**CSE 545 – Artificial Intelligence – Final Project – Job Scheduling -Wisdom of Crowds**

Jordan Richardson and Kyle Akridge

BA Computer Science and BS Computer Science and Engineering

Speed School of Engineering

University of Louisville, USA

[Jkrich05@louisville.edu](mailto:Jkrich05@louisville.edu) and [kmakri01@louisville.edu](mailto:kmakri01@louisville.edu)

**Introduction**

This project revolves around applying a solution to an NP-Complete Problem titled /Job Scheduling. NP-Complete Problems are a class of problems in computational theory that can be NP and NP-Hard. First you should understand what these mean, P represents polynomial time and are problems that can be solved quickly (in polynomial time) by deterministic solutions. NP means non-deterministic polynomial time and are problems for which a solution can be verified quickly (in polynomial time) but may not be quickly solvable. NP-Hard means that an algorithm for solving this problem can be translated into one for solving any NP Problem.

Key features of NP-Complete problems revolve around the fact that they are solvable and verifiable in polynomial time by a non-deterministic machine, however, no known algorithm can solve these problems quickly on a deterministic machine.

Deterministic means that given a particular input and state, it will always produce the same output and transition to the same next state, which means that non-deterministic does not work this way and results can vary for the same input. These problems are important because many real-world problems in different fields are NP-Complete, making the study of these problems crucial for developing efficient algorithms.

Job Scheduling is an NP-Complete problem that involves scheduling a series of jobs, each consisting of a sequence of tasks to be processed on different machines, while each task has its own completion time and cannot be reorganized. The objective is to minimize the total time taken to complete all jobs and tasks, while only re-arranging jobs.

This concept can be hard to understand so let’s analyze a specific problem. Consider a series of 10 jobs, each job has 10 tasks that must be completed on a machine and a completion time for each task.

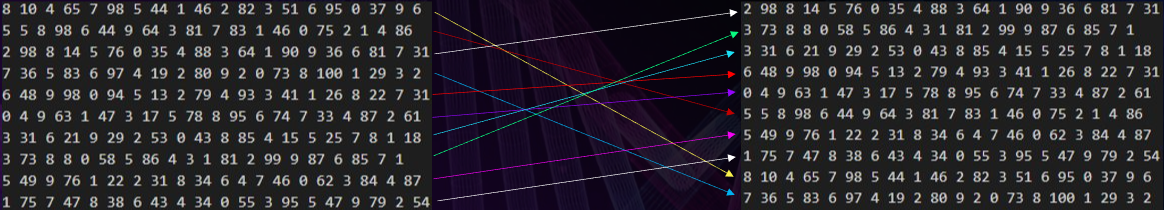
A black background with white numbers

Description automatically generated

This is what your data would look like with 10 jobs, and 10 machines and times in each job, it becomes a 10x20 array due to the machine and its completion time taking up two spots.

However, we cannot rearrange the machines/tasks that make up each job, as in real world applications of this problem, you cannot rearrange an assembly line and still end with the same product.

So, we must rearrange entire jobs to try to manage time. An example of how this works is below.



So before sorting of these jobs, they would take around 3802 seconds (about 1 hour) to complete, while after sorting they only take 3549 seconds (about 59 minutes) to complete. While this may not seem like a significant impact, this can be a permanent improvement for a business that can last for years to come, and that is where a significant impact can happen.

Let’s look at the math for this specific example.

Given that after sorting we save around 253 seconds for each job cycle.​

A company repeats these job sets around 22.7 times a day given 3802 seconds for a whole set, as (((3802)/60)/60) = 1.05 hours. We can then divide 24/1.05 to get 22.7\*365\*10, which means they complete around 82,855 individual jobs a year.​

A company utilizing Job Scheduling repeats these jobs sets around 24.3 times a day given 3549 seconds for a whole set, as (((3549)/60)/60) = 0.98 hours. We can then divide 24/0.98 to get 24.3\*365\*10, which means the complete 88,695 individual jobs in a year.​

This is a difference of 5840 in individual job completions, which can potentially be a huge profit increase many companies may be missing out on if they complete jobs without utilizing Job Scheduling.​

Here are some key points that lead to Job Scheduling being an NP-Complete problem.

*Complexity -* The complexity of job scheduling increases significantly as the number of jobs grows. This is because each additional job adds a new layer of possible permutations for the order in which jobs can be processed.​

The number of possible variations in job sequences grows factorially with the number of jobs, which means the problem's solution space expands rapidly, making it computationally intensive to find the optimal sequence.​

*Scale -* As the number of machines increases along with the number of jobs, the dimensionality of the problem also increases. This scaling issue means that the computational resources required to solve the problem exactly can quickly become prohibitive.

The complexity of finding the optimal solution in a job scheduling problem is not just exponential; it is factorial, which is even more daunting as the scale increases.**​**

*Real World Implications -*In real-world manufacturing and production settings, job scheduling complexity can lead to significant bottlenecks. The ability to effectively schedule jobs has a direct impact on the efficiency of the entire operation.​

Delays in job scheduling can cause cascading delays in production, leading to missed deadlines, increased costs, and reduced customer satisfaction.​

This will help you to understand not only why we chose this problem, but why it has real world implications today, and the importance of solutions to NP-Complete problems in modern times.

**Approach**

Our approach for Job Scheduling revolves around evolution through genetics and the wisdom of a crowd of solutions. We applied this to Job Scheduling using a *Genetic Algorithm* or GA henceforth, that can be computed and uses randomness along with the passing of best traits to children, to obtain a better solution, then after multiple iterations, combining these solutions to create a new solution. While this may sound complicated, it is very straightforward. Below is an overview, but further down are detailed explanations.

The GA starts with the given number of generations, population size, crossover rate, mutation rate, and tournament size.​

The GA functions by first creating an initial population of "X" schedules of jobs that are randomly created from the initial schedule that is being optimized.​

The GA then calculates the fitness of all schedules created from the population and documents the best schedules for the generation based on the minimum time to complete the jobs.​

A new population is then created based on the best individuals identified from their fitness which is known as elitism.​

Then tournament selection is used to select parents for the reproduction of the new population.​

Crossover is then performed between two parents essentially combining them into one.​

Mutation is also sometimes performed which makes small changes to the offspring produced from elitism and crossover, this helps to introduce more diversity into the population.​

The population is finished off by adding the offspring from the above steps and the initial elite schedules obtained.​

This process is continued until the termination condition is reached, which in this case is the number of generations initially specified.​​

The GA starts with creating an *initial population*.

*Initial Population* - The initial population is generated by creating multiple random job schedules from the data file.​

Everyone in the population is a list of jobs, and each job is represented as a list of tuples.​

Each tuple contains a machine and the time required for that machine to complete its part of the job.​

The function randomly shuffles the order of jobs for everyone in the population.​

This ensures a diverse set of starting solutions.​

After creating an Initial population, the GA then calculates the fitness (time to completion of jobs) of each member of the population and sorts them, it then selects a few best fitness individuals according to a number provided by the user, and has them navigate through a series of functions created to induce variability on a small scale in an attempt to keep their good fitness, while also allowing for variability in the next generation. These methods are namely *tournament selection*, *crossover*, and *mutation*, and these continue until an entirely new population is made from the best individuals of the previous generation.

*Tournament Selection* – Tournament selection is a method of selecting individuals from the population for breeding.​ The tournament selection function randomly selects a group of individuals from the population and then chooses the fittest individual among them.​ The size of the group is determined by tournament size which is decided by the user.​ This method balances between selecting the best individuals for breeding and maintaining diversity in the population.​

*Crossover* – The crossover process combines parts of two parent individuals to create new offspring.​ It does this by iterating over pairs of genes (jobs) from both parents.​ If a job from the first parent hasn't been added to the child yet, it's added​. If it has, the function tries to add the corresponding job from the second parent.​ This process continues until the child has a complete set of jobs which means that the offspring inherit characteristics from both parents, promoting the combination of successful traits.

*Mutation* – The mutation process introduces variation into the population.​ It randomly selects pairs of jobs within an individual and swaps them. ​ The number of swaps is determined randomly, up to half the length of the job list.​ This randomness in mutation helps to explore a wide range of solutions and prevents the algorithm from being stuck in local optima.​

These processes happen generation after generation, recording the best, and continuing. This allows the algorithm to progress to a local optima, and the variability induced by these function prevents it from getting stuck to early. We have the process reiterate multiple times, as defined by the user, and record the top 20% of solutions. We then take these solutions and analyze them to create a *Wisdom of Crowds* solution.

*Wisdom of Crowds* - The Wisdom of Crowds (WoC) is a concept that involves creating a solution based on the best-performing solutions in your population. We have a function that identifies and stores the top-performing solutions based on their fitness. This subset represents the 'best' solutions found by the genetic algorithm so far.​

We then have a function that generates a new solution based on the most common job sequences in the best solutions.​ It does this according to the following:

It counts how often each job appears in each position across all top solutions.​

The function then builds the WoC solution by selecting the most common job for each position. The selection is based on the frequency of appearance and ensuring that jobs are not duplicated.​

If any jobs are missing in the constructed solution (to ensure that every job is represented), they are added to the WoC solution.​

This process effectively creates a new solution that represents the 'wisdom' of the best solutions, assuming that the most common choices among the best solutions are likely to lead to a good overall solution​.

This is our approach to creating a viable solution for the Job Scheduling NP-Complete problem, below we will talk about how it performed.

**Overall Results**

WOC can have a massive impact both on time and the outcome, however it can also be very conditional in the aspect of the size of the dataset, generations, population size, and number of runs. This is because the longer the GA is given to run, the larger the variety of local optima’s each run may meet increases. Some local optima’s may be very close to the most optimal solution; however, some may be very far away from it. This can lead to the WOC creating a sub-optimal solution. Here is a table showing the effects of population size and generations on Wisdom of Crowds, vs the GA’s best results.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 40 Jobs / 10 Machines for all sets | Population 10 Generations 20 | Population 20 Generations 20 | Population 30 Generations 20 | Population 40 Generations 30 | Population 50 Generations 30 | Population 50 Generations 40 | Population 75 Generations 75 |
| Run 1 Best | 14025 | 13407 | 13472 | 13360 | 13217 | 13415 | 13062 |
| Run 2 Best | 13610 | 13788 | 13900 | 13090 | 12753 | 13453 | 12876 |
| Run 3 Best | 13623 | 13599 | 13907 | 13343 | 13093 | 13257 | 12825 |
| Run 4 Best | 14120 | 13205 | 13366 | 13144 | 13115 | 13092 | 12774 |
| Run 5 Best | 14025 | 13487 | 13667 | 13275 | 13501 | 12748 | 12971 |
| WOC Best | 13576 | 12948 | 13357 | 13150 | 13120 | 13076 | 12684 |

So, from this table you can get a basic grasp of how the WOC works, it has a theoretical sweet spot between time invested into the algorithm, and the Final WOC Results vs Final GA Results.

We have identified this spot to be either one of two areas.

For large datasets, if time is a concern, you want to minimize the population and allow a middle range for generations to get a fast execution and WOC generally performs much better than GA’s best results.

However, if time is not a concern, the more time you are willing to invest into the algorithm, the more it performs better because most solutions meet similar local optima’s which then enables the WOC to find optimal combinations between them.

You are generally better off either investing a lot of time, or little time, and when you meet in the middle of these, the GA will sometimes outperform the WOC, due to WOC being a combination of the GA’s results.

**Data (Describing the data we used)**

Although there were some established JSP files online that provided the optimal or suboptimal solutions to the problems, most of the solutions online were randomly generated using code or algorithms. Because of this we decided to create our own algorithm that would make our JSP files. In this way we have the ability to create and test as many files as we possibly wanted or needed. These files are created by running our “JobGenerator.py” script. You can specify the number of jobs and the number of machines, and the

**Data Results and Represented Results**

We will be completing this section using set datapoints in our GUI. We will NOT be including the best “path” for any of these due to the size of the output. Job Scheduling is a very large dataset compared to some NP-Complete problems such as TSP.

The following input values are as follows, if they are underlined, the values will not change for all runs:

Jobs: 50

Machines/Times: 20

Mutation Rate: 0.1

Crossover Rate: 0.70

Population Size: \_\_\_

Generations: \_\_\_

Elitism Count: \_\_\_

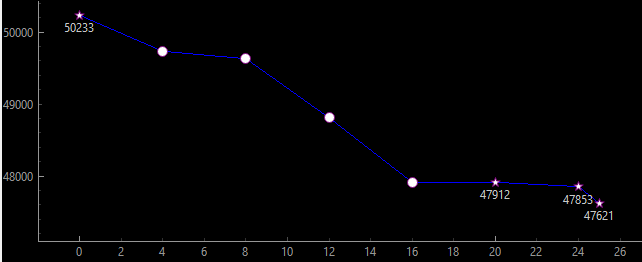
Tournament Size: \_\_\_

Number of Runs: \_\_\_

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 50 Jobs and 25 Machines in each | 25 Pop  25 Gens  1 Elite  3 Tou Size | 40 Pop  40 Gens  3 Elite  9 Tou Size | 75 Pop  25 Gens  3 Elite  9 Tou Size  10 Runs | 125 Pop  15 Gens  9 Elite  27 Tou Size  10 Runs | 100 Pop  50 Gens  9 Elite  27 Tou Size  5 Runs | 150 Pop  25 Gens  9 Elite  27 Tou Size  5 Runs | 75 Pop  200 Gens  3 Elite  9 Tou Size  5 Runs |
| Run 1 GA Bet | 48771 | 48550 | 47735 | 47746 | 46048 | 47183 | 45296 |
| Run 2 GA Best | 48277 | 47162 | 48034 | 47422 | 46550 | 47533 | 45172 |
| Run 3 GA Best | 48638 | 47601 | 47540 | 47234 | 46525 | 45579 | 44468 |
| Run 4 GA Best | 48255 | 47952 | 47579 | 47181 | 45871 | 47607 | 45513 |
| Run 5 GA Best | 47853 | 46973 | 46706 | 47891 | 46725 | 46119 | 45719 |
| Run 6 GA | N/A | N/A | 48205 | 47355 | N/A | N/A | N/A |
| Run 7 GA | N/A | N/A | 47810 | 48216 | N/A | N/A | N/A |
| Run 8 GA | N/A | N/A | 48402 | 46533 | N/A | N/A | N/A |
| Run 9 GA | N/A | N/A | 47092 | 47734 | N/A | N/A | N/A |
| Run 10 GA | N/A | N/A | 45934 | 47669 | N/A | N/A | N/A |
| WOC Best | 47621 | 47626 | 46468 | 47138 | 45168 | 46032 | 44427 |

Below are graphs for each of these runs. They plot a portion of the points to make the graph, and the last point is WOC appended to each run. The point before that will be the best result for that run of the GA.

First Data set is with 25 population, 25 generations, 1 elites, 3 tournament size, and 5 runs.



Second Data set is with 40 population, 40 generations, 3 elites, 9 tournament size, and 5 runs.

A graph of a constellation

Description automatically generated with medium confidence

Third Data set is with 75 population, 25 generations, 3 elites, 9 tournament size, and 10 runs.

A graph with dots and lines

Description automatically generated

Fourth Data set is with 125 population, 15 generations, 9 elites, 27 tournament size, and 10 runs.

A graph with dots and lines

Description automatically generated

Fifth Data set is with 100 population, 50 generations, 9 elites, 27 tournament size, and 5 runs.

A graph with dots and lines

Description automatically generated

Sixth Data set is with 150 population, 25 generations, 9 elites, 27 tournament size, and 5 runs.

A graph with dots and lines

Description automatically generated

Seventh Data set is with 75 population, 200 generations, 3 elites, 9 tournament size, and 5 runs.

A graph of a graph

Description automatically generated with medium confidence

**Discussion**

In the implementation of the Wisdom of Crowds (WoC) for Job Scheduling, the aggregation of solutions is achieved through a voting system among the top-performing solutions.

The algorithm first determines a subset of elite solutions by selecting the top 20% based on their fitness levels. These represent the most efficient job sequences identified by the genetic algorithm over multiple runs.

For the creation of the WoC solution, the algorithm examines the position of each job within these elite sequences and creates votes for their placement. The Job Schedule that garners the highest votes for a specific position is then chosen to be a part of the WoC solution. This collective decision-making process ensures that the WoC solution is a composite of the most frequently successful job placements. The final WoC solution is further optimized by adding any jobs that were missing from the voting process, thus preserving the complete set of tasks.

This solution then serves as a strong candidate for subsequent population initialization, where it is diversified through slight mutations. These variations are explored in the hope of stumbling upon an even more optimized sequence. By leveraging the collective intelligence of the best solutions, the WoC approach seeks to strike a balance between exploration and exploitation in the search space, aiming to converge towards an optimal job scheduling configuration.

**References**