Finding critical edges in schedules for re-entrant manufacturing machines

Amir Shimoni - van Delft
Supervisor(s): Eghonghon Eigbe, dr. Neil Yorke-Smith
EEMCS, Delft University of Technology, The Netherlands
20-6-2022

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

Critical constraints in re-entrant flexible manufacturing systems(FMSs) schedules are those constraints that for some change to their weight (and only the weight), could make the sequence of operation in the schedule infeasible. This paper describes how to find critical constraints by representing the benchmark as a graph and finding its strongly connected components and by exploiting the properties of FMSs. We also find we can force the schedule to have non-participating constraints at some points in the sequence. Finally, we see two possible relationships between critical constraints and describe how these can be found. We conclude that the points in the sequence where non-critical edges lay, have infinite temporal flexibility.

1 Introduction

Many different measures of flexibility exist for Flexible Manufacturing Systems(FMSs) [1]. In this paper, rather than coming up with another flexibility measure, we will try to assess the flexibility of constraints over a schedule themselves by their potential to break the schedule's sequence. We will distinguish between those constraints with the potential to break a schedule (participating/ critical constraints) to those that do not. By this division, we learn about points of infinite temporal flexibility in the schedule, how these can be found, and how we can force them into a schedule. These points of infinite flexibility should help schedules be more resilient to unexpected occurrences in the schedule and the production process.

In this paper we try to answer the following questions:

- How can we use and alter known algorithms to isolate what constraints could possibly cause a schedule to become infeasible and which do not participate at all
- How can we find relationships between constraints that might have an accumulating contribution to making a schedule infeasible
- How can we force certain constraints to be non-participating

The main contribution found in this paper is the methods to find critical and non-participating constraints. We describe under what conditions such constraints are formed and what measures can be taken to force them into a schedule. We also describe a measure for strong dependency of constraints and present an algorithm to find such dependency.

This paper is structured into eight parts. Section 2 elaborates on the problem and describes it in more detail. Section 3 discusses related work done by others. In section 4 we present and explain the methodology and the solution approach for solving the problem presented in the paper. Section 5 goes over the results produced in this paper and analyses them. Section 6 goes into responsible research aspects of our work. Section 7 will go over opportunities for future work. Finally, section 8 highlights the conclusions we came up with during our research.

2 Problem description

Potential to break a schedule has to do with the pair of operations a constraint is applied to. We are looking for constraints that can break a schedule by changing their weight, but the participating operation must remain the same. Our goal is to come up with an algorithm to find those constraints quickly and efficiently. Moreover, we are looking for ways to create a schedule sequence such that we force some non-participating constraints in chosen points in the sequence. Finally, we want to find relationships between critical constraints that indicate how a couple of constraints are tightly bound or if their relationship is unlikely to have a joined contribution to breaking a schedule.

In this paper, we represent a schedule in the form of a graph as described in [2] or in figure 1, where every node is an operation and an edge from nodes a to b with weight w implies b takes place at least w units of time after a. This representation makes some algorithms accessible in order to solve the problem, as we are going to see in chapter 4.

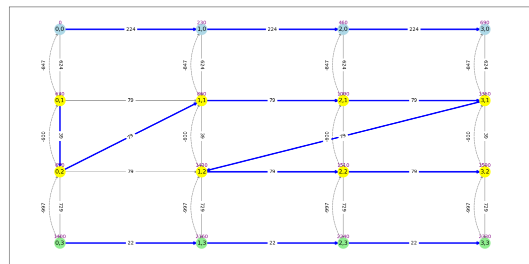

Figure 1: An example of a schedule. Edges represent time constraint between pairs of operation(positive edges - upper bound constraint, dashed/negative edges - minimum bound constraints). Blue edges represent the sequence of operations. A larger picture of this figure can found in appendix B

3 Related work

Work has been already been done to define different types of flexibility for FMSs, one of those is temporal flexibility[1], which is especially relevant to us as we show how infinite temporal flexibility in schedules can be achieved in this paper. The way schedules are represented as graphs in this paper follows the representation described in [2] with due edges(dashed) always having a negative value and not being flipped in direction.

It has also been shown already that following this graph representation, a benchmark only has a valid assignment if no positive cycle exists[2; 3], which comes in handy for determining whether a schedule is valid and how it might become infeasible.

In this paper, several solution approaches are presented that involve graph algorithms suggested by earlier papers. The first one attempts to find all cycles in a graph. While we use the algorithm by Johnson to do that [4] other algorithm for the same goal exist [5; 6]. Another approach uses the Bellman-Ford algorithm to detect positive/negative cycles [3]. Finally, we consider algorithms for finding strongly

connected components(SCCs) such as those mentioned in [7; 8]

4 Solution approach

In order to solve the problem at hand, we are going to first use and adapt existing graph algorithms to get a solution. The reason we chose this approach is that while the problem can be completely described as a graph, without ignoring any given information or creating ambiguity [2], there exist many algorithms to analyze graphs that we suspected could be useful for this problem. More specifically for representing a schedule as a graph, we can show that for a schedule to be infeasible, it must have a positive cycle [2]. Critical constraints are thus depicted as cyclic edges, acyclic edges are not able to cause the schedule to become infeasible. We are going to show that it is possible to come up with an algorithm to find cyclic and acyclic edges, and thus use this algorithm to isolate critical constraints. We will then try to find those critical constraints by exploiting the properties of FMSs, consider how we can manipulate a schedule to have non-critical constraints, and examine the relationships between critical constraints

4.1 Finding critical edges using graph algorithms

Looking back at my research question "How can we use and alter known algorithms to isolate what constraints could possibly cause a schedule to become infeasible and which do not participate at all?", it is now clear that we should be looking for algorithms for finding cyclic and acyclic edges or algorithms that can be altered to return those edges. To do this, we are going to attempt three different approaches.

Finding all cycles

The first method we used for finding cyclic edges is to find all cycles in the graph. For this, we used the algorithm by Johnson [4] (although other algorithms for this purpose exist). A nice advantage of this approach is that it also reports the cycles themselves, and not only the edges participating in any cycle. While it is not required for the research question, it is useful for some things beyond our research such as assessing schedule slack.

The problem with this approach is that it is quite complex. Running this has a complexity of $O((V+E)*(C+1))$, where V is the number of nodes, E is the number of edges, and C is the number of cycles. Since C is exponential in terms of E [9] we get a non-polynomial complexity.

Multi Bellman-Ford

Another way to find cyclic edges utilizes the Bellman-Ford algorithm. While by itself, the Bellman-Ford algorithm can not detect (a)cyclic edges, it can be used to detect positive or negative cycles [3]. We used this property and changed each edge individually, setting its value to infinity(in practice this was a very high value), then we ran Bellman-Ford on the adapted graph. Since the edge is set to positive infinity, if it is part of a cycle, the cycle must have a positive value, otherwise, that would mean the edge does not participate in any cycle at all. Bellman-Ford has a worst-case complexity of $O(V * E)$ [3], running it for every edge gives us a worst-case complexity of O($V * E^2$). One advantage of this approach is that it can be parallelized as each execution of the Bellman-Ford algorithm is independent of the others. With this approach, we showed that our problem is solvable in polynomial time. Pseudo-code for this algorithm can be found in A.2.

Using an algorithm for finding strongly connected components

Finally, to come up with the most time-efficient algorithm presented in this subsection, we consider the relationship between (a)cyclic edges to strongly connected components(SCCs). We tried to solve the problems we faced using DFS combined with all sorts of approaches. One of them was searching the graph while keeping track of the search tree, but it failed in the case where one cycle "cuts" another(by cut we mean, one cycle is a result of removing nodes from the other. See 2 for illustration). We suspected we had to reduce cycles to single components to overcome this. What we then tried is as such: upon finding a cycle we remove the nodes of the cycle and the edges that participate in the cycle, replacing them with one node and keeping the edges going in and out of the cycle. This approach would eventually leave us with the acyclic edges connecting the remaining nodes. We then also came to realize that each node left represents a strongly connected component(SCC) in the graph and that this is essentially what our algorithm does. Fortunately, algorithms for finding SCCs already exist.

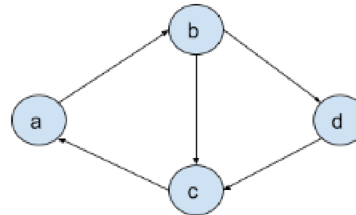

Figure 2: cycle a-b-c "cuts" cycle a-b-d-c

We will now clarify what (a)cyclic edges have to do with SCCs. Inside an SCC, every node is reachable from every other node. For any edge we take going from a to b (where a and b are part of the same SCC) there will be a path from b to a that completes a cycle. Since a and b are arbitrary nodes, we conclude all edges connecting two nodes of the same SCC are cyclic. On the contrary, if there exists an edge going from node a1 in one SCC to a2 in another SCC it implies that every node in the second SCC is reachable from any node in the first SCC(as we can get from any node in the first SCC to a1, then onto a2, and from a2 to any node in the second SCC). For this edge to be cyclic there must be a path going back from the second component to the first (so we can go back from a2 to a1). If such a path exists, we can use the same process as before to conclude that any node in the first SCC should be reachable from any node in the second one, just going the other way around. We would then get that every node in both SCCs is reachable from any node in any of the two SCCs, which means there would only be one SCC. This is a contradiction, and therefore edges connecting different

SCCs are acyclic.

Realizing this relationship between cyclic edges and SCCs, we can now use an algorithm for finding SCCs to divide the graph into its strongly connected components, and then extract the cyclic edges.

Theorem 1

1. *When modeled as a graph, critical FMS constraints will be represented as cyclic edges, while those that are non-participating will be represented as acyclic.*
2. *Edges interconnecting different strongly connected components(SCCs) are acyclic, while those that connect nodes within the same SCC are cyclic*

From 1 and 2 we conclude that critical constraints in FMS can be found by running an algorithm for finding SCCs and then comparing for each constraint(edge) the SCCs of the nodes they are connecting.

Some of these algorithms, such as the Kosaraju-Sharir algorithm, have a worst-case time complexity of $O(V+E)$ [7]. Separating the edges into cyclic and acyclic after finding the SCCs is just a matter of comparing the SCC of the source of the edge to the SCC of its destination. We thus conclude that this approach dominates the other solutions presented here so far in terms of time complexity. For finding SCCs we used the Kosaraju's-Sharir algorithm implementation given by the NetworkX library [10] and the algorithm in A.1 to extract the edges given the SCCs.

4.2 Detecting non-participating constraints by their characteristics in FMSs.

Having looked at the results for the previously mentioned algorithms, we came to realise that a non-participating constrain will always be one that is applied from the last operation on the re-entrant machine on some job J to the first operation on the re-entrant machine for the following job J+1 or a constraint that is applied to the same pair of jobs as the described constraint(this is explained later in 4.3). So far we tried to apply graph algorithms to find (a)cyclic edges. In this subsection, we use the previously mentioned property to find (non)critical constraints.

Iterating all edges

The first most obvious solution is to iterate all edges, marking edges as described before as non-critical as well as any edges connecting the same two jobs. Iterating all edges can be done in linear time complexity. let p be the number of operations per job in our benchmark, for each edge where we find a checkpoint we need to add additional p edges. Since we can have at most J checkpoints and $J * p < E$ the complexity for this approach is $O(E)$. Pseudocode for this approach can be found in A.4

Iterating sequence edges

Since the defining factor for (non)critical constraints are the sequence edges, we can use the same approach as before, looping only over the sequence constraints. That is of course under the condition that we have a list of the sequence edges. The number of sequence edges is as big as the number of operations we have on the re-entrant machine over all jobs, thus we have $J * r$ sequence edges where r is the number of operations per job on the re-entrant machine. Following the same reasoning with the complexity analysis as for the previous method, the complexity for this method is $O(ch * p + J * r)$ where ch is the number of checkpoints.

Retrieving relevant edge

For determining whether a specific constraint is critical, we can look up the necessary sequence constraint as described before. Not finding one implies the constrain considered is critical. Time complexity, in this case, depends on the implementation of the graph, and can be $O(1)$ for some data structures like an adjacency matrix by using the appropriate nodes as indexes and directly finding the constraint, if one exists.

Iterating operation sequence

We do not have to represent the problem as a graph to solve our problem. If we have the order of operations(or we are willing to sort them) we can simply follow every pair of adjacent operations and check if they follow the properties mentioned before. If so, the constraint involving those operations and any constraint involving operations from the same two jobs will be non-critical. The process is similar to the one for iterating sequence edges and so the complexity is $O(ch * p + J * r)$ as well. Pseudocode for this approach can be found in A.5

4.3 Forcing non-participating constraints on schedules

To create schedules with non-participating constraints we need to consider how to create different SCCs in the graph representing the schedule. Taking the example of the graph in B we see that the operations of the same job, represented by the same column, will always be in a cycle with one another. Edges directing to the right towards a later job, exist from every node in the graph(except for the last job). We thus observe that a lack of edges going left would mean there is no way to move back(towards earlier jobs) in the graph, which forces the graph to break down into many different SCCs. If there are no left-going edges at all - one for each job. Since we have to visit all nodes, the only way to avoid getting back is to finish all operations on the re-entrant machine of all jobs up to some job k, before starting any operation in job k+1. This is also visualized in the graph as a sequence edge going from the last operation of job k in the re-entrant machine to the first operation of job k+1 on the same machine.

In fact, any time we have a non-cyclic edge all parallel edges(parallel being connecting operation of the same two adjacent jobs) are ought to be non-cyclic too, all representing non-critical constraints. The point in time between two jobs connected by acyclic edges has infinite temporal slack, we call such point a "checkpoint" as the process can be halted at this point, assessed, and altered with no time constraints applied. With the realization presented in the last two paragraphs, it is simple to see how we can come up with schedules that contain such checkpoints. To do that we need to run the

solver on a subset of the jobs up to the point where we want to have non-participating edges, and then run it for the set of jobs following that, adjusting the start time to match the end time of the first set of jobs(plus whatever time is required by the constraints).

Criteria for forcing checkpoints

When considering where to put these checkpoints, and how many of them to force, we observe three criteria: First is the desired level of flexibility. The more flexible we want the schedule to be, the more points of infinite temporal flexibility we should use. Second is the desired productivity level. When breaking up the benchmark into several components to force checkpoints, we force the solver to come up with a new schedule. The more checkpoints we force the less freedom we give to the scheduler when looking for a schedule that is as productive as possible. We thus need to keep in mind that forcing many checkpoints comes with a potential cost in productivity, and we need to consider, depending on the case, whether this cost is worth the added checkpoints and to what extent. Third, we look into the usability of these checkpoints per use case. These checkpoints can be used to halt the process in order to compute a new schedule completely, have a human intervene, pause the process for an indefinite period, or do any other action that has to not be constrained by time, depending on the context of the industry and the production process.

4.4 Finding relationships between critical constraints

First, we need to describe the kinds of relationships critical constraints might have. Since all edges in an SCC must be participating in a cycle together(as you can always reach their source node) we understand that manipulating any pair of constraints in the same SCC might have an accumulating effect that can break the schedule. We do observe, however, that some of these cycles are not simple cycles, which implies they are built of smaller cycles, at least one of which has to be positive(and thus break the schedule) for the non-simple cycle to be positive. We, therefore, distinguish between pairs that are tightly bound by being in the same simple cycle and thus being able to both contribute to its value and break it, to those that are loosely bound, not sharing a simple cycle and thus not being able to both contribute to making a cycle positive without any of them individually making a simple cycle positive.

We now want, given a critical constraint, to find what other constraints are tightly bound with it, or in other words: In a graph representation, what other edges participate in the same simple cycle. To do that we apply a depth-first search from the destination of the edge to its source following the algorithm in A.3. We mark nodes with no outgoing edges ”dead” and the source node, ”cycled”. We then traverse the graph, keeping track of the visited nodes in our path. In line 10 we check if we revisit a visited node, which is not allowed in a simple cycle, if so we step back. If we do not reach an edge case, in lines 16-18 we do a recursive call with every node directly reachable from the current node, setting the node value to ”cycled” if any of the recursive calls lead to a cycled node. Otherwise, setting it to ”dead” if any node of the recursive call was marked dead, otherwise, unvisited. In lines 12-13 we have an edge case where we reach the source or an edge labeled cycled, which will result in other nodes in the path leading there being set to cycled. Similarly, in lines 14-15, if we reach a dead-end or a node that is already marked ”dead”, we will return dead. After the recursive call, we set the edge that lead us to the node to the same value the node has.

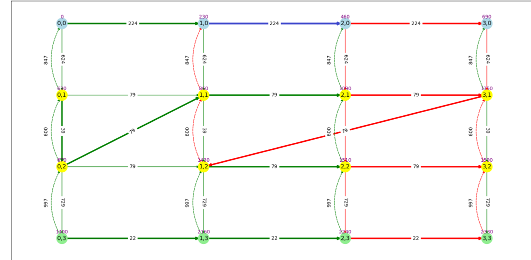

Figure 3: A chosen critical constraint(in blue) and its tightly bound constraints (in red)

5 Experimental Setup and Results

In this section, we will discuss the setup we used for assessing the results in terms of running time, validity, and the number of checkpoints to appear in a schedule organically. We will also reflect on the results and provide a brief explanation to them.

5.1 Validity

While we provide a logical explanation in the previous chapter of how the algorithms work and why they are applicable to the task at hand, we decided to validify that they all indeed return the correct value. This was done by assessing the results of one of the algorithms by inspection and then running some code to confirm that the results for selected schedules are identical among different algorithms. For finding tightly bound constraints, this was done by inspection as well.

5.2 Running time

Running the script that uses Kosaraju’s algorithm, the running time averaged 55.7278 ms on the set of the biggest schedules (with 500 jobs each) and averaged 3.3024 ms on a smaller set of test schedules. Multi Bellman-Ford averaged 9.963 s on the smaller set. On the subset of the small set of schedules containing only 10 jobs, Multi Bellman-Ford averaged 184.1068 ms. This method was not tested on the larger set of larger jobs as execution time would be too high. We then examine the run time for the approach finding all the cycles. We ran this only on the schedules containing 10 jobs as the complexity of this algorithm is significantly higher and found an average run time of 26.9177 s. For finding the cycles we used again the implementation by NetworkX [10]. We ran the methods described in section 4.2 on the set of the biggest

schedules. First, we ran the script that implements the approach iterating over all edges and we found an average runtime of 2.5688 ms. For the approach iterating only sequence edges, we got runtime of 0.298 ms. Iterating the operation sequence itself yielded an even faster runtime of 0.1541 ms. Finally, we ran the script implementing the approach for only determining whether a specific edge is critical for all edges of one big schedule(500 jobs) and got an average runtime of 1.3316μs. Note that these running times are descriptive of the algorithm execution itself, not including any prior data processing applied before any of these algorithms such as reading the file and constructing the graph. All scripts ran on an ”HP ZBook Studio G5” laptop. These results clarify the superiority of the solution that uses an algorithm for finding SCCs over the other approaches running graph algorithms, as well as the superiority of the methods described in 4.2 in terms of running time.

	Minimal set (10 jobs)	Test set	500 jobs set
Johnson	26.9176923s	N/A	N/A
Multi B-F	0.1841068s	9.9639568s	N/A
Kosaraju’s	N/A	0.0033024s	0.0557278s
All edges	N/A	N/A	0.0025688s
Sequence edges	N/A	N/A	0.000298s
Operation sequence	N/A	N/A	0.0001541s

Table 1: Runtime results for the different critical constraints finding algorithm over 3 sets of jobs.

5.3 Number of checkpoints in industry schedules

We also examined the number of checkpoints that naturally appears in real industry scheduling problems with schedules generated by the scheduler described in [2; 11]. We ran our algorithms on three groups of 150 schedules, each consisting of 500 jobs and 3 machines with one of them having one re-entry. The three groups are the results for the same set of 150 benchmarks with different parameters given to the solver. The first one (A) is a result of the solver favoring productivity over flexibility, in set B flexibility and productivity were balanced, and in set C flexibility was favored over productivity. However, the parameters for sets B and C ended up yielding the same results, making sets B and C identical.

We first examine the number of checkpoints for set A. We found 109 of the schedules ($\sim$72.7%) to not have any checkpoints at all. 37 of them ($\sim$24.3%) had one checkpoint and only 4 of them ($\sim$2.7%) had two checkpoints. We then run it for sets B and C. Here we find much greater variability. While 26 of the schedules ($\sim$17.3%) have no checkpoints at all, 13 schedules had as many as 71 checkpoints ($\sim$8.6%), being the highest number of checkpoints we found. The rest of the schedules had some number of checkpoints in between as shown in 5.

We thus observe that schedules that are designed to be flexible, will be more likely to have a high number of checkpoints in them, while those designed for maximizing productivity, will only have a few.

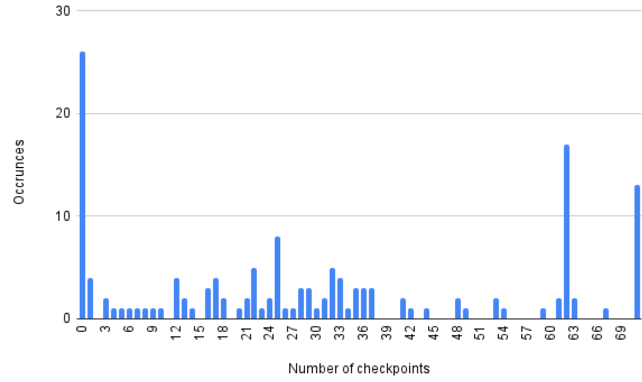

Figure 4: Number of checkpoints (x) and the number of schedules with this amount of checkpoints (y) in set B

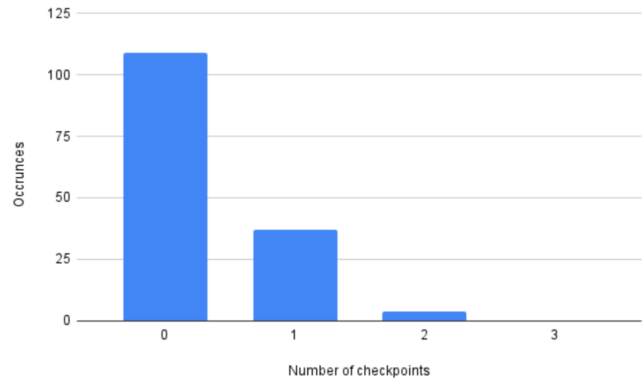

Figure 5: Number of checkpoints (x) and the number of schedules with this amount of checkpoints (y) in set A

6 Responsible Research

In this section, we will discuss the ethical aspects of the study, how it can affect people and how our results can be reproduced.

First, we think about whether this research can harm someone or discriminate against some group. Since we only deal with efficiency and flexibility of schedules, it is unlikely any harm can be done by using the findings of this paper.

We then look at the reproducibility parts of the research. Since all processes described here are deterministic, they can be reproduced exactly. Implementing the algorithms described in the paper will yield the same results we got for a given schedule. The only variation we expect is in running time, which is highly dependent on the machine. We, therefore, mention the hardware we used for our experiments

7 Future Work

In this section, we are going to discuss potential future work to be done on the subject, and how this research can be extended. While we believe there is not much to improve from our solutions for deterministically finding critical constraints in terms of time complexity, we do think more research can

be done as to how this can be used for improving the functioning of and the process executed by FMSs. We believe working with real-life examples of FMSs could reveal interesting ways to use these previously described checkpoints. In the paper, we describe how to find checkpoints in the process that are the result of the sequence picked by the solver, as well as how we can force the solver for specific checkpoints. In reality, a more balanced approach might be viable. One that does not deviate much from the ideal schedule but also roughly adheres to desired checkpoints. Finally, the paper suggests two different relationship types between critical constraints, and while it seems like they give us important information about how different constraints interact with each other, we did not have the chance to dive into possible ways to exploit these relationships for better performing or more flexible schedule.

8 Conclusions

We have shown that we can isolate all constraints of a schedule to critical and non-critical quickly enough to be run online. We also describe what type of constraints could be critical, and how finding a group of non-critical constraints creates a point of infinite temporal flexibility in the schedule. Points of infinite temporal flexibility can be forced into a schedule by altering its sequence and used to create checkpoints where any measure can be applied to the scheduler or the process without being subjected to time constraints. Pairs of constraints can be classified together to be tightly or loosely bound, describing whether they can have an accumulating contribution to breaking a schedule(without any of them individually breaking a schedule).

References

[1] J. P. Shewchuk and C. L. Moodie, “Definition and classification of manufacturing flexibility types and measures,” *The International Journal of Flexible Manufacturing Systems*, pp. 325–349, 1998.

[2] J. V. Pinxten, U. Waqas, M. Geilen, T. Basten, and L. Somers, “Online scheduling of 2-re-entrant flexible manufacturing systems.” *ACM Transactions on Embedded Computing Systems*, 2017.

[3] E. D. Demaine and C. E. Leiserson, *Introduction to Algorithms*, 2015.

[4] D. B. Johnson, “Finding all the elementary circuits of a directed graph,” *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975. [Online]. Available: https://doi.org/10.1137/0204007

[5] J. L. Szwarcfiter and P. E. Lauer, “A search strategy for the elementary cycles of a directed graph,” *BIT*, vol. 16, no. 2, pp. 192–204, 1976.

[6] G. Loizou and P. Thanisch, “Enumerating the cycles of a digraph: A new preprocessing strategy,” *Information Sciences*, vol. 27, no. 3, pp. 163–182, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020025582900238

[7] M. Sharir, “A strong-connectivity algorithm and its applications in data flow analysis,” *Computers Mathematics with Applications*, vol. 7, no. 1, pp. 67–72, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0898122181900080

[8] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972. [Online]. Available: https://doi.org/10.1137/0201010

[9] A. Arman and S. Tsaturian, “The maximum number of cycles in a graph with fixed number of edges,” *The Electronic Journal of Combinatorics*, vol. 26, no. 4, 2019. [Online]. Available: https://doi.org/10.1137/0204007

[10] NetworkX developers, “Networkx analysis in python,” last accessed 1 June 2022. [Online]. Available: https://networkx.org/

[11] R. van der Tempel, J. van Pinxten, M. Geilen, and U. Waqas, “A heuristic for variable re-entrant scheduling problems.” *ES reports*, vol. 2018, no. 2, pp. 336–341, 2018.

A results

A.1 Kosaraju's Algorithm

Algorithm 1 extract critical edges from SCCs

1: **for** every edge in the graph **do**
2: source = Source node of the edge
3: destination = Destination node of the edge
4: **if** SCC of source = SCC of destination **then**
5: Add edge to list of critical edges
6: **return** critical edges

A.2 Multi Bellman-Ford

Algorithm 2 Multi Bellman-Ford

1: **procedure** GETBIGNUMBER
2: $bigNumber \leftarrow 0$
3: **for** every edge in the graph **do**
4: $bigNumber \leftarrow bigNumber+$ Absolute value of edge weight
5: **return** $bigNumber + 1$
6: **procedure** SINGLE BELLMAN-FORD VARIATION
7: $distances = bigNumber, bigNumber, \ldots$
8: **for** every edge in the graph **do**
9: destination = Destination node of the edge
10: **if** SCC of source = SCC of destination **then**
11: Add edge to list of critical edges
12: **return** critical edges

A.3 Connected edges

Algorithm 3 Connected edges

1: **procedure** FIND CONNECTED EDGES
2: $unvisited = 0\ \ \&\ \ dead = 1\ \ \&\ \ cycled = 2$
3: $nodeStatus \leftarrow unvisited, unvisited, \ldots$
4: $visited \leftarrow False, False, \ldots$
5: $edges \leftarrow$Empty Set
6: $s \leftarrow$source of edge
7: $d \leftarrow$destination of edge
8: **return** FINDCONNECTEDEDGESHELPER($dest = s, prev = s, current = d$)
9: **procedure** FIND CONNECTED EDGES HELPER
10: **if** visited[current] == False **then**
11: **return** unvisited
12: $visited[current] \leftarrow True$
13: **if** $nodeStatus[current] == cycled \lor current == destination$ **then**
14: $nodeStatus[current] \leftarrow cycled$
15: **else if** $nodeStatus[current] == dead \lor \#outgoingEdges(current) == 0$ **then**
16: $nodeStatus[current] \leftarrow dead$
17: **else** MAX(
18: **for** every outgoing edge of current **do**
19: call function recursively with the nodes of the edge being the new prev and current
)
20: $visited[current] \leftarrow unvisited$
21: **if** $nodeStatus[current] == cycled$ **then**
22: add the edge going from prev to current to set of edges
23: **return** $nodeStatus[current]$

A.4 Iterate all edges

Algorithm 4 Iterate all edges

```
1: procedure ITERATE ALL EDGES
2:     for every edge e in the graph do
3:         if the operation a of the source of e is the last one on the re-entrant machine AND the operation b of the destination of e is the first one on the re-entrant machine AND the job k of b is one bigger than the job j of a then
4:             Add edge e to list of critical edges
5:             for every operation in job j do
6:                 Add the edge going from (j, op) to (j + 1, op) to critical edges
7:     return criticalEdges
```

A.5 Iterate sequence

Algorithm 5 Iterate sequence

```
1: procedure ITERATE SEQUENCE
2:     for every operation op in the sequence do
3:         if this is the first operation then
4:             skip to next iteration
5:         if the previous operation a is the last operation on the re-entrant machine AND the current operation b is the first one on the re-entrant machine AND the job k of b is one bigger than the job j of a then
6:             Add the edge from the previous operation to the current operation to list of critical edges
7:             for every operation in job j do
8:                 Add the edge going from (j, op) to (j + 1, op) to critical edges
9:     return criticalEdges
```

B schedule example