Improving the execution time of the FOCS algorithm implementation to enable real-time optimized EV scheduling

Loonstra, Joas
Bart, Duncan
Chen, Kuan-Hsun

***Abstract*—The Flow-based Offline Charging Scheduler (FOCS) and Flow Under Local PEnalties Solver (FULPES) algorithms were made to schedule the charging of electric vehicles (EVs). The FOCS algorithm schedules the EVs without accounting for any charging guarantees a parking lot might provide. The FULPES algorithm is based on the FOCS algorithm, however it does provide a schedule which adheres to the charging guarantees. This increases execution times significantly. Due to the increase in execution time, real-time implementation could not be reached. This report details how the goal to speed-up the FOCS algorithm by a factor 100 was met. Finally achieving a speed-up of 185 times, reducing execution time from 300 ms to 1.6 ms for 200 simultaneously charging EVs.**
Methods such as porting from Python to C++, improving memory locality and changing variable types were used. The speed-up amount decreases as the instance size increased, due to the need for a slower fall-back method in solving maximum flow problems. This makes the current solution not yet applicable for larger parking lots.

I. Introduction

Charging electrical vehicles (EVs) can have a high toll on the electrical grid, possibly even causing power outages [1]. This suggests that the charging of EVs should be scheduled in order to mitigate the peak power consumption of parking lots. For this reason the Flow-based Offline Charging Scheduler algorithm (FOCS) was developed. FOCS creates a schedule which can be used to charge these EVs effectively and within a timely manner [2]. However, one thing which the FOCS algorithm does not take into account is the sudden departure of some EVs.
Therefore the Flow Under Local PEnalties Solver (FULPES) algorithm was created. It is based on FOCS, however the algorithm takes the charging guarantees provided by some parking lots into account. FULPES takes a long time to execute, therefore we aim to aid in the real-time adaptation of the FULPES algorithm.
In large-scale settings of 200 to 400 parking spots, the current FULPES implementation can take up to 60 seconds to compute a full schedule. Although no specific running time constraint is imposed in the closing statements of the paper on FULPES by Winschermann et al. [2], we adopt a 1-second runtime target. This reflects the need for quick re-evaluation in real-time EV-charging scenarios where EVs arrive and depart at random frequencies. This means that a minimum speed-up factor of 60 is necessary.
However, the publicly available FULPES implementation when starting this project was not yet fully functional, therefore the decision was made to improve FOCS instead of FULPES. This will still aid in the improvement of FULPES since it is based on the FOCS algorithm. To ensure robustness under edge cases and provide performance headroom, this research targets a 100x speed-up, reducing the total execution time to at most 1 second for the largest instances. The original FOCS code takes on average 300 ms to calculate a schedule. Consequently, this corresponds to a target of 3 milliseconds per 200-spot schedule, forming the performance benchmark for real-time adaptation.
This research will look into how the FOCS EV-charging algorithm can be optimized to efficiently calculate a 200-spot EV-charging lot schedule within 3 milliseconds.

Code is publicly available at: https://github.com/JoasLoo/FULPES

II. Theoretical background

This section covers what flow networks are, how the maximum flow can be found and how we will use them to schedule EV charging. Then multiple methods of reducing the run-time of the original FOCS program will be explored.

A. Flow Networks

Flow networks are a set of vertices connected via a set of edges. All of these edges are given a capacity, through which that amount of flow can pass. The flow is an abstract variable, however it can be translated to a lot of things. It can mean quantities of water, amounts of power, or anything which is desired. For this report, the flow through the network will correspond to the flow of power to an EV.
Each network has a designated source and sink node. The maximum flow is the amount of flow that can be sent from the source to the sink. In other words, a maximum flow problem includes finding a feasible flow through a flow network, which has the highest possible flow rate. All nodes except the source and the sink must have the same flow going into the node as going out. This means that nodes cannot generate or store flow.

B. Maximum flow algorithms

Maximum flow algorithms try to find the maximum flow from a source node to a sink node through vertices. The maximum flow is found for the given capacities and connections. These connections form a network of edges between vertices with a capacity. The edges have a capacity which is used to find the maximum flow from the source to the sink. There are multiple methods of finding the maximum flow. For example the Goldberg and Tarjan maximum flow algorithm is based on the height of a node and the excess flow that is situated in that node [3]. Whereas the Ford-Fulkerson algorithm finds all paths and starts filling each path 1 unit of flow at a time, until the maximum flow is reached [4].

Different algorithms have been created in order to reduce the running time of finding the maximum flow. The worst-case running time of the Edmonds-Karp algorithm is denoted as $O(ve^2)$ [3, 5] where v is the number of vertices and e is the number of edges. The Shortest Augmenting Path algorithm (SAP) has a worst-case time complexity of $O(v^2e)$ [6]. The older Ford-Fulkerson algorithm has a time complexity of $O(vf)$ where f is the maximum flow in the graph [4].

Figure 1 shows an example graph as initially constructed by FOCS. The Edmonds-Karp algorithm uses a Breadth First Search (BFS) method to find paths to fill in. It searches for all possible neighbour nodes from the current node. From node *s* it puts nodes *j1*, *j2*, *j3*, *j4* and *j5* in a queue, from these nodes it repeats the process until it encounters the sink (*t* in the example) then it exports the path from the source to the sink. Eventually the first path it finds is $s \rightarrow j1 \rightarrow i1 \rightarrow t$, it will find the limiting capacity along this path, which is 3.36 in this case. After which it fills in this flow for all edges in the path. This search then continues until there are no more paths available for additional flow.
The Shortest Augmenting Path (SAP) algorithm works differently. It starts by assigning a height label to each node, with the sink set to height 0. The heights increase as you move away from the sink. For example, in the graph in figure 1, all nodes *i* are given height 1, nodes *j* get height 2, and the source node *s* is assigned height 3.
Using these heights, the algorithm searches for augmenting paths. From the starting node, it looks for a neighbor with a lower height—this indicates that flow can be pushed in that direction. If there is residual capacity on the edge and the neighbor's height is exactly one less than the current node's, then that edge can be added to the path. SAP uses Depth First Search (DFS) to explore the graph along these available edges and construct valid paths for pushing flow. The search starts at *s*, finds *j1*, and from *j1* continues, to find *i1* and then to *t*. After which a path is found, the flow is filled in and the search continues. If a dead end is found, where there is no residual capacity to any edge, the height of that node is increased by one in an attempt to include backward edges in the next search. Once no more path can be found using SAP, it finishes by having another algorithm as a fallback, this fills in the last few remaining edges.

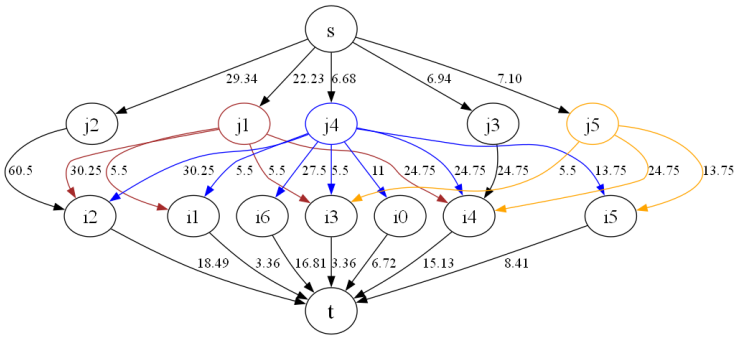

Fig. 1. Graph created during FOCS, using the first 5 entries of the data file DEMSdata_FOCS_v1 as provided by L. Winschermann. Drawn using the Graphviz Preview application.

C. FOCS

The Flow-based Offline Charging Scheduler algorithm (FOCS), is an algorithm which aims to find the optimal EV charging schedule for a parking lot filled with EVs. The graphs consist of 4 layers, the source and sink and 2 layers in-between which determine the power output to each EV [2], see figure 1. The edges indicate the maximum power flow that can reach each state within the graph. With this set-up a maximum flow algorithm can be used to effectively schedule the charging of electric vehicles. The solving phase of the FOCS algorithm utilizes a sub-algorithm, for solving maximum flow problems. This sub-algorithm is executed in multiple rounds in order to achieve the required result. The entire FOCS algorithm has a worst-case time complexity of $O(v^2x)$. Here v is the amount of vertices in the digraph and x is the worst-case time complexity of the used maximum flow algorithm [2]. The main innovative aspects of FOCS are the generation and updates of the graph. Where the generation happens in the setup phase and the updates and solving happen in the solving phase. The FOCS algorithm takes 5 rows of input from a dataset to calculate the power to each EV. First a job-list is created, where each job is an EV. This list is used to create the first layer connected to the source. The capacity from the source to each job is equal to the "total_energy_Wh" variable, which is the total amount of energy the car is requesting. In figure 1 this is the *j*-layer. The second layer is the time intervals which the jobs are in, seen as the i-layer in the example graph. Time is divided into slots of 900 seconds, the arrival and departure time of each EV is measured in these timeslots. The edges are then created between the jobs and the timeslots they are in. The capacity of these edges is based on the maximum amount of energy that can be charged in that time interval. The last edges, from each timeslot to the sink, are initiated to 0 and later updated during the solving phase of FOCS.

The updating of the graph is done in two steps, the first is defining critical and subcritical jobs in the *j*-layer. If the energy demand can be fulfilled for a job, then that job is set as critical. After all jobs have been flagged as either critical or subcritical, all critical jobs are removed from the graph, thereby shrinking the graph. After this step, an extensive formula, which can be found in the original work of Winschermann et al [2], is used

to calculate the new capacities of all edges.

D. FULPES

The FOCS algorithm functions as expected, however parking lots have charging guarantees which need to be met. These charging guarantees can sometimes be visible when entering a parking lot. They often state how much power is guaranteed to have charged by a certain time. These charging guarantees were not accounted for in the FOCS algorithm, therefore another iteration of FOCS was created, namely the Flow Under Local-PEnalties Solver algorithm (FULPES). FULPES is based on FOCS, however with extra requirements. These charging guarantees are implemented in the algorithm as different sub-graphs with their own specific charging guarantees and limits per time interval. These sub-graphs are instantiated by adding an additional layer next to the *j*- and *i*-layer, which will be set as the sink. This new implementation made the time complexity a lot larger, since it has become $O(|v||e|x)$. The difference between FOCS and FULPES lies only in the charging guarantees. For the algorithm this means; the initiation of the graph to be solved. The FULPES graph has an additional layer before reaching the sink. This extra layer adds extra vertices and edges, and needs to be solved independently. Thereby requiring the algorithm to solve multiple sub-digraphs per iteration, increasing the execution time.

III. Improving execution time

Improving the execution time of a program can be done in a few steps. Some of the more straightforward ones are porting the code to a faster language and using Compressed Sparse Row (CSR) format for storing graphs. These will be the first steps to be discussed in this section. After applying those, a few other optimizations are discussed, tested and added to further improve performance. Subjects that will be discussed are Multithreading, Memory locality and Algorithm changes.

A. Porting to C++

At the time of starting the project, the newest publicly available algorithm of FULPES was not fully functional, therefore the decision was made to improve the execution speed of the FOCS algorithm. Once the FULPES algorithm is available to be implemented with a shorter execution speed in mind, the FOCS implementation can be adapted to include charging guarantees. This effectively turns the FOCS implementation into FULPES.

The available code for FOCS was written in Python, this code utilizes the NetworkX library. Python is a high-level coding language which means that there is very little optimization capabilities in terms of improving execution times, therefore the whole FOCS code was ported to C++. In C++, there are many options for fine-tuning performance, particularly when it comes to memory management, loop execution, and compiler settings. While many performance tweaks require changes inside the code itself, compile options are applied externally in the compile command.
These compile commands can be used in C++ since the code is converted from the original code into an executable file, where the compiler sees all of the code and can selectively improve parts. In Python, the code is directly run from the coding file, meaning it performs a lot of checks whilst executing.

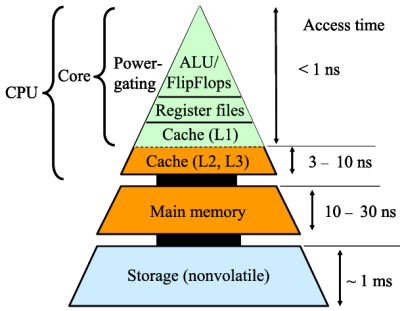

Fig. 2. Computer memory access pyramid including rough time estimates [9].

B. Memory access

Optimizing memory hierarchy is crucial for enhancing system performance and efficiency [7]. By improving cache utilization and therefore reducing the cache miss rate, execution times can be reduced. Figure 2 displays the different levels of memory including rough estimates of the time it takes to read data from these memory levels. Copying data to these different levels happens in blocks. This means that data which is located next to one another is copied to higher level storage at the same time. This can be done in C++ by pre-allocating memory. When memory is pre-allocated, it attempts to include all of the adjacent memory locations in the storage of that array. As a result, when pulling a block of memory into L1 Cache, it contains a larger portion of the needed data. Once this data is located in the L1 Cache, the memory access time decreases, increasing performance. The better memory locality increases not only the access time, however also decreases the cache miss rate. The miss rate is when a core attempts to find data in a cache layer, which that specific data is not stored in. Each cache miss results in a miss penalty, which can take up to 20 ns. Therefore, reducing the miss rate directly lowers the total execution time [8].

The original FOCS code utilizes strings of characters as vertex names, however this is generally slower than using integers since integers are stored as 4 Byte numbers. Whereas an array of ASCII characters, string variables, can be any length of bytes which is not always pre-defined. However, the variable length isn't the only disadvantage, string variables in C++ take up at least 32 Bytes as seen in our tests. Since it is not pre-defined, more checks and conversions need to happen when processing string variables. Since it takes up more storage, it thereby reduces the cache efficiency.

C. Algorithm improvements

During the execution of the FOCS algorithm, a very specific graph is made which always has the same shape. For this

reason, spending more time researching the fastest maximum flow algorithm for that specific shape could significantly improve the algorithm runtime. At the start Edmonds-Karp is used as a maximum flow algorithm due to the simplicity of the algorithm, however switching to the SAP algorithm can significantly increase the execution time of the FOCS algorithm. Researching these different maximum flow algorithms could be the key to improving execution time further.

D. Multithreading

The last discussed method of improving execution time is multithreading. When using multithreading, the workload is divided over multiple threads on the processor. For example when dividing the work-load over two cores, at most the execution time is almost halved. It does not entirely divide the execution time by the amount of cores, since the overhead of spawning and securing cores can take up a lot of time. This can even result in slowing down the execution. Securing data means that it is protected from reading whilst writing to that data slot. It also prevents multiple cores from writing to that register at the same time.
To make multithreading successful in the case of FOCS, a maximum flow algorithm should be chosen which supports parallelization. Edmonds-Karp and Dinitz's algorithm do not support parallelization because of their recursive nature. The next iteration of the algorithm is dependent on the last iteration's results. An algorithm which supports parallelization is the push-relabel algorithm. This algorithm sets heights to nodes and then looks per node how much flow can be sent to neighbours. The algorithm can parallelize pushing the flow and setting heights. Every thread starts at a node and attempts to push flow to all neighbours. This will result in a great speed-up, if the amount of vertices is much greater than the amount of nodes. Which is not the case for FOCS, however multithreading will still be tested.

IV. Experiments

For the experiments the Edmonds-Karp algorithm was implemented first, since it was necessary before an attempt at Shortest Augmenting Path could be made. The SAP algorithm used in the original FOCS implementation is from the Python NetworkX library. This implementation uses a fall-back to the Edmonds-Karp algorithm, and therefore needs a functional Edmonds-Karp algorithm. This means that all of the implementations in C++ use Edmonds-Karp as a maximum flow algorithm, except if stated otherwise. But first the experimental setup will be provided in order to achieve reproducibility.

A. Experimental setup

All tests were performed on the same computer. The processor used is a Ryzen 7 5800X, which has 64 kB of L1 cache per core and 512 kB of L2 cache per core and 32 MB of shared L3 cache [10]. All tests were repeated 500 times using loops, then the execution time was averaged over all of the tests. During the tests the computer was not being used for other active programs.

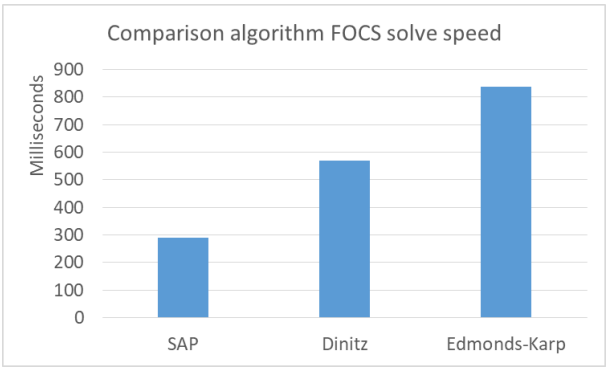

Fig. 3. Original FOCS Python implementation different maximum flow algorithm, FOCS execution time comparison. Execution time is averaged over 100 repetitions. All instance sizes set to 200, on a linear scale.

B. Base tests

Using the original Python implementation of the FOCS algorithm, a few different maximum flow algorithms can be compared for the graphs used in FOCS. This list of algorithms was the list as included in the Python NetworkX library. Figure 3 shows the 3 different algorithms on the x-axis and the execution time in milliseconds on the y-axis. As can be seen in the figure, from the tested algorithms Shortest Augmenting Path turns out to be the fastest for FOCS graphs. Therefore the SAP algorithm will be used as a benchmarking tool to test the relative speed-up.

All C++ code was made from the start onwards, where only inspiration was taken from online sources, no code was directly copied. This explains why the basic and CSR [11] steps, in figure 4, are slower than the original implementation in Python. The code was not optimized, it was purely made to achieve the correct result. Optimizations that were made to improve this unoptimized code, lied mostly in the addition of CSR. Since at first the edge array existed, however there was no edge pointer array. This meant it was looping over the entire array until it found the edge it was looking for. CSR made the look-up time a lot shorter. However, once CSR was implemented the C++ implementation was still slower than the original implementation. This can be seen in figure 4. Despite using a faster programming language, the lack of memory optimization and the use of a slower maximum flow algorithm still resulted in longer execution times.

C. Memory allocation tests

The aforementioned cache layout is one of the main reasons why memory locality and data representation were the next steps for optimization. These improvements resulted in a speed-up of 42 times. This makes execution times go from more than 1600 milliseconds to just below 40 milliseconds, as is visualised in figure 4. A key speed-up factor was in creating multiple value- and row pointer-arrays for different purposes. The improved memory locality meant that the cache was better used, reducing lookup times and more than halving the cache miss rate.
Another important improvement was switching from strings to integers for node identifiers. Integers take up less space

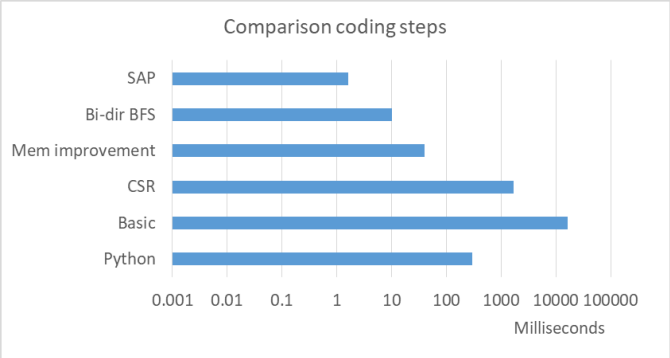

Fig. 4. Comparing different steps taken to improve execution time, all using the same test file, 200 instance size and 200 repetitions. Execution time is the average over the amount of repetitions. Plotted on a logarithmic scale. X-axis displays the average execution time in milliseconds. Y-axis shows the different steps taken where 'Python' is the original FOCS implementation.

in memory, allowing more data to fit in the L1 cache and reducing the frequency of cache evictions. Strings are stored as ASCII characters on standard computers, however in C++, std::string has an overhead. Storing a string consisting of 1 character takes up 32 Bytes. This can be tested using the 'sizeof' command in C++. Another advantage of integers is that less checks are required when using them, since they are always the same length.

For example, forward and reverse name maps were created specifically to speed-up Breadth-First Search. The neighbours of nodes were put into vectors, which were then organized in a vector where the index corresponds to the node ID. This way nodes are stored in such a way that chunks of memory often include multiple nodes. The vectors were ordered by node ID, in order to achieve easier access in the cache layers.

One of the concerns with this approach was the memory usage, since multiple large vectors could take up too much space. While this approach increased the number of large vectors, the total memory usage (excluding test data) remained within 1 to 2 MB for instance sizes of 200, depending on the stage of the FOCS algorithm. For reference, the original Python implementation used approximately 50 times more memory during execution. As a result, memory usage was not considered a bottleneck.

D. Multithreading

The push-relabel algorithm was tested on a standard graph created when solving FOCS for an instance size of 200. The test was done in a benchmarking tool as provided by D. Bart [12], as well as a benchmark for the Shortest Augmenting Path algorithm. For the created graphs, the push-relabel algorithm was always three orders of magnitude slower than the SAP algorithm. The tests were performed with combinations between 1 and 16 threads. For this reason no further attempts were made at implementing a different, parallelizable algorithm.

This, as discussed earlier, only yields a significant speed-up if the number of neighbours is high. However, in FOCS, the number of neighbours per node relative to the number of nodes is very low. On average, for an instance size of 200, there are 254 nodes with a total of approximately 5800 edges. This means that each node in the *j*-layer has approximately 28 neighbours in the *i*-layer. Therefore, parallelizing the execution of very little work results in a slower execution time due to the overhead.

Parallelizing does not only apply to the maximum flow algorithm, but can also be done on single loops. This was tested, but showed slower results due to the overhead of spawning threads and securing data, which outweighs the benefit when dealing with relatively small loop bodies.

E. Final results

The final code was then tested using a test data file. In this final code, all improvements discussed before are implemented, this means optimal memory locality, the SAP algorithm and more. Different instance sizes were compared in relative execution time to the original Python implementation. This is displayed in figure 5, with the execution time in logarithmic scale, measured in microseconds on the y-axis. The x-axis depicts the different instance sizes tested. The speed-up factor decreases from 185 times, at an instance size of 200 to 40 times with an instance size of 2000. This change is deducted from figure 6, which displays the speed-up amount on the y-axis and the instance size on the x-axis. This means that the improved code becomes relatively slower when instance size increases.

This decrease in efficiency is mainly due to the need for a fallback to the Edmonds-Karp algorithm for clearing up. When the instance size increases, so does the flow obtained from the fallback method. Although our SAP and Edmonds-Karp implementations are faster for smaller instance sizes they have not been thoroughly optimized. Whereas the Python implementation uses the highly optimized NetworkX library. This results in the SAP implementation from NetworkX becoming relatively faster for larger flow graphs.

In the end, the execution time for instance size 200 is, on average, 1.6 milliseconds.

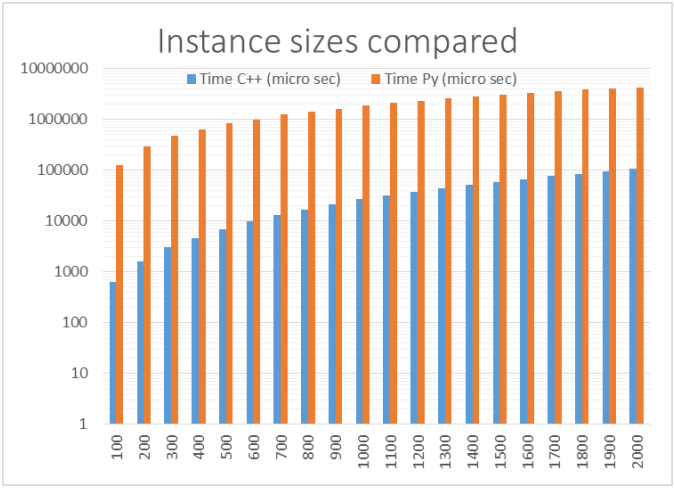

Fig. 5. Final implementations compared with different instance sizes, comparing the FOCS Execution time. FOCS execution time is the average over 500 repetitions of different instance groups.

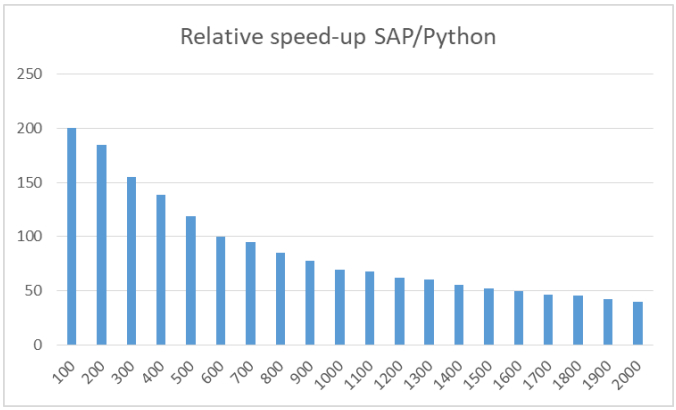

Fig. 6. Final implementation relative speed-up of SAP over Python. Comparing FOCS execution time.

V. Conclusion

The FOCS algorithm is accelerated more than a hundred times, from 300 ms in Python to well under 3 ms in an optimized C++ version for a 200 instance size. This feat was achieved by refining memory accesses, and implementing maximum flow algorithms based on the refined memory. Larger instance sizes do not reach the goal of 100x speed-up due to the inefficiency of the Edmonds-Karp fallback method, which along with the SAP implementation have not been thoroughly optimized. The next step would be to adjust the FOCS integration to include charging guarantees, making it into the FULPES algorithm. Other improvements could be made by finding better maximum flow algorithms, or optimize those used to achieve a greater speed-up for larger graphs. Although no faster alternative has been found so far for FOCS. This work enables practical, real-time EV charging management in parking lots.

References

[1] CNN, "Spain says april blackout was caused by grid failures and poor planning, not a cyberattack," 2025. Accessed: 2025-06-20.

[2] L. Winschermann, M. E. T. Gerards, A. Antoniadis, G. Hoogsteen, and J. Hurink, "Relating electric vehicle charging to speed scaling with job-specific speed limits," 2025.

[3] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *Journal of the Association for Computing Machinery*, vol. 35, no. 4, pp. 921–940, 1988.

[4] WsCube Tech, "Ford-fulkerson algorithm," 2025. Accessed: 2025-03-20.

[5] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, no. 2, pp. 248–264, 1972.

[6] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993. See pages 218–219 for the Shortest Augmenting Path algorithm time complexity.

[7] M. Vaithianathan, "Memory hierarchy optimization strategies for highperformance computing architectures," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 6, p. 24–35, Jan. 2025.

[8] M. Hugue, "Cache memory." https://www.cs.umd.edu/~meesh/cmsc311/cache/node4.html, 2002. Accessed: 2025-06-16.

[9] Spin-transfer torque magnetoresistive random-access memory technologies for normally off computing (invited), "Layered structure of computer systems: Typical access times for smartphone, personal computer, and high-performance computing systems." https://www.researchgate.net/figure/Layered-structure-of-computer-systems-Typical-access-times-for-sm fig1_263004188, 2025. Accessed: 2025-06-02.

[10] TechPowerUp CPU Database, ""AMD Ryzen 7 5800X Specs"." https://www.techpowerup.com/cpu-specs/ryzen-7-5800x.c2362, 2025. Accessed: 2025-06-19.

[11] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2 ed., 2003.

[12] D. Bart, "Maxflow benchmark." https://github.com/dbart-utw/max_flow_benchmark, 2025. Accessed: 2025-06-19.