Colab: https://colab.research.google.com/drive/1cz5v248VBO16dRgrUj9Lt3CbEpu_ZWIr?usp=sharing

GitHub:
https://github.com/johnsonlarryl/csce_5210_pre_release/tree/main/job_scheduler

Design Approach:
The software implementation I used was the same as outlined in my design. The lower-level implementation used an Object-Oriented Design (OOD) and Object-Oriented Programming (OOP) approach from a design pattern perspective. This resulted in a cohesive software architecture where classes and methods performed very specific functions where objects collaborated with one another for city map generation, trip simulation, and traffic analysis. This provided a clean separation between objects and clearly defined contracts or interfaces between objects.

In addition, the algorithmic approach used simulated annealing (SA), which is a constraint satisfaction problem (CSP). The SA algorithm "is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. The name of the algorithm comes from the annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties.[1] The SA decomposition of the problem formulation is a decision-making process based on the evaluation of the objective function at different points in the state space, guided by a Boltzmann probabilistic mechanism for escaping local optima. However, there is no guarantee that the solution will find global minimum.

In order to represent the schedule that is ultimately fed into algorithm a network graph represents the job scheduling system. More specifically, each node represents a job operation, and each edge represents the machines processing these operations in parallel using the weights on each edge as the compute time for each operation. This is based on the configuration parameters that are fed in such as the number of machines and number of operations per machine. Lastly each job is separated by a list that specifies the order of the job based on the successor function that randomly replaces jobs. In addition, operations representing a sequence of tasks can also be rearranged. However, the jobs must be executed in sequence based on the project requirements.

R3:
The makespan computed was 27 as the best solution for the job sequence = [4, 1, 5, 3, 2]. This is true in various scenarios due to the constraint of the problem to process the tasks for each operation in sequence. In addition, each job must finish completed before another job executes thereby causing some idle time even though the system is performing parallel processing.

[1] https://en.wikipedia.org/wiki/Simulated_annealing

R4:

The Job Schedule was randomly generated based on the following requirements:

Number Jobs	50
Number of Machines	5
Operations per Job	3
Random times per operation	(5, 50)

Trials for various executions of the aforementioned sequences:

Makespan Trial 1	526
Makespan Trial 2	517
Makespan Trial 3	511

R5:

The Job Schedule was randomly generated based on the following requirements:

Number Jobs	50
Number of Machines	3
Operations per Job	5
Random times per operation	(5, 50)

Trials for various executions of the aforementioned sequences:

Makespan Trial 1	710
Makespan Trial 2	762
Makespan Trial 3	724

R6:

There appears to be the optimal solution returned on various trials. This is due in large part to the constraint that each job must finish before the next job can execute. As a result, without compare every single solution of the various permutations it would be difficult to verify these results without an exhaustive computational experiment. It is safe to say that various job sequences with similar constraints (same number of jobs, same quantity of groupings of operations, same number of machines, and the same number of operations per machine) will return similar results.

Furthermore, due to the probabilistic nature of the SA algorithm it further guarantees that the solution will return a solution that is as least as good as the optimal one. But it is not guaranteed as it escapes the local optima. But it cannot conclusively compute the global minima.

I was able to achieve a full solution working with random schedule generation with values even outside of the requirements of the project. The solution can also allow to try experimentation with other schedule optimization problems outside of SA to the modular nature of the code. Lastly, I was automated testing and an OOD/OOP solution that can allow for an extensible architecture as long as domain objects and functions utilize a similar interface that describes the constraints the problems (machines, jobs, etc…).

In terms of optimization using various combinations of configuration parameters it appears that the higher numbers of machines will have the most effect on the makespan (ie. minimized the most thereby reducing job scheduling execution time).