Splitting the Zones of Feasibility for a Given Schedule

Kasper Wolsink
Supervisor: Eghonghon Eigbe
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

1 Abstract

Flexible manufacturing systems (FMS) such as large industrial printers can be modeled as re-entrant flow-shops with a set of constraints of processing times, setup times and relative due dates. This is a non-deterministic system in which the actual values for these constraints can be different from the ones originally estimated. Therefore, a way to quickly determine the ranges of constraint values for which a given schedule becomes feasible or infeasible can become highly useful. In this paper, a heuristic algorithm is proposed that can efficiently split the zones of feasibility for a given schedule. With the right settings, an accuracy of 99.9% can be achieved with relatively few queries on 2-dimensional examples.

2 Introduction

An optimal schedule for a flexible manufacturing system (FMS) is a schedule such that it has the minimal possible make-span. The make-span is defined as the time of completion of the last executed operation. One such FMS in the real world is a large industrial printer. This system can be modeled as a re-entrant flow shop with machines, jobs, operations and a set of constraints [1]. In the case of a large industrial printer, this is a non-deterministic system. For example, one of the printer components might make a slight error, which will result in a longer setup time than originally anticipated. It might be the case that a given schedule for this printer is optimal, but not robust; the schedule easily becomes infeasible after due to a slight variance in constraint values. This can be problematic as it can be computationally expensive to calculate a near-optimal schedule for such an FMS due to this being an NP-complete problem [2].

In this paper, we will not focus on finding schedules for an FMS itself. Rather, we will explore the robustness of a given schedule by splitting the zones of feasibility. The main research question we will be trying to answer are the following:

- Can we find an efficient way to determine for any two constraints when a given schedule becomes feasible/infeasible for any two arbitrary values for these constraints?
- Can we generalize this method to n constraints?
- Can we make a trade-off between accuracy and running time for these methods?

The paper is divided into the following sections: in section 2, a formal definition of the re-entrant flow shop and a schedule is given, along with their graph representation. We also define the notion of the feasibility boundary in this section. Next, in section 3 we discuss the related work in this field. In section 4

we introduce a heuristic algorithm that can be used to accurately estimate the feasibility boundary. In section 5 we elaborate on the experimental setup used and discuss the obtained results. Sections 6 and 7 are dedicated to responsible research and future work respectively and we conclude in section 8.

3 Problem definition

In this section, the re-entrant flow shop is defined along with the definition of a schedule. Next, a graph representation for this flow shop is introduced. Finally, the notion of a feasibility boundary is defined.

3.1 Re-entrant flow shop definition

The re-entrant flow shop considered in this paper is very similar to the one introduced in [1]. However, a slightly different notation is used in this paper. Here, the re-entrant flow shop is defined by the tuple (M, J, O, ϕ, P, S, D) as follows:

- **Machines -** M is a set of machines that execute the operations.
- **Jobs -** J is a set of jobs that have to be executed. Each job consists of a set of operations. The number of operations for each job is equal.
- **Operations -** O is the set of operations. Each operation belongs to a job and each job has an equal amount of operations. As in [1], the k_{th} operation of job j can be denoted as $o_{\text{j,k}}$.
- **Re-entrance vector** The re-entrance vector ϕ is adapted from [1] without modification.
- **Processing times -** P is a function defined as $P : O \to R$
- **Setup times -** S is a function defined as $S : O \times O \to R$. For a schedule to be feasible, it must hold that for any two operations o_{i} and o_{j} such that the start time of o_{j} is greater than the start time of o_{i}, the start time of o_{j} must at least be $S(o_{\text{i}}, o_{\text{j}}) + P(o_{\text{i}})$ after the start time of operation o_{i} if S is defined for these operations. If S is not defined for these operations, the start time of o_{j} must be at least $P(o_{\text{i}})$ after o_{i}.
- **Relative due dates -** D is a function defined as $D : O \times O \to R$. For a schedule to be feasible, it must hold that for any two operations o_{i} and o_{j} such that the start time of o_{j} is greater than the start time of o_{i}, the start time of o_{j} must at most be $D(o_{\text{i}}, o_{\text{j}})$ after the start time of operation o_{i} if D is defined for these operations. If D is not defined for these operations, no such constraint exists.

3.2 Schedule definition

Next, a schedule is defined as a mapping of operations to start times. For each operation o, $T(o) \mapsto R$ gives the start time of that operation. A schedule is said to be feasible if it obeys the following constraints:

- The start times for each operation do not violate the constraints introduced by the processing times, setup times and relative due dates.
- Operations within a job must be done in order. This means that if $l > k$, $T(o_{\mathrm{j,\ l}}) > T(o_{\mathrm{j,\ k}})$ must hold.
- The equivalent operations across jobs must be done in order. This means that if $i > j$, $T(o_{\mathrm{i,\ k}}) > T(o_{\mathrm{j,\ k}})$ must hold.

3.3 Graph representation of the re-entrant flow shop

The re-entrant flow shop defined above can be represented as a weighted directed graph. Not only does this give us a clear visual representation of the model, but it also allows us to determine the feasibility of a schedule by detecting negative cycles using the Bellman-Ford algorithm [3, 4].

The graph representation is as follows. Each operation can be represented as a node. Let $n(o_{\mathrm{i}})$ be the node that represents o_{i}. For each setup time $S(o_{\mathrm{i}}, o_{\mathrm{j}})$, an edge $(\mathrm{o_i},\ \mathrm{o_j})$ is added with a weight of $S(o_{\mathrm{i}}, o_{\mathrm{j}})$. For each relative due date $D(o_{\mathrm{i}}, o_{\mathrm{j}})$ an edge $(o_{\mathrm{j}}, o_{\mathrm{i}})$ is added with weight $-D(o_{\mathrm{i}}, o_{\mathrm{j}})$. A given schedule can then be added to the graph as follows. If o_{j} is the first operation to start after o_{i}, an edge $(n(o_{\mathrm{i}}), n(o_{\mathrm{j}}))$ will be added with a weight equal to $P(o_{\mathrm{i}})$, or if there already exists an edge $(n(o_{\mathrm{i}}), n(o_{\mathrm{j}})$, its weight will be incremented by $P(o_{\mathrm{i}})$. This will be done for all operations in the order dictated by the given schedule.

3.4 Feasibility border

The processing times, relative due dates and setup times are all constraints defined by some mapping of operations to a positive real value. We can treat these constraints as variable values which we can either increment or decrement as we desire. Consider a subset of these constraints of a given system defined as C. Now consider an $|C|$-dimensional Euclidean space X. We can now take any arbitrary vector $v = \{v_1, v_2, ..., v_{|C|}\}$ within X and define v such that it represents a configuration of values for the constraints in C where v_i represents the value of the i_{th} constraint of that vector. Since X is a continuous space and the constraints in C take on continuous values, there is an infinite amount of such vectors in X. We can now label each of these vectors either $True$ or $False$ to denote the feasibility of this vector by taking its configuration and deciding the feasibility of the system under this configuration by some oracle B. B can be any algorithm that can verify the feasibility of a schedule under a set of constraints. In this research, the Bellman-Ford algorithm was used for this purpose [3, 4]. This results in an infinite labeled vector space.

Since the re-entrant flow shop considered in this paper is a system of linear constraints, the feasible zones of a schedule can be separated from the infeasible zones in such a Euclidean space by one or more $|C|$-dimensional-hyperplane segments as demonstrated in [5]. These hyperplane segments together form the feasibility border (also referred to as the feasibility boundary) for that particular schedule. A vector lies on the feasibility boundary if and only if the vector is labeled $True$ and there exists at least one vector entry v_i such that constraint C_i can not be made any tighter than v_i without the schedule becoming infeasible. The goal of this paper is to find a close estimate of this feasibility boundary for a given schedule in a reasonable amount of time.

4 Related work

As mentioned before, we are looking at scheduling problems specifically for flexible manufacturing systems such as large industrial printers. The authors of [1] touch upon this exact problem and give a formal definition of this system that is similar to the one proposed in this paper. Additionally, a heuristic scheduler is proposed by the authors which can generate a schedule for a given flexible manufacturing system. Many such heuristics exist. For example, the one proposed in [6].

Once such a schedule has been generated, its feasibility of it can be determined by the Bellman-Ford algorithm [3, 4]. The Bellman-Ford algorithm can detect if a given schedule and re-entrant flow-shop model produces negative (or positive depending on notation) cycles in which case a schedule is deemed to be infeasible.

Finding the values of constraints for which a schedule remains valid can be thought of as finding the feasible zones for said schedule. This has been done before, for example in [5] a method is proposed to find the feasible hyperplanes of a schedule for different values of constraints. This method can handle parameterized tasks such as the ones present in the scheduling problem discussed [1], but may not terminate in a feasible amount of time. In [7] the authors also concern themselves with finding the feasible zones of schedules but they focus on a slightly different problem. Here, performance characterization is addressed rather than just the feasibility of a schedule, resulting in a set of performance regions for different schedules rather than feasible and infeasible regions. The approach only takes into account deterministic systems, which do not fit the description of the re-entrant flow-shop considered here.

5 Splitting the zones of feasibility for a given schedule

In this section, we will introduce the EstimateBorder algorithm that can estimate the feasibility boundary for a given configuration. The algorithm first constructs a $|C|$-dimensional grid graph where each node represents an unlabeled vector as defined in section 3.4. Then the set of cut-edges is extracted from this

graph through the S^2 algorithm [8] which has a very efficient implementation for grid graphs.

We note that the feasibility boundary can be thought of as a set of hyperplane segments as demonstrated in [5]. We estimate these segments by interpolating between points that lie on the cut-edges to construct a set of $|C|$-dimensional hyperplanes that together form an estimate of the feasibility boundary.

The algorithms reliably works for $|C| = 2$. The algorithm has also been implemented for $|C| = 3$, but produces unpredictable results. Implementations for $|C| > 3$ are left for future work due to the geometric complexity quickly rising in higher dimensional configurations.

Algorithm 1 EstimateBorder: estimation of the feasibility boundary

```
1: function EstimateBorder(configuration, resolution, budget, bs_depth)
2:     G ← GridGraph(resolution)
3:     B ← CreateOracle(configuration)
4:     cut_edges ← S²(G, B, budget)
5:     return Interpolate(cut_edges, B, bs_depth)
6: end function
```

The EstimateBorder algorithm is shown in **Algorithm 1** and accepts four arguments:

- *configuration*: The configuration to run the algorithm on. This includes a flow-shop, a schedule, a subset of constraints and the domains for each constraint. The algorithm will estimate the feasibility boundary over the predefined domain of each constraint. The range for the domain is determined either through expert knowledge or extracted from the data itself.
- *resolution*: This is the resolution of the constructed grid graph. A *resolution*$\times$ *resolution* grid graph is said to have resolution of *resolution*.
- *budget*: This is the budget of the S^2 subroutine as described in [8].
- *bs_depth*: This is the maximum depth of the BinarySearchBoundary subroutine.

On line 2, we construct a grid graph G. On line 3 we define the oracle B for this specific *configuration*. In this example, we use the Bellman-Ford algorithm [3, 4] as an oracle. On line 4 we make a call to the S^2 subroutine resulting in a set of cut-edges (subsection 5.1). We then interpolate over this set on line 5 and return the result (subsection 5.2).

5.1 The S^2 subroutine

In this subsection, we will take a look at the S^2 subroutine of this algorithm as described in **Algorithm 2**. This algorithm is a slightly modified version of [8]. It accepts three arguments: the grid graph G, the oracle B and the

budget *budget*. The algorithm differs from [8] in that it returns the set of cut-edges instead of the set of all queried nodes. It also separates the queried nodes by their labels in $feasible$ and $infeasible$. This is done because the MSSP subroutine called in line 19 needs these sets separated. The MSSP subroutine calculates the shortest-shortest path and bisects it as described in [8]. Since G is a grid graph, an efficient implementation of the MSSP subroutine was used as seen in the appendix.

Algorithm 2 S^2: shortest shortest path [8]

```
1: function S²(G, B, budget)
2:     feasible ← ∅
3:     infeasible ← ∅
4:     cut_edges ← ∅
5:     while True do
6:         x ← Randomly chosen unlabeled vertex
7:         do
8:             feasibility ← B(x)
9:             if feasibility then
10:                Add x to feasible
11:            else
12:                Add x to infeasible
13:            end if
14:            budget = budget + 1
15:            Remove from G all edges whose two ends have different labels and
    add them to cut_edges
16:            if |L| >= budget then
17:                return cut_edges
18:            end if
19:        while x ← MSSP(G, feasible, infeasible) exists
20:    end while
21: end function
```

5.2 The Interpolate subroutine

In this subsection, we will take a look at the INTERPOLATE subroutine as shown in **Algorithm 3**.

Algorithm 3 Linear interpolation

```
1: function InterpolateSS(cut_edges, B, bs_depth)
2:     cut_edges ← SortSS(cut_edges)
3:     segments ← ∅
4:     base_points ← ∅
5:     hyperplane ← null
6:     while Not every cut-edge is marked as interpolated do
7:         for all edge in cut_edges do
8:             if edge is marked as interpolated then
9:                 continue
10:            end if
11:            if length(base_points) < |C| then
12:                Mark edge as interpolated
13:                border_point ← BinarySearchBoundary(edge, bs_depth, B)
14:                if border_point is not co-linear with the other points in
   base_points then
15:                    Add border_point to base_points
16:                    if length(base_points == |C|) then
17:                        hyperplane ← Hyperplane constructed through the
   points in base_points
18:                    end if
19:                end if
20:                continue
21:            end if
22:            intersection ← Intersection of hyperplane and edge
23:            if intersection exists then
24:                Mark edge as interpolated
25:            end if
26:        end for
27:        Add hyperplane to segments
28:        base_points ← ∅
29:        hyperplane ← null
30:    end while
31:    return segments
32: end function
```

The core idea behind this subroutine is that it very accurately calculates the feasibility boundary for a select few points, and can then constructs a $|C|$-dimensional hyperplane by interpolating between these points. This hyperplane then accurately estimates the feasibility boundary for some of the other cut-edges, without needing further calculations.

To do this when $|C| = 2$, we sort the set of cut-edges by one of their nodes in a way that the order represents a permutation such that the shortest path from the node of the first edge to the node of the last edge passes through all the nodes of all the edges in between only once. An intuitive way to think about

this is that the edges are ordered in such a way that you can draw a continuous line from the first edge to the last edge while crossing every edge in between only once. This sorting is done by the SortSS subroutine in line 2 of **Algorithm 3** where we create a 2-nearest-neighbor graph of the set of cut-edges and for each node we generate a depth-first-search pre-ordered permutation of the graph. We then pick the permutation such that the length of the path through all nodes is minimal, resulting in the correct order. This subroutine is shown in **Algorithm 4**.

Algorithm 4 SortSS: Sorting mechanism for $|C| = 2$

```
1:  function SortSS(cut_edges)
2:      nodes ← Either the positive or negative nodes of cut_edges
3:      nn_graph ← 2-NN Graph of nodes
4:      preorderings ← ∅
5:      for all n in nn_graph do
6:          p ← Calculate DFS-preorder from nn_graph starting from n
7:          Add p to preorderings
8:      end for
9:      permutation ← pre-order from preordrings with the shortest path
10:     sort cut_edges according to permutation
11: end function
```

The ordering described above does not reliably lead to the correct result for $|C| > 2$ and is the main reason the current algorithm can not be generalized to higher dimensions. Finding an algorithm that correctly orders the set of *cut_edges* for n dimensions is left for future work and is further discussed in section 8.

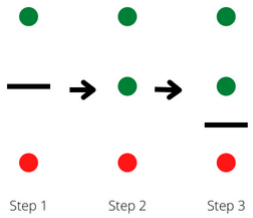


Figure 1: The BinarysearchBoundary subroutine visualised.

After we have sorted the set of *cut_edges*, we iterate through it. We accurately calculate the point where the feasibility boundary intersects the current edge through binary search. This is done by bisecting the edge and querying its label through B. This then creates a new interval within which the feasibility border must lie (Figure 1). We repeat this *bs_depth* times which results in an

accurate estimate of the feasibility border. We then do this for the first $|C|$ edges such that the resulting points are non-co-linear and we construct a hyperplane through these points as seen in lines 14 to 17 in **Algorithm 3**. We then calculate the intersection of this hyperplane with each of the remaining edges and if this intersection exists, we assume the respective edge belongs to the same border segment and label it as interpolated as seen in lines 22 to 25. We then repeat this entire ordeal and generate new hyperplanes until all edges are marked as interpolated. The resulting set of hyperplane segments is denoted as *segments* and returned in line 31. These hyperplanes together accurately estimate the feasibility boundary.

6 Experimental setup and results

In this section, we will discuss the experiments conducted. We will later mention the results and provide some analysis of our findings. For the main set of experiments, we only consider $|C| = 2$ as higher dimensions currently produce unreliable results. Additional experiments were conducted for $|C| = 3$ and the results are briefly discussed in the appendix.

6.1 Experimental setup

The goal of these experiments is twofold. First of all, we want to verify that the algorithm introduced in section 5 performs well on a wide range of configurations. If we succeed in doing this, we will gain some confidence in the correctness of the algorithm. Secondly, we want to test the possibility of trading off accuracy for an improved running time.

In these experiments, we will estimate the feasibility boundary using the algorithm introduced in section 5 for different 2-dimensional constraint configurations. Different types of constraints are expected to generate different shapes of feasibility boundaries. By verifying the algorithm works on different configurations, we can make some assumptions about its effectiveness and robustness.

The experiments are conducted as follows: the set of constraints C will be initialized by arbitrarily picking some constraints of different types. Let i be the initial value of a constraint, then the feasibility border will be estimated over domain $[i - 30\%, i + 30\%]$ for each constraint.

Then, 300 random samples will be generated over the same range and we will classify them using the estimated feasibility boundary. This is done by checking on which side each sample lies for every border segment in *segments*. If the sample lies on the feasible side of every segment, we will classify it as feasible. Otherwise, we will classify it as infeasible. We will assume each border segment extends out infinitely on both sides, although this is not shown in the graphs. After this, we will query the actual label for each sample through our oracle B, allowing us to calculate the accuracy score which is simply defined as the amount of correctly labeled samples divided by the total amount of samples.

Alongside the accuracy, we can measure the running time of the algorithm

and the number of times oracle B was queried. The oracle used in this example, is the Bellman-Ford algorithm [3, 4]. The worst-case running time of this algorithm is $O(|E||V|)$ where $|E|$ and $|V|$ are the number of edges and vertices in the problem respectively [9]. Therefore, for configurations with large amounts of edges and vertices, it becomes crucial we query B as few times as possible.

Finally we would like to note that the amount of feasible and infeasible samples over the range of constraint values might be heavily skewed towards either one of the labels. Therefore, it is entirely possible that the algorithm achieves a high accuracy score while actually incorrectly estimating much of the border segments. This means that an accuracy score significantly below 100% should be seen as a faulty estimation of the feasibility boundary.

For the experiments we will introduce the following variables:

- **C**: The subset of constraints C we run the algorithm on. The exact values of these sets can be found in the appendix (Table 2).
- **b**: The *budget* parameter.
- **d**: The *bs_depth* parameter.
- **q**: The amount of time the oracle B is queried.
- **t**: The running time of the algorithm (in ms). Note that these experiments were conducted on consumer hardware. The running time of a configuration relative to other configurations is of interest, rather than the absolute running time.
- **Acc**: The accuracy of the estimated feasibility boundary.
- **FS X**: A flow shop of X jobs and its corresponding schedule. The full configurations are attached in the Appendix.

Note that we chose not to vary the *resolution* parameter, as it would generate exponentially more results that could not be fitted inside this paper. The *resolution* parameter was set to a constant value of 10. A full parameter sweep is left for future work.

6.2 Results

Parameters			FS 20			FS 50			FS 100		
C	**b**	**d**	**q**	**t**	**Acc**	**q**	**t**	**Acc**	**q**	**t**	**Acc**
1	10	1	12.4	128.2 ms	76.9%	15.3	178.4 ms	73.4 %	12.0	390.4 ms	74.4%
		10	30.0	164.2 ms	74.8%	37.5	342.5 ms	75.6 %	38.0	1112.6 ms	79.9 %
		100	237.5	533.3 ms	76.2%	220.0	1509.4 ms	74.3 %	210.0	5474.8 ms	74.1 %
	30	1	36.0	508.5 ms	96.7 %	36.4	651.1 ms	97.0 %	34.0	1258.3 ms	97.4 %
		10	**70.0**	**581.6 ms**	**100** %	**74.0**	**936.1 ms**	**100 %**	**74.0**	**2301.0 ms**	**100 %**
		100	430.0	1344.5 ms	100 %	430.0	3276.5 ms	100%	430.0	11332.9 ms	100 %
	50	1	54.0	571.8 ms	97.3 %	54.4	788.5 ms	97.8 %	54.0	1796.1 ms	98.3 %
		10	94.0	630.0 ms	100 %	90.0	1027.7 ms	99.9%	90.0	2743.4 ms	100%
		100	450.0	1281.4 ms	100 %	562.5	3404.8 ms	100 %	450.0	11783.6 ms	100 %
2	10	1	18.7	164.5 ms	95.3 %	14.5	234.2 ms	95.5%	16.7	2502.6 ms	89.2 %
		10	**36.7**	**244.6 ms**	**100 %**	**30.0**	**596.9 ms**	**100 %**	40.0	7185.8 ms	92.1%
		100	210	1085.8 ms	100 %	210.0	4497.7 ms	100%	212.5	9482.5 ms	92.3 %
	30	1	35.6	486.4 ms	95.9 %	34.0	765.1	94.6%	36.0	6321.2 ms	95.9 %
		10	50.0	608.4 ms	100 %	50.0	1389.9 ms	100%	70.0	12321.2 ms	97.6 %
		100	230.0	1584.4 ms	100 %	230.0	6262.1 ms	100%	**430.0**	**54487.4 ms**	**98.0 %**
	50	1	54.0	585.5 ms	96.3 %	54.0	948.0 ms	96.2 %	56.0	7926.0 ms	95.3 %
		10	70.0	724.0 ms	100 %	70.0	1397.2 ms	100%	90.0	14801.5 ms	97.1 %
		100	250.0	1385.6 ms	100 %	250.0	9242.6 ms	100%	450.0	53707.1 ms	97.3 %

Table 1: Results for a 2-dimensional configuration

As seen in Table 1, the EstimateBorder algorithm can reliably estimate the feasibility boundary with close to 100% accuracy given the correct values for the parameters. For each configuration, the setup that resulted in the highest accuracy with the lowest running time is shown in **bold**.

First, we take a closer look at the *budget* parameter **b**. Table 1 shows that setting **b** too low can result in a faulty border estimation. This can be seen when $\mathbf{C} = 1$ for all flow-shop configurations. When the budget is set too low, the S^2 subroutine (**Algorithm 2**) does not produce enough cut-edges such that it reliably captures all segments of the feasibility boundary. Increasing **b** solves this issue but also increases the running time significantly.

It should also be noted that for some configurations a low budget reliably produces an accurate result. This can be seen in the configuration where $\mathbf{C} = 2$ of both FS 20 and FS 50. In these configurations, the actual feasibility boundary only consists of one straight segment. Therefore, we technically need only two cut edges to correctly interpolate the entire feasibility boundary. Contrary, in a configuration where $\mathbf{C} = 1$ for example, the actual feasibility boundary consists of more than 1 segment. As a result, a low budget **b** is not sufficient. We can therefore conclude that underlying domain knowledge is required to set the budget **b** as low as possible - since it minimizes run time - while still setting it sufficiently high enough that it correctly captures the underlying segments of

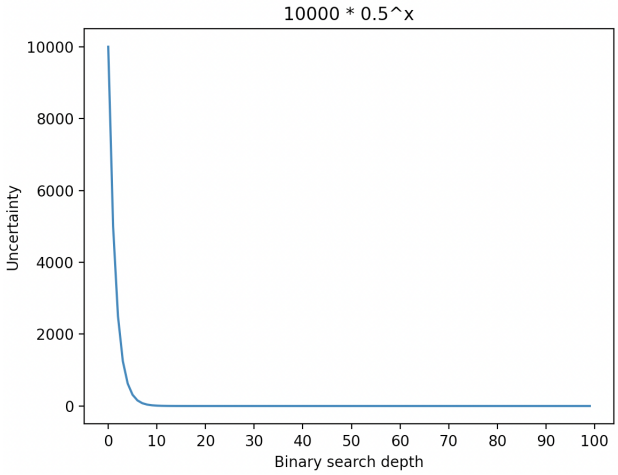
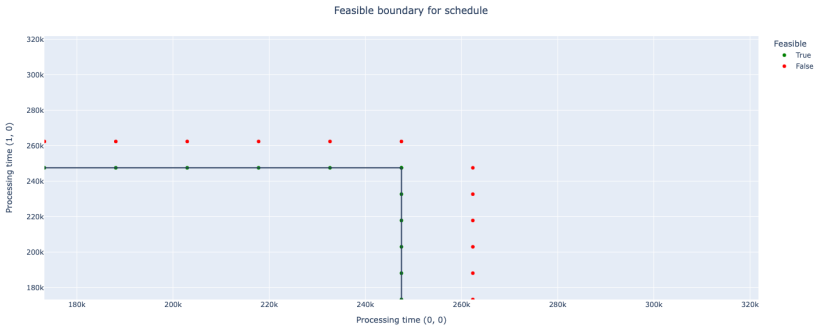


Figure 2: Exponential decrease of uncertainty visualised

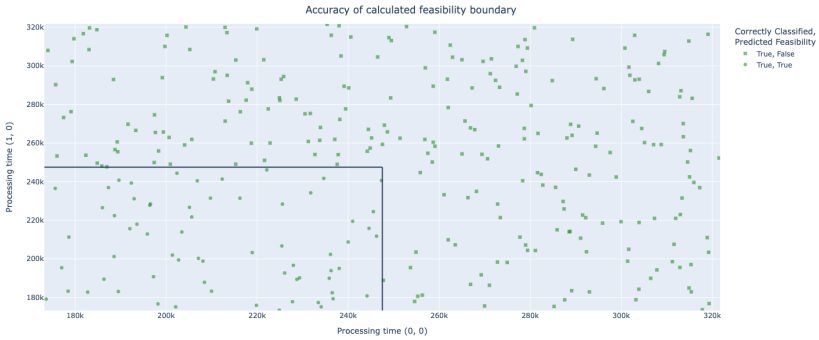
the feasibility boundary.

Next, we zoom in on the *bs_depth* parameter **d**. This parameter determines how accurately we estimate the actual feasibility boundary when we interpolate between cut-edges. Based on the results of Table 1, we can state that increasing **d** generally increases the accuracy up to a certain point if the **b** is sufficient. There is a significant increase in accuracy from $\mathbf{d} = 1$ to $\mathbf{d} = 10$, but a barely noticeable one from $\mathbf{d} = 10$ to $\mathbf{d} = 100$. This is to be expected though, as each iteration of the BinarySearchBoundary subroutine exponentially decreases the uncertainty of the feasibility boundary estimate on a certain cut-edge by halving it. In Figure 2 we can see that even starting from an uncertainty of 10000, it quickly approaches 0 after only a few iterations. We can quickly calculate the optimal value for **d** with the following formula: $m >$ length of the current $cut\_edge * 0.5^d$ where m is the maximum amount of uncertainty we are willing to tolerate. The length of the current *cut_edge* is the uncertainty we start with, as we know the actual feasibility boundary passes somewhere through this edge. Calculating this value is important as increasing **d** any further significantly increases the number of queries and the running time while no longer improving accuracy.

Finally, we look at the running times for different flow-shops. We can see that as the number of jobs increases, the running time grows exponentially. This is because the Bellman-Ford algorithm [3, 4] used as our oracle has a worst-case run-time of $O(|V||E|)$ [9]. As the number of jobs in the flow shop increase, so do both $|V|$ and $|E|$. Therefore, it becomes increasingly important to optimally tune the **b** and **d** parameters to minimize the amount of queries as the number of jobs increases.

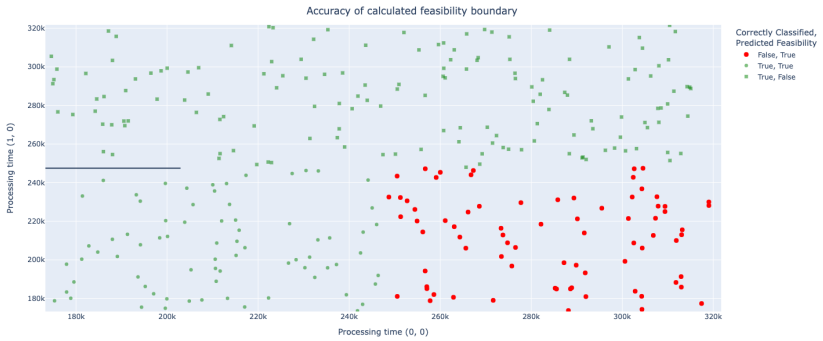


(a) The nodes of the cut-edges produced by the S^2 algorithm. The border is interpolated through the edges and is shown as a line.

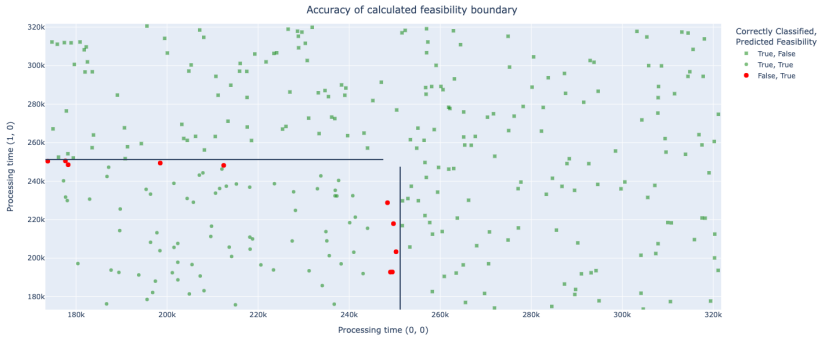


(b) The predicted border is evaluated by generating 300 random samples and comparing their true label to their predicted one.

Figure 3: Graphs for FS 20, C=1, b=30, d=10. The border is accurately estimated.



(a) b=10, d=10. Not all border segments are captured.



(b) b=30, d=1. All border segments are captured but inaccurately.

Figure 4: Graphs for FS 20, C=1 with incorrect border estimations.

7 Responsible research

To ensure reproducibility of the conducted experiments, the configurations of flow-shops are attached in the appendix. Additionally, pseudo-code for the proposed algorithms is provided.

Some outliers were found after conducting the experiments. However, it was quickly concluded that these outliers were not caused by an exceptional configuration, rather they were caused by either an incorrect setup or a bug in the implementation of the algorithms. Therefore, the decision was made to not include these results.

The algorithms proposed in this paper are heuristic by nature. Therefore it is likely not guaranteed that the correct results will be produced for every configuration. During this research, a wide range of configurations were tested. Not all of these results are shown in this paper, as there are too many of them. The results in this paper are randomly selected from all the obtained results and are not cherry picked to fit the narrative.

Since no sensitive data was used during the entirety of this research, we see no other ethical pitfalls that need to be discussed in this section.

8 Future work

In this section, we will discuss the current limits of our findings and propose subjects for future research.

Although the proposed algorithm shows promising results for a 2-dimensional case, the current implementation does not yet lend itself to higher dimensions. Currently, we can not reliably interpolate the correct hyperplanes through the cut-edges when $|C| > 2$. In the INTERPOLATE subroutine (**Algorithm 3**), we sort the cut-edges according to the SORTSS subroutine (**Algorithm 4**), and then iterate through them in order and construct a hyperplane through the first $|C|$ edges that are not co-linear. If we are able to sort the cut-edges by the hyperplane segment they belong to in the actual feasibility boundary, the INTERPOLATE subroutine will correctly construct accurate estimates of these hyperplanes. The SORTSS subroutine reliably succeeds in sorting the edges correctly for a 2-dimensional case but does not generalize to higher dimensions. Therefore, finding an algorithm that correctly sorts the cut-edges for n dimensions will likely mean we can generalize the INTERPOLATE subroutine to n dimensions.

Next, we note that we have not performed a full sweep for the parameters of the ESTIMATEBORDER method (**Algorithm 1**) in this paper. We have decided not to vary the *resolution* parameter, and have only considered a limited range of possible values for each constraint. A more complete parameter sweep could be conducted in the future.

9 Conclusions

The algorithm proposed in this paper can accurately determine the feasibility boundary for two arbitrary constraints of a given flow shop and schedule. With the right configuration, a classification accuracy of 99.9% can be achieved over the predetermined range with relatively few queries.

This means that both the first and third research questions posed are answered (section 2). The second research question is partly answered. A way to generalize the algorithm to higher dimensions is proposed but not successfully implemented. If an implementation of this is realized, we can reliably estimate when any schedule for any flow-shop might break for any subset of constraints. This might prove the be useful in real-world applications such as the scheduling of a large-scale FMS.

References

[1] Joost Van Pinxten et al. "Online scheduling of 2-re-entrant flexible manufacturing systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (2017).

[2] Michael R Garey, David S Johnson, and Ravi Sethi. "The complexity of flowshop and jobshop scheduling". In: *Mathematics of operations research* 1.2 (1976).

[3] Richard Bellman. "On a routing problem". In: *Quarterly of applied mathematics* 16.1 (1958).

[4] Lester R Ford Jr. *Network flow theory*. Tech. rep. Rand Corp Santa Monica Ca, 1956.

[5] Alessandro Cimatti, Luigi Palopoli, and Yusi Ramadian. "Symbolic computation of schedulability regions using parametric timed automata". In: *2008 Real-Time Systems Symposium*. IEEE. 2008, pp. 80–89.

[6] Caixia Jing, Wanzhen Huang, and Guochun Tang. "Minimizing total completion time for re-entrant flow shop scheduling problems". In: *Theoretical Computer Science* 412.48 (2011).

[7] Joost Van Pinxten, Marc Geilen, and Twan Basten. "Parametric scheduler characterization". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.5s (2019).

[8] Gautam Dasarathy, Robert Nowak, and Xiaojin Zhu. "S2: An efficient graph based active learning algorithm with application to nonparametric classification". In: *Conference on Learning Theory*. PMLR. 2015, pp. 503–522.

[9] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008, pp. 55–78.

A Appendix

A.1 MSSP subroutine

Algorithm 5 MSSP [8]: Optimized for grid graphs.

```
1: function MSSP(G, feasible, infeasible)
2:     deque_feasible ← new Deque
3:     deque_infeasible ← new Deque
4:     visited_feasible ← ∅
5:     visited_infeasible ← ∅
6:     for all n in feasible do
7:         Append (n, G.Neighbours(n)) to deque_feasible
8:         Add n to visited_feasible
9:     end for
10:    for all n in infeasible do
11:        Append (n, G.Neighbours(n)) to deque_infeasible
12:        Add n to visited_infeasible
13:    end for
14:    while deque_feasible and deque_infeasible do
15:        n, neighbours ← deque_infeasible.popleft()
16:        for all neighbour in neighbours do
17:            if neighbour not in visited_feasible then
18:                Add neighbour to visited_feasible
19:                Append (neighbour, G.Neighbours(neighbour)) to
   deque_feasible
20:                if neighbour in visited_infeasible and neighbour not in infea-
   sible then
21:                    return neighbour
22:                end if
23:            end if
24:        end for
25:        n, neighbours ← deque_feasible.popleft()
26:        for all neighbour in neighbours do
27:            if neighbour not in visited_infeasible then
28:                Add neighbour to visited_infeasible
29:                Append (neighbour, G.Neighbours(neighbour)) to
   deque_infeasible
30:                if neighbour in visited_feasible and neighbour not in feasible
   then
31:                    return neighbour
32:                end if
33:            end if
34:        end for
35:    end while
36: end function
```

A.2 Configurations used for results

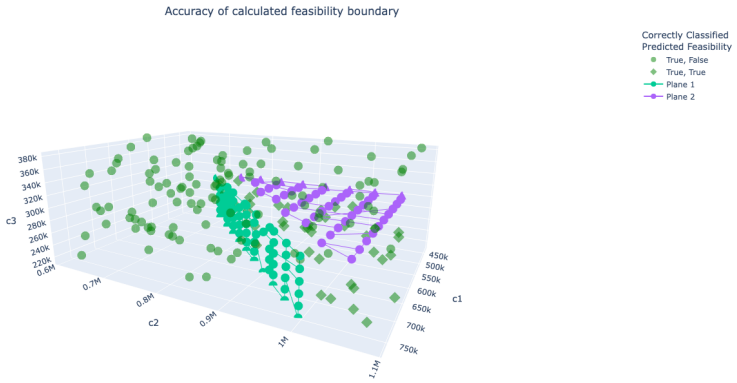
C	
1	$[P(o_{0,0}),\ P(o_{0,1})]$
2	$[S(o_{3,1}, o_{3,2}),\ D(o_{3,1}, o_{3,2})]$

Table 2: Constraint sets for C

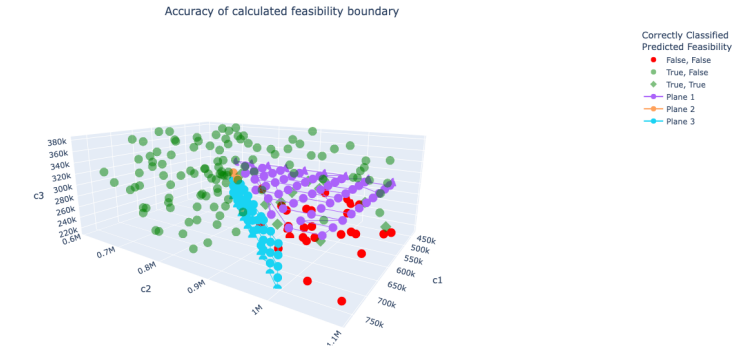
The configurations of FS 20, FS 50 and FS 100 mentioned in chapter 6.2 can be found at `https://github.com/kwolsink/Feasibility_bound_schedules`. A parser for these configurations is included. The constraint sets used for C can be seen in Table 2.

A.3 Results for $|C| = 3$

As stated in section 5, the EstimateBorder algorithm only reliably works when $|C| = 2$. However, an implementation for $|C| = 3$ was realised. This version of the algorithm does not produce the correct result reliably. This is mainly due to the fact that the SortSS subroutine (**Algorithm 4**) does not produce the correct ordering of cut-edges every time. This is because the SortSS may produce many permutations in a 3-dimensional example of which some will lead to a correct result while others will lead to an incorrect result. Therefore, running the EstimateBorder algorithm multiple times on a 3-dimensional configuration will produce different results each time, of which some are correct while others are incorrect. This is shown in Figure 5.



(a) A correct example.



(b) An incorrect example.

Figure 5: Both a correct and incorrect example for $|C| = 3$. Both trials were conducted under the exact same settings, showing the unreliability of this method. The planes are displayed as colored points.

If an algorithm were to be developed for $|C| = n$ like SortSS is for $|C| = 2$, we might be able to generalize the algorithm to any dimension.