EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Scheduling and optimization of heuristic production printer scheduler

Pasupula, S.V.V.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

EINDHOVEN UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Scheduling and Optimization of Heuristic Production Printer Scheduler

---

Author:
S. V. V. PASUPULA
s.v.v.pasupula@student.tue.nl

Supervisors:
dr. ir. Marc GEILEN
ir. Umar WAQAS

Electronic Systems
Department of Electrical Engineering

26th September 2016

Océ technologies, Venlo

Abstract

Scheduling and Optimization of Heuristic Production Printer Scheduler

A *Production Printer* can print hundreds of thousands of sheets at a fast pace. These are also called Large Scale Printers (LSPs) and are mainly used to print large volumes of sheets within less time, for example pamphlets, books with large number of pages, etc. These printers can also handle wide variety of media types (A3, A4, A5, B2, B3, etc). To print on such large scale systems, the resources need to be automatically allocated and printer components should be scheduled to perform different operations at different times perfectly such that the print job finishes quickly. A scheduler is one such program that can understand the work requirement and allocate resources in such a way that the main objectives of these production printers are met, i.e., high speed, high quality and low cost. The scheduler discussed in Waqas et al. [9], takes a heuristic approach (referred as 'Heuristic Production Printer Scheduler' in the coming sections) to do the scheduling. This algorithm looks at the scheduling problem as a re-entrant flow shop problem and solves it by using 'heuristics' after modeling the different tasks to be performed and their inter-dependencies as a 'constraint graph'.

This heuristic approach has shown to generate better schedules[9] than the algorithm that is currently being used in the printer . To use this algorithm on a real world large scale printer, it needs to be an 'online' scheduler. For a scheduler to be categorized as an 'online' scheduler, the most important requirement is that, it should not be a bottleneck to the printer performance. The prototype printer for which the scheduler is designed can print each sheet in a print job made of smallest supported sheet size every 400ms. In order to do so, it needs schedule of each sheet in less than 400ms. But the initial version of this Heuristic production printer scheduler is slow and as the number of sheets to be scheduled increases, the runtime to schedule also increases. This makes the scheduler a bottleneck for the performance of the printer. So, the goal of the project is to optimize the algorithm so that it will not be a performance bottleneck.

This thesis proposes various optimization techniques along with a generic technique that can compute schedules of each sheet in the print job in less than 400ms so that the printer can print each sheet in 400ms. The prototype printer, while printing sheets of smallest supported size at its fastest supported speed, is assumed to hold up to 92 sheets. The schedule of a sheet cannot be computed independently, but is influenced by other sheets in the print job. Therefore, the initial optimizations concentrate on reducing the runtime of jobs with 92 or less number of sheets. The original version of the 'heuristic' scheduling algorithm in its worst case takes almost 2.5 seconds to schedule 92 sheets, which is nowhere near the target runtime of 400ms. So, different optimizations are proposed to the scheduling algorithm which have given a maximum speedup of up to 12 times. But, when number of sheets increase more than 92, the runtime will also increase. To avoid this, a generic optimization solution is proposed which when combined with one of the previous optimizations will reduce the scheduling runtime per sheet to less than 400ms irrespective of the number of sheets in the print job. These optimizations in the scheduler resulted in scheduling runtime per sheet to less than 250ms. But, this gain in runtime, has cost the scheduler in quality of schedules computed. While in very few cases quality of schedules improved by as much as 10%, but on an average, a decrease in quality of almost 10% is observed in the schedules generated. Altogether these optimizations contributed to reach the goal of the project.

Acknowledgments

This thesis gives the work done for the graduation project on the scheduler for a production printer developed at Océ technologies, Venlo. From the beginning of the project several people provided much needed inspiration, which acted as a driving force behind the success of the work.

First I would like to thank Prof. Marc Geilen, my supervisor at TU/e for his guidance. During the tenure of the project, his supervision has not only encouraged me to ask tough questions but also to come up with better solutions. His supervision has had a great influence on the project and added much more value to the work. It has been a privilege to work under his supervision.

Next, I would like to mention Umar Waqas, my supervisor at Océ technologies and at TU/e. I would like to thank him for giving me a chance and allowing me to work on this project. I greatly appreciate his very patient effort to teach me "how it is done", to solve every problem I faced. His constant support and mentoring have been a great inspiration towards completing this work and have made me a better researcher.

I appreciate the invaluable feedback from Joost van Pinxten. His support has got me out of some really tricky situations. I would like to thank Lou Somers, Jack Kandelaars, and Patrick Vestjens, colleagues at Océ technologies for their support and feedback. On a personal note, I would like to mention Sethuraman for being not only a fun traveling mate to and from Venlo during almost 8 months of this project, but also for being a great friend. Thanks to Raja, Gonzalo for their friendship which made the journey of graduation much more memorable.

Finally, I would like to thank my family and all the friends back at home in India for being a constant source of inspiration. Without these people, it would have been difficult for the project to reach its current state.

Contents

Abstract **iii**

1 Introduction **2**
1.1 Production Printers . . . 2
1.2 Constraint graphs . . . 4
1.3 Review of online scheduling algorithms . . . 4
1.4 Re-entrant flowshop problem . . . 5
1.5 Path scheduling algorithms . . . 6
1.6 Bellman-Ford algorithm . . . 6
1.7 Contributions . . . 13
1.8 Report overview . . . 13

2 Problem definition **14**

3 Heuristic Production Printer Scheduler **15**
3.1 Production printer constraint graph model . . . 15
3.2 Heuristic approach . . . 21

4 Related work **26**
4.1 Optimizing the Bellman-Ford algorithm . . . 26
4.2 Runtime optimizations . . . 27

5 Research, implementations and results **28**
5.1 Runtime optimizations & Results . . . 28
5.1.1 Avoiding invoking Bellman-Ford algorithm twice . . . 28
Optimization technique . . . 28
Optimization Statement . . . 29
Implementation . . . 30
Correctness Reasoning . . . 30
Results and Analysis . . . 30
5.1.2 Optimizing Bellman-Ford by *'early break out'* from the algorithm . . . 32
Optimization Statement . . . 32
Optimization technique . . . 32
Algorithm . . . 33
Proof of Correctness . . . 33
Worst case analysis . . . 35
Results . . . 35
Conclusion . . . 37
5.1.3 Optimizing Bellman-Ford by *marking the modified vertices* . . . 38
Optimization statement . . . 38
Optimization technique . . . 38
Algorithm . . . 38
Results . . . 38
Conclusion . . . 39
5.1.4 Optimizing the Bellman-Ford algorithm by marking the modified vertices and *relaxing them in parallel* . . . 41

Optimization statement 41
Algorithm and results 41
Results 41
Conclusion 42
5.2 A *generic* optimization technique 43
5.2.1 Algorithm 43
5.2.2 Conclusion 43

6 Conclusion and Future work **44**
6.1 Conclusion 44
6.2 Future work 44

A Experimental setup **45**

B Test set **45**

C Box plots **46**

D Heuristics **46**

Bibliography **47**

Chapter 1

Introduction

A large scale printer is a combination of multiple *machines*, each capable of performing a task independently. As there are different machines serving different purposes, the task of printing one sheet can be divided into multiple sub-tasks. Because each machine can execute only one sub-task at a time, computing a schedule is divided into two parts, 'sequencing' and 'scheduling' the sub-tasks. *Sequencing* is to establish the order of sub-tasks that a particular machine will perform, while *scheduling* assigns each sub-task a start time at its respective machine [1].

Sequencing of these sub-tasks in these large scale printers is a special case of *flowshop* problem called 're-entrant flowshop problem' [9], explained in detail in section 1.4. Before going into details of sequencing and scheduling, this chapter first introduces the working of production printers in section 1.1. Section 1.2 will explain constraint graphs and section 1.3 reviews online scheduling. Section 1.5 explains the need for a path-algorithm and section 1.6 explains Bellman-Ford algorithm which is used to schedule.

1.1 Production Printers

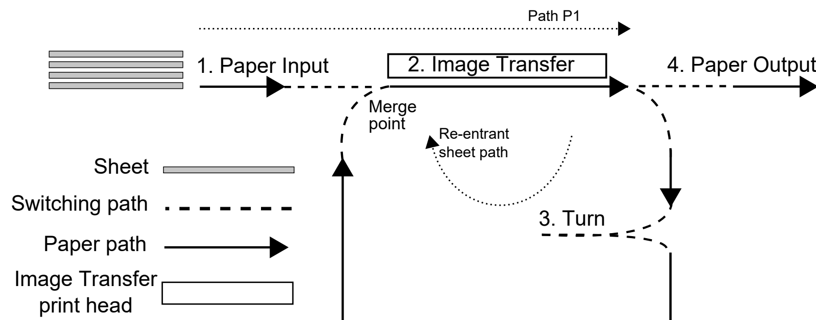


FIGURE 1.1: Paper path of a printer. (adapted from [9])

As discussed in the Introduction above, to complete the task of printing one single sheet, each machine has to perform a unique sub-task. The task of printing one sheet is defined as a *job* and sub-tasks are defined as *operations*. The numbered labels in figure 1.1 shows different machines present in the production printer.

1. *Paper Input* (PI): This machine feeds the appropriate media/sheet to the printer.
2. *Image Transfer* (IT): This machine prints the image on to the paper.
3. *Turn* (T): This machine turns the paper so that *Image Transfer* machine can print on the other side of the paper.
4. *Paper Output* (PO): This is the machine where print jobs finally accumulate/stacked.

The *paper path* in a printer is defined as the combination of machines a paper has to go through to finish the job. A *job* as explained above is a set of ordered operations which define the order of machines the paper has to move along. There are two types of jobs. A *simplex* type job, where a sheet is printed on only one side, so it needs to go through Image Transfer machine only once. It's corresponding paper path will be {PI, IT, PO}. A *duplex* type job where a sheet is printed on its both sides. Hence, a duplex sheet needs to go through the Image Transfer machine twice. But before its second pass, the sheet needs to be turned. Therefore, it's corresponding paper path will be {PI, IT, T, IT, PO}. Figure 1.1 also shows the paper path of a duplex sheet in the printer. The print on first side of a sheet is referred as 'first pass' and the print on second side is referred as 'second pass'.

Irrespective of job type, either simplex or duplex, sheets are inserted into the printer from Paper Input machine and are passed to Image Transfer in the main path P1, where the first side of the sheets is printed. A sheet of simplex job type, gets its first pass and leaves the printer at the Paper Output. But the sheet of duplex job type need to re-enter the main path P1 at the **merge point** (shown in figure 1.1) after turning at the Turn machine. The path through the Turn machine until merge-point is called as *re-entrant* path. The most important task of the scheduler is here at the merge point, "to decide between which two sheets's does a re-entering sheet's second pass execute its print operation". These decisions are called *interleaving* decisions and are taken such that delay due to setup time will be as minimum as possible so that the last sheet in the queue can come out at Paper Output machine at the earliest possible time.

Because from merge-point to merge-point there can be multiple sheets present, these sheets are referred to as *buffer*.

A simplex type job does not need any sequencing, because once that sheet enters the printer at Paper Input, it has fixed path to Image Transfer and then out of the printer at Paper Output. So, if simplex jobs and duplex jobs are mixed, it will cause gaps in the re-entrant path, leading to a complicated scheduling problem involving more delays due to the gaps. Therefore, for simplicity it is assumed that all sheets are of *duplex* job type. Therefore, even the sheets which are to be printed on only one side (simplex job type), enter the path *P1* and pass through Image Transfer twice with now work done during the second time.

Since each machine in the printer does some work on a sheet, there are different delays that affect the printing time of all the jobs that enter the printer. The time taken by each machine to execute an operation on a sheet is defined as *processing time*. But before processing the next sheet in the sequence, a machine may have to perform certain setup steps. For example, depending on the next sheet's thickness the Image Transfer machine might need some time to either heat or cool the ink. This causes additional delay in starting the execution of operation on the next sheet in sequence. This delay changes depending on the sequence of sheets and is defined as *setup time*. If two similar sheets follow one after the other into the Image Transfer machine, then the machine need not perform setup steps twice because the machine is already in the required state for the second sheet. Since this setup time depends on the sequence of sheets, the delay can also be termed as the *sequence dependent setup time*.

The setup time between the two operations gives the minimum amount of time the second operation need to wait before it is executed after the first operation. This means, setup time gives the lower bound for the delay. Because of the scheduler sequences in a way that successive operations have as small setup time as possible, there is a chance that, some operations with higher setup time do not get to execute at their corresponding machine at all. To avoid this, an upper bound is introduced for every operation. This is defined as the *relative due date*. The relative due date gives the maximum amount of time an operation can wait after its predecessor in the sequence has finished its execution. The scheduler models these operations and delays between these operations as a constraint graph, with each operation as a vertex and the delays make up the edges. Constraint graphs are explained

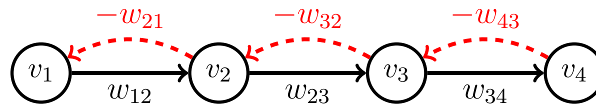


FIGURE 1.2: An example constraint graph.

in section 1.2. Using all these parameters of *re-entrance*, *processing times*, *sequence dependent setup times* and *relative due dates* the scheduler uses a heuristic approach to minimize the total duration of the schedule.

Total duration of the schedule, also denotes the "time to complete the last job in the queue" and is referred to as *makespan* in graph terminology. The goal of the scheduler is to minimize the makespan of the schedule by changing the sequence of operations executed on a machine while maintaining the order of sheets intact.

1.2 Constraint graphs

A constraint graph is a set of vertices connected using set of edges called directed edges. A directed edge is an edge between two vertices with a source and destination. Because the vertices in the constraint graph are connected using directed edges, the graph is also called a *directed graph*. Each vertex in the graph represents a variable or a task. Edges define the constraints for the variables or tasks. Each edge will have a value attached to it, called an edge weight. This edge weight denotes a minimum distance constraint that a variable or task has to satisfy to go from vertex at the source of the edge to the destination of the edge. To differentiate the edges that go in different directions between the same two vertices, positive and negative edge weights can be used.

Consider the graph shown in figure 1.2. Each vertex represents a task and the edge weights represent the *minimum* delay between the tasks. With the word *minimum* as the key, the start times of these tasks (t_{v_1}, t_{v_2}, t_{v_3} and t_{v_4}) depend on the edge weights between the vertices. For instance, the constraints for the start times can be written as, ($t_{v_2} \geq t_{v_1} + w_{12}$) and ($t_{v_1} \geq t_{v_2} - w_{21}$). In this graph, the edge with weight w_{12} gives distance between vertex v_1 and v_2, but to show the distance between vertices 2 and 1, $-w_{21}$, a negative edge weight is used to differentiate the direction of the edge.

1.3 Review of online scheduling algorithms

Scheduling is a process to decide when a certain task will be performed at certain resource in a system. Scheduling is needed to improve Quality of Service parameters, such as to increase throughput (total amount of work completed per unit time), to decrease response time (time to begin work after resources are being assigned), to decrease latency (time to complete work), etc. Achieving one of these quality of service targets often conflict with each other, so, the main task of the scheduler is to find a compromise between them and yet get better performance from the system. There are many scheduling algorithms that can be used depending on the requirement of the applications, such as First In First Out, Round robin, etc. [12]

Online scheduling is a method where the algorithm has to make scheduling decisions without complete information of the tasks to perform. In the case of the production printers, it is actually not the lack of information, but the chance of new information that might come into the print queue as part of new print jobs. Therefore, a scheduler for a production printer need to satisfy two things to be called an online scheduler. First, the scheduler should be able to compute schedules at a pace faster

than the printer's printing. Only then the printer can actually output the sheets at regular intervals. Second requirement is that, the scheduler should be able to consider information about new print jobs joining the queue while making its scheduling decisions. In both cases, a delay can lead to delay in completing the job. So for the scheduler proposed in Waqas et al. [9] to be used in the real world printer, it needs to satisfy both these conditions.

1.4 Re-entrant flowshop problem

A scheduling problem is a **flowshop** type problem, when there are m machines in a fixed sequence and jobs can either be executed in exactly same sequence of machines or not. As there are m machines, a job is also divided into m operations, where each job is executed at all the machines, an operation at each machine. Irrespective of the sequence of jobs executed at a machine, the sequence of machines and the sequence of operations should be same and to finish a job, the operation sequence should be maintained. Each machine executes same operation belonging to different jobs, one job after the other. As the sequence order has to be maintained, once a machine starts executing an operation, preemptions to the executing operation are not allowed [1]. The machine can begin next operation in the sequence only if the previous operation has finished executing completely. A typical flowshop example can be a water bottling plant unit with 3 machines as shown in figure 1.3.

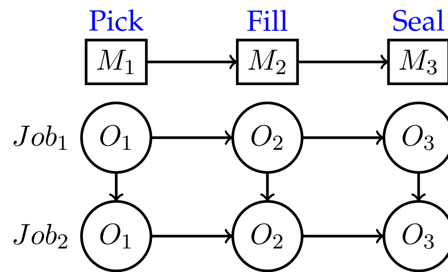


FIGURE 1.3: A water bottling unit as an example to show typical flow shop problem.

The above flowshop problem deals with a water bottling unit with three machines are $\{M_1, M_2, M_3\}$ each performing one task {Pick, Fill, Seal} respectively. Two jobs are shown Job_1 and Job_2, each represent bottling of a bottle.

The edges represent the order of operations. In the same job, the order of operations are O_1, O_2 and O_3 in that order. In a machine the sequence of operations will be [(Job_1, O_1), (Job_2, O_1)] on machine M_1, [(Job_1, O_2), (Job_2, O_2)] on machine M_2 and [(Job_1, O_3), (Job_2, O_3)] on machine M_3 respectively.

Re-entrant flow shop problem: As shown above, in a typical flowshop, the job moves forward over the machines and do not re-enter any machines again. Not all problems can be solved in this manner. There are many real world systems, where to complete a job, more than one sub-tasks need to be processed in a same machine. In such cases, a the flowshop problem is modified so that there are more operations when compared to number of machines and some machines execute sub-tasks more than once on a job. Because the job re-enters into a machine, these type of problems are called re-entrant flowshop problem. Figure 1.4 shows a machine with three machines. But in this system, unlike a typical flowshop problem, to complete the job, the second machine need to do a sub-task twice, shown by operations o_2 and o_3. The order of operations in each job will be, o_1, o_2, o_3 and o_4. Only in machine M_2, because there are two operations executed per job, the order of operations for a job is important. In each job at machine M_2, first operation o_2 will be executed and only after that operation o_3 will be executed.

There are many real world problems which can be solved using the re-entrant flowshop model and one of them is the production printing system.

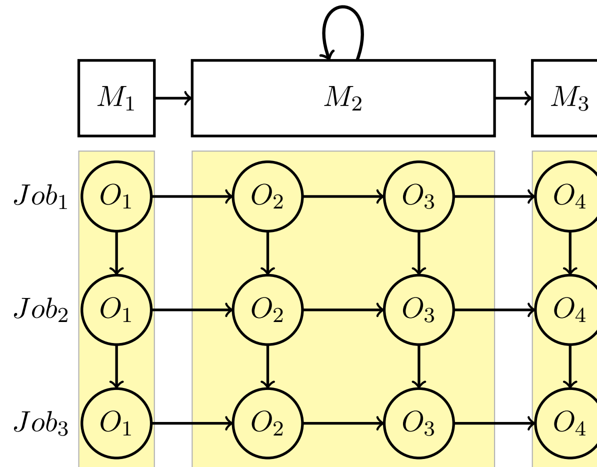


FIGURE 1.4: The operation to machine mapping of a job in a 3 machine re-entrant flowshop problem.

1.5 Path scheduling algorithms

Solving the scheduling of jobs on a production printer as a *re-entrant* flowshop problem provides proper sequence as a solution to the operations re-entering into the same machine from different jobs. For example, operations o_2 and o_3 of all jobs Job_1 to Job_3 executed in Image Transfer, shown in figure 1.4. But as discussed in the beginning of this chapter, to complete the solution, the sequencing need to be validated and if valid, the sequence of operations need to be given their start times. This is referred to as scheduling.

When there are many jobs and every job re-enters into one of the machines, the *re-entrant* flowshop problem can given many sequencing solutions. For each sequence, the scheduling algorithm should be able to do following things.

1. Assign all operations are with a start time, beginning with first job's first operation starting at t=0.
2. Check if the constraints on the sequence of operations is maintained to execute on each machine. Invalid sequences should be discovered and eliminated.

   In case of the production printers, detecting invalid sequences is important because they will cause sheet collisions inside the printer. Therefore, such sequences need to eliminated.

The path scheduling algorithm apart from being able to do the above two things, it should also be able to handle constraint graphs which have edges of both polarities, positive and negative. Bellman-Ford algorithm is one such algorithm which apart from being able to solve these two problems, can also handle constraint graphs with both positive and negative edges. Therefore the heuristic scheduler [9] uses Bellman-Ford algorithm to validate the sequences and schedule the operations of the print jobs. Bellman-Ford algorithm along with an example is explained in the section 1.6.

1.6 Bellman-Ford algorithm

Bellman-Ford algorithm is one of the popular Single Source Shortest Path (SSSP) algorithms. To find the shortest path in a graph G(V,E), where V is set of vertices and E is the set of edges, from source

vertex to all other vertices, Bellman-Ford algorithm works on the principle of *relaxing*. For example, consider the graph shown in figure 1.5.

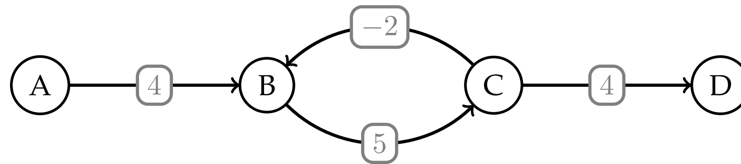


FIGURE 1.5: Constraint graph to explain the Bellman-Ford algorithm.

Before beginning with relaxing, first step of the algorithm is to initialize the first distances of all vertices. Since the distance is computed from the source vertex, it assigned a distance 0. In graph shown in figure 1.5, assuming vertex A as the source vertex, distance of vertex A, d_A is set to 0. The Bellman-Ford algorithm need to compute the distance to remaining vertices. So, the distance of other vertices is set to infinity (the maximum value possible). Therefore, $d_B = \infty$, $d_C = \infty$ and $d_D = \infty$.

Step I "The algorithm initializes the first distance of source vertex to 0 and all other vertices to ∞ (maximum possible value)."

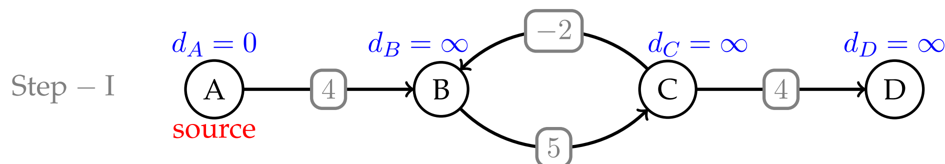


FIGURE 1.6: Constraint graph after initializing the distances of the vertices.

After initializing, relaxing of the edges is the next step. An edge is relaxed if it is proved that the vertex at the destination end can be reached earlier through this edge. Therefore, relaxing is a conditional computation needed to calculate the distances of vertices based on the edges to the vertices and their edge weights.

Relaxing can be explained as, "If for an edge (u,v), beginning at vertex u and ending at v, with weight w_{uv}, relaxing computes the distance of the vertex v, based on the condition that, if initial assigned distance d_v of v is greater than the sum of d_u, distance of the vertex u and the edge weight w_{uv} of edge from vertex u to v. If the condition ($d_v > d_u + w_{uv}$) turns out to be true, then, vertex v's distance d_v is set to ($d_u + w_{uv}$)".

Step II "Relax all the edges in the graph in each iteration for (n-1) iterations."

An example for relaxing is explained in figure 1.6. The edge between the vertices A and B is relaxed based on the condition ($d_B > d_A + 4$). Because d_A is 0 and the condition becomes true, which means vertex B has a shortest path from vertex A. Therefore, the value of d_B is changed to $d_A + 4 = 0 + 4 = 4$.

"Depending on the order in which the algorithm relaxes the edges, the outcome at the end of an iteration can change."

Figure 1.8 shows the distances computed in the first iteration after all the edges in the graph are relaxed once with the order shown in figure using the numbers (i,ii,iii and iv). If a vertex's distance is

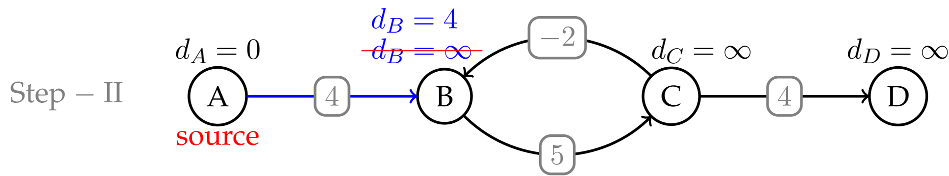


FIGURE 1.7: Constraint graph after relaxing the edge from vertex A to B.

changed multiple times, it is shown by listing how the distance changed, in order of edges relaxed, from *bottom to top* with the corresponding edge that caused the change in parenthesis beside the distance. In this order, after relaxing all edges once, the distances of vertices A, B, C and D are 0, 1, 6 and 10 respectively.

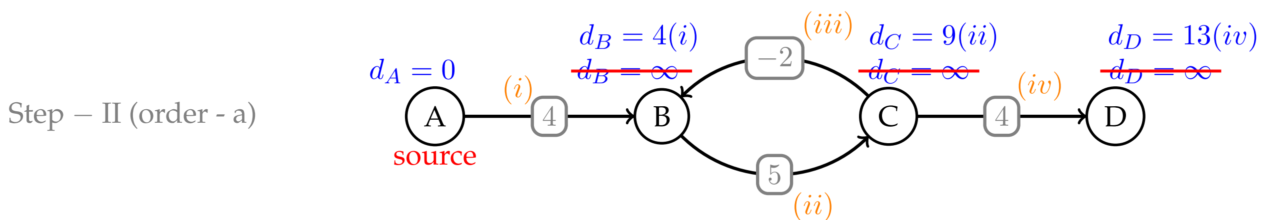


FIGURE 1.8: Constraint graph after relaxing all the edges in 1st iteration.

Figure 1.9 shows the outcome of relaxing all the edges once, if the edges are relaxed in a different order.

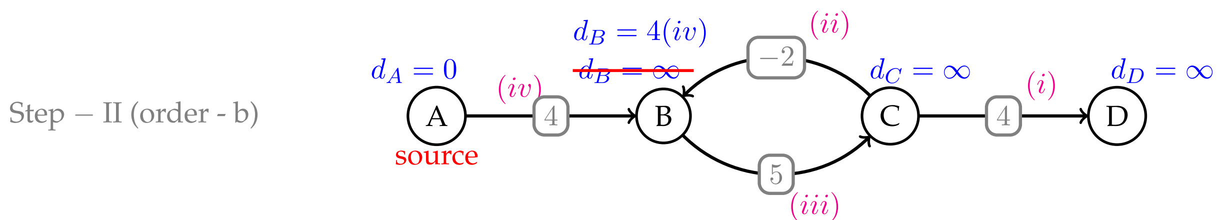


FIGURE 1.9: Constraint graph after relaxing all the edges in a different order in 1st iteration.

In this order, after relaxing all edges once, the distances of vertices A, B, C and D are 0, 4, ∞ and , ∞ respectively. The distance of vertices B and C did not change because the edges incoming to them are relaxed first and while relaxing, the vertices that these edges originated at still have their distances set to ∞. Any addition or subtraction to ∞ will not have any affect on the resulting value. Hence, these values remain same.

There are many such combinations of edges possible. The important point here is that, depending on the order in which the edges are relaxed, the distances computed in an iteration might change. But independent of this order, final distances of all vertices after required (n-1) iterations will have shortest possible values, i.e., all vertices will have shortest paths from the source vertex. Figure 1.10 shows that the distances of vertices computed after 3 iterations, irrespective of the relaxing order, will have same values.

Step - III “All the edges are relaxed for one final time after (n-1) iterations and if even one edge relaxes the distance of a vertex, it means there is a cycle in the graph. Using this the algorithm reports

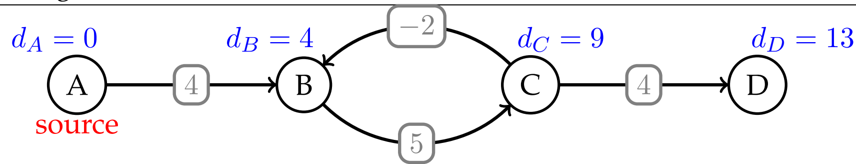


FIGURE 1.10: Constraint graph after relaxing all the edges in 2nd iteration.

if there is a cycle in the graph or not."

After relaxing all the edges in each iteration for (n-1) iterations, the Bellman-Ford algorithm can also report presence of negative cycles in the graph. A *cycle*, is set of edges which begin at a vertex v, traverse through the graph via one or more vertices and end at the same vertex v. This cycle becomes a *negative cycle* when the sum of edge weights of the edges in this cycle is negative. When there are negative cycles in the graph, shortest paths to the vertices in that cycle cannot be computed, even after relaxing the edges for (n-1) times.

For example, consider the same graph used in above figures, but with one change, the edge weights of edges (B,C) and (C,B) are swapped by keeping the polarity of these weights same. The graph is shown in figure 1.11. There is a cycle in the graph, which originates at B and ends at B traveling via vertex C. This is a negative cycle because the sum of edge weights is negative (w_{BC} + w_{CB} = 2 + (−5) = −3). There was a cycle prior to the swap too (in graph 1.5), but that cycle is not negative, because in that graph, sum of edge weight of cycle (B, C, B) is positive (w_{BC} + w_{CB} = 5 + (−2) = 3).

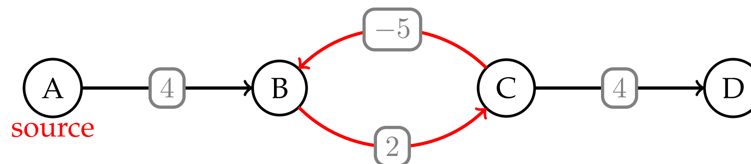


FIGURE 1.11: Constraint graph with a negative cycle.

After the first iteration of relaxing the edges in the graph 1.11, figure 1.12 shows the distances computed for each vertex.

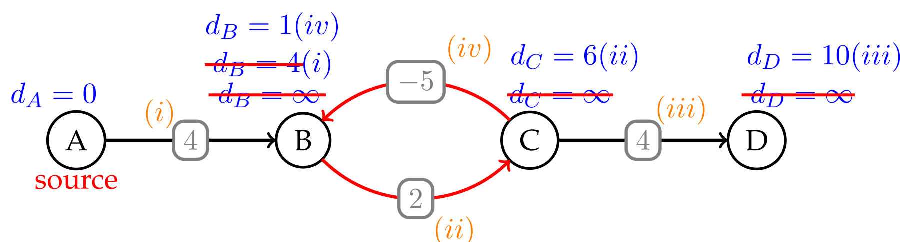


FIGURE 1.12: Constraint graph with a negative cycle, after 1st iteration of relaxing all the edges.

Continuing the relaxing, if all edges are relaxed for the second time, the result is shown in figure 1.13. Any following iterations of relaxing will continue reducing the distances of vertices B, C and D even after (n-1), i.e., 3 iterations. So, it is impossible to compute shortest distances to any of these vertices. Hence, after (n-1) iterations if any edge is successfully relaxed, then it means the graph has a cycle and the algorithm can report the same.

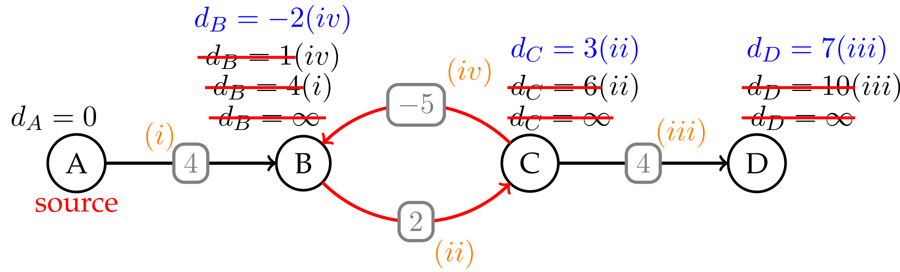


FIGURE 1.13: Constraint graph with a negative cycle, after 2nd iteration of relaxing all the edges.

The flow chart of the Bellman-Ford algorithm following all three steps is shown in figure 1.14.

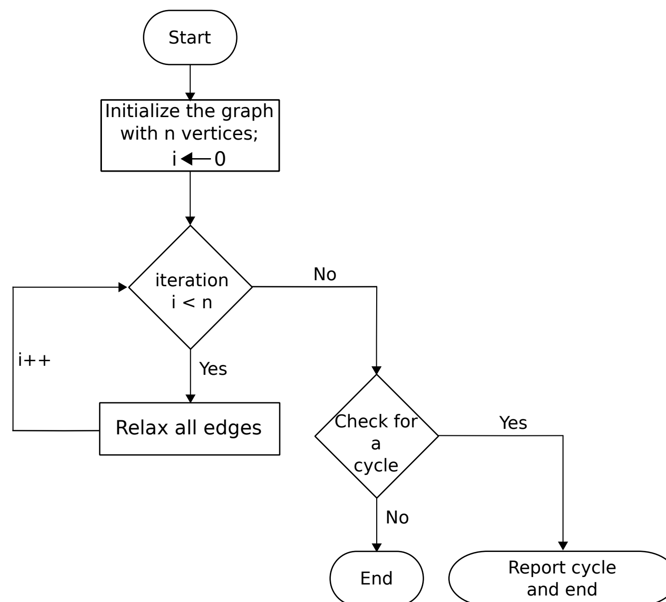


FIGURE 1.14: Flow chart of the Bellman-Ford algorithm. (adapted from [5])

The algorithms 1, 2 and 3 show the pseudo code for the steps I, II and III in that order of the Bellman-Ford algorithm, discussed above. These algorithms define the graph using G(V, E), V denotes set of vertices and E denotes set of edges.

To access a vertex, array notation is used, for example $V[0]$, $V[1]$, $V[k]$ denote vertices 1st, 2nd and $(k+1)^{th}$ vertices in the vertex set V. To access the distance of a vertex, dot notation is used, i.e., $V[1].distance$ gives the distance of 2nd vertex from the source vertex.

Similarly, an edge has the information about where the edge starts, where the edge ends and the edge weight. If e is an edge, $e.src$ gives the vertex where the edge e begins, $e.dst$ gives the vertex where the edge e ends and $e.weight$ gives the edge $e's$ weight.

The algorithm 1 gives the pseudo code to initialize the first distances to each vertex in the graph. The algorithm 2 shows the pseudo code for the relaxing all edges for one time in an iteration, a (part of Step-II) of the Bellman-Ford algorithm. The algorithm 3 checks if the graph has any cycles and reports the result. The algorithm 4 shows the pseudo code of the complete Bellman-Ford algorithm

Algorithm 1 Initializes first distances of all vertices in the graph. (adapted from [6])

```
1: procedure INITIALIZE(V, source)
2:
3:     //(Step - I) Initialize all vertex distances to a default value.
4:
5:     Create i ← 0
6:     for i < n do
7:
8:         if i = source then
9:             //source vertex distance is set to zero(0).
10:            V[source].distance ← 0
11:
12:        else
13:            //All other vertex distances are set to infinity(highest possible value).
14:            V[i].distance ← ∞
15:
16:        end if
17:        i ← i+1
18:    end for
19:
20: end procedure
```

Algorithm 2 Relaxes all edges in the graph *once*. (adapted from [6])

```
1: procedure RELAX(E, V)
2:
3:     //Relax all edges in the graph for one time.
4:     for all edges e ∈ E do
5:         if V[e.dst].distance > V[e.src].distance + e.weight then
6:             V[e.dst].distance ← V[e.src].distance + e.weight
7:         end if
8:     end for
9:
10: end procedure
```

shown in flowchart 1.14 using the algorithms 1, 2 and 3 which invokes all the methods of the three procedures which are shown in figure 1.14.

Algorithm 3 Checks if the graph has any cycles. (adapted from [6])

```
1: procedure NEGATIVE_CYCLE_CHECK(E, V)
2:
3:     //(Step - III) Relax all edges once and see if any vertex distance changes.
4:
5:     for all edges e ∈ E do
6:         if V[e.dst].distance > V[e.src].distance + e.weight then
7:             //If the above condition is true, it means there is a cycle in the graph.
8:             //This reports the presence of cycle.
9:             return true
10:        end if
11:    end for
12:
13:    //The algorithm reaches here only if there are no cycles.
14:    //This reports that there are no cycles in the graph.
15:    return false
16: end procedure
```

Algorithm 4 Complete Bellman-Ford algorithm using algorithms 1, 2 and 3

```
1: procedure BELLMAN_FORD_ALGORITHM
2:     //E is the list of edges, V is the list of vertices.
3:     Create E, V
4:
5:     //Before beginning relaxing, a source vertex should be defined.
6:     //source is set 0, because in arrow notation, 0 means the first element of the array.
7:     Create source ← 0
8:
9:     //Step - I
10:    //INITIALIZE sets the initial distance of vertices to a default value (shown in algorithm 1).
11:    INITIALIZE(V, source)
12:
13:    //Bellman-Ford algorithm iterates for n-1 times.
14:    Set i = 0;
15:
16:    //Step - II
17:    for i < n - 1 do
18:
19:        //RELAX, relaxes each edge in E once. (shown in algorithm 2).
20:        RELAX(E,V)
21:
22:        //increment the iteration variable by 1 to move to the next iteration.
23:        i = i+1
24:
25:    end for
26:
27:    //Step - III
28:    //Relax all the edges once again to check for the presence of a cycle (shown in algorithm 3).
29:    NEGATIVE_CYCLE_CHECK(E,V)
30:
31: end procedure
```

1.7 Contributions

Removed due to confidentiality.

1.8 Report overview

The first chapter introduces production printers and important components of a production printer, followed by explaining Constraint graphs and online scheduling algorithms. This chapter also explains how the production print scheduling problem is modeled as a flowshop problem and also how the path scheduling algorithms, especially the Bellman-Ford algorithm, are used by the scheduler to compute schedules for jobs. Chapter 2 explains the scheduling algorithm and the heuristics used to arrive at a scheduling solution. Chapter 3 defines the problem this thesis project aims to solve. Chapter 4 gives the related work studied about the optimizations that can help improve the runtime of the scheduler, concentrating on the Bellman-Ford algorithm. Chapter 5 discusses the optimizations implemented. Along with these optimizations, chapter 5 explains the modifications done to the heuristic scheduling algorithm that can reduce the scheduling runtime per sheet to less than 400ms. Finally chapter 6 concludes the thesis and proposes future work.

Chapter 2

Problem definition

Although the Waqas et al. [9] proposed heuristic scheduler gives better schedules than the scheduler currently in use[9], the main requirement for it to be used in the printer is that it should be an online scheduler. An online scheduler is a scheduler which can make scheduling decisions as and when it gets new information about the jobs to be scheduled and most importantly, it is not a bottleneck to the performance of the printer. A prototype printer considered for this project is assumed to be able to print upto 150 letter size sheets (300 sides) per minute, i.e., a sheet every 400ms. For the printer to be able to do so, the scheduler should compute schedules at least at the rate the printer can print sheets or perform even better. Keeping that in perspective, this chapter defines the problem statement of the graduation project.

The problem for this project is defined using a prototype printer proposed by Océ technologies. Currently, when the printer gets a print job, for example to print a book, or to print large number of pamphlet sheets, the scheduler has to compute the schedule before starting printing. But the proposed heuristic scheduler is taking a long time because it is scheduling all the sheets in the job. This means that, for any large job lined up, the printer needs to wait a long time to start printing. A delay in computing schedule will delay the start of print job which in turn affects time to complete the job.

To avoid this, a generic solution is proposed that will compute schedules for the sheets in the print job without any need for the printer to wait longer period of times. Because the prototype printer can print at the rate of 400ms per sheet, the goal of the project will be optimize the scheduler such that, scheduling time per sheet is less than 400ms.

Goal. Optimize the scheduler such that, scheduling time per sheet is less than 400ms.

As discussed before, in the worst case, this prototype printer can hold up to 92 sheets (of smallest sheet size). So, different optimizations are aimed to reduce the runtime to schedule 92 sheets. But the scheduler's run time analysis for jobs with 92 sheets has shown that the scheduling runtime is too high, reaching almost 2.5s in worst cases. With this runtime profile, as the number of sheets to schedule increase, scheduling time increases. In order to reach the goal of scheduling runtime per sheet to be less than 400ms, first the jobs with 92 or less sheets are optimized to have a runtime less than 400ms. But since the approach taken to solve the scheduling problem is 'heuristic', runtime also depends on the heuristics chosen. So, a generic optimization is discussed in the end which will get the scheduling runtime to less than 400ms irrespective of the heuristics chosen or the number of sheets or the type of sheets in the job.

Note: Since the majority of test cases used to benchmark the scheduler (discussed in Appendix B) have 100 sheets in their print job description, the optimizations results also consider 100 sheets to benchmark instead of 92.

Aim. Improve runtime of the scheduler to less than 400ms for jobs with less than 100 sheets.

Chapter 3

Heuristic Production Printer Scheduler

As explained in section 1.4 of chapter 1, the scheduler models the scheduling of sheets in the production printer as a re-entrant flowshop problem and solves it using the constraint graphs. This chapter explains the heuristic approach taken by the scheduler. Section 2.1 explains how constraint graphs are used to model the scheduling the problem.

3.1 Production printer constraint graph model

Figure 1.1 shows the machines and the path a sheet needs to take to finish printing and come out of the printer. Since each job is assumed to be of duplex type, the order of machines is also given in section 1.1 of chapter 1. Figure 3.1 gives the order of machines in the production printer.

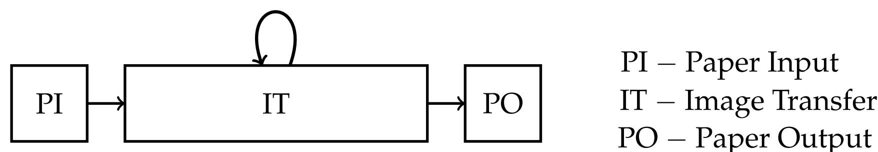


FIGURE 3.1: The order of machines in a production printer.

The order is from left to right, first, the sheet enters into the printer from Paper Input machine. Once entered, the sheet is moved to the printer Image Transfer machine where it is printed on and then the sheet is moved out of the printer on to a stacker via the Paper Output machine. Since every sheet is assumed to be a duplex sheet and therefore have to re-enter the machine to be printed on the second side, this re-entry into the machine is denoted using the *loop* on top of the Image Transfer machine. Because of the same reason, the Turn machine is not shown in the order. It will only be included as a delay between the first side and second side print operations.

To model the problem into a constraint graph, the model shown in figure 3.1 for machines is used. Paper Input and Paper Output machines do one operation each per sheet i.e., Paper Input inputs the sheet and Paper Output outputs the sheet. But the Image Transfer machine has execute two operations per sheet, each operation denotes a print on each side of the sheet. So, printing of one sheet can be shown as an ordered set of four operations, first operation is to input the sheet (o_1) at the Paper Input (PI), the second and third operations (o_2, o_3) show printing on each side of the sheet at Image Transfer (IT) and finally fourth operation (o_4) is to output of the printer at Paper Output (PO) machine. Figure 3.2 shows the operations as vertices and the edges between the operations enforce the order from first operation to the last.

Section 1.1 of chapter 1 explains different delays that can affect the start times of the operations at different machines. For an operation to begin executing at its machine, it should wait until its predecessor operation is executed along with the time that is needed for the machine to complete its setup. Predecessors can be recognized using the edges incoming to a vertex. So, each edge will have

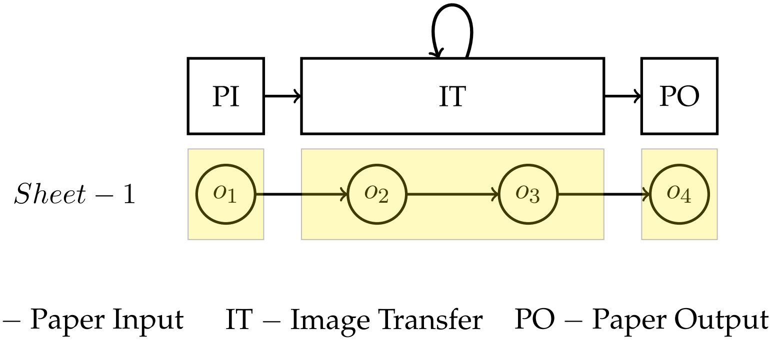


FIGURE 3.2: The constraint graph of a sheet to be printed.

a weight which is the sum of predecessor operation's processing time (t_p) and setup time (t_s) needed between the two operations, shown as (t_p + t_s). This edge is referred as a forward edge and gives a constraint for the minimum time that is needed .

There is also a maximum time that an operation can wait, called reverse due date (t_d). This is shown using an edge incoming to the operation but with negative weight. This edge is referred to as a reverse edge. Figure 3.3 shows the constraint graph of one sheet with both forward and reverse edges.

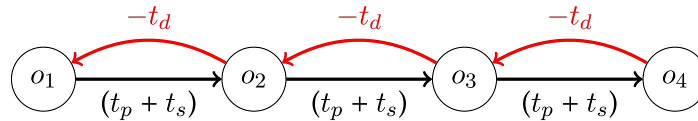


FIGURE 3.3: The constraint graph of a sheet to be printed.

For example, the forward edge constraint between operations o_1 and o_2 in figure 3.3 can be written as, $t_{o_2} \geq t_{o_1}$ + (t_p + t_s). t_{o_1} and t_{o_2} are the start times of operations o_1 and o_2, t_p is the time needed to process operation o_1 and t_s is the setup time between the operations o_1 and o_2. This constraint gives the minimum time operation o_2 needs to wait before it can begin after o_1 has started executing. The reverse edge constraint between operations o_2 and o_1 can be written as, $t_{o_1} \geq t_{o_2} - t_d$. This gives the maximum time that o_1 can wait before it needs to begin its execution.

The scheduling algorithm computes start times of operations of all sheets. The set of start times of all these operations is referred to as a *schedule*.

Figure 3.2 and 3.3 show the constraint graph for one sheet. When there are multiple sheets to be printed, the graph should not only need constraints to show the order of operations, additional constraints are needed to enforce the order of sheets. Extra set of edges are used to join similar operations of different sheets to enforce these constraints. To denote the constraints between Figure 3.4 shows the constraint graph with multiple sheets.

The operations that are to be executed on a machine from each sheet are shown in blocks and operations in each block are referred to as a *group* (shown as G_1, G_2 and G_3 in figure 3.4).

The group of vertices G_1 and G_3 are mapped to machines Paper Input (PI) and Paper Output (PO) and need no sequencing because there is only one operation per sheet that is mapped to these machines and because these sheets are ordered, the corresponding operations are also ordered.

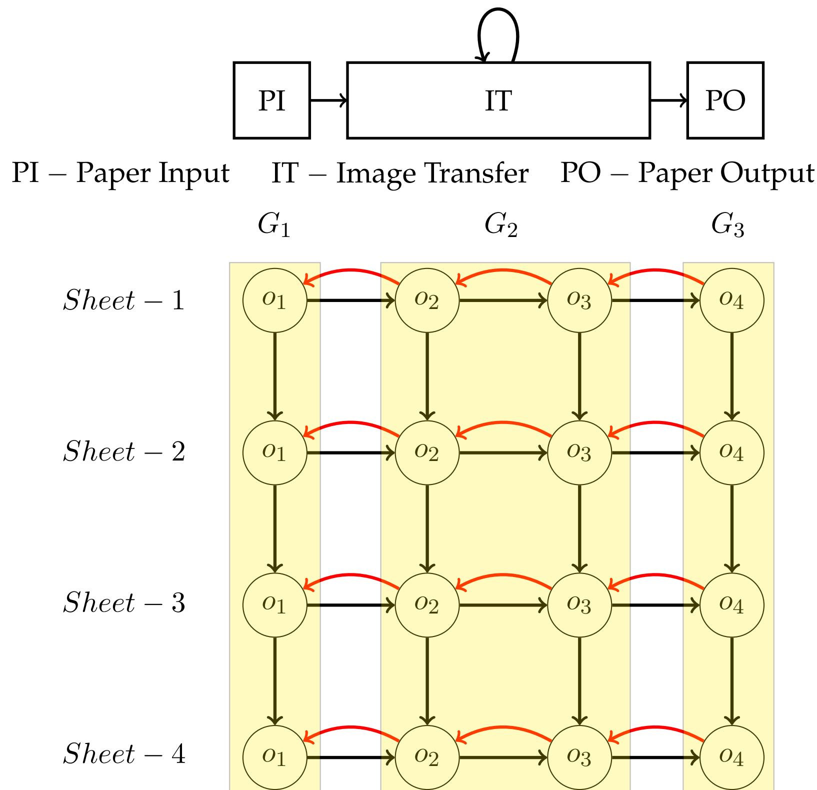


FIGURE 3.4: Constraint graph with multiple sheets.

The remaining group is G_2. The operations of this group are mapped to Image Transfer machine. Unlike the other groups, this group has two operations from each sheet. So, this provides the scheduler an opportunity to re-order the operations so that *makespan* is kept minimal.

All the edges shown in figure 3.4 need to be satisfied, if not the computed schedule will be invalid and will lead to errors during printing. Therefore, these constraints on these edges form *compulsory constraints*, denoted by E_C.

With these compulsory constraints, the *default ordering* of the operations will be, o_2, o_3 of a sheet will execute and only after these two operations are finished, next sheet's o_2, o_3 can start. This order goes from first sheet to last sheet. But, because the printer can hold a *buffer* of sheets before it can begin printing second side prints of sheets, as long as the compulsory constraints are satisfied, there can be other sequences too. To find these orderings in the operations in group G_2, the scheduling algorithm uses extra constraints. These constraints are shown using dotted edge in figure 3.5 and will represent the amount of time that each sheet can take before it can begin its second side print. Scheduler can find a different ordering for the operations in group G_2 using these dotted edge constraints which when compared to the default ordering can result in a lesser makespan. Since the constraints on these edges are not compulsorily needed to be satisfied and are only used when necessary, these set of edges are referred to as optional constraints, denoted by E_O.

In the printer paper path shown in figure 1.1, the only place the scheduling algorithm can make decision for a different ordering is at the *merge point*. The decision that the scheduler needs to take is, between which two sheets' print operations can a different sheet coming from re-entrant path can enter into main path P1 at the merge point and execute its print operation. This decision is called

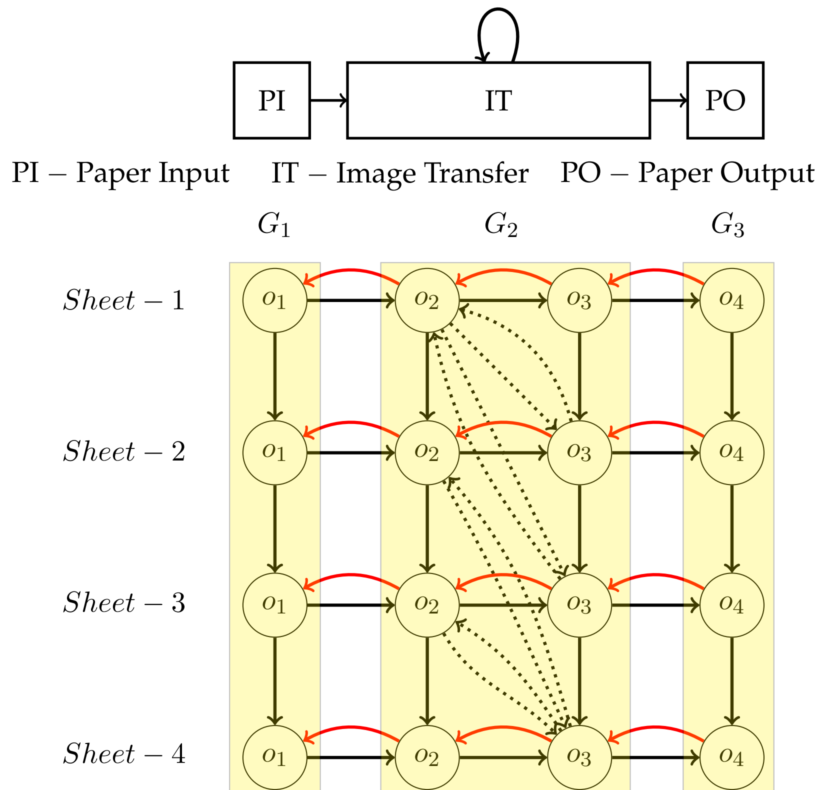


FIGURE 3.5: Constraint graph with multiple sheets.

an *interleaving*. Because this decision is made on a sheet that is coming from re-entrant path, an interleaving always refers to a decision, "between which two sheet's print operations, a sheet's *second print* can be executed".

For example, consider the four sheets shown in figure 3.6. For convenience, the constraint graph in figure 3.4 is rotated by 90 degrees and is shown in figure 3.6. In this figure, each column denotes a sheet. As explained above, the scheduler need to find an interleaving for all the sheets in the queue. In default ordering, each sheet is interleaved with the next sheet's first print. In an ordering which has different sequence of operations when compared to default case, the dotted lines show the interleavings. For example, the incoming and outgoing edges from operation o_3 of $Sheet-1$ show that, the second print of $Sheet-1$'s second print operation goes in between $Sheet-2$ and $Sheet-3$ instead of itself and $Sheet-2$'s first print operations. Similarly the $Sheet-2$'s second print operation goes between $Sheet-3$ and $Sheet-4$'s first print operations.

An interleaving is an ordered tuple, $ot((a,b),(b,\ c))$ where a, b, and c denote vertices. The ordered tuple $ot((a,b),(b,\ c))$ also means that the operation denoted by vertex b is executed after a is executed and before c is executed. An interleaving is said to *feasible* if the ordering does not create a cycles in the compulsory constraint graph. Only after an interleaving is found to be feasible, the edges of this interleaving between vertices (a, b) and (b, c) are added to compulsory constraint edges E_C.

The *scheduling problem* is to find the feasible set of ordering tuples in OT such that all the vertices in all the groups which have multiple operations per job mapped to a same machine are ordered such that the *makespan* is kept minimal.

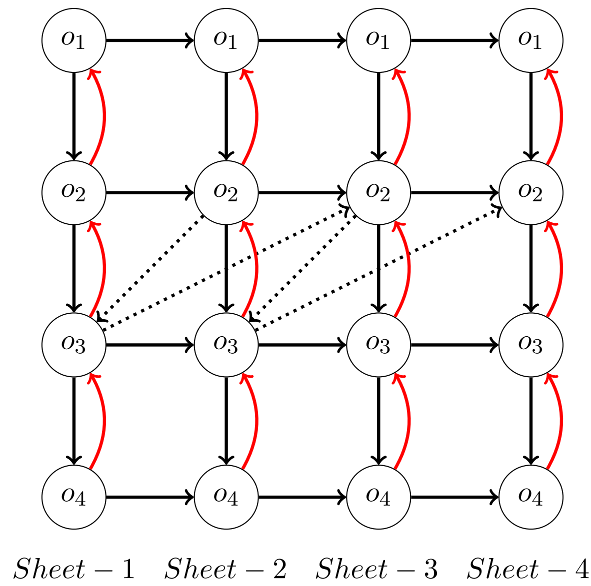


FIGURE 3.6: Constraint graph example with sheets after interleaving.

There are three types of interleavings possible to schedule a sheet in queue.

Regular interleaving

This interleaving denotes a sheet's ordering between "first prints" of two other sheets in the queue. This is shown in figure 3.7. It will be referred to as an *interleaving* from here on in the report.

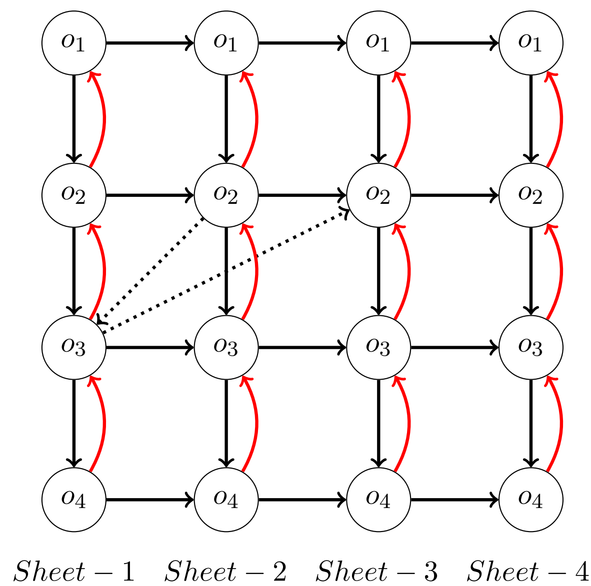


FIGURE 3.7: Constraint graph example with *regular* interleaving.

Exceptional interleaving

This refers to an interleaving when a sheet's second print operation is interleaved with first and second print operations of a sheet in the queue. This is different from the above discussed regular interleaving because this interleaving is only with one sheet (its first and second print operations), instead of two different sheets' first print operations. In example shown in figure 3.8, because of the exceptional interleaving the Sheet-3 can begin printing on its first side only after Sheet-2 finishes its second side print.

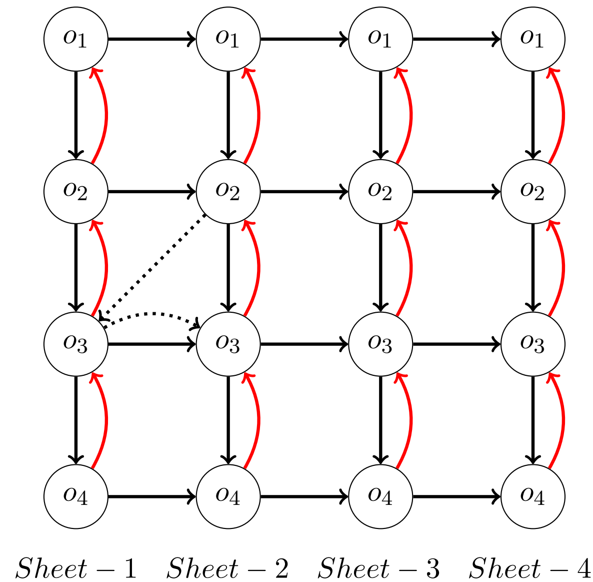


FIGURE 3.8: Constraint graph example with *exceptional* interleaving.

Flush interleaving

This interleaving, as its name suggests will flush the printer buffer before next sheet starts its first print operation. In example shown in figure 3.9, because of the flush interleaving the Sheet-4 can begin printing on its first side only after Sheet-3 finishes its second side print.

"Because the two interleavings, exceptional and flush are used to clear the printer buffer, these two are time expensive interleavings."

Use of Bellman-Ford algorithm to compute start times

The Bellman-Ford algorithm explained in section 1.6 computes the shortest paths to the vertices from the source vertex. But for the production printer scheduler, as explained, above, all compulsory constraints need to be satisfied before an operation can start its execution. This means that if an operation has two incoming edges that belong to E_C, then constraints on both edges need to be satisfied, i.e., the operation can only start its execution after both the operations on the other side of the incoming edges have finished their execution. This means, instead of computing shortest path, to compute the start times, *longest path* need to be computed.

"The source to destination longest path, can be computed by inverting the signs of edge weights on the constraint graph."

For simplicity, all the optimizations are explained on the algorithm that computes the shortest paths, but just by changing the sign of the edges, the same optimizations will work to compute the longest path.

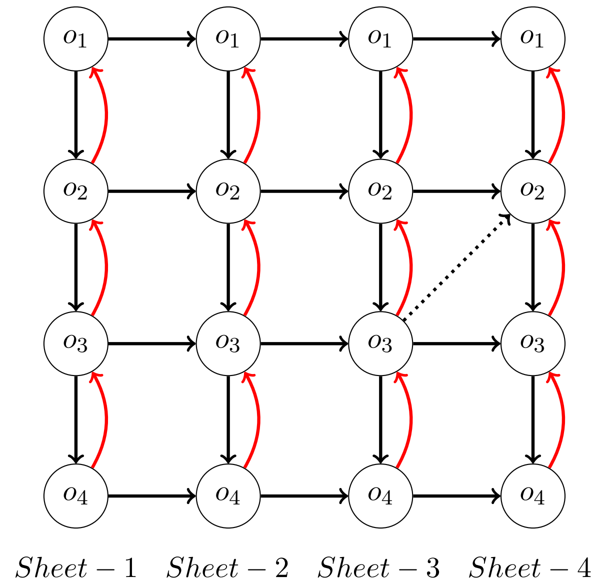


FIGURE 3.9: Constraint graph example with *flush* interleaving.

3.2 Heuristic approach

Heuristic approach is a *practical way of exploring solution space* for a problem. The scheduler's problem here is to explore the different solutions of re-ordering of operations on machines which have more than one operation per job mapped to them in order to minimize the makespan.

As explained in section 1.1 of chapter 1, the scheduler needs to decide between which sheets does the re-entering sheet enter the main path P1 and is printed on its second side. In flowshop and constraint graph terminology, an ordering tuple need to computed for each re-entering operation, say ((a,b), (b,c)), where b denotes the re-entering operation, and a and c represents the two operations between which b is executed. For such a re-entering operations there can be more than one valid ordering tuple and the makespan of the final schedule depends on how the heuristic algorithm selects a final ordering tuple. The heuristic approach that Waqas et al. [9] presents is a generic algorithm which will compute schedule for constraint graphs of re-entrant flowshop type problems. The production printer scheduling is one such flowshop problem with one re-entrant machine. Since the problem has only one re-entrant machine there is only one group for the scheduler to re-order.

Creating interleaving solution space

The algorithm has to compute the ordered tuples for each re-entering operation, called interleaving and is denoted by OT^{sol}. To do so, first the algorithm needs a solution space to explore the interleaving solutions. There are two ways to compute interleaving solution space for re-entering operations.

1. Forward interleaving

Interleavings are computed from first sheet to last sheet in that order. Figure 3.10 gives an example of forward interleaving. In the example, interleaving begins from $Sheet - 1$ and figure shows that $Sheet - 1$ is interleaved with $Sheet - 3$ and $Sheet - 4$. So, ordered tuple for $Sheet - 1$ can be written as (($Sheet - 3$, $Sheet - 1$), ($Sheet - 1$, $Sheet - 4$)). This means the $Sheet - 1$ can execute its second print after the first side of $Sheet - 3$ has finished its execution and before $Sheet - 4$ has started its first side print.

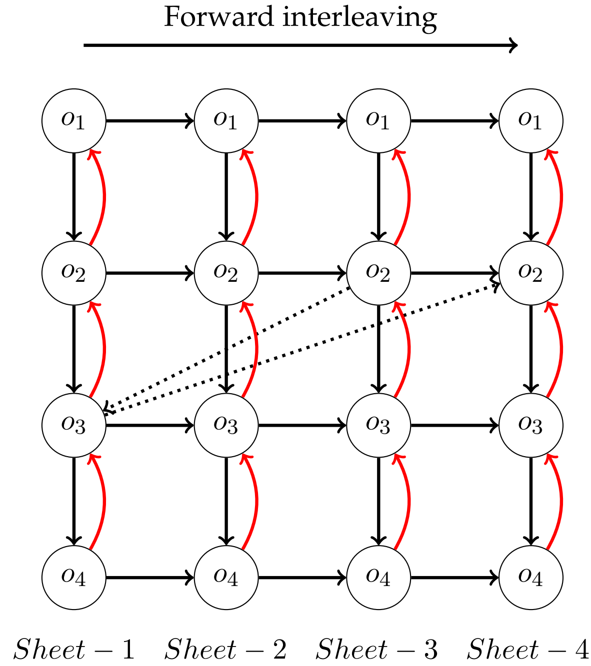


FIGURE 3.10: Constraint graph example with forward interleaving.

2. Reverse interleaving

In reverse interleaving, interleavings are computed from last sheet to first sheet. In the example, the interleaving begins at $Sheet - 4$. But since it is the last sheet and there are no sheets after it in the queue, there is no interleaving needed for the last sheet. So, interleaving begins at the pen-ultimate sheet, $Sheet - 3$. $Sheet - 3$ has only two interleavings possible, either the exceptional interleaving or flush interleaving. $Sheet - 2$ now has all three types of interleavings possible and so can $Sheet - 1$. Figure 3.11 shows both $Sheet - 2$ and $Sheet - 3$ with flush interleaving, i.e., $Sheet - 3$ can begin its first side print only after $Sheet - 2$ has finished its both side prints and $Sheet - 4$ can begin its first side print only after $Sheet - 3$ has finished its both side prints.

Having an option to flush when interleaving a sheet is preferred because it will act like a fall-back in case of any errors during scheduling. In forward interleaving, shown in example 3.10, when the scheduler tries to interleave $Sheet - 2$'s second print operation, we can see that $Sheet - 2$ has no option to flush because $Sheet - 1$ is interleaved between $Sheet - 3$ and $Sheet - 4$. But, as shown in figure 3.11, with reverse interleaving it is always possible to have this option to flush for every sheet. "Therefore, if reverse interleaving is used, it is a definite possibility that every sheet will have a valid interleaving choice."

So, the production printer scheduler that's discussed below will use the reverse interleaving technique to create the interleaving solution space.

Heuristic metrics

The scheduler takes a heuristic approach by using three metrics to understand the effect of an ordering tuple on the makespan. They are, *productivity*, *flexibility* and *distance*.

Metric 1. *Productivity* of an ordering tuple measures the impact of the ordering tuple on the makespan. To calculate the productivity metric of the ordering tuple ((a,b),(b,c)), first the scheduler computes the

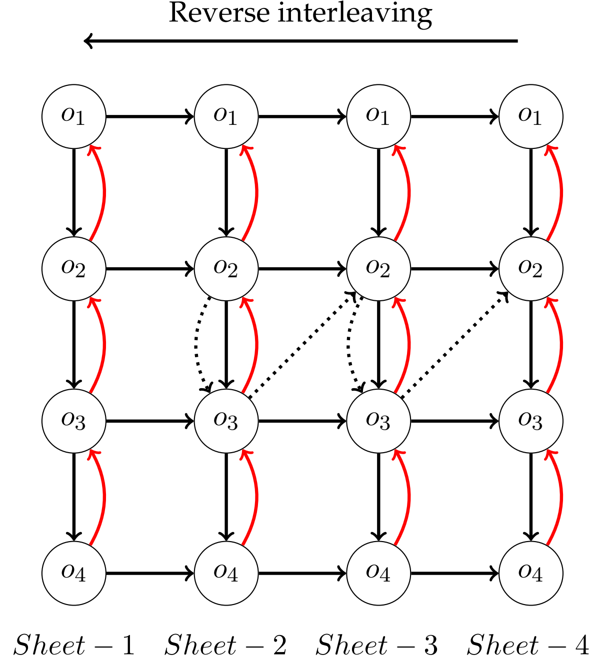


FIGURE 3.11: Constraint graph example with reverse interleaving.

maximum possible increase in start time of the operation c (denoted by $d_{x,b}$), that can be caused by one of the ordering tuples of b (denoted by OT_b^x), where x and b denote the group and vertex which are currently being ordered.

$$d_{x,b} = \max_{ot \in OT_b^x} \max\left(t_c, \max\left(t_b, t_a + w(a,b)\right) + w(b,c)\right)$$

Productivity metric denotes how restrictive the ordering tuple is. So, P_{ot} = 0 denotes a highly productive ordering tuple. The productivity of an ordering tuple can be given by the equation :

$$P_{ot} = \frac{\max\left(t_c, \max\left(t_b, t_a + w(a,b)\right) + w(b,c)\right) - t_c}{d_{x,b}}$$

Metric 2. The *flexibility* of an ordering tuple ((a,b),(b,c)), measures the effect of edges over the due date constraints of vertex b. If D_b contains all the negative edges going out of vertex b, the flexibility is given as,

$$F_{ot} = \min_{(b,x) \in D_b} \frac{max(t_b, t_a + w(a,b)) - t_b}{-w(b, b-1) - w(b-1, b)}$$

The above equation F_{ot} denotes the ratio of excess usage of the time by the ordering tuple to the total allowed time by the due date constraint. The terms t_b and t_x are the start time of the vertices b and x, computed from the partial solution (the compulsory constraint set E_C and OT^{sol} at that point of time).

Metric 3. *Distance* denotes the length of interleaving of the vertex c. It is given by the following formula.

$$D_{ot} = \frac{|V| - dist_c}{|V|}$$

where |V| denote the number of vertices and $dist_c$ denote the minimum number of edges from source vertex v_s to c.

The *weight* of an ordering tuple is computed by weighted sum of all the three metrics calculated for each ordering tuple. It is given by the following equation.

$$R_{ot} = \kappa_P \times P_{ot} + \kappa_F \times F_{ot} + \kappa_D T \times D_{ot}$$

The relative weights, κ_P, κ_F and $\kappa_D T$ indicate the relative importance to the metrics and sum upto 1.

"The heuristics are modeled in such a way that, as the weight of an interleaving option increases, its rank decreases. Therefore, lower the weight, higher the rank."

The complete heuristic approach is explained in algorithm 5.

The Heuristic Production Printer Scheduling algorithm

The algorithm as explained above uses reverse interleaving, so the interleaving begins at the penultimate sheet in the queue. The algorithm is explained in steps below.

Step-1: The algorithm selects a sheet's second side print operation and computes all its valid interleavings.

Step-2: All the valid interleavings are weighted and ranked using the metrics mentioned in section 2.2. The interleaving with lowest weight will be given highest rank.

Step-3: Check the feasibility of the interleavings in the order of their ranking.

Step-4: Highest ranked feasible interleaving will be selected and its edges are added to compulsory edges set E_C and the interleaving will be added to the final set of ordered tuples OT^{sol}.

While computing each sheet's interleaving the schedule (start times) of the operations of sheets in the queue will be computed and will be used during ranking of the interleavings. This is denoted by T in the algorithm.

Final step: The algorithm finally returns the schedule after all sheets are interleaved.

Algorithm 5 Heuristic approach to re-order the vertices and compute their schedule. (from Waqas et al. [9])

```
1: procedure HEURISTIC_SCHEDULER(set of first print ops V_x, set of second print ops V_y)
2:     OT^sol = ϕ
3:     T = ϕ
4:     for all second print operations o_3 in V_y do
5:         compute and update T with start times of operations in the graph
6:         //Step-1
7:         compute set of valid interleavings for o_3
8:         //Step-2
9:         compute weight and rank all valid interleavings
10:        //Step-3
11:        check feasibility interleavings
12:        //Step-4
13:        add the edges of this interleaving to E_C
14:        add the interleaving to OT^sol
15:    end for
16:    //Final step
17:    return T
18: end procedure
```

Chapter 4

Related work

In this chapter different optimizations are studied which will help accelerate the scheduler at its bottlenecks. As it will be explained in the next chapter (chapter 5) the major bottleneck of the heuristic production printer scheduling algorithm is the Bellman-Ford algorithm. Therefore, the related work will concentrate in the literature that studies different Bellman-Ford algorithm optimizations and acceleration methods.

4.1 Optimizing the Bellman-Ford algorithm

The Bellman-Ford algorithm is a shortest path computing algorithm. Depending on the graph for which the shortest path is computed for, the first step is to study the amount of work that the algorithm actually need to do to compute the shortest path of the graph. Not all graphs need all the (n-1) iterations of edge relaxing to compute the shortest path from source to destination. A basic optimization described in *Improvements* section in [10] uses this idea and suggests that the algorithm can be immediately terminated if in two consecutive iterations, none of the vertex distances change. Implementing this and studying the results also gives an idea about the complexity of the graph and will give a basic idea on how much work is actually being done by the algorithm to compute shortest path distances. The results from this experiment can be used to set targets for further optimizations to achieve.

Many literature works use the way the Bellman-Ford algorithm relaxes the distances of the vertices to their advantage. Because relaxing an edge does not depend on the other edges, as long as the conflicts of memory writes are handled, as the number of the edges increase, they can be relaxed in parallel with proper synchronization.

The implementation of Cohen and Dallas [3] is using Adjacent matrix to represent the graph. The implementation parallelizes the graph by dividing "evenly" between the threads. Even though the word *evenly* does not give any specific information on how the graphs are divided (by the area of graph spread of graph or by the number of vertices), the single core pseudo code seems to suggests that the vertices might be divided equally in number between the threads. The experiments are conducted on 1, 2, 4 and 8 processors to compute the shortest path for different graphs with different vertex degrees. Two graph configurations G3 and G4 implemented in [3] are very similar to the constraint graphs generated for the production printer. The graphs of G3 and G4 are made of 2000 and 4000 vertices respectively with a vertex degree of 3 and the production printer constraint graphs with maximum vertex degree of 3. So, from work done in [3] the results of G3 and G4 explain the behavior of Bellman-Ford algorithm on the multi core system. The results show that,

(a) The total speedup is less 2.5,

(b) As the number of processors increases from 4 to 8, the speedup actually decreases.

The requirement in the case of the production printers is to obtain a speedup more than 11.

A parallel implementation discussed in [2] uses a GPU to parallelize the Bellman-Ford algorithm and uses a frontier queue based vertex relaxing method. In this method, multiple edges outgoing from different vertices are relaxed in parallel on a Graphics Processing Unit. Whenever a vertex's distance is changed because of a successful edge relaxation, that particular vertex is then added to a queue. Only the edges outgoing from these vertices are then relaxed which propagates the shortest path computation from one modified vertex to the next. This explains the way the total work is restricted to only those that affect the final distance and unnecessary edge relaxations are avoided, resulting in faster path computations.

4.2 Runtime optimizations

Apart from improving the runtime of the Bellman-Ford implementation, there are many other optimizations that can be possible in other parts of the heuristic scheduler that will have an effect on the final run time. First is to use "faster" sequence container types, vectors or lists depending on how and where they are being used. STL *Vector*s provide random access to its elements which can be used to make accessing edges, vertices and arrays that stored schedules from memory during relaxation algorithm by their position. This is very important in case of the path computing algorithms because of the random connections between vertices in the graph. Whereas STL *List* does not support random access and to reach a certain position in the list, an iterator is needed beginning from either end of the list which will consume more time [7].

Chapter 5

Research, implementations and results

Océ, the company at which this work is done, decided that the work should be kept confidential. Because this report is written as a public version, the optimizations are explained in detail only if they are directly used from already published literature and others are kept confidential. Such optimizations are explained in a separate confidential report.

This chapter explains the research work done to study the optimizations needed for the scheduler. The results obtained due to these optimizations are explained and analyzed.

Section 5.1 shows different runtime optimizations that are implemented on the heuristic scheduler and their results are explained. The results of each of the optimization are analyzed and conclusions are explained using which next optimization are motivated. The optimizations in section are on jobs with 100 or less sheets, therefore, section 5.2 explains a generic optimization which will optimize the scheduling runtime per sheet to less than 400ms irrespective of number of sheets in the print job.

5.1 Runtime optimizations & Results

This section lists the initial runtime optimizations for the scheduler. As explained during the problem definition (chapter 2), these mainly concentrate on optimizing the scheduler such that, the runtime to compute schedule for any job with 100 or less sheets is less than 400ms.

5.1.1 Avoiding invoking Bellman-Ford algorithm twice

Optimization technique

The algorithm 6 recaps the heuristic scheduling algorithm explained in algorithm 5 of chapter 3 with some extra details that are needed to understand the optimization that will be discussed later in this section.

Figure 5.1 shows a Gantt chart of the runtime profile of the scheduler when a 92 sheet job is scheduled. Looking at this Gantt chart, it can be observed that the scheduler has two bottlenecks. They are,

1. Computing and updating start times *which takes 51% of total execution*
   To rank the list of interleaving choices made for a particular sheet, the algorithm needs the start times of the sheets prior to the interleaving. This uses the Bellman-Ford algorithm to compute these start times.

2. check interleaving feasibility (which takes almost 47% of total execution)
   Before adding an interleaving's edges to the compulsory constraint edges set E_C, it should be checked if that particular interleaving will cause any cycles in the constraint graph. To check this, Bellman-Ford algorithm's capability of detecting cycles is used.

Algorithm 6 Heuristic approach to re-order the vertices and compute their schedule. (from Waqas et al. [9]) with appropriate comments.

```
1:  procedure HEURISTIC_SCHEDULER(set of first print ops V_x, set of second print ops V_y)
2:      OT^sol = φ
3:      T = φ
4:      for all second print operations o_3 in V_y do
5:          //computing start times uses the Bellman-Ford algorithm.
6:          compute T, start times of operations in the graph
7:          //Step-1
8:          compute set of valid interleavings for o_3
9:          //Step-2
10:         compute weight and rank all valid interleavings
11:         //Step-3
12:         //select the first feasible interleaving.
13:         //feasibility is checked using Bellman-Ford algorithm is used.
14:         select the first feasible interleaving
15:         //Step-4
16:         add the edges of this interleaving to E_C
17:         add the interleaving to OT^sol
18:     end for
19:     return T
20: end procedure
```

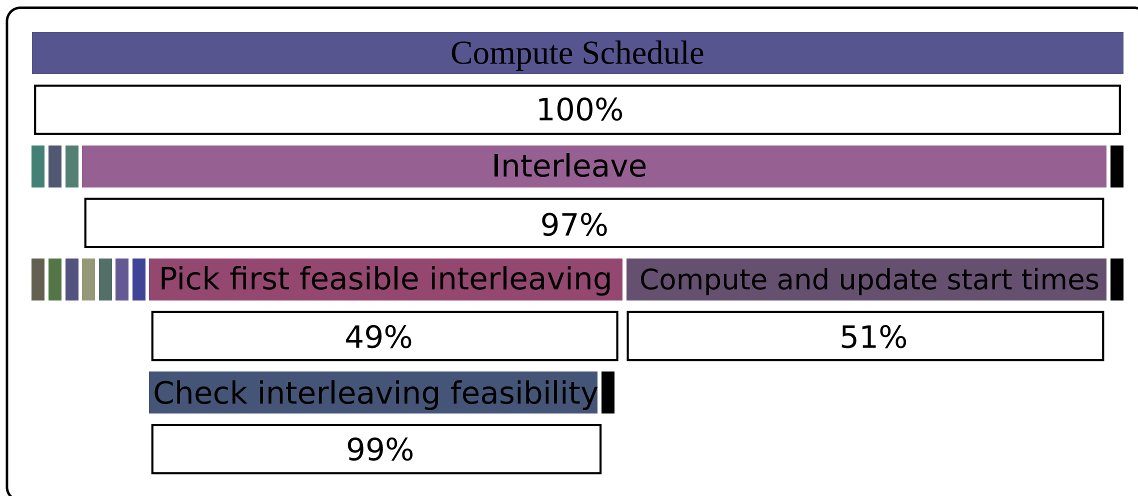


FIGURE 5.1: Gantt chart of the runtime profile of the Heuristic scheduler for scheduling 92 sheets.

Both to compute and update start times and to validate interleavings, same Bellman-Ford algorithm is used. To optimize the runtime, the algorithm is modified to use the Bellman-Ford algorithm only once and get both the outcomes needed.

Optimization Statement

"During the feasibility check that uses the Bellman-Ford algorithm, the start times will also be computed. As part of step 3, feasibility check ends when the algorithm finds the first feasible interleaving. After this feasibility check is done, in the next iteration start times are computed on the same graph.

So, instead of invoking the Bellman-Ford algorithm just to compute start times, while validating the interleaving, start times are also obtained from the Bellman-Ford algorithm."

Implementation

Modified algorithm is shown in algorithm 7.

Algorithm 7 Optimized heuristic approach to re-order the vertices and compute their schedule. (from Waqas et al. [9])

```
1:  procedure HEURISTIC_SCHEDULER(set of first print ops V_x, set of second print ops V_y)
2:      OT^sol = ϕ
3:      T = ϕ
4:
5:      //For the first time, T should be computed
6:      compute T for the initial graph.
7:
8:      for all second print operations o_3 in V_y do
9:          //Step-1
10:         compute set of valid interleavings for o_3
11:         //Step-2
12:         compute weight and rank all valid interleavings
13:         //Step-3
14:         //select the first feasible interleaving.
15:         //feasibility is checked using Bellman-Ford algorithm is used.
16:         //this also returns the start times of the graph.
17:         select the first feasible interleaving, also return computed T
18:         //Step-4
19:         add the edges of this interleaving to E_C
20:         add the interleaving to OT^sol
21:     end for
22:     return T
23: end procedure
```

Correctness Reasoning

Validating a choice is done on the list of sorted choices one after the other until a valid choice is found. As explained during 'reverse interleaving' in chapter 2, because finding a valid choice is a definite possibility for every sheet's second print, it will also be a definite possibility that 'T' will have valid values to rank the interleaving choices of next sheet's second print. Hence, the schedules computed after this optimization will be same as the schedules computed before the optimization.

Results and Analysis

Results after this implementation are shown in figure 5.2 which shows box plots of the runtime distributions.

The test set is explained in Appendix B and the platform used is *CPU-I*(configuration is given in the Appendix A).

As expected, this optimization removed one of the bottleneck's shown in figure 5.1 and a speedup of almost 2 times is observed after optimizing. The maximum speedups obtained in the runtime for different types of job types is given in table 5.1.

	Homogeneous(S)	Varying Thickness(T)	Varying Length (L)	Booklet-A (BA)	Booklet-B (BB)
after avoiding double call	1.7	1.7	1.6	1.7	1.8

TABLE 5.1: Maximum speedup obtained in the execution of Heuristic scheduler after the above discussed optimization is implemented in the Heuristic scheduler.

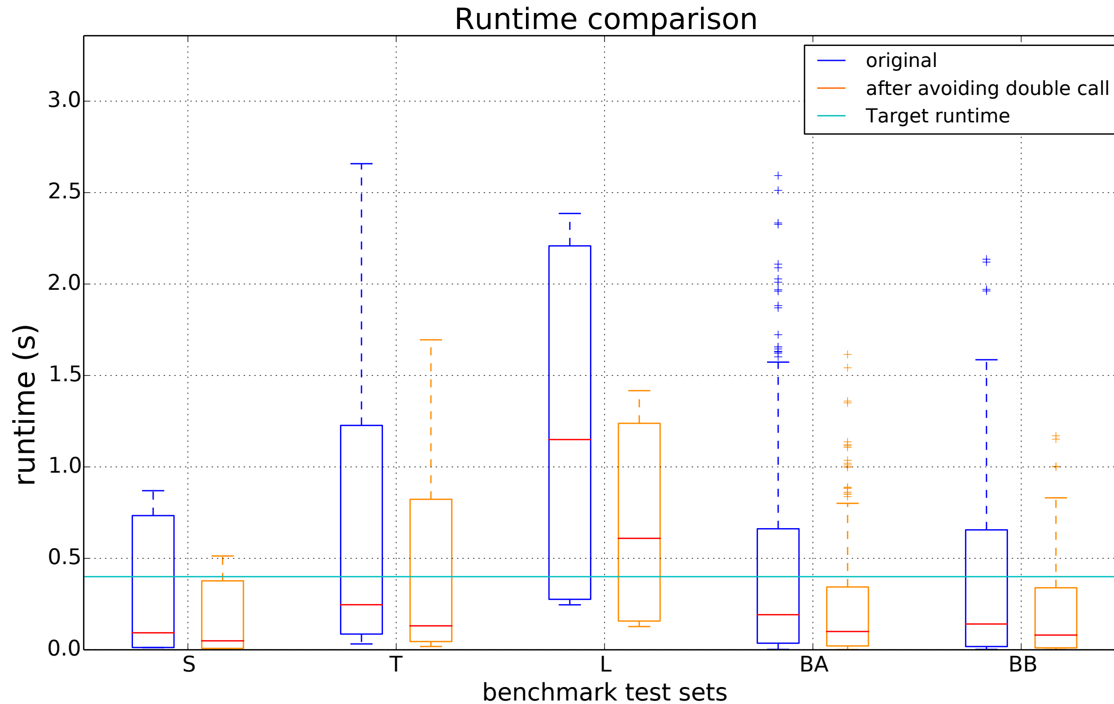


FIGURE 5.2: Results after optimizing to invoke the Bellman-Ford algorithm only once during validating an interleaving choice.

5.1.2 Optimizing Bellman-Ford by *'early break out'* from the algorithm

As explained in the previous section 5.1.1, one of the major bottlenecks shown in figure 5.1 has been eliminated, but the runtimes of the scheduler are still above the target runtime. To improve it further, the optimizations will concentrate on how the Bellman-Ford algorithm can be improved. In this section, one of the optimizations that utilizes the Bellman-Ford algorithm's capability to be able to compute the shortest path in iterations less than what the algorithm actually demands is discussed.

This is an optimization over standard Bellman-Ford algorithm, which is also explained in section V of Kumar, Misra, and Tomar [6].

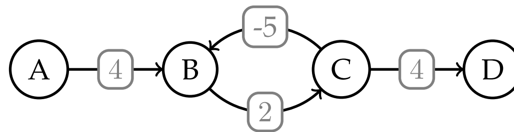
Optimization Statement

" If it is observed that, for two consecutive iterations in the Bellman-Ford algorithm the distances of vertices in graph are not changed, then 'breaking out' of the algorithm at that point does not change the final result. ([6], [10]). "

Optimization technique

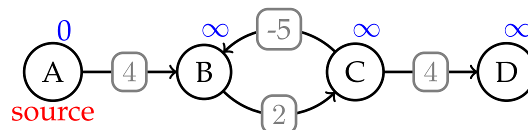
Before going into details of the optimized Bellman-Ford algorithm and its results, the technique is explained by example. This helps to understand where the Bellman-Ford algorithm can be optimized and how it affects the runtime of Bellman-Ford algorithm and the scheduler.

Consider the following graph with four vertices.



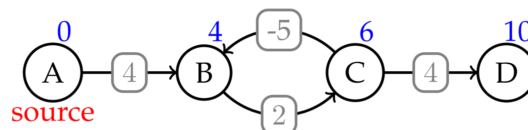
We follow the steps as given by the algorithm 4.

First step is to 'initialize' the vertices with initial distances. We assume vertex A as the source vertex.

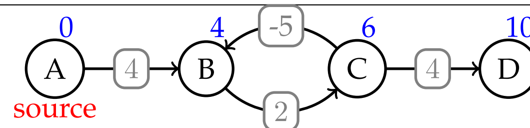


In the second step, all the edges in the graph are relaxed for 3 iterations (as there are 4 vertices). Below graphs show the distance of each vertex from source after each iteration.

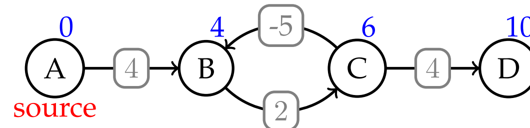
After 'iteration 1':



After 'iteration 2':



After 'iteration 3':



After the first iteration, in any of the next iterations the distance values of any vertex does not change. So, in a general scenario, while relaxing a graph once it is ascertained that distance of vertices have not changed for two consecutive iterations of the Bellman-Ford algorithm, then it means that any following iteration will not make further changes to vertex distances. The optimization discussed in this section makes use of this point to see if final iterations in the Bellman-Ford computation can be skipped. Depending on the size and complexity of the graph, there is a chance that this optimization can show good improvement in runtime of the Bellman-Ford algorithm.

In a graph $G(V, E)$, for each iteration skipped,

(a) 2xE number of additions can be avoided,
(b) E number of condition checks can be avoided.

This improvement in runtime is seen without any effect on the makespan of the schedules. The algorithm and reason for its correctness after optimization is discussed in the following sections.

Algorithm

The modified Bellman-Ford algorithm using the above discussed optimization is shown in algorithms 8 and 9.

Proof of Correctness

To prove this optimization technique gives correct results, the proof of correctness of the original Bellman-Ford algorithm is given below. The optimization is explained at the end of the proof.

For a graph represented by G(V,E) where V is set of vertices and E is set of edges,
(i) $\delta(v)$ represents the shortest distance to vertex $v \in$ V.
(ii) $d[v]$ represents the final computed distance to vertex $v \in$ V.

Bellman-Ford algorithm: For a graph G(V, E) without any negative cycles reachable by s, by relaxing all the edges in each iteration, at the end of (n-1) iterations,

for every vertex reachable from s, $d[v] = \delta(v)$, $\forall\ v \in$ V.

Corollary: If a value $d[v]$ fails to converge after (n-1) iterations, there exists a negative cycle in the graph reachable by s.

Proof of correctness by *induction*
Let V be the vertex set and $p = \{v_0, v_1, v_2, v_3 ... v_{n-1}, v_n\}$ with source = v_0 and destination = v_n be the

Algorithm 8 Complete Bellman-Ford algorithm using algorithms 1, 9 and 3.
(Algorithm 9 is adapted from [6])

```
1: procedure BELLMAN_FORD_ALGORITHM
2:     //E is the list of edges, V is the list of vertices.
3:     Create E, V
4:
5:     //Before beginning with steps that are discussed above, a source vertex should be defined.
6:     //source is set 0, because in arrow notation, 0 means the first element of the array.
7:     Create source ← 0
8:
9:     //Step - I
10:    //INITIALIZE sets the initial distance of vertices to a default value (shown in algorithm 1).
11:    INITIALIZE(V, source)
12:
13:    //Bellman-Ford algorithm iterates for n-1 times.
14:    Set i = 0;
15:
16:    //Step - II
17:    for ((i < n) and graphModified = true) do
18:
19:        //RELAX (algorithm 9) returns true if a vertex distance is modified.
20:        graphModified ← RELAX(E,V)
21:
22:        //increment the iteration variable by 1 to move to the next iteration.
23:        i = i+1
24:
25:    end for
26:
27:    //Step - III
28:    //Relax all the edges once again to check for the presence of a cycle (shown in algorithm 3).
29:    NEGATIVE_CYCLE_CHECK(E,V)
30:
31: end procedure
```

shortest path from source to destination. The algorithm needs to compute this path from source to destination in (n-1) iterations.

After first iteration of relaxing all edges in E, the algorithm will have computed the shortest distance adjacent to the source vertex.

$$\mathrm{d}[v_1] = \delta(s, v_1)$$

After second iteration, the vertices adjacent to vertex v_1 will have their shortest distances assigned to them. Let v_2 be the adjacent vertex to v_1,

$$\mathrm{d}[v_2] = \delta(s, v_2)$$

"For each iteration, the shortest path computation moves at least one step closer to the destination vertex."

So, by induction, it can be proved that, after (n-1) iterations, all of the vertices including the destination vertices will have their shortest distances (δ) computed.

Algorithm 9 Bellman-Ford edge relax algorithm. (adapted from [6])

```
1: procedure RELAX(E, V)
2:
3:     //first initialize graphModified to false; set to true only if a vertex distance is modified.
4:     graphModified ← false
5:
6:     //Relax all edges in the graph for one time.
7:     for all edges e ∈ E do
8:         if V[e.dst].distance > V[e.src].distance + e.weight then
9:             V[e.dst].distance ← V[e.src].distance + e.weight
10:
11:            //set graphModified to true only if a vertex distance is modified.
12:            graphModified ← true
13:
14:        end if
15:    end for
16:
17:    //graphModified will be true if a vertex distance is modified; false otherwise.
18:    return graphModified
19:
20: end procedure
```

Proof of correctness after optimization

The optimization statement can be *proved by contradiction* that if it is allowed to break early out of the Bellman-Ford algorithm if none of the vertex's distance is unchanged, the distance values of vertices will still be correct.

Lets say for two consecutive iterations (k and k+1 $<$ (n-1)) before the final iteration, the distance values of all vertices remain unchanged. And in the next iteration, i.e., (k+2)th iteration, the distance value of one of the vertices change. This is not possible because, as the Bellman-Ford algorithm is proved above, at least one edge will be relaxed in an iteration to reach the destination in its shortest path. But since two consecutive iterations have same distance values then it means no edges are relaxed which means the shortest paths have already been computed. Therefore it would be contradictory to say an edge is relaxed after two iterations have shown same distance values.

After computing the distances, the algorithm continues to check for the presence of cycles ($corollary$) in the graph as it is done in the original Bellman-Ford algorithm (shown in algorithm 3).

Worst case analysis

For the Bellman-Ford algorithm shown in algorithm 4, the worst case is either when there is a cycle in the graph or the path from source to destination includes n number of vertices. Similar is the case in the optimized version shown in algorithm 8. The optimized version is designed to break out of the algorithm only if there are no changes seen in the graph for two consecutive iterations. But, if there is a cycle in the graph, then in each iteration, the distance of one of the vertex in the cycle will be modified, which ensures that execution does not break out before finishing all the 'n-1' iterations.

Results

The experiments are run on the scheduler after implementing the above optimization. Test set is explained in B and the platform used to run the experiments is *CPU-I* (configuration is given in A).

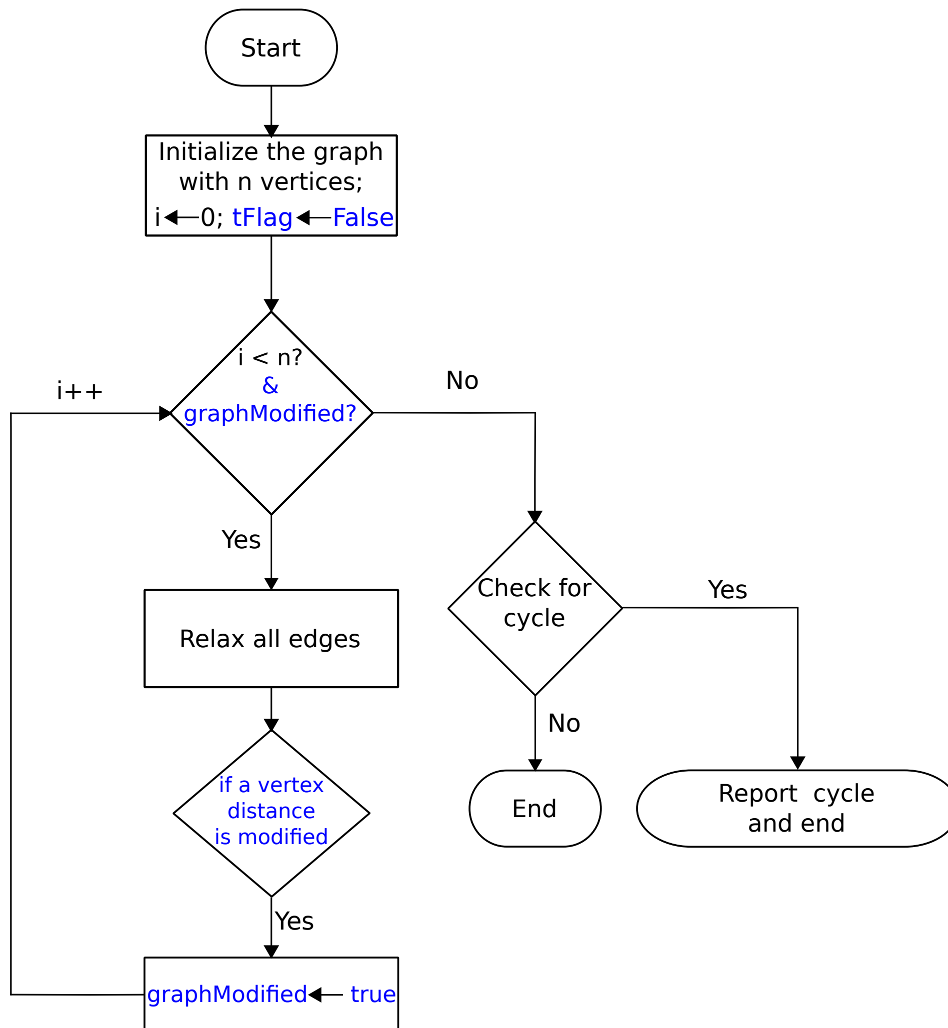


FIGURE 5.3: Flow chart of Bellman-Ford algorithm with 'early termination' allowed (adapted from [6]).

Table 5.2 shows the maximum speedup obtained by different optimizations after this optimization is implemented on the scheduler.

	Homogeneous(S)	Varying Thickness(T)	Varying Length (L)	Booklet-A (BA)	Booklet-B (BB)
original	1	1	1	1	1
after avoiding double call	1.7	1.7	1.6	1.7	1.8
after avoiding double call + early-break out	5	3.2	4.8	6	8

TABLE 5.2: Maximum speedup obtained in the execution of Heuristic scheduler after the above two discussed optimizations are implemented in the Heuristic scheduler.

Figure 5.4 shows the performance improvement obtained with this optimization compared to original scheduler.

After the current optimization, the average runtime of the scheduler has come down to less than the *target* runtime of 400ms. But there are still majority of test cases whose runtimes fall above the target line of 400ms.

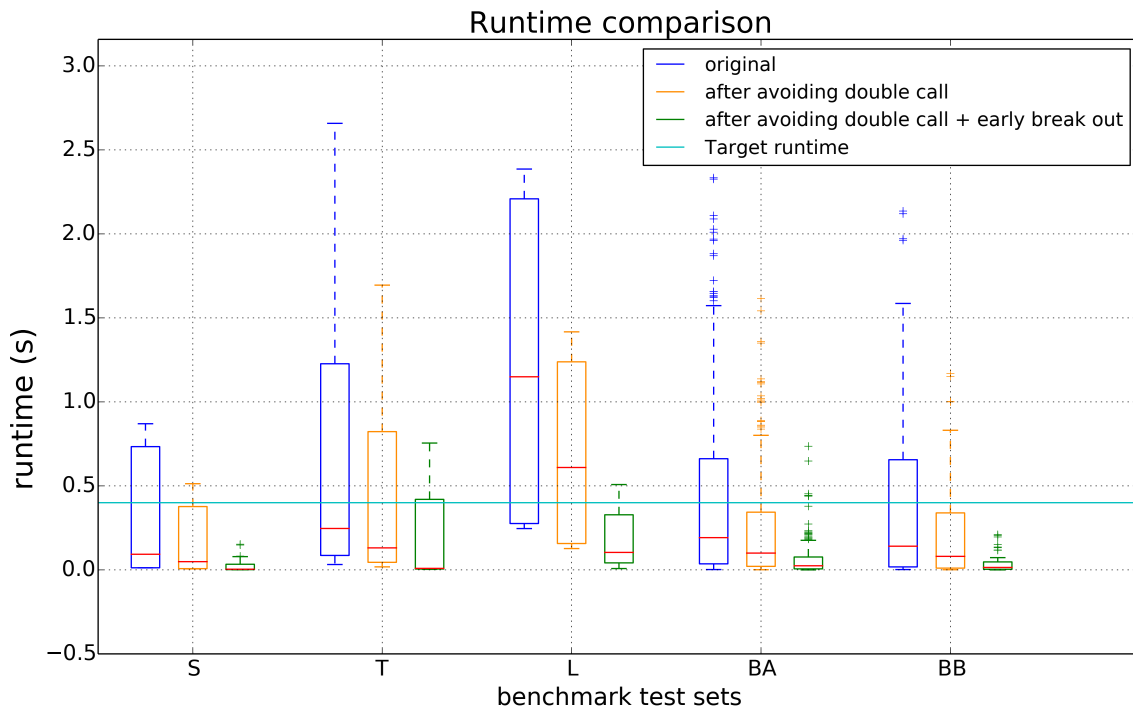


FIGURE 5.4: Runtime results of the Heuristic scheduler after *early break out* from the Bellman-Ford algorithm.

Conclusion

From above figures, it can be seen that the runtime has improved for the scheduler compared to the original implementation. But there are still some test cases and outliers with 100 or less sheets that still have their runtime more than 500ms. So, further optimizations to improve the runtime are studied.

5.1.3 Optimizing Bellman-Ford by *marking the modified vertices*

The previous implementation of 'early break out' from the Bellman-Ford algorithm has shown that the values of the edge weights are in such a way that the Bellman-Ford is computing the final distances of all the vertices before finishing all the (n-1) iterations. This also indicates that many of edges are not effective, i.e., edges are successfully relaxed very few times in the iterations to compute the actual vertex distances.

The optimization discussed in this section will try and reduce the number of edges relaxed further by looking only at the vertices whose distances are changed due to relaxing. Whereas, in previous optimization, in each iteration, all edges are attempted to relax. This optimization uses an adapted version of Bellman-Ford implementation explained in [8]. To maintain confidentiality, the IP department at Océ has suggested to not publish the details of this optimization in the public document.

Optimization statement

"Relax only the edges which are outgoing from the vertices whose distance is previously modified."

Optimization technique

Removed due to confidentiality.

Algorithm

Removed due to confidentiality.

Results

	Homogeneous(S)	Varying Thickness(T)	Varying Length (L)	Booklet-A (BA)	Booklet-B (BB)
original	1	1	1	1	1
after avoiding double call	1.7	1.7	1.6	1.7	1.8
after avoiding double call + early-break out	5	3.2	4.8	6	8
after avoiding double call + modified vertex list	12.5	3.4	7.25	7.8	9

TABLE 5.3: Maximum speedup obtained in the execution of Heuristic scheduler after the above three discussed optimizations are implemented in the Heuristic scheduler.

The experiments are run on the scheduler after implementing the above discussed optimization. Test set is explained in B and the platform used to run the experiments is *CPU-I* (configuration is given in A).

Figure 5.5 shows runtime profile of the heuristic scheduler after implementing this optimization in the Bellman-Ford algorithm. Although there are some test cases and outliers which do not meet the target runtime of 400ms, the Inter Quartile Range (IQR) of the runtime distributions that shows

almost three quartiles of the test cases with 100 or less sheets have their runtime less than 400ms. When compared to previous optimization discussed in section 5.1.2, even though the results are almost same, because the IQR is below the target runtime and also because of this optimization does least amount of work, the 'modified list' version of the Bellman-Ford algorithm will be considered as the better version and is recommended for using in the scheduler.

The speedup obtained after this optimization for each of the job type is given in table 5.3. Figure 5.6 compares the runtime profile of the scheduler using current optimization with runtime profiles of optimizations discussed before. Each set of box plots represents a type of job explained in Appendix B.

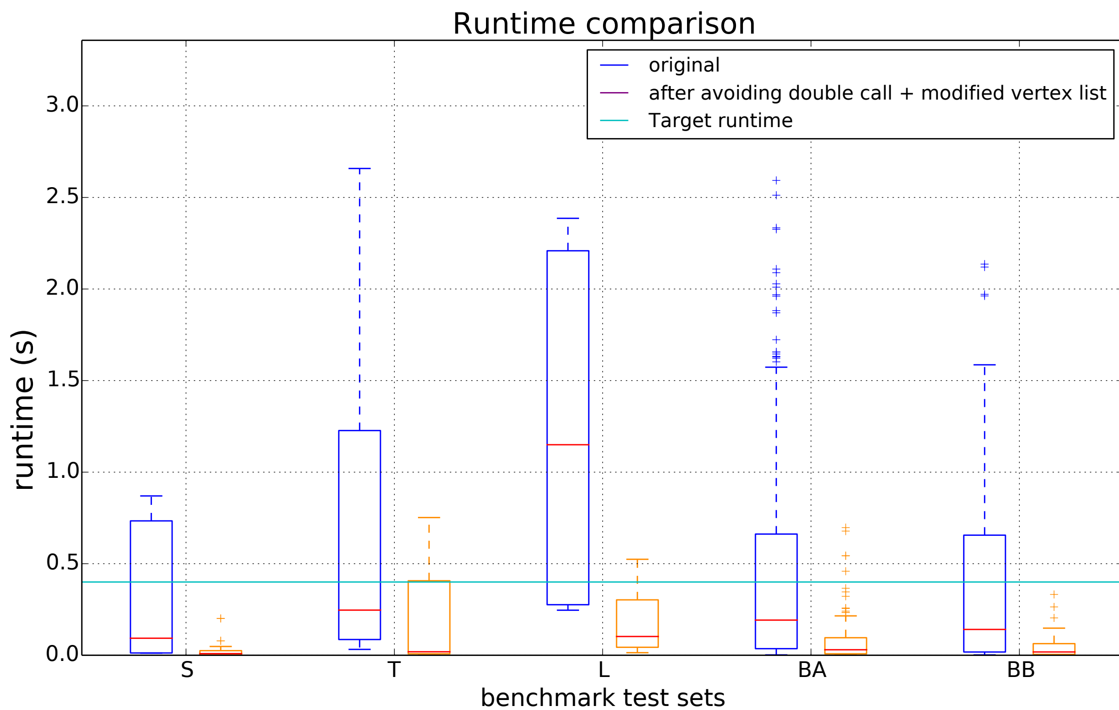


FIGURE 5.5: Runtime of the Heuristic scheduler after implementing the above optimization of *marking the modified vertices* in the Bellman-Ford algorithm compared to original scheduler execution profile.

Conclusion

The results show that the runtime of the Heuristic scheduler has improved when compared to the original implementation. The Inter Quartile Ranges of the corresponding box plots show that current optimization gives better runtime than all other previous optimizations. Although there are certain cases whose *maximum*'s fall outside the target line, judging by the size and distribution of Inter Quartile Ranges, the optimization using the 'modified lists' is a better optimization than the 'early break out' version of the Bellman-Ford algorithm.

As this version is the fastest version of the other optimizations, the next optimization that can help improve the runtime would be to parallelize the algorithm. Many literature works have shown improvement in the Bellman-Ford algorithm runtime profile when executed on a parallel platform. So, as the next step, the algorithm is implemented on a parallel platform.

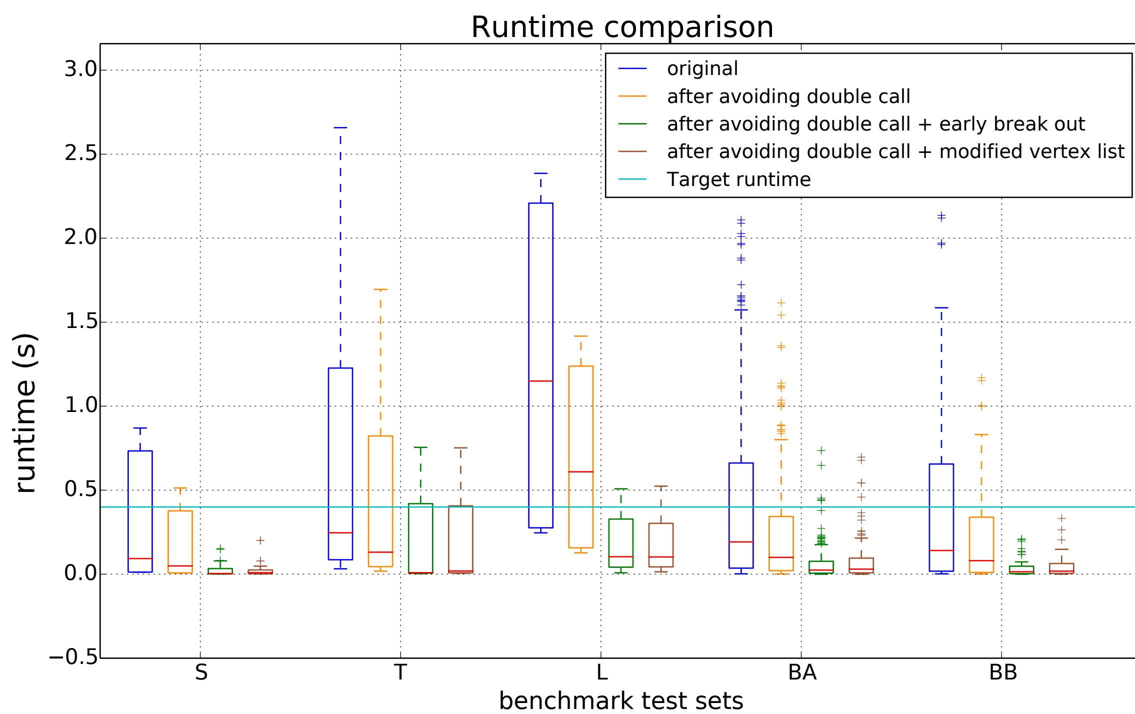


FIGURE 5.6: Runtime of the Heuristic scheduler after implementing the above optimization in the Bellman-Ford algorithm compared to all other optimizations discussed above.

5.1.4 Optimizing the Bellman-Ford algorithm by marking the modified vertices and *relaxing them in parallel*

In an attempt to further reduce the runtime, using the research done during the preparation phase of the project, the Bellman-Ford algorithm is parallelized.

As the last discussed optimization has shown better runtime profile than other optimizations discussed, a parallel version of that implementation of the algorithm is implemented here by taking some ideas from related works. Because this optimization is done on the previous optimization, the details regarding this optmization are also kept confidential.

Optimization statement

To compute distances to each and every vertex in the graph, the Bellman-Ford algorithm expects to compute the distance between vertices with the help of edge weights. This process, as discussed many times until now is called 'relaxing'.

To parallelize any algorithm, there must be some section of code which is executed again and again. This section of code should not only execute multiple times but also, one execution should be independent from the other. Independence is measured in terms of the data that is being accessed, both to read and write. More the independence between executions, more parallelism can be put in place to get better runtime. In case of the Bellman-Ford algorithm, the step of relaxing is that part of the code which is run multiple times (on every edge). Since this is the same code, relaxing each edge can be done in parallel. Better results can be obtained if the edges to be relaxed are 'destined to' different vertices, because, in that case, when all edges that are relaxed in parallel can write to memory without any conflicts, giving maximum independence between parallel threads.

Algorithm and results

Removed due to confidentiality.

Results

To execute the scheduler using parallel version of Bellman-Ford algorithm, *CPU-II* is used. OpenMP is used to implement the parallel version. The experimental setup is given in A. Test set is given in Appendix B.

Figure 5.7 shows runtime of the scheduler with parallelized version of the 'modified vertices list' Bellman-Ford algorithm when compared to sequential version. Figure shows box plots (explained in Appendix C) of runtime distributions of the scheduler after different optimizations on different test sets.

Looking at the results, it can be observed that, the scheduler using the parallel version of Bellman-Ford algorithm is slower than when scheduler used the version explained in section 5.1.3. Although during the execution 3 threads are executed in parallel every time, the scheduler seems to have slowed down.

Multiple reasons individually or collectively can be the reason for this behavior. First being, multiple executions of the Bellman-Ford algorithm within a small amount of time. It is observed that for a test case with 100 sheets, the Bellman-Ford algorithm is invoked at least 100 times. And every time the Bellman-Ford algorithm is invoked, Open-MP creates and after their execution the threads are destroyed. This process of creating threads is known as 'forking'. This leads to an unnecessary overhead and can delay the execution, ending up with a slower parallel version than the sequential

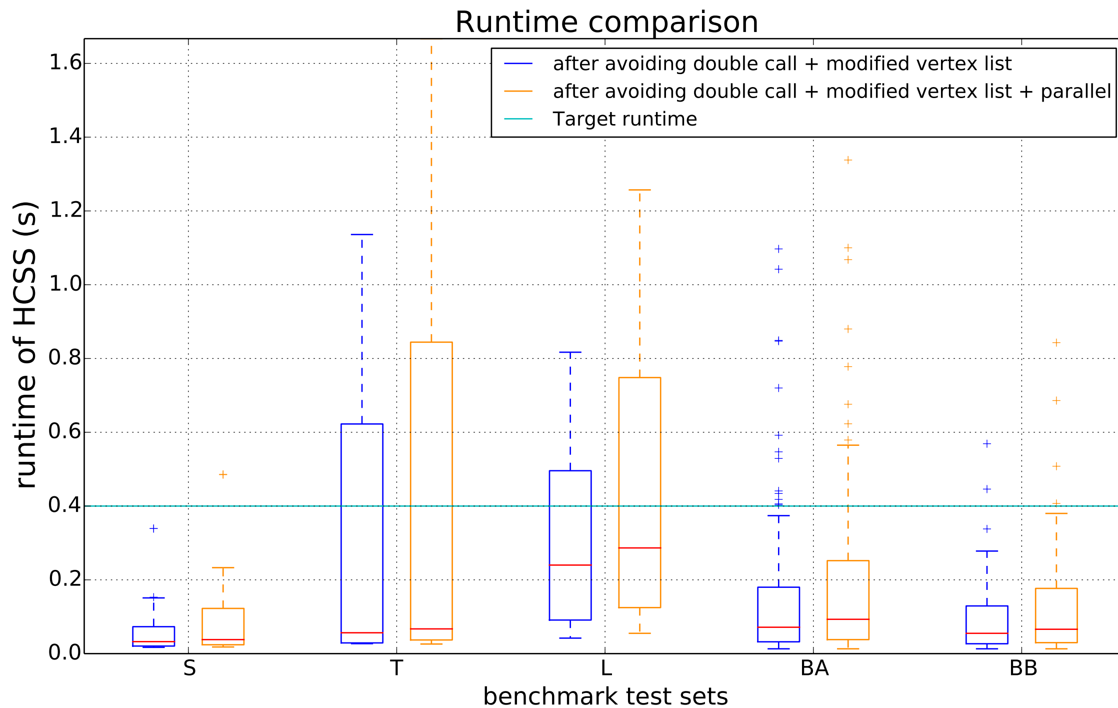


FIGURE 5.7: Runtime of the Heuristic scheduler after implementing the above optimization of *marking the modified vertices* ***relaxing in parallel*** in the Bellman-Ford algorithm compared to sequential version.

version. Second reason that could have affected the runtime is the lack of independence between parallel threads. This is not the case in this implementation because, although multiple threads read same data, they will always write to different data locations because out going edges end up in different vertices. So, even if relaxing all of the outgoing edges are successful, writing to memory will be at different locations which shows complete independence between the threads. So, 'forking' seems to cause the extra delay that is observed in the results.

Also, related works discussed in chapter 4 deal with large graphs and have shown speedups when compared to sequential versions, they execute the Bellman-Ford algorithm only once, unlike here in the scheduler. Executing only once will mean that the threads are created only once and destroyed finally, so there is no need to consider the overhead due to forking and show better results.

Conclusion

Following this development in the runtime profile of the Heuristic Scheduler, a generic optimization technique is studied that gives more freedom in terms of runtime.

5.2 A *generic* optimization technique

To schedule jobs of sheets on a printer, printer does not need schedule for all the sheets in the job. For example, a pamphlet print job requires the printer to print almost 100,000 sheets of a particular type, say smallest media type available. Not only scheduling 100,000 sheets takes a long time, but also for the printer to print, it does need schedule for all the 100,000 sheets, it just needs the schedule for the number of sheets that fit in the printer at a point of time. So, the scheduler can be modified to supply the printer schedules of only those sheets that will be in the printer at that point of time. The only constraint is that, for the printer to be able to output one sheet for every 400ms, the schedule for the number of sheets that fill printer should be made available in 400ms.

The scheduler is modified to compute schedules in this way and more details of this technique and its results are given in the full confidential report.

5.2.1 Algorithm

Removed due to confidentiality.

5.2.2 Conclusion

Removed due to confidentiality.

Chapter 6

Conclusion and Future work

The heuristic scheduling algorithm is optimized so that it can be used as an online scheduler. Different optimizations for the scheduler are proposed after which the runtime to schedule each sheet has reduced to less than 400ms. In this chapter, section 6.1 concludes the thesis work and section 6.2 proposes possible future work on the scheduler.

6.1 Conclusion

Removed due to confidentiality.

6.2 Future work

Removed due to confidentiality.

Appendix A

Experimental setup

The experiments for the optimizations done on the scheduler are performed on two systems due to restrictions on the platforms. The first three optimizations explained in sections 5.1.1, 5.1.2, 5.1.3 and scheduler after implementing generic optimization technique are executed in Linux operating systems running on an Intel Core i7 CPU running at 3.4GHz with 16GB of RAM. This system is referred as *CPU-I*. For the optimization explained in section 5.1.4, a system platform that supports multi-processing is needed, so the algorithm is executed on a system running Ubuntu 14.04.1 on Intel Core i7 CPU running at 2.67GHz with 12GB of RAM. This system is referred as *CPU-II*.

Appendix B

Test set

The test set consists of 5 types of jobs. The first category is of homogeneous type where every job has only one type of sheet, shown as S in figures. The second category denotes set of jobs where each job has sheets with varying thickness, shown using T. The third test set contains test cases of print jobs with sheets that have 5 blocks of varying lengths with each block having 10 or 20 sheets. Two categories, booklet-A (BA) and booklet-B (BB) are of two booklet type test cases. Booklet test cases have two types of sheets called cover sheets and body sheets with varying thickness and length. Cover sheets have higher thickness than body sheets. Booklet-A test cases have varying number of cover sheets (from 1 to 3) with same sheet lengths for cover and body type sheets. Booklet-B has one cover sheet with cover sheet having higher sheet length when compared to a body type sheet. (adapted from Waqas et al. [9]).

Appendix C

Box plots

Box plots, which are also called box and whisker diagrams gives a way to show how set of numerical data is distributed through the quartile representations. A box plot is summarized using (minimum, first quartile, median, third quartile, and maximum). There are also outliers shown using whiskers which are either 3xIQR (Inter Quartile Ranges) above third quartile or below the first quartile. (Adapted from [4], [11])

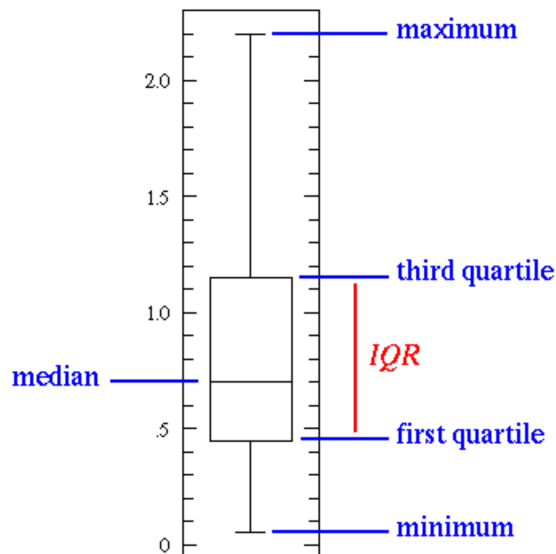


FIGURE C.1: Box plot example (adapted from [4])

Appendix D

Heuristics

Heuristic metrics

The algorithm, as explained in chapter 2 uses 3 heuristic metrics to rank the interleaving choices of each sheet. To compute ranks of each interleaving option, the algorithm uses three relative weights given by κ_P, κ_F and κ_{DT}. During the computation of ranking, these relative weights are given a constant value. The values assigned to each of the weights are,

$$\kappa_P = 0.7$$

$$\kappa_F = 0.25$$

$$\kappa_{DT} = 0.05$$

Bibliography

[1] Ibrahim M Alharkan. “Scheduling theory”. In: *Algorithms for sequencing and scheduling*. 2005, pp. 1.1–1.24.

[2] F. Busato and N. Bombieri. “An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.8 (2016), pp. 2222–2233. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2485994.

[3] David Cohen and Matthew Dallas. “Implementation of Parallel Path Finding in a Shared Memory Architecture”. In: (). URL: http://kesshoryu.com/files/Parallel_Paper.pdf.

[4] Bob Jones university Computer Science department. *Box Plot: Display of Distribution*. [Online; accessed 09-August-2016]. 2016. URL: {http://www.physics.csbsju.edu/stats/box2.html}.

[5] In-Kyu Jeong et al. “Accelerating a Bellman–Ford Routing Algorithm Using GPU”. English. In: *Frontier and Innovation in Future Computing and Communications*. Ed. by James J. (Jong Hyuk) Park et al. Vol. 301. Lecture Notes in Electrical Engineering. Springer Netherlands, 2014, pp. 153–160. ISBN: 978-94-017-8797-0. DOI: 10.1007/978-94-017-8798-7_19. URL: http://dx.doi.org/10.1007/978-94-017-8798-7_19.

[6] S. Kumar, A. Misra, and R.S. Tomar. “A modified parallel approach to Single Source Shortest Path Problem for massively dense graphs using CUDA”. In: *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*. 2011, pp. 635–639. DOI: 10.1109/ICCCT.2011.6075214.

[7] ProgrammingInC++. *Vector and List - Part 1*. [Online; accessed 08-August-2016]. 2016. URL: {http://www.programmingincpp.com/vector-and-list.html}.

[8] R Sedgewick and K Wayne. *Algorithms: Pearson Education*. 2011.

[9] Umar Waqas et al. “A Re-entrant Flowshop Heuristic for Online Scheduling of the Paper Path in a Large Scale Printer”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. DATE ’15. Grenoble, France: EDA Consortium, 2015, pp. 573–578. ISBN: 978-3-9815370-4-8. URL: http://dl.acm.org/citation.cfm?id=2755753.2755882.

[10] Wikipedia. *Bellman-Ford algorithm*. [Online; accessed 08-August-2016]. 2016. URL: {https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm}.

[11] Wikipedia. *Box plot*. [Online; accessed 09-August-2016]. 2016. URL: {https://en.wikipedia.org/wiki/Box_plot}.

[12] Wikipedia. *Scheduling (computing) — Wikipedia, The Free Encyclopedia*. [Online; accessed 08-August-2016]. 2016. URL: {https://en.wikipedia.org/w/index.php?title=Scheduling_(computing)&oldid=700486388}.