567	0	0
Random Facet	1.381605	-324.202865741405	2860	-1	0
Clarkson	3.493717	-324.202865741403	2945	-1	0

Table 30: Results for the share2b dataset.

G.30 Stocfor1

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	8.487626	-25468.6697284763	4043	-1	0
Danztig	8.189625	-25468.6697284763	4043	-1	0
Steepest Edge	11.316533	-25468.6697284763	4043	-1	0
Random Edge	0.255702	-25468.6697284763	143	0	0
Random Facet	1.369028	-25468.6697284847	2782	0	0
Clarkson	8.531733	-25468.6697284763	4065	-1	0

Table 31: Results for the stocfor1 dataset.

G.31 Vtpbase

Pivot Choice	CPU Time	Function Value	Iterations	Exit Flag	Failure
Bland	0.000494	143152.061160208	215	0	0
Danztig	0.000107	143152.061160208	215	0	0
Steepest Edge	0.000107	143152.061160208	215	0	0
Random Edge	0.000116	143152.061160208	215	0	0
Random Facet	0.055229	143152.061160208	350	0	0
Clarkson	0.025806	143152.061160208	429	0	0

Table 32: Results for the vtpbase dataset.