Finding upper bound for tightening of individual constraints in flexible manufacturing systems

Angel Karchev
Supervisor(s): Eghonghon Eigbe, Neil Yorke-Smith
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Delays and complications in different schedules are a common and widely applicable issue in modern society. These problems, if severe enough can cause a preexisting schedule to become infeasible, thus creating additional problems with varying levels of severity. The focus of this work is to showcase a systematic method for modelling potential disturbances in the execution of a schedule, and to provide algorithms which can help analyze the maximum bounds in which each constraint of the schedule is allowed to change without making the schedule infeasible. Presented is a method that is proven to provide results for the entire schedule with cubic complexity with regard to the number of jobs, as well as a heuristic method with results that approximate the correct ones, but runs considerably faster.

1 Introduction

Scheduling problems are ones of great importance for society and have been for thousands of years. Currently, solutions to different varieties of scheduling are being utilized in a wide range of areas [12] [15] [14]. While such problems can be fairly trivial for smaller sets of actions and constraints (for instance if an individual would only have to purchase groceries then use them to prepare a meal at any arbitrary point during the day, the solution is quite easy to find), the time it takes to find a solution increases drastically with an increase of conditions and constraints (due to most scheduling problems being NP-complete[18]). As such, due to the lack of a catch-all solution that runs in polynomial time, various techniques that take into account the intricacies of specific scheduling variations have been devised to generate feasible schedules based on sets of operations and constraints. Presented in this paper is one such type of problem, as well as a method for analysis of solutions to the problem and the effect that changing constraints can have in them.

Problem background

The aim of this research is to gain an understanding of how scheduling for an industrial printer is carried out, and how the constraints of a particular schedule affect performance. In particular, this research focuses on an abstracted, more theoretical version of a scheduling problem, making its results more widely applicable. A more generalized abstracted form of the goal of the research group as a whole is to quantify the extent of change a given schedule can withstand before requiring revision.

Research Question

The research question assigned in particular is : "How much do constraints have to change individually to cause a schedule to become infeasible?".

This question of great importance to scheduling for an industrial printer, as well as numerous other tasks that can be reduced to a scheduling problem. Due to a wide array of possible issues or changes in an environment depending on the context, scheduling problems often change with respect to the conditions under which a given schedule holds. Planes can be delayed due to unexpected weather conditions[2], projects development can fall behind due to poor assumptions in their underlying planning. Constraints of problems are in a constant flux. As such, identifying the extent to which constraints can change, while still

having a schedule be effective is a topic of great interest.
This question can be broken down into three sub-questions:

- Which constraints have great (if any) impact on a schedule's performance?
- How much does each individual constraint affect performance?
- What would an algorithm that determines this for an entire schedule, as opposed to a single constraint look like?

2 Related Work

The bulk of the academic work already done on flexible manufacturing systems has focused on various algorithms and heuristics for creating schedules [6] [11] [10], compliant with the various constraints of a given scheduling problem or a set of scheduling problems. Oftentimes these papers aim to create, or approximate the "optimal" schedule with regard to some predefined metric (most commonly maximum lateness in execution time of the entire schedule [17] or minimizing total completion time [8]). Analysis of already existing schedules has received considerably less attention however, thus warranting this paper. Fortunately, a lot of work has been put toward creating formal definitions of a flexible manufacturing system. This work in particular will stick closely to the setup detailed in [19]. Like in that work, the mapping of operations to machines is assumed, but the flow of operations, and the sequence of jobs is fixed, however, instead of assuming that a product is processed exactly twice by exactly one of the machines, we take the considerably broader problem of having products be processed by an arbitrary amounts of machines an arbitrary amount of times. This way, a wider variety of models and schedules can be analyzed, though it is worth noting that one of the methods described works considerably better with the more simplified setup. Additionally, like in the work referenced [19], due dates are relative to the setup times, and missing any one of them in a given schedule makes the schedule considered infeasible.

3 Problem Description

The problem solved in this paper relates to flexible manufacturing systems (FMS). An FMS processes products according to a particular flow through processing stations [19]. A more complete definition of such a system is given in [19]. In particular, this paper concerns re-entrant FMSs, where products can pass through the same processing stations twice. Such FMSs can be modelled as a re-entrant flow-shop according to the procedure described by van Pinxten, Waqas and Geilen . A re-entrant flow shop with sequence-dependent set-up times and relative due dates, is a tuple $(M, J, r, O, \phi, P, S, SS, D)$ [19].

The objective of the strategies devised for this paper was to create a comprehensive methodology to determine the maximum change in every one of the constraints for which a given, previously feasible schedule can still remain feasible. In turn, this way any change more extreme than the one found can be assumed to make the given schedule infeasible. In terms of real life applications, a change in a single constraint can be a delay caused by a part that takes a certain additional amount of time to repair, a change in a deadline, and many more. Such a change can be represented as an increase of the weight of a single edge in the graph representation of a flow-shop model schedule graph. Since such a schedule is feasible

if and only if there are no positive cycles in its graph representation, the problem can be reduced to finding the individual largest increase of the weight of each edge, for which any further increase, will make the graph contain at least one positive cycle.

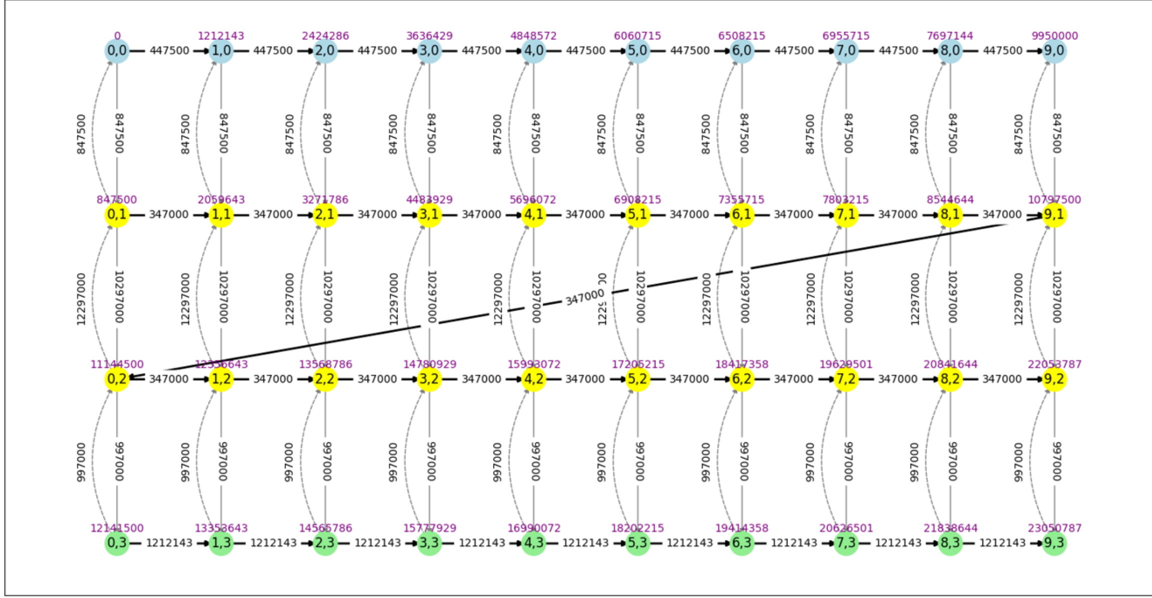


Figure 1: Image representation of the graph of a flow-shop model schedule

4 Algorithm for determining maximum tightening of single constraints

As discussed earlier, the graph representation of a schedule containing any positive cycle indicates that the schedule it represents is infeasible given the constraints under which it is to operate. Therefore, if the change of the value of a single constraint (represented by the weight of an edge) causes at least one positive cycle to appear in the graph, then it can be concluded that this change in constraints has made the schedule infeasible, given that said schedule was feasible prior to the change. Since changing the weight of a single edge in a graph does not change the structure of the graph, this implies that at least one of the negative cycles which exist in the graph needs to turn positive. Since the total weight of a cycle is equal to the sum of all edges it contains[4], this means that in order for a cycle containing an edge e_i to become positive only by changing $w(e_i)$, the change in edge weight $\Delta w(e)$ needs to satisfy $\Delta w(e) > \sum_{n=1}^{c} w(e_n)$, where c is the number of edges in the cycle and e_n is the nth edge in that cycle. Since having at least one negative cycle turn positive causes a schedule to become infeasible, if, for a particular value of $\Delta w(e)$, the above inequality holds for any cycle containing e, then the schedule would become infeasible for that value of $\Delta w(e)$.

Base algorithm

Showcased here is an algorithm that, given a weighed, directed graph G, with no positive cycles, as well as an edge in that graph e, determines the minimal amount $\Delta w(e)$, for which increasing $w(e)$ by any number greater than $\Delta w(e)$ would cause the graph to contain a positive cycle.

The algorithm makes use of Bellman-Ford's algorithm for finding the shortest path in a graph[20]. Bellman-Ford's algorithm requires a graph with no negative cycles in order to function. Since the G contains no positive cycles, inverting the values of all edge weights would cause it to contain no negative cycles.

Lemma 4.1. *Given a feasible schedule, the maximum value by which a single constraint represented by the edge e can be increased can be determined using reducing the schedule into a graph and applying the following algorithm:*

Algorithm 1 Algorithm for finding maximum tightening of an individual constraint

---

Require: g is a graph with no positive cycles and an edge e from node s to node d in g
Ensure: the value returned is the maximum increase of the edge e

$i \leftarrow 0$
while $i < |E|$ **do**
 $w_i \leftarrow -w_i$
 $i \leftarrow i + 1$
end while
$bf \leftarrow Bellman_Ford(d)$
$dist \leftarrow bf[s]$
$delta \leftarrow dist + w(e)$
return delta

Proof. This subsection contains a proof that the algorithm described above correctly determines $\Delta w(e)$. d will refer to the destination of e and s will refer to the source of e. The length of the path from d to s calculated by Bellman-Ford's algorithm will be referred to as L

First, we will prove that every cycle containing e can be represented as the edge e and a path of arbitrary length from d to s. If a cycle contains e, then there must be a way to traverse e an arbitrary amount of times after it has been traversed once. In order to reach a state from which e is directly accessible, an agent must be able to reach s. Additionally, after taking e, a hypothetical agent traversing the graph would always end up at the destination d. Therefore, in order to complete a cycle containing e, an agent would need to traverse a path from its destination to its source.

In the algorithm mentioned above, Bellman-Ford's algorithm is used to find the shortest amongst all paths from d to s, with the values of all edges inverted. With the inverted value of $w(e)$ added to complete the cycle, the result that this algorithm returns is guaranteed to be positive due to the assumption that there are no negative cycles in the initial graph. It is also the total weight of the shortest cycle in the inverted graph containing e, as the distance of the path between d and s is minimized, as Bellman-Ford's algorithm is proven to determine the shortest path[], $w(e)$ is constant, and every cycle containing e being a

combination of e and a path from d to s. This means that this algorithm returns the total weight of the longest cycle in the original graph (since all cycles there are negative, it also returns the weight of the cycle with lowest absolute value).

Intuitively, it can be seen that increasing the $w(e)$ by the computed value $\Delta_1 w(e)$, the total weight of the cycle will become 0, as $\Delta_1 w(e)$ is the inverse value of the total weight. Thus, increasing e by any arbitrary amount more than $\Delta_1 w(e)$ will cause the graph to contain a positive cycle. In order to prove that $\Delta_1 w(e)$ is indeed the minimum possible value of $\Delta w(e)$, a proof by contradiction will be used.

Assume that $\Delta_2 w(e)$ exists, where $\Delta_2 w(e) < \Delta_1 w(e)$ and where incrementing $w(e)$ by any value greater than $\Delta_2 w(e)$, a positive cycle will appear in G. Then there must be a path from d to s with length $L1$, where:

$$L1 + w(e) + \Delta_2 w(e) >= 0 \tag{1}$$

However, we know the following:

$$L > L1 \text{ (Bellman-Ford's algorithm)}$$
$$\Delta_1 w(e) > \Delta_2 w(e) \text{ (given from assumption)}$$

Therefore:

$$L1 + w(e) + \Delta_2 w(e) < L + w(e) + \Delta_1 w(e) \tag{2}$$

$$L + w(e) + \Delta_1 w(e) = 0 \tag{3}$$

As $\Delta_1 w(e)$ is the inverse value of $L + w(e)$, thus $L1 + w(e) + \Delta_2 w(e) < 0$ creating a contradiction.

Therefore $\Delta_1 w(e)$ is the minimal amount, for which increasing $w(e)$ by any number greater than $\Delta_1 w(e)$ would cause the graph to contain a positive cycle.

□

Complexity

Since all operations described in the algorithms are done sequentially and are not repeated, the total complexity of the algorithm is equal to the highest complexity among all operations[1].

The complexity of inverting all edges in the graph is linear with respect to the number of edges, so $O(|E|)$.

The complexity of running Bellman-Ford is $O(|V||E|)$ in the worst case. [20].

Finally, the complexity of adding $w(e) + D(s)$ then returning the result is constant, so $O(1)$

Out of all the terms, $O(|V||E|)$ is clearly the biggest, thus the complexity of the algorithm described above is equal to that of Bellman-Ford.

4.1 Running time for entire graph

To run the algorithm for every constraint, one can simply invert all edges in the graph once, then repeat the entire procedure after that for every edge e_i in the graph. This method has a complexity equal to that of Bellman-Ford multiplied by a factor of $|E|$. This factor can instead be substituted by a factor of $|V|$ by maintaining a matrix of the distances from each node to each other node and calculating the distances starting from each node using Bellman-Ford. This way, instead of Bellman-Ford being run once for each edge in the graph, it can be run once for each node. Typically, nodes possess considerably more edges than they do nodes, up to $|V|^2$ [9]. This optimization comes at the cost of increased memory usage however, as instead of having to track the distance from one node to each other node, the program would have to keep track of the distances between each pair of nodes in the graph. Additionally, from the problem description, it can be determined that each node can have at most four outgoing edges (one setup time edge, one due date edge, one job edge and one schedule edge). Thus the following holds:

$$|E| <= 4 * |V| \tag{4}$$

This means that time complexity-wise in big-Oh notation[1], |E| is equivalent to |V|.

4.2 Potential improvement of algorithm

Since the problem of finding the cycle with highest total weight (thus lowest absolute value of that total weight, since every cycle is negative initially) has essentially be reduced to a shortest path problem, the complexity of the algorithm to solve it is largely determined by the algorithm for finding the shortest path. Dijkstra's algorithm was considered, due to its significantly lower complexity of $|E| + |V| * log(|V|)$ [7], however was concluded to be inapplicable due to requiring the graph to contain no negative edges. Running a shortest path algorithm on the inverted graph will always cause it to have some amount of negative edges, as all job edges, setup time edges and schedule edges are positive.

The Floyd-Warshall algorithm [16] is an algorithm that finds the distance between each pair of nodes in a given graph with complexity $O(|V|^3)$. For an arbitrary graph, this would be an improvement complexity-wise to the $O(|V|^2 * |E|)$ of the Bellman-Ford solution, but since the number of edges is proven to be at most a constant factor of the number of nodes, with that constant factor being smaller than 4, the two complexities are effectively equivalent in terms of big-Oh notation.

A faster, but possibly lacking complete accuracy heuristic approach is also possible. This heuristic assumes that the weights of job edges, setup edges and relative due date edges are similar across the graph. If these assumptions hold, it can also be assumed that for vertex $v1$ and vertex $v2$, both representing job of the same machine, the overall total weight of a route between $v1$ and $v2$ is, on average lower when going through edges of a different machine, due to the sum of a setup time edge and its corresponding relative due date edge is always negative (as otherwise a negative cycle would exist), as well as the previous assumption about the similarity of edge weights. In order to provide a faster runtime, the algorithm described in this subsection assumes that a path between two nodes corresponding to jobs of the same machines, sufficiently close to the shortest path between those two nodes in the entire graph can be found by using only edges where both nodes of the edge correspond to

jobs done by that machine. The algorithm runs the same way the main approach does, two exceptions:

- For edges between two nodes corresponding to different machines, the heuristic approach immediately returns $|S_i + D_i|$, where S_i is the weight of the setup time edge between the two nodes, and D_i is the weight of the due date edge.
- For edges between two nodes corresponding to the same machine, the usual approach is taken, with the difference that instead of running Bellman-Ford for each node in the entire graph and each edge in the entire graph, Bellman-Ford is only run for all nodes corresponding to jobs of the machine and all edges between them.

This reduces the worst case complexity to $O(|V| * |V'| * |E'|)$, where $|V'|$ and $|E'|$ correspond to the number of vertices in corresponding to the machine with the most jobs assigned to it, and the number of edges between those vertices respectively. A downside of this approach is that results from it are not guaranteed to be entirely accurate, as there are cases which can be described which make it very subpar. Particularly, when the assumptions stated do not hold. For instance, if the weights of all edges within a particular machine are considerably higher than the weights of edges within the next one, it may be beneficial to take an edge to the machine with edges with higher weight, offsetting the "cost" of taking a due edge.

5 Experimental Setup and Results

The data set used for the experiments consisted of multiple benchmark problems, as well as a schedule for each problem. The benchmark problems each consisted of three machines - one taking care of two operations and the other two only taking care of one each. There were two main sets of problems - ones with 100 jobs and ones with 500. There was also a smaller set of test problems, which were easier to visualize and debug on. Each schedule corresponding to a problem was generated using the Gurobi solver [5] with the solver set to minimize latest finishing time. All numbers in the data were taken from a model for an industrial printer, and were provided by Canon [3]. Provided in total were 450 schedules with 1f00 jobs and 450 schedules with 500 jobs.

5.1 Metrics used

In order to evaluate an algorithm's effectiveness, three metrics were used:

- Factor of overestimation - A metric used to judge algorithms based on the closeness of their results to those of the proven to be fully correct base algorithm. Calculated by using the average for running the algorithm on the 100-job schedules and the 500-job schedules separately. Since all algorithms described in this paper either overestimate the optimal result or return it, for a single constraint edge, this metric is defined as:

$$(returned_result - optimal_result)/optimal_result + 1 \tag{5}$$

  Here, a score of 1 would mean that the returned result exactly matches the optimal one (thereby overestimating the optimal result by a factor of 1), whereas a higher score would mean that the result is less close to the optimum.

- Average time spent - This metric consists of two values - The average amount of time spent (in seconds) for running the algorithm on the 100-job schedules and the 500-job schedules. The standard deviation for the running times across the 500-job schedules is also provided.

5.2 Machine specifications and properties of the testing code

Provided in this subsection is information regarding the specifics of the environment in which the experiment was conducted. They can be used as a point of reference when attempting to reproduce the results of this work.

5.2.1 Hardware

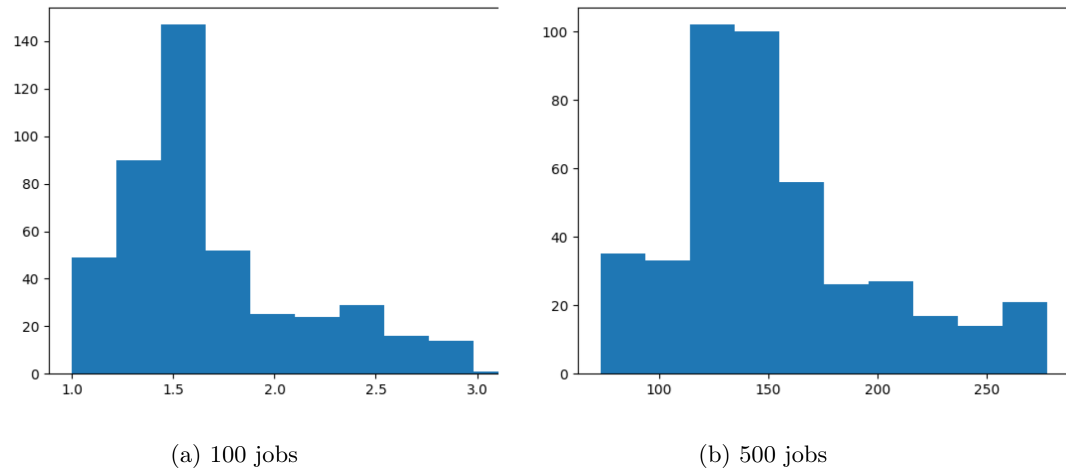
- CPU: Intel Core i7-8750H @ 2.20GHz
- RAM: 16 GB

5.2.2 Software

- Programming language: Python 3.9
- implementation of Bellman-Ford: Networkx library [13]

5.3 Performance of base algorithm

- Factor of overestimation: 1(100 jobs), 1 (500 jobs)
- Average time spent: 1.68s(100 jobs) 153.41s (500 jobs)
- stdev of time spent: 45.48s (500 jobs)



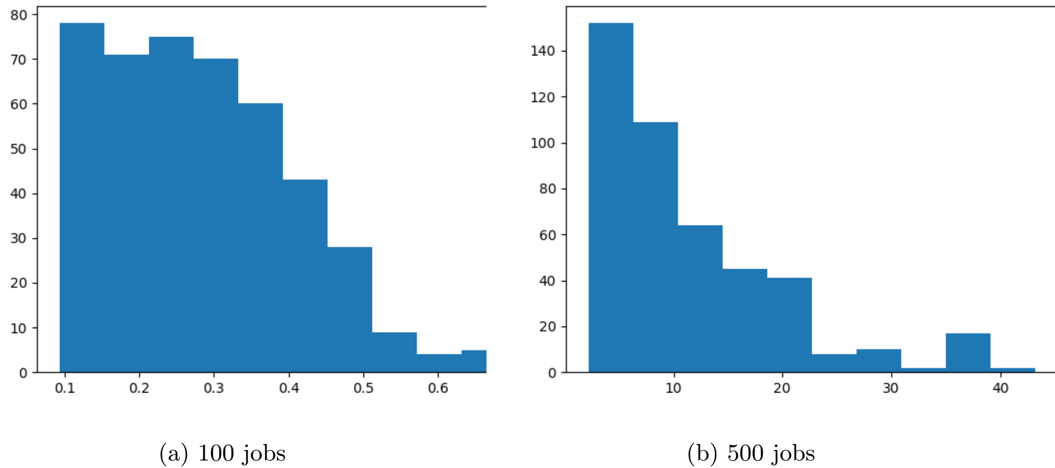
(a) 100 jobs

(b) 500 jobs

Figure 2: Distribution of time taken by the optimal solution algorithm to calculate optimal $\delta w(e)$ for every node in the graph (in seconds)

5.4 Performance of heuristic algorithm

- Average factor of overestimation: 3.02(100 jobs), 2.06 (500 jobs)
- Median factor of overestimation: 2.105 (500 jobs)
- Average time spent: 0.28s(100 jobs) 11.24s (500 jobs)
- stdev of time spent: 8.18s (500 jobs)



(a) 100 jobs

(b) 500 jobs

Figure 3: Distribution of time taken by the heuristic algorithm to calculate optimal $\delta w(e)$ for every node in the graph (in seconds)

6 Discussion

As showcased in the previous section, the time it takes to complete an analysis of a schedule using the base approach increases drastically with an increase of jobs. More precisely, going from 100 jobs to 500, caused an average computation time increase by a factor of more than 90. As such it can be concluded that the base algorithm would not be a suitable choice for computing significantly larger varieties of the problem. In comparison, due to its approximate nature, computation times for the heuristic approach increased by the still large, but significantly less so factor of 40. This was coupled with overall significantly better performance time-wise. The obvious downside was the decrease in accuracy with regard to finding the optimal bound. A median factor of overestimation of 2.105 means that for more than half of the 500-job schedules, the average value found for $\delta w(e)$ is more than double its true value. Additionally, since the values found using the heuristic are always equal to or greater than the true values of maximum changes in constraints, this means that an actual change of constraint equal to a value computed using the heuristic is more likely than not to make the schedule infeasible. A positive way to make use of the computed value could be to run a small number of schedules and/or individual constraints through the optimal algorithm then to determine the average factor of overestimation and then to counteract

the increase with a division by some predetermined multiple of the factor of overestimation. This way, a sort of confidence interval for the true value of $\delta w(e)$ could be created.
Finally, it is worth noting that the dataset used had large differences of the values of setup edges between different machines. As detailed in the section describing the heuristic approach, this type of setup significantly deteriorates its performance with regard to accuracy. It is possible (although mostly author speculation) that the method works considerably better with datasets with more consistent setup times between machines.

7 Responsible Research

The ethical issues concerning this work mostly revolve around the ways in which results produced from the methods described are used. An infeasible schedule can have serious ramifications for thousands if not millions around the world if on a large enough scale. The base algorithm finds the exact bounds at which a schedule becomes infeasible, so some buffer should be considered, in order to never actually reach near points of infeasibility. This holds doubly so if an entity decides to use the heuristic solution, as that one specifically always overestimates how much constraints can be changed at a non-constant rate. Furthermore, it is worth noting that this study performed testing on a limited set of data with very specific characteristics. As such, if one decides to use this research to achieve a related goal, they must consider the context in which the methods detailed here will be used.
To make this research reproducible, theoretical implementations of the algorithms used for experiments have been explained in great detail. Furthermore, the test problems and schedules used in experiments are uploaded in the repository containing this paper. Finally, specifications of the hardware used to run experiments as well as the programming language and library used for Bellman-Ford have been detailed.

8 Conclusion and future work

This paper studied the problem of analyzing schedules for flexible manufacturing systems with regard to constraints. A method to find maximum tightening of individual constraints was introduced. The algorithm described has the advantage of being relatively easy to implement. Additionally, this paper presented a significantly faster approximation of it. The base approach was proven to function correctly, but its $O(|V|^2 * |E|)$ complexity when run for every constraint in a problem proved to be an issue, due to the fast growth in required execution time. Since the bottleneck of the current algorithm is the shortest path algorithm used, the way research into this topic can be continued could be to find a more appropriate shortest path algorithm given the unique set of constraints the model graph possesses. Another future direction would be an improved heuristic approach which sacrifices complete accuracy in exchange for shorter execution time.

References

[1] Sammie Bae. Big-o notation: An introduction to understanding and implementing core data structure and algorithm fundamentals. pages 1–11, 01 2019.

[2] Stefan Borsky and Christian Unterberger. Bad weather and flight delays: The impact of sudden and slow onset weather events. 05 2018.

[3] Canon. Canon website, 2022.

[4] Nitin Chandrachoodan, Shuvra Bhattacharyya, and K. J. Ray Liu. Adaptive negative cycle detection in dynamic graphs. pages 163–166, 01 2001.

[5] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual. `https://www.gurobi.com`, 2022. Accessed: 19-05-2022.

[6] Bo Huang, Yu Sun, Ya-Min Sun, and Chun-Xia Zhao. A hybrid heuristic search algorithm for scheduling fms based on petri net model. *The International Journal of Advanced Manufacturing Technology*, 48:925–933, 06 2010.

[7] Adeel Javaid. Understanding dijkstra algorithm. *SSRN Electronic Journal*, 01 2013.

[8] Caixia Jing, Wanzhen Huang, and Guochun Tang. Minimizing total completion time for re-entrant flow shop scheduling problems. *Theoretical Computer Science*, 412(48):6712–6719, 2011.

[9] Chunhui Lai. On the number of edges in some graphs. *Discrete Applied Mathematics*, 283:751â755, 09 2020.

[10] Anita Lee-Post and Chun Hung Cheng. Metaplanning in fms scheduling. *International Journal of Operations Production Management*, 16:12–24, 06 1996.

[11] JianChao Luo, MengChu Zhou, and Jun-Qiang Wang. Abb: An anytime branch and bound algorithm for scheduling of deadlock-prone flexible manufacturing systems. *IEEE Transactions on Automation Science and Engineering*, PP:1–11, 11 2020.

[12] Fredrik Monsuur, Marcus Enoch, Mohammed Quddus, and Stuart Meek. Modelling the impact of rail delays on passenger satisfaction. *Transportation Research Part A: Policy and Practice*, 152:19–35, 10 2021.

[13] NetworkX library. NetworkX documentation. `https://networkx.org`, 2022. Accessed: 19-05-2022.

[14] Lothar Pantel and Lars Wolf. On the impact of delay on real-time multiplayer games. pages 23–29, 01 2002.

[15] Xiaosong Peng and Guanyi Lu. Exploring the impact of delivery performance on customer transaction volume and unit price: Evidence from an assembly manufacturing supply chain. *Production and Operations Management*, 26, 01 2017.

[16] Ramadiani Ramadiani, D Bukhori, Azainil Azainil, and N Dengen. Floyd-warshall algorithm to determine the shortest path based on android. *IOP Conference Series: Earth and Environmental Science*, 144:012019, 04 2018.

[17] J. Sun. A genetic algorithm for a re-entrant job-shop scheduling problem with sequence-dependent setup times. *Engineering Optimization - ENG OPTIMIZ*, 41:505–520, 06 2009.

[18] J.D. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[19] Joost van Pinxten, Umar Waqas, Marc Geilen, Twan Basten, and Lou Somers. Online scheduling of 2-re-entrant flexible manufacturing systems. *ACM Transactions on Embedded Computing Systems*, 16:1–20, 09 2017.

[20] David Walden. The bellman-ford algorithm and "distributed bellman-ford. 01 2008.