THREE MACHINE ONLINE MAKESPAN WITH ARRIVAL TIMES

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

By

Omer Azmon

May 2013
The thesis of Omer Azmon is approved:

_____________________________  _________________  
Richard J. Lorentz, Ph.D.  Date

_____________________________  _________________  
Richard Covington, Ph.D.  Date

_____________________________  _________________  
John Noga, Ph.D., Chair  Date

California State University, Northridge
Dedication

This paper is dedicated to my father who always wanted me to have a graduate academic degree.
Acknowledgements

Special thanks to Professor John Noga for suggesting this problem.
Table of Contents

Copyright ii
Signature page iii
Dedication iv
Acknowledgements v
List of Figures vii
Abstract viii

1 Introduction 1
   1.1 The Problem Domain ................................................. 1
   1.2 Problem Details and Examples ..................................... 4
   1.3 A Taxonomy of Scheduling Problems ................................ 7
   1.4 Formal Statement ..................................................... 9
   1.5 Prior Work ............................................................ 10

2 The Visualization Tool 11
   2.1 Features Identified .................................................. 11
   2.2 Key Design Decisions ............................................... 13
      2.2.1 Pluggable Algorithms .......................................... 13
      2.2.2 Data and Database ............................................. 14
      2.2.3 Static System Design ......................................... 15
      2.2.4 Combinatorial Generation Design ............................ 15
      2.2.5 Selecting 3D Graph Package ................................. 17
   2.3 Verifying and Validating the Tool ................................ 18
      2.3.1 Verifying the code ............................................. 18
      2.3.2 Validating the tool ............................................ 18

3 Summary and Further Work 22

References 25
List of Figures

1.1 Sample Sequence of Events .............................................. 6
2.1 Entity Relationship Diagram ............................................. 14
2.2 Domain Class Diagram ..................................................... 15
2.3 Permutation Iterator Activity Diagram ................................. 16
2.4 Combination Iterator Activity Diagram ................................. 17
2.5 Increment Minimums Sub-Activity Diagram ............................. 18
2.6 2-Machine Trivial Algorithm Plot Diagram ......................... 19
2.7 2-Machine LPT Algorithm Plot Diagram ............................... 20
2.8 2-Machine SLEEPY Algorithm Plot Diagram ......................... 21
ABSTRACT

THREE MACHINE ONLINE MAKESPAN WITH ARRIVAL TIMES

By

Omer Azmon

Master of Science in Computer Science

This report considers a problem of scheduling jobs that arrive over time on three machines with the goal of minimizing the maximum completion time of a job (the makespan). I will describe my development of a tool and my use of the tool to perform an empirical analysis of algorithms for this problem. This work builds upon a solution for the problem with only two machines.
Chapter 1

Introduction

1.1 The Problem Domain

It is common to find oneself confronted with the issue of how to best allocate resources to various tasks. Whether one is assigning processes to machines, jobs to specialists in a machine shop, or software development tasks to engineers, one is in the business of scheduling. Machine or resource scheduling, therefore, is the process of assigning a group of tasks, jobs, or processes to one or more machines or resources. These assignments are usually made with the intention of achieving a ‘best’ allocation, where this ‘best’ is both defined and constrained by its context. That ‘best’ may be achieved by assigning the tasks or jobs based on some predetermined decision strategy. Such a strategy enters one into the study of algorithms, where procedures are mathematically studied, analyzed, and evaluated. In the study of problems and their algorithms, our problem domain is classified under the sub-heading of machine scheduling.

The term machine denotes the resource, person, physical machine, or computing machine allocated and/or required for the execution or completion of a job. Machines shall be considered a limited resource. If there were an unbounded number of machines, any optimization becomes the trivial case of "use the next machine." Machines might have different capabilities (the type of work they may perform), different capacities (the quantity of work they may perform in a unit of time), and different co-relations with other machines (the execution dependencies between machines, or jobs, such as, two adjacent machines may not work on the same type of job, or some jobs execute faster on some machines). The term machine type refers here not to the characteristics of one machine, but to the characteristics of the set of available machines. For example, if all machines have exactly the same capabilities and capacities, the machines are called identical. Similarly, if the only difference between machines is the speed with which jobs are completed then the machines are called related (if the difference is simply a constant multiplicative factor) or unrelated (if the difference is anything other than a constant multiplicative factor).

When discussing which algorithm is ‘best’, there is a need for an evaluation criteria. This is usually defined as optimizing one or more performance measures. These measures, whether duration, financial cost, or deadlines missed, are called the cost of the algorithm. For some problems there is a natural cost to processing each job and this can be summed to give the total cost of processing all of the jobs. For example, if every job had someone waiting for the outcome then we might be concerned with the time each and every job completes and consider the sum of completion times. However, we can be more concerned with finding a global aspect of the schedule. Often a more meaningful choice is to define the cost as the span of time (the duration) required to process all jobs, that is, the time from when job processing starts to when jobs processing ends. When there are two or more machines, parallel execution enters the equation, and the measurement of span begins to differ markedly from a straight sum.
A very commonly used and well studied criteria is the makespan of a schedule which is the maximum completion time of any job. Note that this is the amount of time that passes from the first job being presented until all jobs are finished. The problem this paper focuses upon will use makespan as the criteria.

There are truly trivial machine scheduling problems, which use makespan as the criteria. For example, if a set of jobs, which are completely independent of one another are to be scheduled on a single machine then any schedule that does not artificially introduce idle time is optimal. In this case, the makespan will be the sum of the processing times of all jobs. However, even relatively simple versions of machine scheduling problems are generally difficult. The problem of scheduling a set of jobs which are completely independent of one another on two machines is NP-Complete. In the real world, since there a definite need to solve problems of this type, there was and is a need for polynomial time algorithms that provide near optimal solutions for these problems. For this reason attention has primarily shifted to finding approximation algorithms.

Approximation algorithms can be divided into two classes depending upon how/when the input is provided to the algorithm. If all relevant information is immediately provided then we purely have a question of the tradeoff between the quality of the solution and the time necessary to achieve it. If the information is provided piece by piece and intermediate decisions need to be made prior to all information pieces becoming available, then we can also study the trade off between lack of knowledge and quality of solution. This second type of approximation is the study of online algorithms.

This important twist for more iterative or incremental environments got renewed interest with the rise of the Internet. The idea was to design algorithms that can better address real-world situations where the algorithm must begin making decisions before all the input data is known or available. Initial studies of these scheduling algorithms assumed that once a job arrives, no more jobs arrive until that job had been scheduled, and that jobs arrive without any time gap. In other words, while jobs must be scheduled in a particular order (without knowledge of later jobs) all jobs are available at time zero. While it is arguable that this assumption makes the problem more tractable, it would seem to reduce the direct applicability to real world problems.

For the online scheduling problem we will consider: time passes. Every job will have both a processing time and an arrival time. Jobs need not be scheduled upon arrival, but only at the point when the algorithm wishes to begin processing on that job. One version of this problem was studied by [CV98] and [NS01]. Online algorithms are more suited for an agile (just-in-time) style of work model. Questions of how to compare different online algorithms to each other and the bounding of the best possible algorithm will be addressed next.

Measuring the quality of an online algorithm is not entirely straightforward. Without immediate full knowledge of the problem instance, it is impossible (even without any time or space restrictions on computation) for an online algorithm to always provide the optimum solution. Still the quality of different online algorithms needs to be measured. A new
Imagine a problem instance consisting of a particular set of jobs. An online algorithm will provide a feasible schedule for this set of jobs. There will also be a truly optimal way to schedule the jobs. The costs of these two schedules can be computed and a cost ratio found. Note that the cost ratio must be at least 1, since the online algorithm cannot outperform the true optimum, and the smaller the ratio is, the closer to optimum the online algorithm performed. If there was one specific set of jobs, which was truly “typical” then computing the cost ratio on that set would be a reasonable measure, but in most cases there is no obvious instance. If instead one calculated the supremum of the cost ratio for every possible set of jobs then one would at least have a worst-case performance guarantee. This is the competitive ratio.

To actually calculate an approximation algorithm’s competitive ratio by the process described is impossible, since the set of job instances is infinite. However, each ratio is a lower bound on the true value. Further, if tested on instances representative of all possible inputs the result should approach the correct value and give insight into where the algorithm is weak and guide a more formal proof.

When analyzing online algorithms, it is often useful to cast the problem into the mathematical field of game theory [BEY98]. In this case, one might imagine the algorithm as being player one and an adversary, player two, submits the jobs. In each move, the adversary reveals a job and the player must respond by creating a schedule. We can group the various types of adversaries that may be constructed into three main categories: Random, Omniscient, and Oblivious. The Random adversary selects jobs, release times, and any other required characteristics according to particular distributions, respectively. This is the easiest adversary to implement in a computer program. The omniscient adversary knows the player’s algorithm, the current state of the game, and all moves that have transpired, and makes every attempt to cause difficulty for the player. On the other hand, an oblivious adversary knows the player’s algorithm, and the current state of the game, but does not keep track of what has transpired while still attempting to cause difficulty for the player. Note that in any finite game a random adversary will eventual produce the same sequence as an omniscient or oblivious adversary would have. Since we are interested only in the worse case scenario, all three are equivalent in deterministic finite games. This and our earlier assumption that all job sets are finite, conveniently, allows us to choose one adversary type for writing the visualization computer program described in Chapter 2, and another for proving theorems.
1.2 Problem Details and Examples

The problem considered in the remainder of this report will be the online three machine-scheduling problem with arrival times and the quality of an algorithm will be judged by its competitive ratio.

Formally, a problem instance will consist of a set of jobs $J = \{j_1, j_2, \ldots, j_n\}$ where each job $j_i = (p_i, r_i)$ has a processing time and a release time. A feasible schedule will define a machine $m_i \in \{1, 2, 3\}$ and a start time $s_i$ for each job such that $s_i \geq r_i$ where if $s_i \leq s_j \leq s_i + p_i$ then $m_i \neq m_j$. An online algorithm must determine $m_i$ and $s_i$ without knowledge of any job with $r_j > s_i$. The cost of a schedule is $\max\{s_i + p_i\}$.

I have selected this problem for investigation because it appears that results for 3 (or the more general $m$) machines has the potential to shed light on many real-life problems, where decisions must be made about resource assignment before all the facts are available.

Note that we may assume, without loss of generality, that there is a job with release time $r_i = 0$ as the time spent waiting for the first job, need not affect either algorithm, optimal or online. Another important point to note here is that we can assume, without loss of generality, that all sets of such jobs are finite, and that each and every job in each set is of finite duration, for if they were not, the makespan is unbounded, resulting in an undefined cost ratio.

The problem is usually informally described as taking place in a manufacturing shop. However, resolving scheduling problems is not limited to manufacturing, and in computer science, for example, these problems arise when executing threads/processes and when communication streams are allocated bandwidth. Scheduling problems may be solved with various goals in mind, such as balancing system load or achieving a given quality of service. Together, the requirement to execute more than one job at the same time (that is, multitasking), and the sequential execution requirement, give rise to the need to determine job execution order (scheduling).

When designing a system, a programmer must consider which algorithm will be the ‘best’ fit for that system. But, there is no universal ‘best’. Rather each solution has its own performance characteristics, which vary based on the constraints placed on the solving algorithm and on algorithm’s implementation. Some of these constraints are explicitly stated in the problem name, while others are implied. The constraints that apply to the three machine online with arrival times problem under investigation follow.

During the processing of any specific set of jobs, the number of machines does not change for any reason, including machine failures. While this simplification may not match reality, it is seen as a stepping-stone to more general problems. Introducing this additional complexity, therefore, must be part of future work. The most likely path is to solve the cases of 3 machines and $m$ machines with this simplification before adding in this additional difficulty. This paper focuses of the problem where the number of machines is exactly three, since one is trivial and two has already been thoroughly solved [NS01]. Therefore, this is the next logical problem to tackle.
All machines are assumed identical in all their characteristics. The scenario where one or more machines are faster or better at some operations is a valid extension of the current problem. The machines must also be independent of the jobs, so that it does not matter on which machine a job is run. Considering related or unrelated machines is another valid extension of the current problem. Finally, the machines may, but need not require any gap, such as setup time, between consecutively executed jobs. If such a gap is applied to every job, it would simply extend each job's processing time. Therefore, the setup time requirement is not considered.

Each job has a processing time which becomes known when it is received. Without this requirement, it is difficult to give any meaning to planned scheduling. This job execution time must be not only known but finite, for even a single unbound job results in an unbounded makespan for the job collection containing it. This, of course, moves any comparison out of the domain of discrete mathematics into infinite sets.

Jobs are, independent of one another, atomic, and once begun cannot be preempted. The choice of atomic jobs, that is jobs that cannot be divided into separate tasks that may be scheduled on different machines, was for simplification. The non-atomic scenario is valid extension of the current problem. The unrelated attribute means that the amount of time required to execute different jobs is unrelated. If this were not true and one could use the results of one job to speed another, that job could be considered also a sub-job of the second job. This would violate the constraint herein that all jobs are atomic. The requirement that the jobs be independent of each other, that is, there is no requirement placed on execution order, is the issue of precedence constraints, which is an important issue which has its own separate category in the taxonomy of problems.

The key constraint that distinguishes the problem studied here from the optimal and online makespan problems is that the job submission is online, that is, the job and its characteristics are unknown to the scheduling algorithm used prior to the job’s arrival time. This is a primary characteristic of the problem being studied here. This means that each job is associated with an arrival time and may not be scheduled before this time. Given that the submission is online, without this constraint, one might have received all jobs at time zero, and thereby losing the online characteristic of the problem. To avoid this and remain an online problem, Chen and Vestjens [CV98] considered the requirement that jobs must be assigned to machines as they arrive before the next job is analyzed. Note that we are willing to consider algorithms that allow a machine to be idle even thought jobs have arrived. I believe that this has more applicability to real life.

Finally, and most importantly, the evaluation criteria selected here limits us to only those algorithms that optimize makespan, where makespan is defined as the duration from the start of execution until the last job completes. Therefore, what is optimized is the degree of parallelism, and what is minimized is the amount of time the machines are idle.

As an example, let us assume for a moment that we have two machines and a problem instance that is a set with three jobs \{ (1, 0), (1, 0), (3, \epsilon) \} and consider an online algorithm that immediately starts a job anytime it is possible. Applying this algorithm, at time 0 we
have two idle machines and two jobs. They are both assigned. So when the third job arrives each of our two machines will be running one job with 1 unit of work. We assign the last job to one machine arbitrarily.

After all jobs complete the online makespan is 4. However, had we assigned the second job to the busy machine, we would have had an idle machine when the last job arrived. This would reduce our makespan to $3 + \epsilon$, which is the optimal for this set of jobs. This gives us a cost ratio arbitrarily close to $4/3$. 

Figure 1.1: Sample Sequence of Events
1.3 A Taxonomy of Scheduling Problems

Algorithms have their own categories and classifications. There are many that are well known: by technique, complexity, and other criteria.

What is less well known is an organization of problems. Using a taxonomy-like analogy, all the problems studied in the field of theoretical computer science might be considered as grouped into classes following the books and chapters of Knuth’s [Knuss] seminal work, and following in the same train one might define orders, families, and groups of algorithms. One of the classes so derived is ‘combinatorial problems’ defined as those problems which can be solved using brute force by checking all feasible solutions. This combinatorial class may then be broken further into several orders including optimization problems, an order which contains many algorithms, which in turn may be grouped into families including the family of scheduling problems.

The scheduling family is divided into (larger and smaller) groups based upon the type of optimization criteria and the types of restrictions applied. Possible restrictions include the number and type of machines, the optimization criteria, and the arrival characteristic of the jobs executed (also known as tasks, or processes).

The Three Identical Machine Makespan with Arrival Times problem falls into this family. Makespan indicates that the optimization criteria is by overall length of time required to process all jobs (as opposed to the sum of job-spans). ‘Identical machines’ indicates that any of the (in our case three) machines can execute (if it is available) any job, and that the cost (duration, span) of said execution would be the same. Arrival time indicates that each job is associated with a time at which it is revealed. This restriction limits the jobs that are eligible for scheduling to those whose arrival time has been reached. The three-machine optimization problem is a member of this family of problems, but the problem I studied is a “cousin”.

Since many of the optimization problems are NP-Complete or even worse, that is, no algorithm can efficiently find optimal solutions (unless P=NP). Since the real world has a need for solutions for huge data sets in reasonable time, this gives rise to the problem of finding an acceptable approximation rather than the optimum. These are the approximation problems. This generates either a tree of approximation problem classifications that mirrors the optimization problem tree, or an extra leaf for each hard optimization problem. In any case, approximations seem to belong to the optimization problem order.

Given an approximation problem, the quality of a solution to that problem is quantified by using the worst-case ratio of the performance of the approximation with the performance of the optimal solution, the approximation ratio.

The importance of this in a discussion on the taxonomy of problems is that in some real world problems, one of the key constraints is that one must begin making decisions before all the data becomes available. This generates a set of problems that have been termed ‘online problems’, and the quality of solutions to such problems are also measured by a competitive ratio. Still, these are not pure approximation problems, for the problem here
is not an attempt to reduce computational complexity, but to provide a near-optimal decision with incomplete data. This means that there are at least two categories of problems with constraints that require near-optimal solutions. Constraints on time and/or space that produce the approximation problems, and the constraint to act on incomplete (partial) data which produce the online problems. Therefore, these problems are not always grouped under optimization problems. Instead, they often receive their own order of near-optimization problems. This order is sub-divided into at least two sub-orders: approximation problems, and online problems. Each of these sub-orders may contain versions of all members of the family found in the optimization order, and specifically, they contain the job scheduling family. The problem under study is a member of the Job-Scheduling family of the On-Line sub-order of problems. This would seem to be the end, but extensive research went into the many members of this family. Some were found to be trivial, and other more central. In their paper on these problems, Lawler et al. suggested to classify these problems based on three criteria: “the machine environment, the job characteristics, and the optimality criterion” [LLKS93]. They introduce a three part notation with each part separated by a vertical bar as follows: 

\[ \text{machine-environment} — \text{job-characteristics} — \text{optimality-criterion} \]

where:

- **machine-environment** is empty for one machine, P for identical parallel machines, Q for uniform parallel machines, or R for unrelated parallel machines. The parallel machines identifier may be followed by the number of machines, if constant and greater than one.

- **job-characteristics** lists any special requirements/specifications of the jobs such as \(\text{pmtn}\) if preemption is allowed, \(\text{prec}\) if there are precedence constraints, \(r_j\) if there are release times. Followed by additional processing requirements, if any.

- **optimality-criterion** indicates what criteria must be minimized, such as \(C_j\) for completion time, \(L_j\) for lateness, \(T_j\) for tardiness, or \(U_j\) for unit penalty. Additional notations, such as \(\Sigma C_j\) for total completion time, or \(L_{max}\) for the maximum lateness, are also available for more involved criteria. Using this 3 field notation the problem under investigation is \(P3|r_j|C_{max}\).

There is no standardized notation to indicate whether the algorithm is online or similar variants. In the tool developed as part of this project, we will append a fourth part that will indicate the algorithm under consideration (such as, \(\text{OPT}\) for the optimal, or \(\text{LPT}\) for longest processing time). Should further distinction be required, such as between the different variants approaching a better solution for 3 machine, we will simply append a hyphen to the well known name, and follow it with the variant name, such as ‘SLEEPY-0-0-a’.
1.4 Formal Statement

This formal statement of the problem is slanted toward making it easier to correlate the results of the visualization tool discussed later, with any formal results. In each execution of the algorithm a collection of jobs $J$ is submitted to the algorithm to be executed on a set of machines $M$. The set of jobs is allowed to have multiple jobs with identical processing and release time.

Though the set is not technically a list (as no ordering is imposed for processing purposes), we will denote the jobs $j_1, j_2, \ldots, j_n$. Each machine in the set of machines $M$ shall be denoted as $M_k$. Since all machines are identical the notation is simply to distinguish them, and since the problem under study is for three machines, the set of machines used is $M = \{M_1, M_2, M_3\}$.

An unspecified algorithm shall be denoted as $A$, while a specific algorithm shall be denoted with its name, such as ‘SLEEPY’.

Each job in the set shall be represented by an ordered pair composed of the processing time, and the release time and notated as $(p_j, r_j)$, where $j$ is the job number. When an algorithm $A$ submits a job to some machine $m_j$ to begin execution at time $s_j$, the result is another ordered pair $(s_j, m_j)$, where $s_j$ is the start time for job $j$, and $m_j \in M$ is the machine that is used to execute that job. Execution for all jobs in some collection $J$, therefore, ends at $\max \{s_j + p_j\}$. This is the makespan of the schedule that $A$ assigns to $J$.

The cost ratio of algorithm $A$ on a set of jobs $J$ is defined as

$$\frac{\text{span}_A(J)}{\text{span}_{OPT}(J)}$$

and will be notated as $\text{costRatio}(A,J)$.

Recall that the competitive ratio of algorithm $A$ is defined as the worst possible cost ratio:

$$\sup_J \text{costRatio}(A, J).$$

In most cases, we will seek an algorithm with the best possible competitive ratio:

$$\inf_A \sup_J \text{costRatio}(A, J).$$
1.5 Prior Work

The problem considered here was first studied by Chen and Vestjens [CV98]. They considered only the LPT algorithm, a greedy algorithm that will immediately start the processing of a job any time there is a unoccupied machine and a job available. They also showed a lower bound of $\frac{3-\sqrt{5}}{2}$ for 2 machines and a slightly smaller lower bound for general $m$.

In 2001, [NS01] presented the algorithm SLEEPY for the 2 machine case, and showed that it achieved the lower bound ratio of $\frac{3-\sqrt{5}}{2} \approx 1.38198$.

There is an immense amount of work on both online problems and scheduling problems. We cite the seminal work below.

One of the earliest papers on scheduling problems which used the idea of comparing results to the true optimum was by Graham [Gra69]. Sleator and Tarjan [ST85] investigated the List Update Problem and Paging in using the concept of online algorithms. Hochbaum and Shmoys [HS87] presented a family of polynomial time (in the input size) algorithms (PTAS) that can produce a solution arbitrarily close to the true optimum (exponential in $\epsilon$ the closeness measure) for the problem without release times. Borodin and El-Yaniv [BEY98] present known results for online algorithms including for machine scheduling and present interesting techniques to prove lower and upper bounds for online problems by exploiting their isomorphism to zero-sum games.
Chapter 2

The Visualization Tool

2.1 Features Identified

The online three-machine makespan with arrival times problem appears to be difficult. The set of possible algorithms is immense. The quality of any given algorithm is not immediately clear. Further, for several natural parameterized sets of algorithms, finding the optimal choice of the parameters has proved elusive to researchers.

Before spending time attempting to prove that an algorithm is the best possible solution, it would be convenient if we could get some sense of how it performs, where its weaknesses are, and whether such weakness could be overcome or are endemic to the online problem. Of the many ways to obtain such a feel for an algorithm and for the characteristics of its worst case scenarios, we chose the development of a tool for visualizing the solution space. While such a tool is no small effort, for a single candidate algorithm, as we expected that we might have to pore through more than one algorithm to find the answer, it was deemed expedient to develop such a tool. The benefits would include the ability to analyze more algorithms and variations faster, and thereby gaining an empirical sense before expending considerable effort to prove (or fail to prove) that the algorithm chosen actually performs well.

As the tool was developed primarily for this research project, and was not developed for long term use and not as an aid to learning a particular language or tool, it was determined that the tools and languages should be chosen with the idea of minimal costs in time and money. This meant a preference for languages supported by open-source tools; and that the choice of tools and language should be limited to those with which the developer is already familiar. Therefore, I selected Java 6 and the NetBeans 7.2 IDE. Furthermore, any library used which is not part of the J2SE 6 platform, should follow the same constraints. Given that the only anticipated user is the developer, no need for any type of online help or user documentation is required. However, the code itself, as the project was expected to require many months of work, and additionally might have to be reviewed, was required to contain sufficient and appropriate Java Doc to allow an experienced Java developer to understand what is going on, take over, and make additions and corrections. This takes care of the forgetfulness aspects. Any database chosen had to follow the requirements of free, open-source, and meeting reasonable performance requirements (given that the sample data sets were anticipated to be very large). Finally, though the code is not intended to be production ready, one must have good confidence in the results, therefore, both unit and functional end-to-end tests were deemed required. Unit test were limited to algorithmic non-UI code with as close as possible to a 100% path coverage. Functional end-to-end tests were required of each algorithm added by running the calculation against a strategically selected sample designed to disrupt that algorithm’s code. No further mandated constraints were placed on the development of the tool.
The functional requirements were grouped and pruned into several key features. The ability to either easily generate a new database with all needed features with one GUI command, or to select one from several earlier created databases, was important to allow the preservation of different test runs and their results. A method, including both algorithmic tools and a UI to generate sample sets and store them, and to create additional samples between existing samples, was required both to allow inspection of target samples and to save the time needed to generate the new samples. Once samples are available, it is required to have a UI to calculate the makespan using various optimal and online algorithms, and store the results for later use. The choice of algorithms that are to be implemented should not be required to be known during the tool’s development. Instead, it should be possible to plug in new or modified algorithms and test their behavior. There also has to be a tool to plot the results in a 3D plot of the online algorithm. The X and Y axes of the graph should be selectable as any two of the number of jobs in a sample versus the largest job in a sample versus the largest release time, with the Z axis showing the makespan for the optimal algorithm and the competitive ratio for online algorithms. Additionally, it was determined that it would be good if there was a way to view which algorithms are available in general, and which have been added to the current database.
2.2 Key Design Decisions

The two most important design decisions were how to make the algorithms pluggable so that new algorithms and variations may be introduced without requiring a re-write or major refactor of the application, and how would the large amounts of data expected be organized and stored, including database selection, and data access design. Lesser decisions included the design of generic combination and permutation generators for use in sample generation and in makespan calculations, and the selection of the package used to generate the 3D visualizations.

2.2.1 Pluggable Algorithms

The first step in achieving pluggable algorithms was the development of a reference single algorithm makespan calculator for the optimal (with and without arrival times). This code was then analyzed and refactored so as to separate into generic code and algorithm specific code. The result were two classes: a BestMakeSpanFinder that performed all the generic work, such as permuting all the job-machine assignments, and then it invoked an instance of the SpanMaker Interface to calculate a makespan for the given arrangement. In that way and given a job sample, the BestMakeSpanFinder would find the best makespan for that sample for any algorithm for which it was provided as an implementation.

To load and bind the various implementations, since the design pattern of the Java Service Provider Interface (SPI) is so widely used in the Java world, it was decided to use it as a design reference for the pluggable MAKESPAN calculation algorithms. In the SPI pattern, “a service is a well-known set of interfaces and (usually abstract) classes. A service provider is a specific implementation of a service. The classes in a provider typically implement the interfaces and subclass the classes defined in the service itself” [Ora85]. In the visualization tool, the service is the SpanMaker, and the various implementations are the providers. These provider are loaded and selected by their self-published algorithm name, which is also used as an attribute of the Algorithm data type in the database. This allows the dynamic inclusion and selection of an Algorithm.

Implementations of the SpanMaker interface, therefore, provide makespan calculation services of a single arrangement of a single sample of jobs to the rest of the application. Each implementation is identified by invoking its getAlgorithm method which provides its identifying name (duplicates are signaled as errors by the SPI mechanism). The implementation of the getMachines method provides the number of machines that are used by that algorithm implementation. The jobs and permutations attributes, which identify the jobs in a sample and the requirements placed on job-machine associations, have setters and getters. The invocation of the makespan method (after setting the jobs and permutations attributes) results in a makespan calculation for a given configuration of jobs. Finally, the implementation of the getJobSubmittalLog method returns a record of each job’s machine assignment, and submission time. To assist with the process of implementing a new algorithm, a DefaultSpanMaker abstract class was provided that is designed to simplify that process. This same class is already in use in the pre-existing Algorithms included with the original tool.
2.2.2 Data and Database

The visualization tool by design is a single user application. Therefore, we should be able to eliminate the cost, administration effort, and communication overhead of a stand-alone database management system by selecting an embedded database. From the list of databases that I was already familiar with, this therefore eliminated Oracle, SQLServer, MySQL, and PostgreSQL. Since the development is constrained to the Java programming language, this limited us further to Derby (aka JavaDB), HyperSQL Database (aka HSQLDB), and to H2, all of which provide the additional benefit of support for in-memory databases as part of the open-source solution, allowing for improved performance in the handling of the expected large data sets. Of these the best performing by far was H2, which is essentially a rewrite of HyperSQL to improve performance and eliminate some cumbersome dependencies. One of the apparent drawbacks plays a big part in its advantage. The database is limited to a max of 64TB. This same limitation, as opposed to the unlimited size design for the other databases, is one of the contributing factors to its speed. The choice, therefore, was obviously H2.

During the functional and acceptance testing phases of the tool, an oversight was discovered. There was no way to snapshot and restart an H2 in-memory database. As the calculation of the optimal makespan for a reasonable sample easily takes tens, if not hundreds of CPU hours, the lack of the capability to checkpoint the operation and continue at a later time was fatal. We were able to locate only one in-memory database that would support such checkpoints, would be available to us, is compatible with Java, and does not require a separate server to run: VoltDB. We were able to obtain a no-charge license for use of the tool limited to this research project. The only changes that were required to the code already developed were some minor changes to the DAO layer. The performance gain over H2 used in disk mode, changed from well over an hour for generating a sample of 3 jobs with durations between one and ten units, and first release times between zero and nine units, to sub-second execution time with VoltDB.

The data that we want to store in the database, at first glance, is very simplistic. However, our first effort was to store the small set of jobs and their submittals inside their ‘Sample’ and ‘Makespan’ records respectively, showed us that under JPA, we are not able to keep the data stored as designed: with the child records stored as columns in the parent table, and yet materializing as Job and Submittal as separate objects their parent object. We next tried JDO (Java Data Objects), which is designed to accommodate more flexible data arrangements, and while it worked on paper, we were unable to locate an open source DAO implementation that would not crash under such an arrangement. We returned to JPA and

![Figure 2.1: Entity Relationship Diagram](image)

next tried JDO (Java Data Objects), which is designed to accommodate more flexible data arrangements, and while it worked on paper, we were unable to locate an open source DAO implementation that would not crash under such an arrangement. We returned to JPA and
split the jobs and the submittals to a dependent table. However, the resulting data objects failed to hide the artificial key (record id) in the sense that everything was readable and writable, and more importantly produced a considerable performance penalty. So, we fell back on an old favorite, Spring DAO. This gave us the necessary control to create DAO objects that behaved just as the requirements and design specified.

2.2.3 Static System Design

The static system design shown here, ignoring details of UI implementation and other implementation specific classes, embodies the data design and the use of algorithms as a service, as described above. However, one design decision should be mentioned here: the representation of the duration (aka run-time), the release time, and the actual submittal time of each job. At first blush, we should implement them as some computer approximation of a real number, such as Float or Double. After all, we know that some algorithms, for example, have optimal submittal times that are related to the $\sqrt{5}$. This, however, overcomplicates a number of issues, given that we are only trying to produce a visualization and not exact result. Still, using integers we could miss the fact that the ideal submittal time is somewhere between two integers. So, we settled on integers with an implied fixed precision as set by the tester with a default for three implied places past the decimal. However, as we expected these values to change and to aid in eliminating some potential coding errors, the decision to note was to make both time and duration opaque data types in our model and not integers.

2.2.4 Combinatorial Generation Design

Since our goal is to produce a visualization of the performance of an algorithm, with a fixed number of machines, and a fixed limit on the number of jobs, where the X-axis is the time the latest job is released, the Y-axis is the run duration of the longest job, and the Z-axis of all samples aggregated at that X-Y projection is either the worst makespan, for optimal algorithms, or the worst competitive ratio for other algorithms. That requires that each
sample is tagged with its longest job, and latest job release time. In the detail data design, we addressed the issue that there could be many samples with the same X-Y coordinates, by assigning each sample an artificial id. This is exactly what the outer algorithm for sample generation does. For each latest job release time in the range selected by the user, for each run duration in the range selected by the user, we invoke the sample generator. The sample generator then must produce all permutations that meet the requirements of the limit on the number of jobs, the latest job release time, and the longest job run duration. It would be possible for each number of jobs to simply permute all possible arrangements of job release time, where at least one job meets the required latest job release time, and cross this with a permutation of all possible arrangements of run duration, where at least one job meets the required maximal run duration, and at least one job is released at time zero. However, a quick examination will demonstrate that the result is full of duplicates, and that it is sufficient to cross the permutation of release times with a combination of run durations (or the other way around). The implementation could be either using the Visitor pattern or the Iterator pattern. The advantage of the Iterator pattern in this case was seen in that we needed to use the permutation generation in the calculation of sample MAKESPAN using a pluggable algorithm, where the Visitor pattern is obviously not a choice. So we set out to produce a generic Permutation Iterator and Combination Iterator, whose only requirement on the set to be permuted was that it have a total order, and with the requirement that the user might request to include only every $n^{th}$ element. This last to allow us to start with 1000, 2000, 3000,... in the sense of 1.000, 2.000, 3.000,... and come back later to fill in the

![Figure 2.3: Permutation Iterator Activity Diagram](image-url)
gaps at interesting spots where we might want more detail.

Figure 2.4: Combination Iterator Activity Diagram

In terms of processing time, the main difference between permutations and combinations is that the first is $k^n$ and the second is $\binom{n}{k}$. In terms of code, this means that if we start with the permutation algorithm, we simply need to change the reset after a selection reaches the maximum value in the set. In case of permutations, we reset to the same minimum every time, while in combination we need to increment these minimums, so we do not repeat combinations. That is the function of the inner algorithm for combination shown.

The last detail to note is that the actual implementation uses a Total Ordered Set abstract data type, which we developed to further isolate these generic generators from the set, which they are iterating.

The MAKESPAN calculation uses the permutation generator to permute the job-machine combination when attempting to find the optimum span that such an algorithm can produce.

2.2.5 Selecting 3D Graph Package

The 3D Graph Package we searched for had to be Open Source and produce 3D surface plots, as oppose to 3D ‘suped up’ business graphics, or 3D images. The goal was to go from a result matrix to graph with as little effort from our side, and yet we needed the result to work. We tested several package and settled on a package hosted by Google Code written by Eric Aro called SurfacePlotter.
2.3 Verifying and Validating the Tool

We performed both verification and validation of the tool before using it to attempt to approach the optimal on-line algorithm for 3-machine MAKESPAN.

2.3.1 Verifying the code

The technique used to verify that code does what we expect is through unit testing of all none-UI object, with code coverage, and static code analysis. The unit tests developed to test that the code does what we expect, have achieved 100% path coverage of all classes, except for one class that included exception catching for exceptions that should never occur and we were unable to produce in testing. We also ran the code through CheckStyle and PMD, and addressed all issues identified. Once we reached our goal on all these fronts, we then reran everything one more time to ensure we are still where we thought we were. This gave us good confidence that the code does exactly what we thought it should.

2.3.2 Validating the tool

The technique used to validate the tool was to use it on algorithms with known results, before we venture into the unknown. We started with a trivial algorithm, and then continued with LPT for 2 machines, and then SLEEPY for 2 machines. For each of these test follows a description of the algorithm tested, the known results, and tool results. Each test was performed in two stages. First, a set of strategically selected test cases was run using end-to-end functional test automation against an expected result determined manually. Then a full sample of two jobs cases with arrival times between 0.000 and 9.000 units, and run durations between 1.000 and 10.000 units was run. This full sample was then spot checked manually, and additionally all apparent outliers were investigated. Every time an issue was identified the full battery of unit tests and the functional tests was rerun from the top.
2.3.2.1 Validating the tool against a Trivial Algorithm

The trivial algorithm used was defined simply to assign a job to the next available machine. The known result for this algorithm is a competitive ratio of $\frac{3}{2}$ for 2 machines.

Figure 2.6: 2-Machine Trivial Algorithm Plot Diagram
2.3.2.2 Validating the tool against 2-machine LPT Online Algorithm

The LPT algorithm sorts the jobs available to schedule by their processing time and then assigns them to the machine with the earliest end time so far. The known result for his algorithm is a competitive ratio of $\frac{3}{2}$ [CV98].

Figure 2.7: 2-Machine LPT Algorithm Plot Diagram
2.3.2.3 Validating the tool against 2-machine SLEEPY

The SLEEPY algorithm is when there are jobs to schedule: 1. If no machine is idle, 
wait; 2. If one machine is idle, sleep until the active machine has run the current job for 
\( \frac{3-\sqrt{5}}{2} \) of its total run duration, and then start the available job with the largest processing
time; 3. If both machines are idle, start the available job with the largest processing time. 
The known result for his algorithm is a competitive ratio of \( \frac{5-\sqrt{5}}{2} \).
I have created an elaborate tool to determine lower bounds for the $P3|r_j|C_{max}$ problem which has been designed, implemented, and thoroughly tested. The tool also provides visual evidence as to whether a particular candidate algorithm performs well, when it performs poorly, and provides guidance for proving upper bounds for candidate algorithms.

I reviewed the database produced for two machines to determine the weaknesses of the LPT algorithm compared to the optimal, that is, at what points does the LPT on-line algorithm fail with respect to the optimal, and what if any are some of the commonalities. The review of these results confirmed my intuition. The only place I found where LPT failed relative to OPT were cases where a longer task is scheduled after a shorter one due to lack of ‘foresight’. This, of course, is a problem only if the sum of the shorter task, and the remaining work on the machine that will finish first is less than the longer task. This is where the SLEEPY algorithm helps by ‘generating’ the ‘foresight’ by waiting a bit before starting to process the shorter task, where the ‘a bit’ is calibrated so as to balance the benefit of waiting with the cost of waiting. We know from earlier work that this the optimal balance point for two machines occurs at $\frac{1-\sqrt{5}}{2} \approx 0.38198$ [NS01].

For the purposes of the discussions here, this amount of time we must wait relative to the current running jobs shall be identified as $\alpha$, and unless specified otherwise we used $\frac{1-\sqrt{5}}{2} \approx 0.38198$ as alpha.

For three machines, intuitively, we expected similar behavior of the LPT and SLEEPY algorithms. To test this, we have prepared an abstract implementation of SLEEPY for three machines where we can provide a vector of 3 delay values for: when all machines are free, when 2 machines are free, and when only 1 machine is free. As an initial run, we choose the following two vectors: $(0, 0, \alpha)$ and $(0, \alpha, \alpha)$.

To calculate the competitive ratios we had to first produce a sample set and calculate the optimum for set of jobs in the sample set. The sample we had chosen was for release times between 0.000 and 4.000 and task durations between 1.000 and 5.000. For the purpose of this test, we only tested the whole numbers permutations. While the calculation of each variation of SLEEPY took minutes, the calculations of the optimal for this limited sample took nearly 300 CPU hours.
The results obtained for the initial runs appear to have confirmed the intuition, as shown in this table of competitive ratios:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comp. Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>1.0000</td>
</tr>
<tr>
<td>SLEEPY (0, 0, α)</td>
<td>1.3818</td>
</tr>
<tr>
<td>SLEEPY (0, α, α)</td>
<td>1.4848</td>
</tr>
</tbody>
</table>

Both these results are better than the $3/2$ promised by LPT and so were encouraging. The next step was to try several intermediate values to determine if the empirical best is $(0, 0, \alpha)$, and if we can do better. The following table summarizes the results so obtained:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comp. Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>1.0000</td>
</tr>
<tr>
<td>Lower bound [CV98]</td>
<td>1.3473</td>
</tr>
<tr>
<td>SLEEPY (0, 0, α/2)</td>
<td>1.3333 LB</td>
</tr>
<tr>
<td>SLEEPY (0, 1/3α)</td>
<td>1.3718 LB</td>
</tr>
<tr>
<td>SLEEPY (0, 0, α)</td>
<td>1.3818 LB</td>
</tr>
<tr>
<td>SLEEPY (0, α/2, α)</td>
<td>1.4156 LB</td>
</tr>
<tr>
<td>SLEEPY (0, α, α)</td>
<td>1.4848 LB</td>
</tr>
<tr>
<td>SLEEPY (0, α/2, α/2)</td>
<td>1.4923 LB</td>
</tr>
<tr>
<td>LPT [CV98], aka SLEEPY (0, 0)</td>
<td>1.5000</td>
</tr>
</tbody>
</table>

From this we can see that we most likely can do better than LPT, and that SLEEPY may be such a route, as expected. However, the fact that $(0, 0, x)$ does better than $(0, x_1, x_2)$ was a surprise, since we expected the reverse.

There appeared to be a problem with the experiment, since the results for the algorithm SLEEPY$(0, 0, \alpha/2)$ were better than the proven theoretical lower bound. However, investigation has shown that this is not error in the experiments or the tool, just a limitation to empirical testing of mathematical values: the worst cases are not exactly at integer values, and yet in the experiments above, we used release time steps of 1.000. That is why SLEEPY$(0, 0, \alpha/2)$ gives us an empirical competitive ratio of $\frac{4}{3}$, which is better than the proven lower bound. However, this is true if one limits the set of job durations and release times to small integers, under 10. So while these results do not give us an exact $\alpha$, nor provide any rigorous proof that SLEEPY is the optimal algorithm with a correct alpha, they do seem to indicate that the solution might very well be in the direction of SLEEPY$(0, 0, \alpha)$.

Therefore, based upon the results found using the tool, I speculate that a natural generalization of SLEEPY would have excellent performance (and perhaps the best possible performance). While there was originally some hope to provide a formal proof that one of the candidate algorithms was actually better than the best currently known algorithm, the
time allotted was insufficient to complete formal proof, and so placed it beyond the scope of this project.

I would have liked to have been able to formally prove that SLEEPY(0,0,\sqrt{2} - 1) has a competitive ratio of \sqrt{2}, and that I would even speculate that a generalization actually achieves the best possible ratio for all \( m \) with this \( \alpha \).

Obviously, more general scenarios involving related or unrelated machines, preemption, job dependencies, and/or machine failures would be of interest.
References


