Simulated Annealing: Generating Neighbors For Job Scheduling

Jonathan Lee with Billianne Schultz and Danielle Covarrubias

December 11, 2018

1 Introduction

Job scheduling is commonly known as allocating a given number of tasks across a set amount of systems or people. The goal of job scheduling is to assign jobs as efficiently as possible in order to minimize the longest time taken to complete a set of jobs. It has many real-world applications such as assigning tasks to employees or managing processes within a computer. There are numerous ways to go about solving this problem. In particular, we will take a closer look at solving the job scheduling problem using the simulated annealing algorithm.

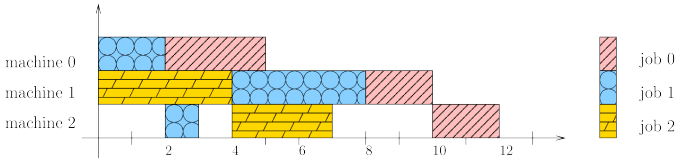

Figure 1: Job Scheduling
Source: `https://developers.google.com/optimization/scheduling/job_shop`

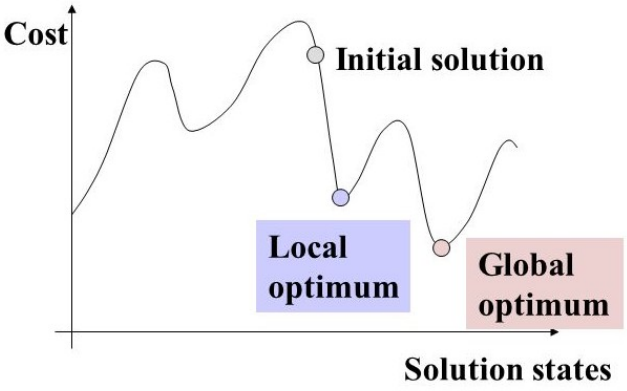

Figure 2: Simulated Annealing
Source: `https://slideplayer.com/slide/7086872/`

This project focuses on solving the job scheduling problem through various techniques to generate neighbors in simulated annealing. Simulated annealing is an algorithm designed to statistically guarantee finding an optimal solution [1]. Simulated Annealing seeks to optimize a solution by generating neighbors in a continuous solution space. This aptly fits the job scheduling problem because its movement is based on whether the generated state is better or not. If a state is more ideal, then it will move to that state, otherwise it will move with a decreasing probability. This probability can be determined by the user in the parameters for simulated annealing. The algorithm terminates when it reaches the goal or when a preset time expires. Again, this parameter can be configured by the user. Despite these attractive qualities in the pursuit of optimizing a problem, simulated annealing is known to be relatively slow compared to other optimization algorithms in addition to having little exploration.

There are numerous methods available for generating neighbors. There is no one method that outperforms the rest at every scenario. This is where testing comes into play. For our experiments, we will test six unique neighbor-generating algorithms: randomly assigning a job to a random person, replacing a random job for a random person, replacing two random jobs for two random people, replacing five random jobs for five random people, finding the person with the highest objective function value and giving their longest job to someone random and finally, finding the person with the highest objective function value and giving their longest job to the person with the lowest objective function value. Our paper will conduct a series of experiments using various methods to generate neighbors for job scheduling, comparing them against each other. The following sections of this paper will detail the methods behind these experiments, analyze the results of our experiments, and provide a glimpse into potential work to be conducted in the future.

2 Methods

These experiments will use six different methods for generating neighbors and then proceed to analyze their runtimes using an objective function. The methods for generating neighbors are as follows:

1. Generate a random solution state
2. Replace one job at random
3. Replace two jobs at random
4. Replace five random jobs at random
5. Find the person with max time and give their longest job to someone random
6. Find person with max time and give their longest job to person with shortest time

The overall objective of conducting these experiments is to analyze various neighbor-generating methods in job scheduling and to compare these techniques against each other. More specifically, the results of the objective function will indicate which method is more efficient, given a set of parameters used in job scheduling. These parameters will include data such as maximum time, number of jobs, and number of people.

The maximum time tells the program how long to continue searching for a solution before terminating. This gives our algorithm a precise cutoff point so that it does not run for unreasonably long periods of time. The number of jobs and people will determine how complex the problem can potentially be. As the number of jobs increase, the task becomes more complex due to an increase in ways to allocate jobs. This can also be said for the number of people. But unlike jobs, more people tend to indicate a speedier completion time since it lessens the chance of over-encumbering one person with too many tasks. For our experiments, we will focus our efforts on analyzing completion time when the number of jobs increase.

We will be using four different job scheduling parameters in our experiments. For each parameter we will run a total of 20 trials per neighbor-generating method. Our job scheduling parameters (where n = number of jobs and p = number of people) are as follows:

1. Max time = 100, n = 10, p = 10
2. Max time = 100, n = 20, p = 10
3. Max time = 100, n = 40, p = 10
4. Max time = 100, n = 60, p = 10

Simulated annealing features its own set of parameters including alpha, starting temperature, ending temperature, and iterations per temperature. All of these values manipulate a temperature function, indicating the current iteration that the algorithm is on [4]. In the first iteration, alpha is applied to the starting temperature by multiplying the starting temperature and alpha together, which results in a new current temperature. In the following iteration, alpha is then applied to the updated temperature using the same process. This process is repeated until the current temperature equals the user-designated ending temperature, which represents when the algorithm should stop running. This is known as the cooling process. This temperature function ultimately serves as a timer for how long the algorithm should run for. These parameters will be held constant throughout our experiments.

The values for the simulated annealing parameters are determined by user. The alpha value is inversely related to the cooling of the temperature. As alpha increases, the time it takes to cool decreases. Alpha is typically set between the 0.8 and 0.99 [4]. The starting temperature provides an initial value used in the temperature function. The starting temperature allows users to control how long an algorithm runs for. An increase in starting temperature, with all other variables held constant, implies an increase in the number of iterations required to reach the ending temperature, thus allowing an algorithm to run longer. The ending temperature serves as a finish point for the algorithm to terminate. In the cooling process, the temperature is cooled in each iteration at some user-defined value; this value is the iterations per temperature in the simulated annealing parameter.

All of these parameters work in tandem with our algorithms to output a value for comparison using a user-defined objective function. The objective function serves as a standard metric to reliably compare the performance of our algorithms. Specifically, our objective function measures the total time to complete all jobs. Essentially, we are looking for the time it takes to complete the longest set of jobs. By using this metric, we can compare the runtimes between algorithms and focus on its performance as the number of jobs increases.

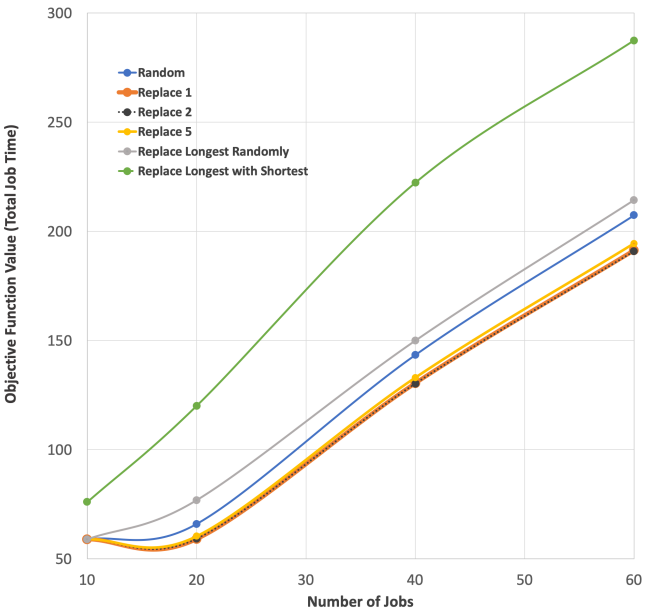

Figure 3: Average Objective Function Value of 20 Runs

The objective function will be crucial for analysis. An objective function can virtually be designed in any way to capture some sort of desired metric for analysis. This metric is ultimately determined by the user. In particular, the objective function in this specific scenario will be used in determining the viability of these neighbor generating techniques. Because we are interested in analyzing the time to completion for these neighbor generating techniques, our objective function will record the maximum time it takes to complete the job shop problem. We achieve this by essentially measuring the longest time it

takes for a person to complete all of their jobs, a common metric used throughout many job scheduling problems.

We conducted 20 trials per technique for each job scheduling parameter and then used our objective function to compare them against the values returned for the other methods using the very same objective function. The objective function for job scheduling seeks to minimize the maximum time taken to complete all jobs [3]. These experiments were then repeated, except with the caveat of a parameter change. The parameter we decided to modify was the number of jobs. By increasing the number of jobs, the problem becomes more complex. Job scheduling is known to be one the most intractable NP-hard optimization problems [2]. Changing only the number of jobs helps pinpoint which techniques would be more optimal (meaning which ones return a lower objective function value) given a set of parameters.

3 Results

Six different neighbor generating techniques were devised for these experiments: Random, Replace One, Replace Two, Replace Five, Replace Longest Randomly, and Replace Longest With Shortest. The first method, Random, generates a random solution state by randomly assigning each job to a person. Replace One replaces one job at random and Replace Two replaces two jobs at random. Replace Longest Randomly takes the person with the longest time and gives the longest job to a random person. It should be noted the person selected to take on this job could very well be the same person with the longest time already due to random chance. And lastly, Replace Longest With Shortest selects the person with the maximum time and gives their longest job to the person with the minimum time.

Table 1: Average Objective Function Value of 20 Trials

Neighbor Selection	10 jobs	20 jobs	40 jobs	60 jobs
Random	59	66	143.4	207.35
Replace One	59	59.05	130.4	191.2
Replace Two	59	59.25	130.3	190.9
Replace Five	59	60.35	133	194.45
Replace Longest Randomly	59	76.9	149.95	214.3
Replace Longest With Shortest	76.05	120.1	222.3	287.4

The parameters for simulated annealing will be alpha = 0.98, starting temperature = 600000, ending temperature = 0.25, and iterations per temperature = 500. These values are used to provide an ending point for our algorithms and will remain constant throughout all experiments. We tested these methods across four job scheduling parameters: jobs = 10, jobs = 20, jobs = 40 and when jobs = 60. We also decided to keep the maximum time and number of people constant to analyze the objective function values when the number of jobs changes. The maximum time and number of people will be held constant at 100 and 10, respectively.

We programmed a Python script to record each method's average objective function value over 20 trials for each job scheduling parameter. The objective function measures the longest time taken for all jobs to be completed. The higher the value, the longer it took a particular method to complete the job scheduling problem. An algorithm with a lower objective function value is considered more efficient than an algorithm with a higher value because it indicates a smaller amount of time to complete the same number jobs.

Table 1 showcases Replace Two to be the most effective algorithm at higher number of jobs whereas Replace One performs the best out of all our algorithms when the number of jobs equal 20. As the number of jobs increases, the problem becomes more complex. At 10 jobs, five of the six algorithms return the same objective function value with the exception of Replace Longest With Shortest. This is likely due to 10 jobs being a relatively low number of jobs, possibly suggesting unnecessary amount of work done to solve a low-complexity problem. At 20 jobs, Replace One returns the lowest value at 59.05 with Replace Two trailing close behind at 59.25. When there are 40 jobs, Replace Two barely manages to outperform Replace with 130.3 and 130.4 respectively. When the number of jobs is set to 60, we begin to see the gap between Replace One and Replace Two grow slightly larger, with Replace Two returning a lower objective function value.

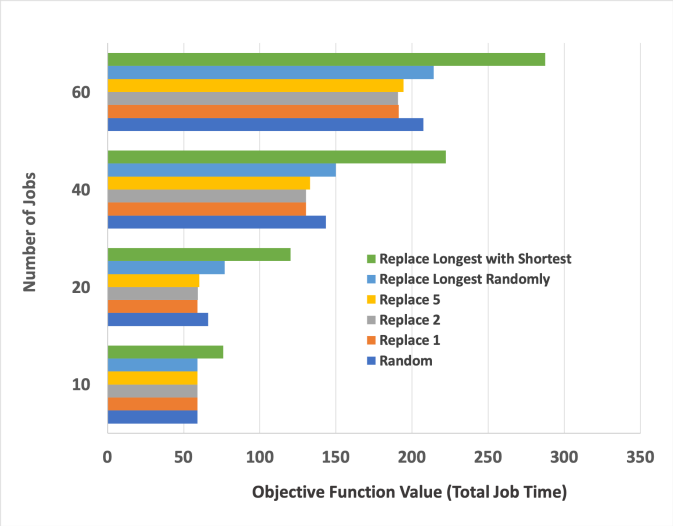

Figure 4: Average Objective Function Value of 20 Runs

When comparing six different methods for generating neighbors, replacing one to two jobs at a time results in a relatively more efficient method for job scheduling given the parameters detailed in our experiments. Figure 3 highlights the staggering difference in completion time between the Replace Longest With Shortest tech-

nique and the rest of our algorithms. This method performs worse than the other algorithms by a considerable margin. According to Table 1, Replace Two, on average, outperforms the rest of the other neighbor generating techniques by consistently returning a shorter completion time as the number of jobs increased.

Surprisingly, Random routinely outperforms two of our algorithms, namely Replace Longest Randomly and Replace Longest With Shortest. This is likely due to Replace Longest Randomly and Replace Longest With Shortest both having a more complex implementation than Random, thus taking a longer time to generate a neighbor than randomly generating neighbors. At each job scheduling parameter, generating a random solution solution state was superior to two of those methods.

4 Conclusion and Future Work

From the experiments conducted, it can be concluded that Replace Two slightly averagely outperforms the other five types of neighbor selection as the number of jobs increases, given our set of parameters. When the number of jobs equals 10, all methods performed the same with the exception of Replace Longest With Shortest having the highest objective function value. As the number of jobs increase to 20, Replace One barely outperforms Replace Two by less than a unit. As the number of jobs increase to 40 and 60, Replace Two begins to slightly outperform Replace One, albeit by less than a unit in these instances.

It is evident that Replace Longest With Shortest is the least effective neighbor generating technique, having the highest objective function value across all tests. But surprisingly, Random outperforms even two of our algorithms at certain parameters. This speaks to the effectiveness of generating a random solution state in simulated annealing. But when randomly replacing one to two jobs, the job shop problem is solved even quicker.

For future experiments, one might be interested in increasing the number of jobs to an even greater number and testing whether if Replace Two continues the trend of outperforming the rest of the other types of neighbor selection as well as capturing marginal differences between each method. It may also be worthwhile to include more methods for generating neighbors or possibly modify the other parameters such as max time or number of people. One could also configure the parameters in the simulated annealing algorithm (such as alpha or starting temperature) to influence the cooling process.

In addition to our current neighbor-generating techniques, other valid considerations could possibly include replace three or four jobs at random. It is evident that under our parameters, Replace Five is made obsolete by Replace One and Replace Two. But it remains to be seen whether Replace Two and Replace One would continue to outperform the previously proposed techniques under our established parameters. It may also prove productive to test these methods at exponentially higher number of jobs, creating a more complex job scheduling problem.

References

[1] Ingber, Lester. "Simulated Annealing: Practice versus Theory," *Pergamon Press Ltd*, vol. 18, no. 11, pp. 29-57, 1993.

[2] Martin, P. and Shmoys, D.B. "A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem," *Springer, Berlin, Heidelberg*, vol. 1084, pp. 389-403, 1996.

[3] Google. `https://developers.google.com/optimization/scheduling/job_shop`. Online, The Job Shop Problem.

[4] Geltman, Katrina Ellison. `http://katrinaeg.com/simulated-annealing.html`. Online, The Simulated Annealing Algorithm.