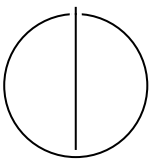
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Analysis of Algorithms for Scheduling with Testing

Klaudia Maria Tarabasz



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

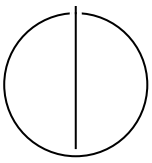
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Analysis of Algorithms for Scheduling with Testing

Implementierung und Analyse von Algorithmen für Scheduling mit Tests

Author:	Klaudia Maria Tarabasz
Examiner:	Prof. Dr. Susanne Albers
Supervisor:	Prof. Dr. Susanne Albers
Submission Date:	16.04.2025



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 16.04.2025 Klaudia Maria Tarabasz

Acknowledgments

I would like to thank Prof. Dr. Susanne Albers for providing me with this interesting topic and answering any questions I had along the way. I am also grateful to my family and friends who helped me with proofreading many versions of this thesis. This help was much needed and I hope to one day reciprocate with the same amount of diligence.

Abstract

Scheduling with testing is a problem with the goal of minimizing cost of produced schedule, which is calculated according to chosen objective function. Certain degree of uncertainty is assumed in algorithms that solve this problem. Namely, jobs that should be scheduled can either be run for the duration of their upper limit on a machine or they can be tested and then run on a machine, which takes previously unknown amount of time. Different algorithms are compared using competitive analysis. In this thesis, I will review some of the algorithms that minimize makespan of schedule and present my implementation of them.

Zusammenfassung

Scheduling mit Tests ist ein Problem mit dem Ziel, die Kosten des erstellten Schedule zu minimieren, die anhand der gewählten Zielfunktion berechnet werden. Algorithmen, die dieses Problem lösen, gehen von einem gewissen Grad an Unsicherheit aus. Zu planende Jobs können entweder für die Dauer ihrer Obergrenze auf einer Maschine ausgeführt oder getestet und anschließend auf einer Maschine ausgeführt werden, was eine bisher unbekannte Zeit dauert. Verschiedene Algorithmen werden mittels Competitive Analyse verglichen. In dieser Arbeit werde ich einige Algorithmen zur Minimierung der Durchlaufzeit von Schedules untersuchen und meine Implementierung vorstellen.

Contents

1 Introduction

Multiprocessor scheduling is a combinatorial problem with a long history. Initially the offline variant of the problem was considered. In its statement the arbitrary number of jobs was given, alongside time needed execution of each of them, and the number of machines on which those jobs should be scheduled. The machines were assumed to be identical and available computation power - infinite. The goal was to minimize makespan of the schedule. An example of an algorithm solving this problem is (2-1/m) - competitive List Scheduling algorithm, proposed by Graham [Gra66] in 1966. Since then, this result was improved and current best algorithm is the 1.9201-competitive algorithm proposed by Fleischer and Wahl [FW00]. The current best result for lower bound, 1.88, was achieved in 2003 by Rudin [RC03].

Already the original problem's name and used terminology (machine, job) pointed at its main application - scheduling of processes on computers with multiple cores. Over the years multiple online variations of the original problem statement were created. Some of them are quite similar to real life problems, where not every information is given upfront. These variations included using different objective functions and introducing concept of testing times for the jobs.

Algorithm with testing allows to recreate the situation where the program is already compiled and ready to run, but we could also decide to optimize (test) this program (which would take some known amount of time) and possibly gain optimised version of the program that would take less time to execute.

This is the general idea behind scheduling with testing. In following chapters, I will formalise its definition, introduce necessary notation and describe some of the algorithms published in the recent years. At the end, I'll present results from my implementation of described algorithms.

2 Preliminaries

2.1 Notation

As briefly stated in the introduction, the job j can be run on a machine for entire duration of its upper limit u_j or tested, which takes $t_j \geq 0$ and then run with reduced processing time p_j. For all jobs the $0 \leq p_j \leq u_j$ is satisfied. Set containing all jobs and set containing all the machines will be denoted as J and M respectively.

The ratio r_j of the job will measure how big the upper limit is, compared to the testing time of the job. It is defined as: $r_j := \frac{u_j}{t_j}$. The minimal running time τ_j of the job j will also be frequently used in later described algorithms. It is defined as $\tau_j := min(t_j, u_j)$.

Through the next chapters the symbol φ will always describe the golden ratio, $\varphi \approx 1.6180$.

To check how good is the schedule produced by an algorithm, the competitive analysis will be used. It is defined as ratio between cost of the algorithm and cost of the optimal solution. Cost is obtained from chosen objective function, like makespan minimisation.

2.2 Categories of algorithms

Algorithms described in the next chapter are part of the class of algorithms that is often called in literature: Scheduling with testing in explorable uncertainty setting. It implies some fact are not known at the start of the algorithm. However, this name still leaves room for interpretation and therefore there exist right now several different algorithms in vastly different settings.

There are several different characteristics of the setting for an algorithm, which allows to categorise them in a few groups. Each describes characteristic how much flexibility the creator of an algorithm has, what are the basic assumptions and what is the goal of the algorithm.

The first one concerns nature of testing. There are three main categories: preemptive, non-preemptive and test-preemptive. Preemptive means that execution of the job can be interrupted at any time and be resumed at any point in the future, possibly on a different machine. On the other hand, when the setting is non-preemptive, the

job cannot be interrupted at all. Test-preemptive setting was introduced not so long ago and serves as the middle ground between the other two. Here, the job can be interrupted only at the completion of test and execution of the job can be resumed with a delay, possibly on a different machine.

Some algorithms include random variables that are used to decide which action should be taken in the next step of the algorithm. Algorithm that allow such thing are created in randomized setting. Otherwise, when for the same input data algorithm always produces the same schedule, the algorithm is deterministic.

All of the algorithms described in the later chapters assume that arrival of all jobs happens at the start of the algorithm. However, such scenario doesn't exactly match some of the real-world applications of said algorithm. Therefore, it is possible that in the future more algorithms will assume setting where jobs arrive in random intervals after start of the algorithm.

Assumptions concerning length of test for each job are divided into two main categories. Times of tests can be uniform, so it always takes one unit of time to run them. Length of the test can also be random. This setting will later also be described as general.

There are also differences in goal of the algorithm. Most prevalent are algorithms which minimise sum of completion times and those that minimise makespan.

All of the algorithms described and implemented in later sections are deterministic, assume arrival of all jobs at the start of the algorithm and minimise makespan of created schedule. Algorithms SBS and BBS have variation for both random testing times and uniform testing times. The Two Phases algorithm assumes random testing times. SBS and BBS in both of their variations are non-preemptive. Two Phases algorithm is test-preemptive.

3 Algorithms for makespan minimisation

In all of the following algorithms in this chapter, all jobs arrive at once, at the beginning, ready to be scheduled.

3.1 SBS algorithm

This algorithm was published in [AE21]. It achieves the competitive ratio of 3.1016. The general idea behind it involves splitting all jobs into three classes: S_1, B, S_2. Order in which the jobs are scheduled depends on which class the job was assigned to. All of the jobs that belong to S_1 will be scheduled first, then all in the class B and after that, algorithm will finish with scheduling all jobs inside S_2. Whether a job inside a class should be tested or not varies from class to class and is described later in this section.

3.1.1 Splitting into classes

Splitting of jobs happens at the start of the algorithm, because of the assumption that every job is available from the very beginning. First, the jobs will be either assigned to set S or set B. Later the set S will be split into S_1 and S_2. Which job is assigned to set S and which to set B depends solely on whether ratio of the job is greater than value of the threshold function, which is defined as follows:

$$T(m) = \frac{(3+\sqrt{5})m - 2 + \sqrt{(38+6\sqrt{5})m^2 - 4(11+\sqrt{5})m + 12}}{6m-2} \tag{3.1}$$

Value of the threshold function depends only on number of machines, which means the same set of jobs could be split into classes differently if we only change number of available machines.

Formally, the jobs are divided into two classes in following way:
$B := j \in [n] : r_j \geq T(m)$
$S := [n] \setminus B$

This division corresponds to saying that all the jobs in B have high enough upper bound compared to their testing times, that they should always be tested.

Now, the set S should be divided into S_1 and S_2. S_1 will contain m jobs at maximum. Each of them should be scheduled on an empty machine at the beginning of scheduling.

Algorithm 1: SBS algorithm

```
1  B ← {j ∈ [n] : r_j ≥ T(m)};
2  S ← [n] \ B;
3  S_1 ← S' ⊂ S s.t. |S'| = min(m, |S|),  τ_{j_1} ≥ τ_{j_2}  ∀j_1∈S', j_2∈S\S';
4  S_2 ← S \ S_1;
5  foreach j ∈ S_1 do
6  |   if r_j ≥ φ then
7  |   |   test and run j on an empty machine;
8  |   else
9  |   |   run j untested on an empty machine;
10 |   end
11 end
12 foreach j ∈ B do
13 |   test and run j on the current least-loaded machine;
14 end
15 foreach j ∈ S_2 do
16 |   run j untested on the current least-loaded machine;
17 end
```

Figure 3.1: SBS

If there are not enough jobs for all of the machines, S_1 is equal S and S_2 is empty. Should there be more jobs in S, than there is machines, then m of those with highest minimal running times will be in S_1, rest in S_2.

3.1.2 Scheduling

Firstly, all of the jobs from S_1 should be scheduled, each on an empty machine as mentioned before. If the ratio of a job j $r_j \geq \varphi$ then the job should be tested. Otherwise it should run untested.

After scheduling last job from S_1, every job that comes next should be scheduled on a least loaded machine. All of the jobs from B should be tested and run on the same machine. Then, all the jobs from S_2 should be run untested. In 3.1 is shown the pseudocode of the algorithm from [AE21].

3.2 Uniform-SBS algorithm

Uniform version of SBS was created for the special case, where testing times of all jobs $j \in J$ satisfy $t_j = 1$. Uniform-SBS uses different threshold function, namely:

$$T_1(m) = \frac{2m - 1 + \sqrt{16m^2 - 14m + 3}}{3m - 1} \tag{3.2}$$

This algorithm reduced used classes to B and S_2 compared to previous one. Additionally, in this case the explicit assignment to classes is not necessary. It is enough to sort the jobs $j \in J$ in non-increasing order according to u_j (note: when the testing times are uniform, $u_j = r_j$) and schedule them in this order. First part of those jobs, consisting of jobs with $u_j = r_j \geq T_1(m)$ form class B, second part of those sorted jobs form class S_2. As previously in SBS for general case, all jobs in B should be tested and run on least loaded machine and all jobs from S_2 should be run untested on least loaded machine.

3.3 BBS algorithm

BBS algorithm [GL21] is a strict upgrade to SBS algorithm. Competitive ratio for $m \to \infty$ is equal $\varphi + \frac{4}{3} \approx 2.9513$ (proof is included in corresponding paper [GL21]).

3.3.1 Splitting into classes

Similarly to SBS, here the jobs are also divided into three disjunctive classes: B_1, B_2, S. The novelty of this approach lies in sorting all jobs in non-increasing order of τ-values and keeping this order after jobs are assigned to their groups. In case some jobs have the same τ-value, they should be ordered non-increasingly according to their upper limits. B_1 will contain first m jobs from this ordered sequence. From the remaining jobs B_2 will contain all jobs j that have the ratio greater than threshold function, $r_j = \frac{u_j}{t_j} \geq T_m^g$. BBS uses following threshold function:

$$T_m^g = \begin{cases} \frac{3\varphi+6+\sqrt{45\varphi+213}}{14}, m = 2 \\ \frac{3m\varphi+4m-4}{4m-1}, m = 1 \text{ or } m \geq 3 \end{cases} \tag{3.3}$$

All remaining jobs will be assigned to S. If there are less jobs, than there is machines, then every job will be assigned to B_1.

3.3.2 Scheduling

Scheduling follows the convention stated in name of the algorithm: BBS. First all of the jobs from B_1 will be scheduled, then all jobs from B_2 and at the end all jobs from S.

Algorithm 1. BBS Algorithm

Input: B_1, B_2^g, S^g;
for $J_j \in B_1$ in increasing order of j **do**
  if $\frac{u_j}{t_j} < \varphi$ **then**
    schedule J_j untested on machine M_j;
  else
    test and schedule J_j on machine M_j;
  end if
end for
for $J_j \in B_2^g$ in increasing order of j **do**
  test and schedule J_j on the least loaded machine;
end for
for $J_j \in S^g$ in increasing order of j **do**
  schedule J_j untested on the least loaded machine;
end for

Figure 3.2: BBS

Jobs from B_1 will be tested and run on the least loaded machine only if ratio of the job, $r_j \geq \varphi$. Otherwise the job will be run untested. All jobs from B_2 will be tested and run on least loaded machine. After algorithm finishes scheduling them, all jobs from S will be run untested on least loaded machine. In 3.2 is the pseudo code of this algorithm, from [GL21].

3.4 Uniform-BBS algorithm

This is again the version of BBS algorithm created for a scenario where all testing times are uniform. The jobs are basically ordered in the same way: non-increasingly by τ-values and in case of a tie non-increasingly by upper limit. Only, in case of uniform testing times, this translates to simply ordering jobs non-increasingly by their upper limit. Jobs are divided into classes in the same way as in general case, but with the use of new threshold function, T_m^u.

$$T_m^u = \begin{cases} \frac{9+3\sqrt{37}}{14}, m = 2 \\ \frac{7m-4+\sqrt{97m^2-68m+16}}{2(4m-1)}, m = 1 \text{ or } m \geq 3 \end{cases} \tag{3.4}$$

3.4.1 Scheduling

If $B_2 = \emptyset$, then scheduling works in the same way as in general case: testing of the jobs in B_1 depends on value of ratio compared to φ and jobs from S are run untested as

Algorithm 2. U-BBS Algorithm

```
Input: B_1, B_2^u, S^u;
if B_2^u = ∅ then
    for J_j ∈ B_1 in increasing order of j do
        if u_j / t_j < φ then
            schedule J_j untested on machine M_j;
        else
            test and schedule J_j on machine M_j;
        end if
    end for
else
    for J_j ∈ B_1 ∪ B_2^u in increasing order of j do
        test and schedule J_j on the least loaded machine;
    end for
end if
for J_j ∈ S^u in increasing order of j do
    schedule J_j untested on the least loaded machine;
end for
```

Figure 3.3: BBS-Uniform

before. However, if B_2 is not empty, all of the jobs from B_1 and B_2 should be tested and run on least loaded machine. Jobs from S are run untested at the end as before.

In 3.3 is the pseudo code describing this algorithm [GL21].

3.5 Two Phases

The basic idea behind this algorithm involves usage of optimal offline algorithm. Using the brute force method is of course not viable in real world. Instead, in implementation presented in the next chapter I will use Sorted List Scheduling algorithm.

The one thing that distinguishes this algorithms from others in this chapter, is test-preemptive setting. As a reminder, this means after testing, running the job could be delayed or potentially executed on a different machine.

3.5.1 Algorithm

As the name suggests, the algorithm is divided into two phases. In each phase offline sub-problem is solved. In this algorithm, all jobs which upper limits are smaller than testing times are set to be scheduled, without testing, in the first phase. For all other jobs, their tests are to be scheduled in the first phase. In the first phase algorithm finds optimal schedule for selected jobs and tests. The second phase consist of finding

an optimal schedule for processing times of jobs tested in the first phase. Then, the new-found schedule is put on top of the one from the first phase.

4 Implementation

I implemented all algorithms described in the previous chapter and created a platform with graphical preview how they perform in different test cases. The overview of the UI is shown in figure 4.1.

4.1 User interface

After launching the application it works on the local host. In the top section of the page (see figure 4.2) user can change parameters of the planned test. That includes number of machines, used file with previously generated or manually set jobs, and algorithms that should be included in comparison.

After clicking 'Submit' the row chart is created in the middle section (see figure 4.3. It scales automatically with the screen size and shows makespan achieved by each algorithm in given test. Row chart template is from D3 Java Script library [D3].

The bottom section shows detailed results from each algorithm that can be expanded (for part maximally expanded results see figure 4.4. There results are visualised using json viewer [and]. On the top layer the algorithms included in the comparison are marked with numbers starting from 0.

Expanding "computationResult" shows how the algorithm divided jobs between machines. Each machine shows which actions were executed in order. Each task executed on the machine includes job type: test, reduced time or upper limit. Task also includes exact time of its execution. In case where the task is run as reduced time, the originally known upper limit of the job is also shown, so it is possible to see how much time was gain through running a test. It is also possible to track which job from the testing file was exactly scheduled here, by checking unique jobID.

4.2 Test generation

The program accepts the json files as valid test files. For them to be visible, they must be placed in resources/jobs folder. Then, after restarting the program, new files will automatically appear in the expandable list at the top of the page.

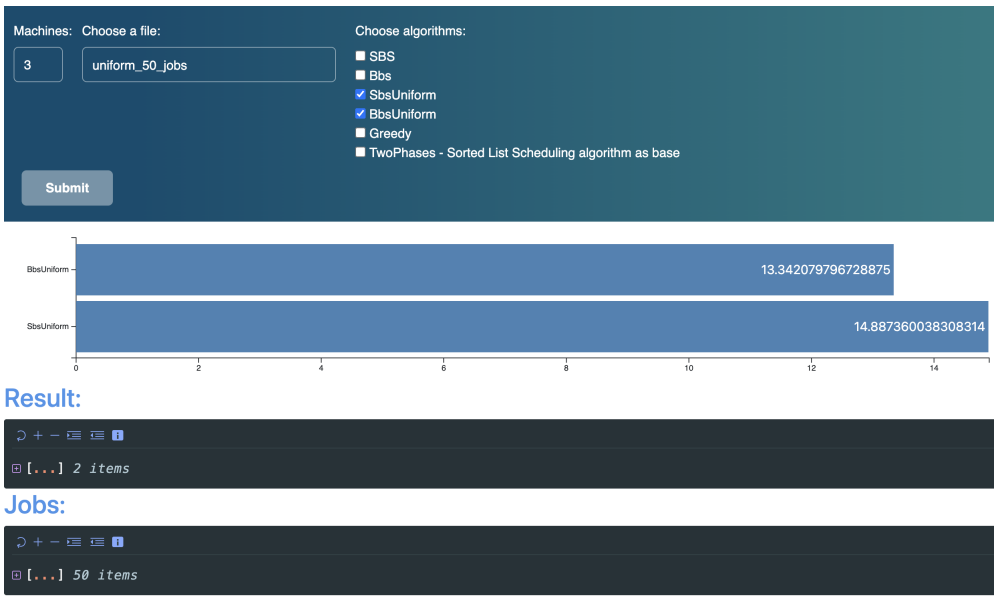


Figure 4.1: UI overview

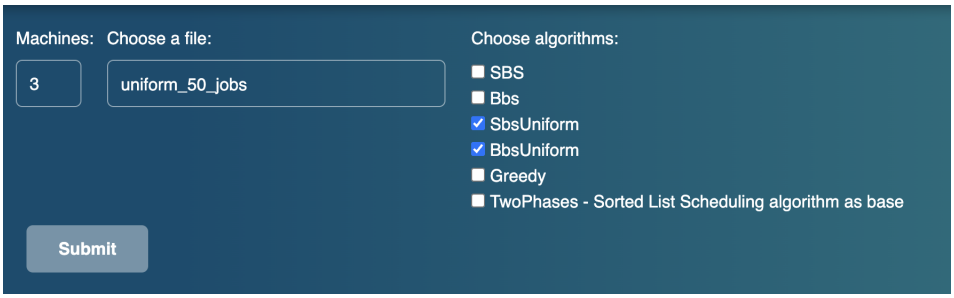


Figure 4.2: Top section of the UI

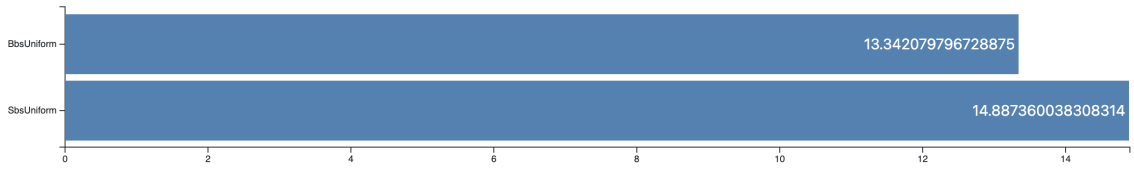


Figure 4.3: Middle section of the UI

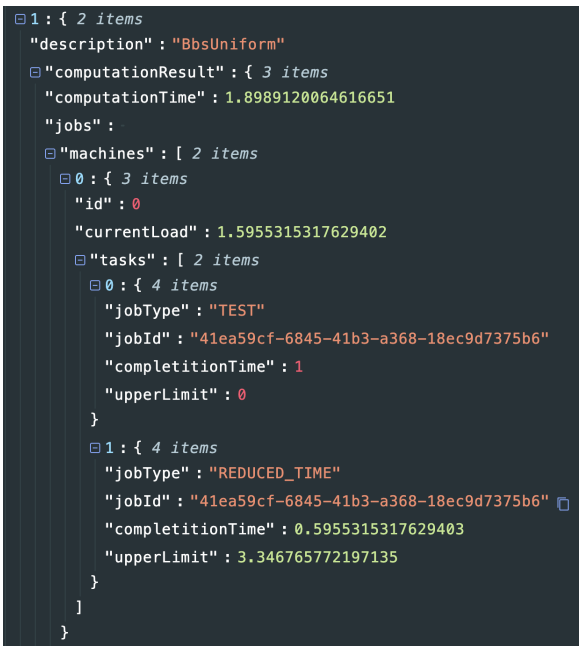


Figure 4.4: Part of expanded results section of the UI

Tests can be generated manually by changing existing files while preserving the general structure of the file, so that it is still recognisable by the program.

More convenient is automatic build-in generation of tests. Test are created with each refresh of the application as long as generateToFile() function in SchedulerService is present. By default this function is placed as a comment in the correct place to avoid creating too many test files by mistake.

Parameters of the created files are adjustable inside SchedulerService class. By default the created file consists of 50 jobs. The upper limit, reduced computation time and test time of each job are random values from exponential distribution with one caveat stating that reduced computation time cannot be bigger than upper limit in any job. In case such thing were to occur, the program reduces obtained reduced computation time value to the size of the upper limit. There is also the option to create json file with uniform testing times instead, while the rest is still taken from exponential distribution. To differentiate the test files, on top of each exists a description field, which may for example contain number of jobs generated in this file or whether the testing times are uniform. It depend solely on the user, the description needs to be changed before running the application, so that it can be included in the generated file.

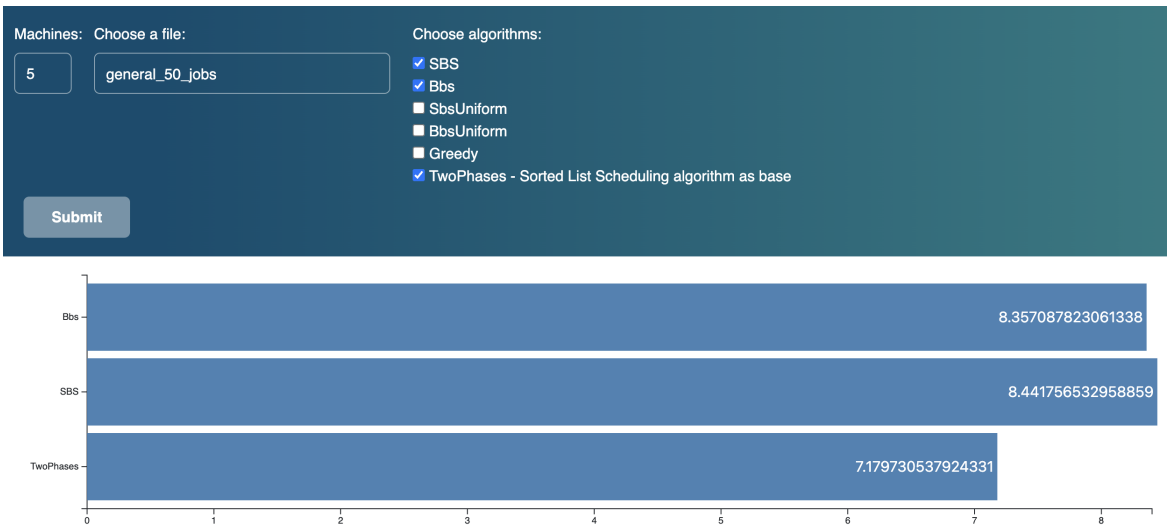


Figure 4.5: Test with 50 jobs on 5 machines

4.3 Comparison of implemented algorithms

The structure of the program makes it quite easy to test the algorithms against multiple test cases and immediately see on the preview how they perform. In many cases they perform similarly to what shows their competitive ratio. For example in figure 4.5, where 50 jobs should be scheduled on 5 machines. SBS performance was the worst, but BBS was not much better. Two Phases algorithm operates in different setting and therefore could postpone executing tested jobs, which was apparently advantageous for this exact test data.

However, as figure 4.6 shows, the randomly generated cases can sometimes show also the reverse results. Here, SBS is marginally fastest, BBS sightly worse and Two Phases algorithm provides the worst schedule.

Competitive analysis, which uses only the worst case for computing the competitive ratio, of course does not imply the BBS algorithm is always better than SBS. It also does not provide any insight on what is the average cost of produced schedule. After reviewing over a 50 generated scenarios, the SBS algorithm seems to perform on average slightly better than BBS, which has better competitive ratio, but a sample of this size is certainly too low to draw any conclusion.

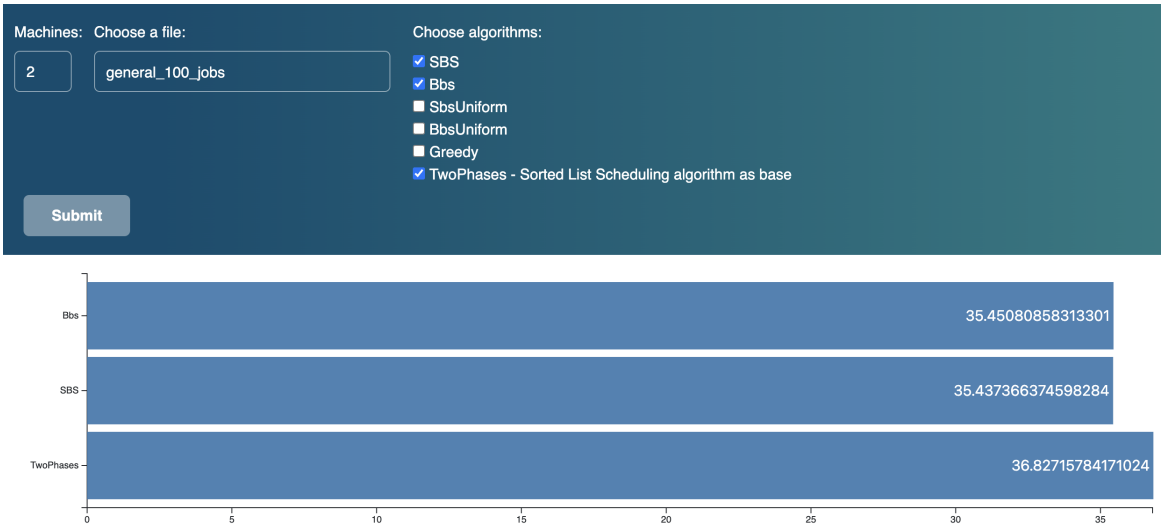


Figure 4.6: Test with 100 jobs on 2 machines

5 Conclusion

After implementing the algorithms presented in this thesis, I came to understand the difficulty in improving them. I also saw that the only metric used to asses value of an algorithm was the competitive ratio. This made me curious what is the average cost of produced schedule of these algorithm and also others I read about while gathering materials for this thesis. I am aware that proving the competitive ratio is already difficult. However one of the real-world application of these algorithms is scheduling of processes in modern computers. With modern processors performing over a 1.000.000.000 operations in one second, it seems to me that even roughly estimated average cost of the schedule created by an algorithm would be valuable in assessment which algorithm should be used in newly created processors. This rough assessment could be obtained by investing more time into expanding presented here program. This expansion would include establishing the database with testing data provided by the program and an extension for batch testing of already implemented algorithms.

From more research-oriented point of view, such extended platform could potentially help with trying out new ideas and pursuing the most promising ones. Testing new idea would then only require writing pseudocode and adjusting it to java syntax. All needed methods like addTask(job, JobType.TEST), which adds test from provided to least loaded machine are already integrated into environment. This whole structure hidden away in various classes makes algorithms easy to read and write, as each of them occupies single file and reading this file is similar to reading pseudocode. After fast implementation, running this program through the night to test new algorithm on tens of thousands cases would bring fairly good estimate on its potential.

List of Figures

Bibliography

[AE21] S. Albers and A. Eckl. "Scheduling with Testing on Multiple Identical Parallel Machines." In: *CoRR* abs/2105.02052 (2021). arXiv: 2105.02052.

[and] andypf. *json-viewer*. https://www.npmjs.com/package/@andypf/json-viewer. Accessed: 2025-04-13.

[D3] D3. *Java$_s$cript$_l$ibrary*. https://d3js.org/. Accessed: 2025-04-13.

[FW00] R. Fleischer and M. Wahl. "On-line scheduling revisited." In: *Journal of Scheduling* 3.6 (2000), pp. 343–353. DOI: https://doi.org/10.1002/1099-1425(200011/12)3:6<343::AID-JOS54>3.0.CO;2-2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1099-1425%28200011/12%293%3A6%3C343%3A%3AAID-JOS54%3E3.0.CO%3B2-2.

[GL21] M. Gong and G. Lin. "Improved Approximation Algorithms fornbsp;Multiprocessor Scheduling withnbsp;Testing." In: *Frontiers of Algorithmics: International Joint Conference, IJTCS-FAW 2021, Beijing, China, August 16–19, 2021, Proceedings*. Beijing, China: Springer-Verlag, 2021, pp. 65–77. ISBN: 978-3-030-97098-7. DOI: 10.1007/978-3-030-97099-4_5.

[Gra66] R. L. Graham. "Bounds for certain multiprocessing anomalies." In: *The Bell System Technical Journal* 45.9 (1966), pp. 1563–1581. DOI: 10.1002/j.1538-7305.1966.tb01709.x.

[RC03] J. Rudin and R. Chandrasekaran. "Improved Bounds for the Online Scheduling Problem." In: *SIAM J. Comput.* 32 (Mar. 2003), pp. 717–735. DOI: 10.1137/S0097539702403438.