Randomized Linear Programming

Lab Report Randomized Algorithms IN4337

Koos van der Linden
4133145

Marieke van der Tuin
4079299

ABSTRACT

For linear programming problem instances that have a relative small dimension compared to the number of constraints, two randomized algorithms, SampLP and IterSampLP, are compared in runtime and asymptotic bounds with Simplex, a deterministic algorithm. It is shown that SampLP performs better than Simplex on instances with small dimension. This is not shown for IterSampLP. By analyzing the asymptotic bounds it is expected that IterSampLP will perform better than both SampLP and Simplex when the number of constraints is much larger and the dimension is still relatively small.

1. PROBLEM INTRODUCTION

The Linear programming problem is concerned with finding the optimum of an objective function limited to a set of constraints. The standard form of a linear program is written in equation 1, with x the vector variables to be determined and A the matrix of constraints that should be less or equal to vector b.

$$\begin{aligned} &\text{minimize } \mathbf{c}^T\mathbf{x} \\ &\text{subject to } \mathbf{A}\mathbf{x} = \mathbf{b} \\ &\text{and } \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{1}$$

with $\mathbf{x}$ the vector variables to be determined and $\mathbf{A}$ the matrix of constraints that should be equal to vector $\mathbf{b}$. If the constraints are in the form $\mathbf{a}^T\mathbf{x} \leq b$ or $\mathbf{a}^T\mathbf{x} \geq b$, slack variables can be introduced to rewrite the problem to the standard form.

In this report, three algorithms that solve linear programming problems are examined: Simplex, SampLP and IterSampLP. The last two of these are randomized algorithms. Simplex is the oldest of these algorithms. This algorithm uses the standard form as an input and has a time complexity of polynomial time on average, but has a worst-case complexity of exponential time. How do SampLP and IterSampLP compare with this? It is expected that these algorithms will perform better on instances with small dimension (the number of variables) and large number of constraints. Examples of problems with relatively small dimension are the smallest enclosing circle problem and the weighted Euclidian one-center problem

2. DESCRIPTION OF USED METHODS

Three methods are compared in this report: Simplex, the only deterministic method, and SampLP, a recursive randomized algorithm, and IterSampLP, an iterative randomized algorithm.

2.1 Simplex

The Simplex algorithm [3] is the most basic algorithm for solving linear programming problems. This algorithm uses the standard form as an input and has a time complexity of polynomial time on average, but has a worst-case complexity of exponential time. This methods works as follows:
The set of constraints H (defined by $\mathbf{A}\mathbf{x} \leq \mathbf{b}$), and the constraint that all variables should be non-negative ($x \geq 0$), define a feasible region. This feasible region is a polytope. In the case of 2 dimensions ($d = 2$), this can be easily seen: all area within the lines defined by H and the axes is called the feasible region.
The algorithm can be split in two phases. In phase I, the feasible region is defined, and an initial solution is found (or the algorithm returns because the problem is infeasible. This is the case when the feasible region is empty). In phase II, the program travels along the edges of the feasible region in the direction that minimizes the objective function. It stops when the optimum is found.

The steps for solving an LP instance in standard form are as follows. First a so called tableau is formed, in which the first row is the objective function, and the other rows are the constraints:

$$\begin{bmatrix} 1 & -\mathbf{c}^T & 0 \\ 0 & \mathbf{A} & \mathbf{b} \end{bmatrix}$$

In each iteration, one column and row in A is chosen with a non-zero pivot. This row is multiplied so that this pivot becomes one. Then this row is subtracted from all other rows. The result is that all elements in the pivot column (except the pivot itself) become zeros. This pivot-row is chosen in such a way to minimize Z, ($Z = \mathbf{c}^T\mathbf{x}$), the top right element in the tableau)). As soon as all elements in the first row are negative, an optimal solution is found. The top right element of the transformed tableau contains the minimum value.
The values of the variables is determined as follows: with the final tableau, discern basic and non-basic variables. In the final objective function, non-basic variables have a non-zero factor, and the basic variables have a zero factor. The value of all non-basic variables are zero. For the basic variables: find the row with the only 1 in the column (the rest is all zero's). The last element in this row (in the transformed $\mathbf{b}$ vector, is the value for this basic variable.

In [5] the runtime of Simplex is analysed extensively. But the authors admit that their understanding of the simplex algorithm is far from complete. It is shown that the average runtime is polynomial, but the runtime depends for a large part on the choice of pivot-rule. Until now, for every deterministic pivot rule a counter example has been found that can only be solved in exponential time, so the worst case running time is exponential.

2.2 Simplex implementations

For testing purposes, two implementations of Simplex were used: Gurobi, and Apache. The Gurobi optimizer is a commercial solver for linear programming problems. It gives several interfaces to use the solver, including Java. With Apache, the Simplex Solver that is included in the open source library of Apache Software Foundation, is meant.
Gurobi is used to determine the feasibility of randomly generated linear programming instances. Apache is used to compare the run time with the two randomized algorithms. Apache is also used in these algorithms, when Simplex is called.

2.3 SampLP

The SampLP algorithm (1) was introduced by Clarkson[1]. This randomized Las Vegas algorithm only gives a speed-up for instances with a small dimension. Examples of these type of instances is the smallest enclosing circle problem and a Chebyshev approximation. For instances with a larger dimension ($n > 9d^2$), the Simplex algorithm is called directly.

The algorithm first selects a random subset of constraints and then recursively calls itself. If the set of constraints is getting small, the Simplex algorithm is called, resulting in an optimal solution. The output vector x of the recursive call is afterwards checked by looking whether it is violating any of the constraints. If no violations are found any more, the solution vector x is correct and is returned.

Algorithm 1: SampLP

Input: A set of constraints H
Output: The optimum $\beta(H)$
$S \leftarrow \phi$;
if $n < 9d^2$ **then**
  return *Simplex(H)*;
end
else
  $V \leftarrow H$, $S \leftarrow \phi$;
  while $|V| > 0$ **do**
    Choose $R \subset H \setminus S$ at random, with $|R| = r = min\{d\sqrt{n}, |H \setminus S|\}$;
    $x \leftarrow SampLP(R \cup S)$;
    $V \leftarrow \{h \in H | \text{vertex defined by } x \text{ violates } h\}$;
    if $|V| \leq 2\sqrt{n}$ **then**
      $S \leftarrow S \cup V$;
    end
  end
  return x;
end

The set V of constraints that violate H is small: it is expected that it contains $\frac{d(m-r+1)}{r-d}$ constraints. Using the Markov inequality this results in $Pr[|V| > 2\sqrt{n}] \leq \frac{1}{2}$. Therefore, the while loop has an expected number of iterations of maximum 2. The set S gets in each of the d phases at most $2\sqrt{n}$ constraints. Therefore, the set $R \cup S$ never exceeds the size of $3d\sqrt{n}$. Checking the possible violation of constraints takes $O(d^2 n)$. This results in a bound of $T(n) \leq 2dT(3d\sqrt{n}) + O(d^2 n)$, for $n > 9d^2$.

If $n < 9d^2$ the simplex algorithm is called. This takes $O(d)^{d/2+O(1)}$.

In total, this results into a complexity bound of

$$O(d^2 n) + \log \frac{n}{d^2}^{\log d+2} O(d)^{d/2+O(1)} \tag{2}$$

2.4 IterSampLP

The more sophisticated IterSampLP algorithm (2) was also introduced by Clarkson[2]. It is also an algorithm which only gives a speed-up if the number of variables is small: if $n < 9d^2$ the Simplex algorithm is called. The iterative IterSampLP uses weighted constraints. For each step it selects a random subset of constraints of size $9d^2$. Next, the Simplex algorithm is called using these values. The result of the Simplex call is used to reweigh the constraints that violate x. A constraint with a higher weight has a higher probability to be selected in the next random choosen set.

Algorithm 2: IterSampLP

Input: A set of constraints H
Output: The optimum $\beta(H)$
$\forall h \in H, w_h \leftarrow 1$;
if $n < 9d^2$ **then**
  return *Simplex(H)*;
end
else
  $V \leftarrow H$;
  while $|V| > 0$ **do**
    Choose $R \subset H$ at random, with $|R| = r = 9d^2$;
    $x \leftarrow Simplex(R)$;
    $V \leftarrow \{h \in H | \text{vertex defined by } x \text{ violates } h\}$;
    if $\sum_{h \in V} w_h \leq (2 \sum_{h \in H} w_h)/(9d-1)$ **then**
      $\forall h \in V, w_h \leftarrow 2w_h$;
    end
  end
  return x;
end

The expected number of iterations of the while loop is $O(dlog(n))$, since the total weight of minimum required constraints grows much faster than the total weight of all constraints, since they are more likely to violate result vector x. In the while loop the simplex algorithm is called. This takes $O(d)^{d/2+O(1)}$. Checking the violating constraints takes $O(dn)$.

This results in a complexity bound of:

$$O(d^2 n \log n) + d \log n O(d)^{d/2+O(1)} \tag{3}$$

3. EXPERIMENTAL RESULTS

To test the performance of the Linear Programming algorithms, two different methods are used: by using benchmarks and by generating random graphs.

First of all, the Simplex implementation of Apache is tested by using the benchmark MIPLIB 2010 [4]. This benchmark has Mixed Integer Problems. We are only considering continuous problems. Luckily, next to the integer objective also the LP (continuous) objective is provided for each problem instance. The Gurobi implementation is assumed to be correct and was previously tested by others using the MIPLIB benchmark. The runtime results for SimplexApache on instances of MIPLIB can be seen in table 6 in the appendix and in figure 1 and 2. Since the benchmark instances are very large and therefore require lots of time to test, only a small subset of the benchmark is used to test the performance of SimplexApache. The choice of this subset was based upon problem size and alphabetic order. Next to recording the required time, the solutions were also checked for correctness. This was the case for all problem instances.

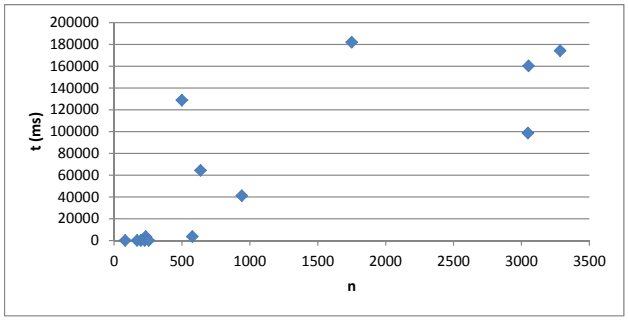

Figure 1: Runtime results for SimplexApache on instances of MIPLIB 2010 [4] compared to the number of constraints: n

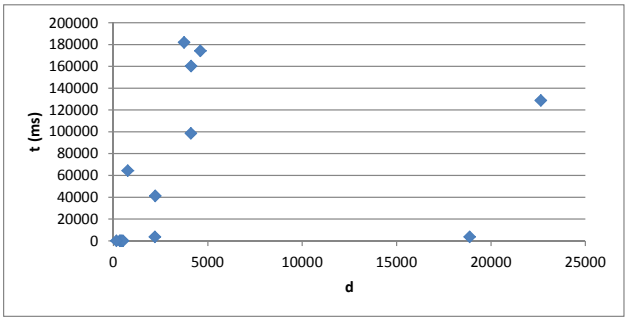

Figure 2: Runtime results for SimplexApache on instances of MIPLIB 2010 [4] compared to the number of variables: d

Unfortunately, this benchmark could not be used to test the performance of SampLP and IterSampLP, because for all instances in this benchmark $n < 9d^2$.

Therefore, to test the performance of the randomized algorithms, a linear programming problem generator is used. This generator generates problems with uniformly random constraints and a random objective function. The feasibility of each generated problem is tested with Gurobi, and feasible instances are stored for later use.

With the generator first a set of feasible problem instances was randomly generated, with d ranging from 2 to 9, and with $n = 10d^3$, so $n > 9d^2$ always holds. For each value of d, four linear programming problems were generated.

Next, the algorithms are run given these problem instances. Each algorithm was repeated 100 times for each problem instance and the results were used to calculate an average runtime. Runtime is measured in two ways: the total runtime and the Simplex runtime. The Simplex runtime is defined as the time the SimplexApache algorithm is run in total. SampLP and IterSampLP both call Simplex for small instances of the original problem. By measuring the time Simplex is running, a comparison can be made that is less dependent on the implementation of SampLP and IterSampLP.

The results can be seen in table 6 and the corresponding figures 3, 4 and 5. In the figures, the runtime is only compared with n, the number of constraints and no longer with d. This is because n depends on d ($n = 10d^3$).

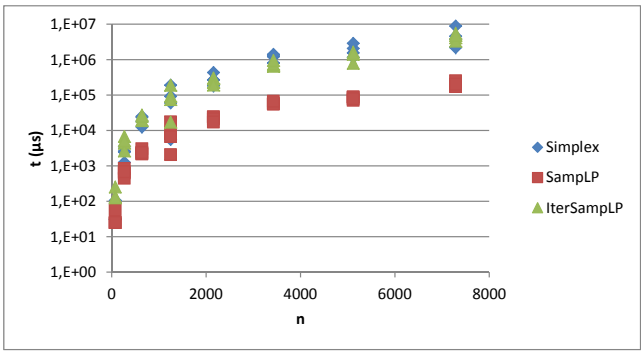

Figure 3: Runtime results for SimplexApache, SampLP and IterSampLP on the randomly generated problems. The total runtime is measured.

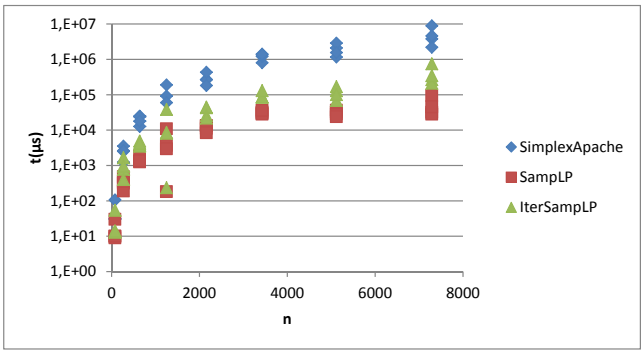

Figure 4: Runtime results for SimplexApache, SampLP and IterSampLP on the randomly generated problems. Only the time Simplex is running is measured.

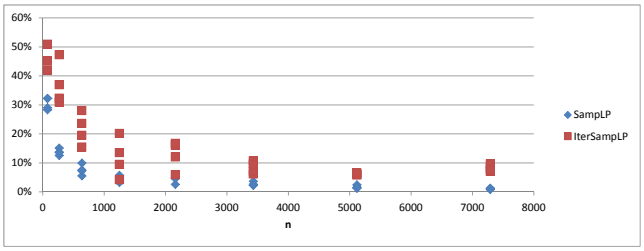

Figure 5: Runtime results for SampLP and IterSampLP on the randomly generated problems. Only the time Simplex is running is measured. The runtime of SampLP and IterSampLP is plotted against the runtime of SimplexApache as a percentage.

4. ANALYSIS OF RESULTS

From figure 1 it can be concluded that the Simplex runtime is generally higher for a larger number of constraints. The relation between n and the required time seems to be quite polynomial, as was expected from the runtime analysis of Simplex.

However, the plot showing the dimension against the time (Figure 2) doesn't exactly seem to follow this polynomial bound. For values of d smaller than 5000 it somewhat holds, but for the 2 larger values of d in the considered benchmark ($d = 18866$ and $d = 22641$), this relation doesn't hold: the running time is much lower than you would expect if there really is a polynomial relationship. It might be the case that - besides values for d and n instances can be marked "easy" or "difficult", for example depending on the number of zeros in the matrix A and the maximum and minimum values.

In figure 3 the runtime of the SimplexApache, SampLP and IterSampLP is considered for our own generated instances. Plotted on a logscale, it can be concluded that the SimplexApache implementation does not behave polynomially on average for our examples. The SampLP and IterSampLP do correctly show their logarithmic complexity bounds. SampLP always runs faster than Simplex.
IterSampLP sometimes is faster than Simplex, but sometimes is not.

The runtime was also considered by the time the Simplex algorithm was running. These results are shown in figure 4 and 5. It can be seen that the SampLP algorithm only requires 8% of the Simplex running time on average compared to Simplex Apache.
IterSampLP requires a bit more Simplex time (19%), since it is called in every iterative step to compute the optimal solution of the random subset. Compared with figure 3 it can be seen that IterSampLP should have a lot of time consuming operations next to calling the Simplex algorithm. This is indeed the case: reweighting after each step takes quite some time. A more efficient implementation of this reweighting process might cause a better performance of IterSampLP in general.

SampLP performs in our experiments better than IterSampLP. This is as expected if you look into the runtime analysis. For instances with larger values for n, IterSampLP theoretically should perform better than SampLP. However, these instances are such large that they are not runnable on a simple machine without generating Out Of Memory or Heap Space errors.

5. DISCUSSION OF THE BENEFIT OF RANDOMIZATION FOR THE PROBLEM

It has been shown that SampLP gives an enormous speed-up compared to the Apache Simplex implementation.
IterSampLP sometimes does this, but is expected to do so for large problem instances. It is therefore useful to use the randomized versions. However, the presented randomized algorithms do not work for all instances. It only gives a speed-up if $n > 9d^2$. Else, the Simplex algorithm is called immediately. This requires very specific problem instances with small dimensions. The smallest enclosing circle problems are for example perfectly solvable with randomized algorithms, since these always have a dimension of 3.

Although randomized algorithms provide a speed-up, a professional LP solver like Gurobi still performs better for instances with a large n. The randomized algorithms might be improved by using the Gurobi solver instead of calling the Simplex algorithm. For larger n this should give a speed-up, since the Gurobi solver should be more optimized than a general Simplex optimization. If then the runtime of the randomized algorithms and Gurobi is plotted (figure 6), it can be seen that Gurobi still performs better than the randomized algorithms. Concluding: the randomized algorithms do not outperform a professional solver like Gurobi (for large n).

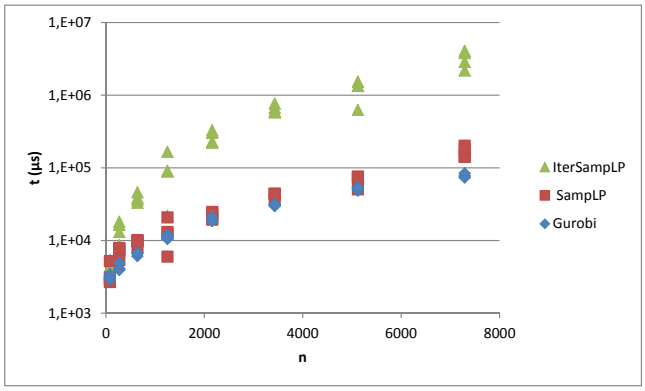

Figure 6: Runtime results for Gurobi, SampLP and IterSampLP on the randomly generated problems. SampLP and IterSampLP use Gurobi instead of Simplex for obtaining an optimal solution

6. CONCLUSIONS

The performance of three algorithms, among which two randomized, was tested on linear programming problem instances. The randomized algorithms that were tested,
SampLP and IterSampLP, were specifically designed to perform better on problem instances with small dimension. In the performance analysis this was indeed the case for SampLP. The deterministic algorithm Simplex was well out performed by SampLP. For some instances SampLP required in average only 3% of the running time of Simplex to solve a

problem. With smaller dimensions the gain was less significant, caused by the larger setup cost. Although SampLP performed much better than the tested Simplex implementation, it did not perform better than the commercial linear programming solver Gurobi.
IterSampLP did not out perform Simplex. When the runtime bound of IterSampLP is considered, this can be understood. The test cases that were generated for the analysis were too small to make use of this sophisticated algorithm. To be able to run bigger problem instances that might show the benefit of IterSampLP, better test equipment is required with more memory and computation power.
In this analysis the number of constraints was dependent on dimension of a problem. For better analysis these two values can be chosen separately (as long as the dimension is relatively small). Another suggestion is to generate random problems from a problem domain, instead of totally random. By generating problems from a specific domain, the runtime analysis might be more useful.

7. REFERENCES

[1] K. L. Clarkson. A las vegas algorithm for linear programming when the dimension is small. In *IEEE 29th Annual Symposium on Foundations of Computer Science*, pages 452–456, 1988.

[2] K. L. Clarkson. Las vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM (JACM)*, 42(2):488–499, 1995.

[3] G. B. Dantzig. Maximization of a linear function of variables subject to linear inequalities, in activity analysis of production and allocation. pages 339–347, 1951.

[4] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[5] D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, 2004.

APPENDIX

Table 1: Runtime results for SimplexApache on instances of MIPLIB 2010 [4]

Problem Name	n	d	t (ms)
30n20b8	576	18,866	3,641
50v-10	233	2,196	3,666
acc-tight4	3,285	4,608	174,289
acc-tight5	3,052	4,114	160,245
acc-tight6	3,047	4,105	98,612
beasleyC3	1,750	3,750	182,036
bnatt400	5,614	3,600	Stopped after 3h40m
cov1075	637	757	64,453
enlight13	169	338	36
enlight14	196	392	17
enlight15	225	450	21
enlight16	256	512	22
enlight9	81	162	3
ex1010-pi	1,468	25,200	Stopped after 2h50m
g200x740i	940	2,220	41,186
ger50_17_trans	499	22,641	128,743

Table 2: The runtime results (in microseconds) for SimplexApache, SampLP and IterSampLP on randomly generated linear programming problems with d ranging from 2 to 9 and $n = 10d^3$. The runtime is an average over 100 repetitions. A distinction is made between total runtime and the time that Simplex is running.

#	n	d	SimplexApache	SampLP total	SampLP Simplex	IterSampLP total	IterSampLP Simplex
1	80	2	31	25	10	127	13
2	80	2	106	57	30	255	54
3	80	2	31	25	9	135	14
4	80	2	31	25	10	128	14
1	270	3	3518	843	482	6767	1663
2	270	3	2498	627	314	4419	773
3	270	3	1249	449	189	2677	404
4	270	3	2591	681	353	5213	958
1	640	4	24974	3044	1815	27322	4874
2	640	4	23685	2223	1317	19794	3663
3	640	4	12578	2169	1250	18999	3526
4	640	4	17826	2374	1348	25389	4220
1	1250	5	91605	8571	3663	93396	8672
2	1250	5	190610	17437	10941	189220	38456
3	1250	5	5599	2080	181	17335	234
4	1250	5	59826	6676	2950	75248	8101
1	2160	6	266887	24334	13178	301377	43118
2	2160	6	269059	24017	13366	291398	45011
3	2160	6	434566	21049	11186	231637	25728
4	2160	6	182747	17205	8382	192605	22020
1	3430	7	1402486	56464	31761	650137	86166
2	3430	7	805804	57138	29486	685525	85931
3	3430	7	1209336	55658	27997	753164	86389
4	3430	7	1349811	67305	35871	950145	130263
1	5120	8	2087614	86721	32143	1428837	130015
2	5120	8	2891915	87656	36512	1643975	171009
3	5120	8	1170917	71039	28429	797186	68344
4	5120	8	1545349	78305	23893	1470287	101232
1	7290	9	8944001	252987	95365	5591116	751477
2	7290	9	3770595	171645	31653	4177553	267840
3	7290	9	2190897	171699	28134	3349625	210678
4	7290	9	4572966	201903	45554	4859949	348285