IE 515 - Term Project Report

Rıza Özçelik

December 16, 2018

1 Introduction

This project contains various solutions and their comparisons to a hypothetical network package transfer problem for an Internet Service Provider(ISP) company. The goal of the ISP company is to send as many network package per second as possible to a target from a source through its communication lines which have certain capacities, meaning that they allow only a certain amount of bits to go through per second. Since an ISP company cannot connect all possible sources to all possible targets directly, it uses server computers that its customers connect for package transfer. Thus, any package is transferred to the servers from the source and then it is delivered to the target. For the purposes of this project, the case where there is a single provider and a single target in the system, which is the case for a single transfer, is focused.

This document is organized as follows: Next section 2, explains the proposed solution model. In section 3, algorithms that are used in the solution are described. In section 4, implemented algorithms are compared in different aspects and in the discussion, the results are evaluated by comparing the algorithms. Finally, document is ended with a conclusion.

2 Solution

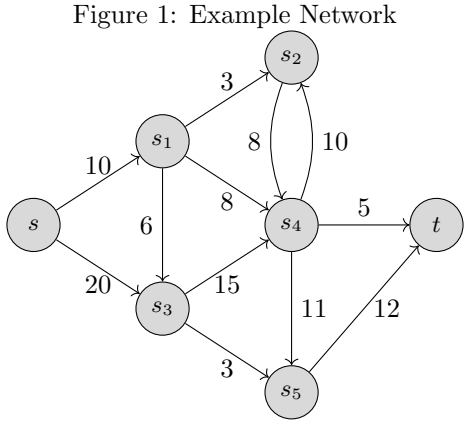
Problem is modelled as a maximum flow problem, where each server is modelled as a node as well as the source and target. Communication lines those connect these servers are the directed arcs, whose capacities are determined by the communication line capacity. These arcs are directed because of the fact that, ISP company may choose sending packages in only one direction through a line.

Given this model, pushing as much flow as possible from source to target corresponds to sending as many package as possible per second. Thus, solving the constructed maximum flow problem solves the problem of ISP as well.

In Figure 1, a pictorial representation of an imaginary scenario is shown. In this specific scenario, source (s) is connected to server 1 (s_1) and server 3 (s_3) with communication lines that have capacities 10 and 20 bits per second

respectively. Intermediary nodes also have connections in between, which allows server to server transfers. Finally, server 4 (s_4) and server 5 (s_5) are connected to the target (t) with capacities 5 and 12 bits per second respectively.

Figure 1: Example Network



3 Algorithms

To solve the constructed maximum flow problem, two main approaches are implemented with different versions, namely, capacity scaling and linear programming. Capacity scaling is implemented using different path finding techniques and update mechanisms, whereas for linear programming, scipy and gurobipy are used to solve the same formulation separately.

3.1 Capacity Scaling

Capacity scaling is a combinatorial approach to solve maximum flow problem that is based on well-known Ford Fulkerson algorithm. It modifies Ford-Fulkerson such that at each iteration, large amount of unit flow is pushed from source to target. To do so, it uses a parameter, Δ which is used to ignore the edges in the residual graph whose capacity is less than Δ. This way, at each step, only the pushes that are larger than Δ is considered. When a new push is impossible with current Δ, Δ is divided by 2. The algorithm terminates when Δ becomes zero and no extra flow can be pushed. See the algorithm 1, for pseudo-code.

Algorithm 1 Capacity Scaling Algorithm

1: $\Delta = \lfloor 2^{logU} \rfloor$
2: **while** $\Delta > 0$ **do**
3: **while** An augmenting path exists in residual graph with Δ **do**
4: Identify an augmenting path P in residual graph
5: δ = Minimum residual capacity in P
6: Augment δ unit of flow along P
7: **end while**
8: $\Delta = \Delta/2$
9: **end while**

For the purposes of this work, this algorithm is modified in two aspects. First aspect is the path finding procedure on the residual graph. Path finding can be done in various ways and in this project, BFS and DFS are implemented and compared. To do so, both algorithms are started from the source node and iterations are stopped when the target. Though the complexities of both algorithm are the same, their inner mechanisms are different. Since DFS traverses the graph as deep as possible and BFS traverses the graph layer by layer, effect of the path finding algorithm is a point of consideration.

Second aspect that is considered is Δ update procedure. In the original algorithm, Δ is halved at each update step, yet different update schemes are also possible. In this project, a version which uses a heap of edges is utilized. Since halving is a graph independent solution, heap can help to finding better Δ choices. To do so, all edges are inserted into a min heap with respect to their capacity and they are popped until an edge whose capacity is equal to Δ. In turn, Δ is set to the capacity of the edge that is seen in the middle during the popping process.

3.2 Linear Programming

Every maximum flow problem can be formulated as a linear program and integrality of the result is guaranteed due to the totally unimodularity of the formulation. To do so, for each edge, a decision variable is added which denotes the amount of flow that goes through the edge. Capacities are added as constraints on these decision variables and for each node, an additional constraint that enforces the node flow balance is added. Note that in a maximum flow problem, this is not the case for source and target which potentially breaks the totally unimodular structure. To preserve it, an artificial edge from target to source with infinity capacity is added. Finally, the objective is to maximize sum of all decision variables apart from the one that corresponds to artificial edge, since it will push all the flow back from target to source. Below you can find the resulting formulation:

$$
\begin{aligned}
&\text{maximize} && \sum_{(i,j)\in A-\{t,s\}} x_{ij} \\
&\text{subject to} && \sum_{i:(ik)\in A} x_{ik} - \sum_{i:(ki)\in A} x_{ki} = 0, \quad \forall k \in V \\
& && 0 \leq x_{ij} \leq u_{ij}, (i,j) \in A
\end{aligned} \tag{1}
$$

In this formulation, s, t stands for source and target nodes relatively and A, V stands for the arc and vertex set. Lastly, x_{ij}, u_{ij} stands for flow on the arc from node i to node j and u_{ij} stands for the capacity of the same arc. This formulation is implemented with two different tools: One using *scipy's linprog* module and the other is *gurobipy*. The comparison of the tools are done in the discussion.

4 Experiments

To compare the different implementations of the above mentioned approaches, five different experiments are designed. They are designed to test the scalability of the algorithms under varying node counts(N), edge counts(M) and capacity distributions with different mean, variance and skewnesses. At each setup, different algorithms are run on graphs that are randomly drawn from corresponding distributions and their total running times are recorded. Remarks that are found important are visualized and presented below.

4.1 Experiment 1

This experiment aims to provide a broad comparison of the algorithms' execution time and to have an idea of the effect of each parameter. All five implementations are run on graphs with different node/edge counts, density[1] and capacity distributions.

Node counts are set to 50, 150 and 300, whereas density takes the values of 0.2, 0.5 and 0.8. For each plot, other variables apart from the plotted one are fixed. Note that for *scipy*, at certain points measurements do not exist, since algorithm did not terminate due to excessive memory usage. Furthermore, plots of this experiments are in the log scale, since running time of combinatorial and linear programming solutions are at quite different scales. Average execution time of each algorithm over all configurations can be seen in Table 1 and see Figure 2 for the result of different graph configurations.

In addition to testing graph configurations, edge capacity distributions are also changed for each algorithm. To do so, distributions with low/high mean and standard deviation are generated as well as left/right skewed and no skew distributions. Each configuration is tested for each algorithm and the results are in Figures 3, 4 and 5. For clarity, each line is sorted by execution time prior to plotting.

[1] Density of the graph is computed as M/N^2

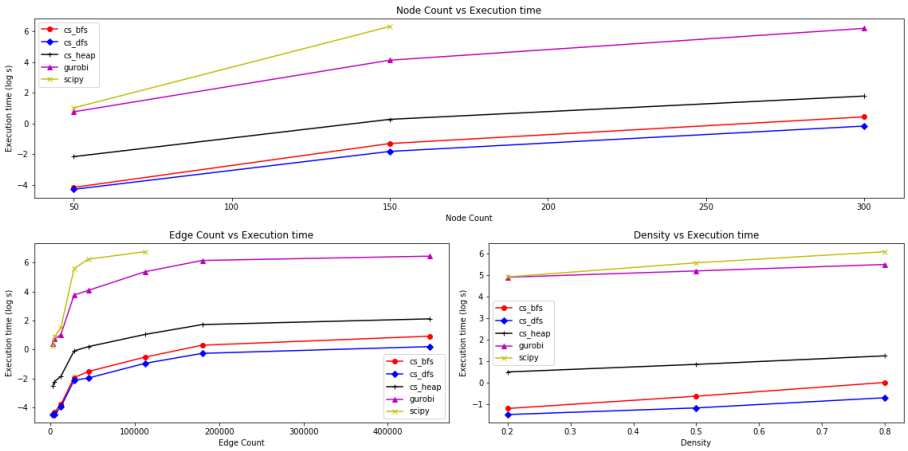


Figure 2: Varying Graph Configurations for All Algorithms

Table 1: Average Execution Time of Algorithms

Algorithm	**Execution Time(s)**
CS-DFS	0.35
CS-BFS	0.61
CS-HEAP	2.46
GUROBIPY	183.86
SCIPY	276.78

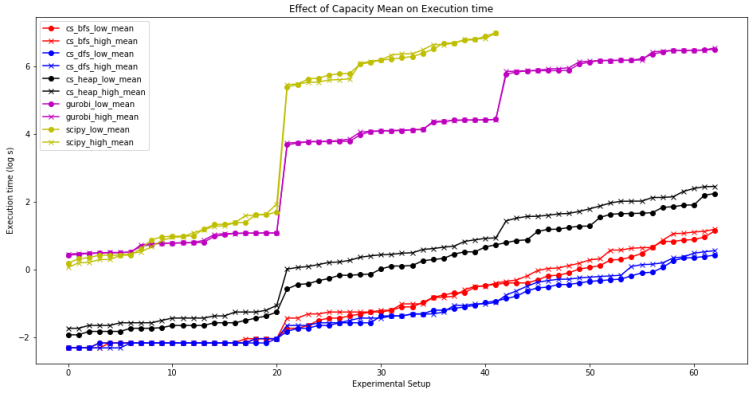


Figure 3: Varying Mean for All Algorithms

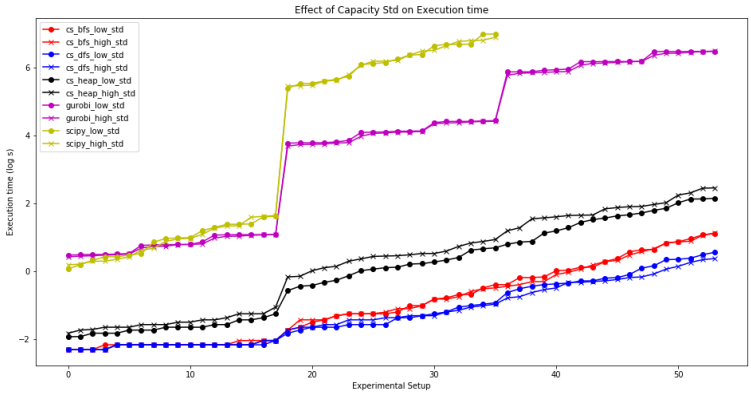


Figure 4: Varying Standard Deviation for All Algorithms

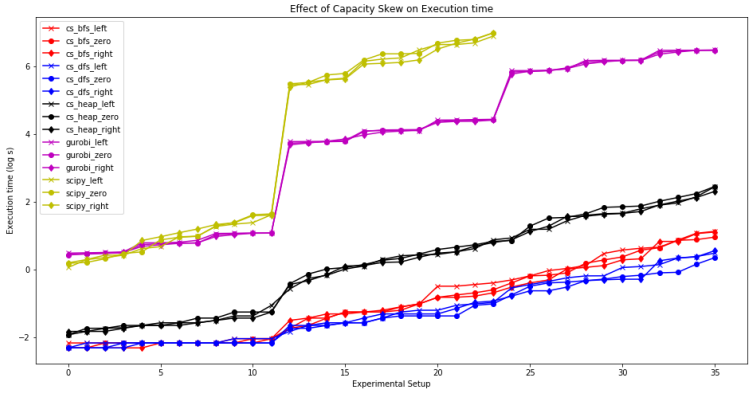


Figure 5: Varying Skew for All Algorithms

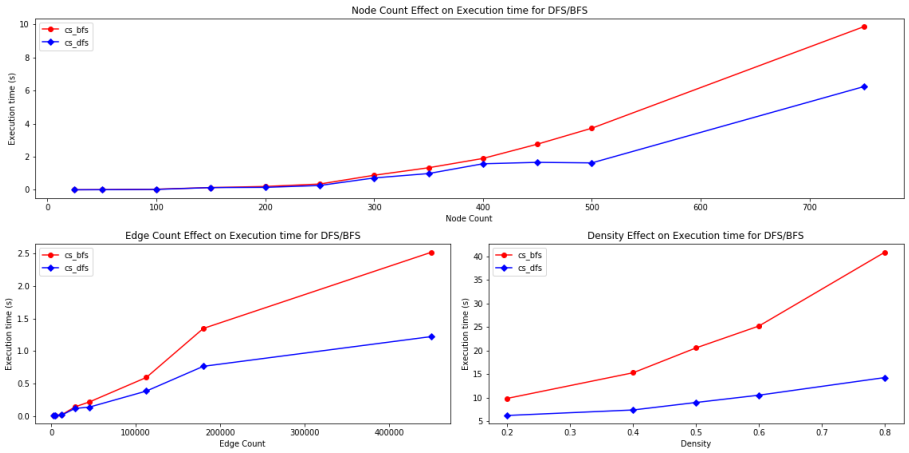


Figure 6: Varying Graph Configurations for CS-BFS and CS-DFS

4.2 Experiment 2

This experiment inspects effect of the path finding algorithms on capacity scaling. To measure this effect, capacity scaling is run with BFS and DFS separately with different graph configurations. Node count is changed from 25 to 750 gradually whereas density took values of 0.2, 0.4, 0.5, 0.6, 0.8. For this experiment, data distribution is fixed to a normal distribution with mean 50 and standard deviation 30. See Figure 6 for the corresponding results.

4.3 Experiment 3

This experiment aims to reveal the time execution difference between LP and CS approaches. Thus, capacity scaling with BFS and LP with gurobipy are run on node counts that increases from 25 to 250 gradually as well as densities that are 0.2, 0.5 and 0.8. Capacity distributions are the same as experiment 2. The results can be seen in Figure 7.

4.4 Experiment 4

This experiment is designed to reflect the effect of heap modification to capacity scaling. Thus, both of the original and heap versions are run on different data and graph configurations. Node counts are varied from 25 to 300 whereas density takes the value same as previous experiment. Moreover, skewness of the data is changed to left, right and zero skew to observe the effect. The results are displayed in Figures 8 and 9.

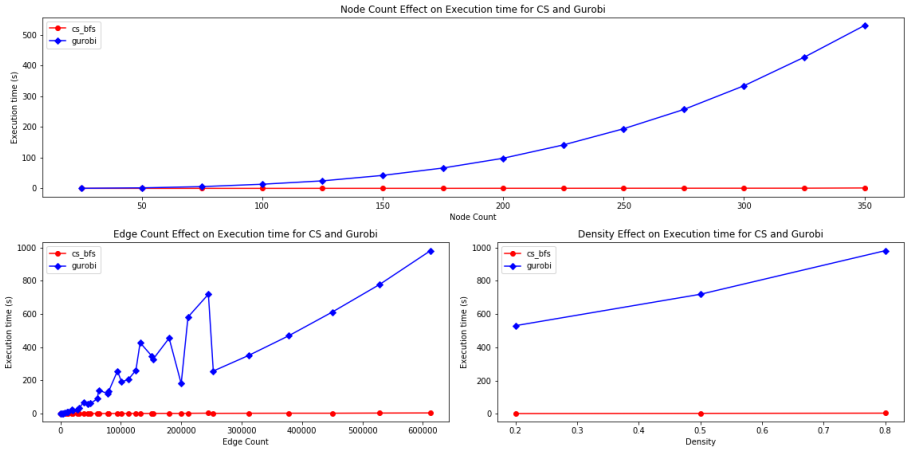


Figure 7: Varying Graph Configurations for CS-BFS and gurobipy

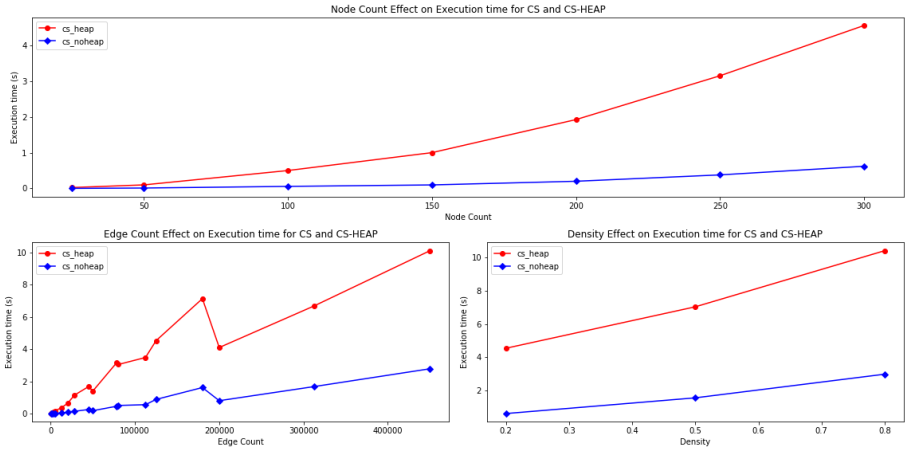


Figure 8: Varying Graph Configurations for CS and CS-HEAP

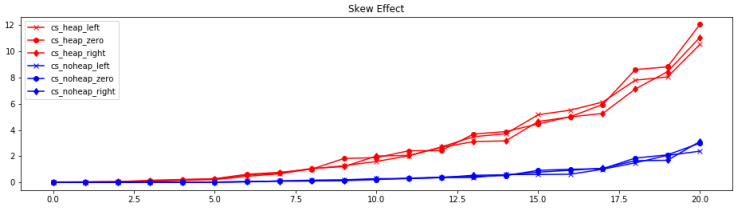


Figure 9: Varying Skewness for CS and CS-HEAP

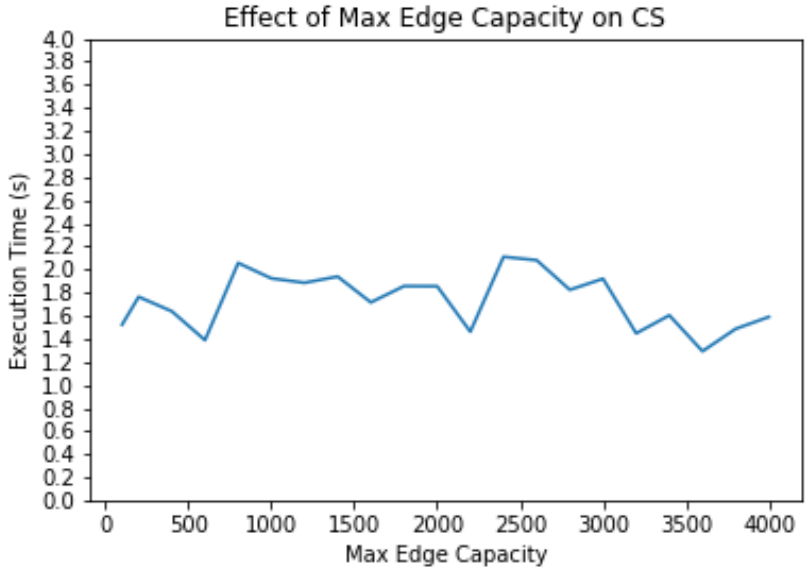


Figure 10: Varying Max Edge Capacity for CS-BFS

4.5 Experiment 5

This experiment is designed to reflect the effect of maximum capacity in the graph which is a theoretical parameter of the capacity scaling algorithm's running time. In this case, solver is fixed to CS with BFS, node count is fixed to 300 and density is fixed to 0.5. Edge capacities are drawn from 21 different uniform distributions with mean varying from 50 to 2000. Note that for uniform distribution changing the mean changes the upper bound as well, which means that maximum capacity is changed from 100 to 4000. Results are as in the Figure 10.

5 Discussion

In experiment 1, all of the algorithms are compared under different circumstances. As seen in table 1, on average, capacity scaling is considerably faster than linear programming. As can also be seen in the resulting plots, independent of the experiment configuration, combinatorial approaches outperform linear programming in terms of execution time. The difference is so open that, the plots are in log scale to reveal the increase in execution time as graph gets larger. This comparison is done with further detail in experiment 3.

In experiment 1, we can also see that, execution time is highly effected by N and M and tends to increase linearly with respect to these parameters. Density also seems important and the plot suggests a quadratic relation between

execution time and density[2]. Plots for each algorithm are parallel which means that they are all effected in the same way. Given that theoretical analysis of capacity scaling also proposes a linear relation with respect to M and N, these findings are accurate. For linear programming, since underlying implementation that is used in gurobipy is not known such a comparison would not be accurate. For scipy implementation, we can see that for the graphs above 150 nodes, it creates memory error which shows its inefficiency compared to gurobipy. Due to this reason, it is not tested further on the remaining experiments.

Another points that can be drawn from this plot is that using DFS for path finding is consistently more efficient than using BFS and updating Δ with the help of a heap slows the algorithm down. These points are examined further in experiments 2 and 4 respectively and validated.

In Figures 3, 4, 5, we can observe the effect of capacity distribution for the algorithms. To generate 3, each algorithm is run on graphs with with high and low capacity edges. As expected, for linear programming solution, this creates almost no effect. Though maximum edge capacity is a theoretical parameter for capacity scaling, apart from the heap version, it seems to have no effect and this is validated in experiment 5. In Figures 4 and 5 we can see that standard deviation and skew also seems irrelevant with running time. Though, intuitively graphs whose edges are mostly populated in lower capacities and graphs whose edges are mostly populated in larger capacities can differ in execution time, experiment results deny this intuition.

In experiment 2, effect of the path finding algorithm that is used to identify an augmenting path is analyzed. Though BFS and DFS have the same time and space complexity in theory, their graph traversal is quite different which makes this question an intriguing one. In figure 6 we can see that both of them react the same way to all variables, yet DFS based path finding results in consistently faster termination. Especially in density plot, we can see that BFS is having trouble to find a path to the target as graph gets more crowded. This also complies with the intuition. Since more dense plots mean more nodes in a layer and BFS traverses the whole layer before moving to the next one, it takes longer to reach to the end. On the other hand, DFS always tries to get as deep as possible and this greedy approach seems to perform better for path finding. Thus, though theoretically there is no difference between using DFS and BFS for path finding in capacity scaling, DFS looks as a more suitable choice in practice.

Experiment 3 is conducted to see the performance difference between capacity scaling and linear programming. Since experiment 1 does not propose a significant effect of capacity distributions, analysis is done only on the varying M,N and density values. In figure 7, we can clearly see that as N increases, linear programming approach takes quadratically more time whereas capacity scaling is much more robust. The gap between two approaches are also clear in increasing edge count and density. These plots show that, capacity scaling algorithm is apparently more scalable than linear programming in terms of ev-

[2]Note that plots are in log scale.

ery parameter. Note that in this experiment BFS is used for path finding in capacity scaling and gurobipy implementation of linear programming is utilized. Yet, drawn results look robust to such implementation details.

Experiment 4 is designed to see if using a heap for Δ update has a prominent effect on execution time. To do so, both versions are run on different graph and edge capacity configurations. Figure 8 proposes that heap harms the performance independent of the configuration and results in a slower overall performance. This is due to the extra cost of maintaining a heap. Note that since edge capacities in residual graph is not static, we need to use a dynamic heap where we can change the values in the nodes or store versions of the edge capacities. This definitely introduces an extra load on the algorithm. In Figure 9, both algorithms are compared under different skewnesses to see if one is more robust than the other but the resulting plot does not offer such a relation.

Experiment 5 is conducted to analyze the effect of maximum edge capacity in a graph which is a parameter of capacity scaling's time complexity. For this purpose, maximum edge capacity is changed from 50 to 4000 gradually while other parameters are kept fixed. Yet, execution time does not increase constantly and correlation coefficient of maximum edge capacity and execution time is -0.2 which also proposes independence of two variables. Thus, theoretical parameter does not show a prominent effect on the execution time in practice.

6 Conclusion

In this project, various approaches to maximum flow problem is implemented and compared under different configurations. As also revealed by the experiments, capacity scaling based approaches are significantly faster than linear programming solution. Yet, both of the approaches succeeded to find exact and optimum solutions to the problem.

For capacity scaling, using a heap does not make sense since its maintenance slows down the algorithm significantly. For path finding though, DFS significantly increases the algorithm's performance and stands as a suitable choice. Moreover, if an heuristic such as euclidean distance between nodes are available, it can also be used with A* search algorithm for more clever path finding.

In the linear programming world, though the problem formulation is quite nice and efficient solvers exist, it is much slower than combinatorial approaches and should be avoided when possible. If linear programming is a must, gurobipy is definitely a better option than scipy, since it is both faster and consumes less memory.

Acknowledgements

Codes and detailed experiment results and configurations are available in GitHub