Dalton Dayhoff

Generalized Optimal Scheduling

A simulated annealing approach

Utah State University
MAE 5370
December 8, 2023

Abstract

The purpose of this exercise is to solve a generalized optimal scheduling problem. The problem formulation can be changed to fit a real problem, likely with many more constraints. Simulated annealing is a metaheuristic optimization algorithm based on the process of annealing a metal. This approach is good for problems in which neighboring solutions are both meaningful and plentiful. Optimal scheduling is perfect for this as one change to a schedule is a neighboring solution. Overall, this marriage worked well as a solution within 10% of the maximum value is found even though the maximum value is actually unobtainable.

1. Problem Description

One of the best uses for metaheuristic optimization algorithms is optimal scheduling as most are NP hard. The scope of these problems is almost limitless as they have been applied to problems such as satellite constellations, transportation schedules, job shops, workforce scheduling, etc. The most general parts of these problems are tasks, some sort of collection mechanism, and an objective function that uses certain aspects of the tasks. The objective function is covered in section 2.

1.1. **Tasks.**

Tasks are the meat of any optimal scheduling problem. There are often more tasks than is possible to collect given a time constraint. This is one of the major reasons the problem is NP hard. There is no way to know when tasks should be in the schedule, let alone whether or not a task should be included, without testing each and every solution. The purpose of metaheuristics is to explore this solution space in a smart way. Examples of tasks to complete are places on the Earth to image, bus stops, charging stations, work shifts, etc.

Tasks in the current problem are generic. They have three major components: position, duration, and value. The domain is a 100 by 100 grid of points. Each task is a point on this grid, thus creating a cost that is the distance between concurrent tasks. The duration corresponds to how long it takes the collector to complete a task. The value is simply how valuable the task is to the overall objective. For this problem all three components of the tasks are randomly generated.

1.2. **Collection.**

The method of collection is the most variable component of an optimal scheduling problem. These can be anything from a constellation of satellites to a set of employees to a space to place jobs. Many of these problems have a multitude of actors, which makes the solution space even more complicated. The difference between having one collector versus multiple is with multiple there are different optimal scheduling problems for each actor. This means the objective function will be a summation of the objective functions for each of these schedules. That is why, for this general optimal scheduling formulation, there is only one actor.

The lone actor in this general formulation moves around the grid to collects the tasks. The actor can move one block in any direction (including diagonally) at each step. This means the collector is always at a point on the grid and cannot take the most direct path to the tasks. This is taken into consideration in the cost function (more on this is section 2), which is part of the objective function. Ideally, this collector always moves to the next closest task after completing the previous task in the schedule. This means there may be some high value targets that are not collected because they are out of the collectors way. All of this is to say, the optimzation algorithm tends to search out "good" but more robust areas of the solution space rather than the most optimal solution.

2. Optimization Formulation

The objective function for this project is relatively simple. The overall value of any given task is

$$Score(X_i) = X_i.value - time(X_i, X_{i-1}) \tag{1}$$

where the time function calculates the distance between the tasks and multiplies this value by 1.25 due to the path the collector must travel. Using this, the overall objective function for the schedule is

$$f(x) = \sum_{0}^{n} X_i.value - time(X_i, X_{i-1})$$

where n is the number of tasks in the schedule. The only external constraint applied to the problem is

$$t_{total} <= 600s$$

to calculate the total time

$$t_{total} = \sum_{0}^{n} X_i.duration + time(X_i, X_{i-1})$$

This is just the summation of the travel time between the tasks and the duration of the task collections. Therefore, the optimization problem can be written as

$$\text{Max} \sum_{0}^{n} X_i.value - time(X_i, X_{i-1})$$

$$\text{s.t.} \sum_{0}^{n} X_i.duration + time(X_i, X_{i-1}) <= 600$$

3. Algorithm and Implementation

Simulated annealing is employed to address the problem outlined in sections 1 and 2. This is a meta-heuristic optimization algorithm based on the real life process of annealing a metal. Annealing is a process by which a metal is heated and slowly cooled in order remove internal inconsistencies. That is to make the metal more uniform and easier to work with. The ideas behind simulated annealing are similar by using randomness to add, remove, and move events around in the schedule.

Prior to delving into the algorithm itself, certain properties of simulated annealing need to be discussed. First of all, one of the main properties of metaheuristic optimization algorithms is they tend to find different solutions for every run due to their inherent randomness. This means for most scenarios, the algorithms do not find the optimal solution. However, for simulated annealing, there is a more positive side to this coin. Given infinite time, simulated annealing will always find the optimal solution given good design choices. This is because the randomness means anything can happen at any time; thus, given infinite time every possibility will happen. This is to say as long as the solutions space is completely connected, or reachable from the initial solution, then the optimal solution will be found. All of this is to say the simulated annealing often finds very good solutions but, due to limited resources, rarely finds the optimal solution.

Now to go over the implementation for this problem. The most simple way to do this is by going over each function individually. Starting with the overall planning function and continuing to each helper function as needed.

```
def plan(self, schedule: Schedule) -> Schedule:
    # Initializing possible solutions
    current_schedule = schedule
    current_score = current_schedule.calculate_score()
    previous_schedule = dc(schedule)
    previous_score = previous_schedule.calculate_score()
    self.scores.append(previous_score)
    self.times.append(previous_schedule.calculate_total_time())
    # Initilizing SA components
    current_temp = self.starting_temperature
    best_score = current_schedule.calculate_score()
    best_schedule = dc(current_schedule)
    iterations_since_best_score = 0
    while current_temp > self.final_temperature:
        for _ in range(self.max_iter):
```

```
16              (success, current_schedule) = self.run_pass(previous_schedule)
17              if not success:
18                  iterations_since_best_score += 1
19                  continue
20              current_score = current_schedule.calculate_score()
21              if current_score > best_score:
22                  best_score = current_score
23                  best_schedule = dc(current_schedule)
24                  iterations_since_best_score = 0
25              else:
26                  iterations_since_best_score += 1
27              if current_score > previous_score:
28                  previous_schedule = current_schedule
29                  previous_score = current_score
30              else:
31                  if np.random.rand() < generate_acceptance_probability(current_score,
    previous_score, current_temp):
32                      previous_schedule = current_schedule
33                      previous_score = current_score
34              if self.scores[-1] != previous_score:
35                  self.scores.append(previous_score)
36                  self.times.append(previous_schedule.calculate_total_time())
37              if iterations_since_best_score > self.diminishing_iter:
38                  return best_schedule
39          previous_schedule = previous_schedule.ensure_consistency()
40          current_schedule = current_schedule.ensure_consistency()
41          current_temp *= self.cooling_rate
42      best_schedule = best_schedule.ensure_consistency()
43      return best_schedule
```

One major part of simulated annealing the ability to sometimes get worse in order to hopefully find a better solution. This creates the need to compare the previous solution with a current solution. The first few lines of the function set up all the variables to compare while running the algorithm. Next is the initialization of multiple components for SA. The meat of the algorithm is the while loop that comes directly after the initialization steps on line 14. In simulated annealing, there are multiple stages that control the acceptance probability with the current temperature. The temperature is changed after the for loop, meaning each time the for loop runs is another stage. The run pass method is called, which is the next function to discuss, and returns a Boolean and the new schedule. If the schedule could not run a pass, move on to the next iteration of the stage. Next are a few comparisons to decide whether or not to accept the change. Next, if the change was accepted then update the metrics. Finally, if there have been a set number of iterations since the best solution so far was found, then return that best solution. Now to discuss the run pass funciton.

```
1 def run_pass(self, schedule: Schedule) -> Tuple[bool, Schedule]:
2     change_selector = np.random.rand()
3     success = False
4     iter = 0
5     while not success and iter < self.max_attempts_per_pass:
6         if change_selector <= self.remove_weight:
7             success = schedule.remove_random_task()
8         elif self.remove_weight < change_selector <= (self.remove_weight + self.
    move_weight):
9             success = schedule.move_random_task()
10        else:
11            success = schedule.add_random_task()
12        iter += 1
13    return success, schedule
```

First, numpy generates a random number between 0 and 1. This value is used to decide which type of change to make, along with predefined weights for moving or removing a task. There is a possibility of failure with each of the changes based on an increase in time incompatible with the constraints or trying to remove a backbone task. There is only one backbone task, the task that is closest to the start point because there is no point in removing the most optimal first step. Overall, this function tries to make a change and if not it just returns false. Most literature on simulated annealing refer to the schedule this function creates a neighboring solution. Next to discuss the function for generating acceptance weights.

```
def generate_acceptance_probability(first_score, second_score, current_temp):
    diff = abs(first_score - second_score)
    raw_rate = diff / current_temp
    expm_rate = np.exp(raw_rate)
    bounded_rate = min(1.0, expm_rate)
    return bounded_rate
```

The probability of accepting a worse solution is connected to two things, the difference between the scores and the current temperature. The reason for using the exponential is to make sure the rate of accepting worse solutions decreases as the temperate decreases. This is called crystallization. The rest of the code is listed in the appendix as they are not directly part of the simulated annealing algorithm but rather part of the specific implementation of the problem.

4. Results and Interpretation

With problems as complicated as optimal scheduling, there is often no way of knowing if a solution is truly optimal. This problem is no different, as is illustrated in the top portion of fig. 1. The dotted line at the top represents the total score of every available task. This value is significantly more than the actual optimal value as there is no way to know the shortest path through the schedule. This being said, the best solution found is relatively close to the top line if you take scale into account (only about 10% off).

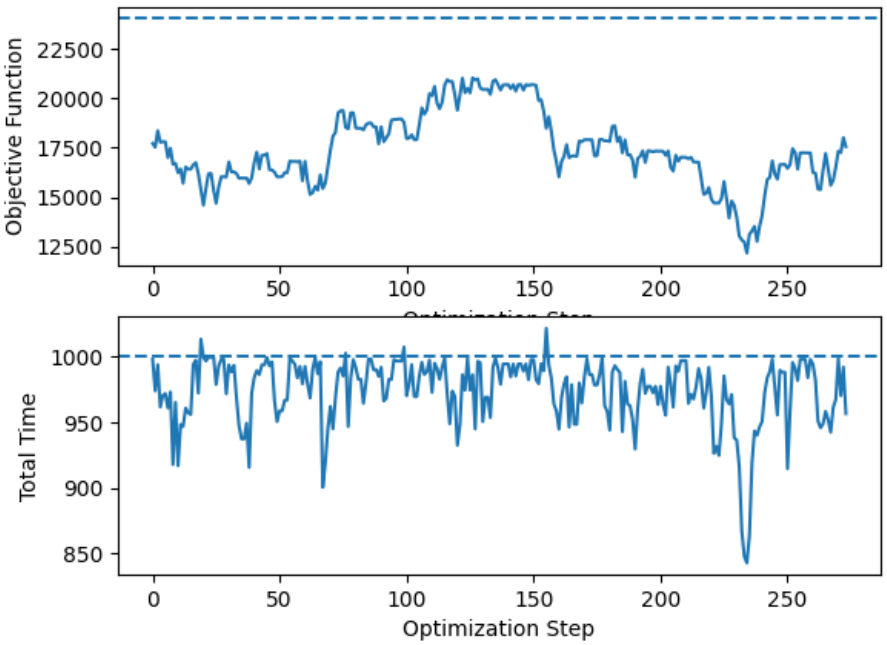

Figure 1. Graphs showing the progression of the schedule

The nature of simulated annealing is illustrated rather while by these graphs. The top shows the constant up and down motion of the objective function score due to the random search of the solution space. The

second graph shows how the overall time of the schedule stays near the maximum allowed time because this allows for more tasks and thus a higher score.

In section 1, the objective function is defined as the sum of the values of the tasks minus the distances between the tasks. The values of the tasks are often worth significantly more than the distance between them. This is why the algorithm does not always go for the closest event because it be worth it to go get a high value task instead. Figure 2 shows a partial progression of the schedule. The red marks are tasks that have been schedule yet not completed. Green is tasks that have been both scheduled and completed. Yellow is unscheduled tasks and blue is the collector. This progression shows that the schedule does tend

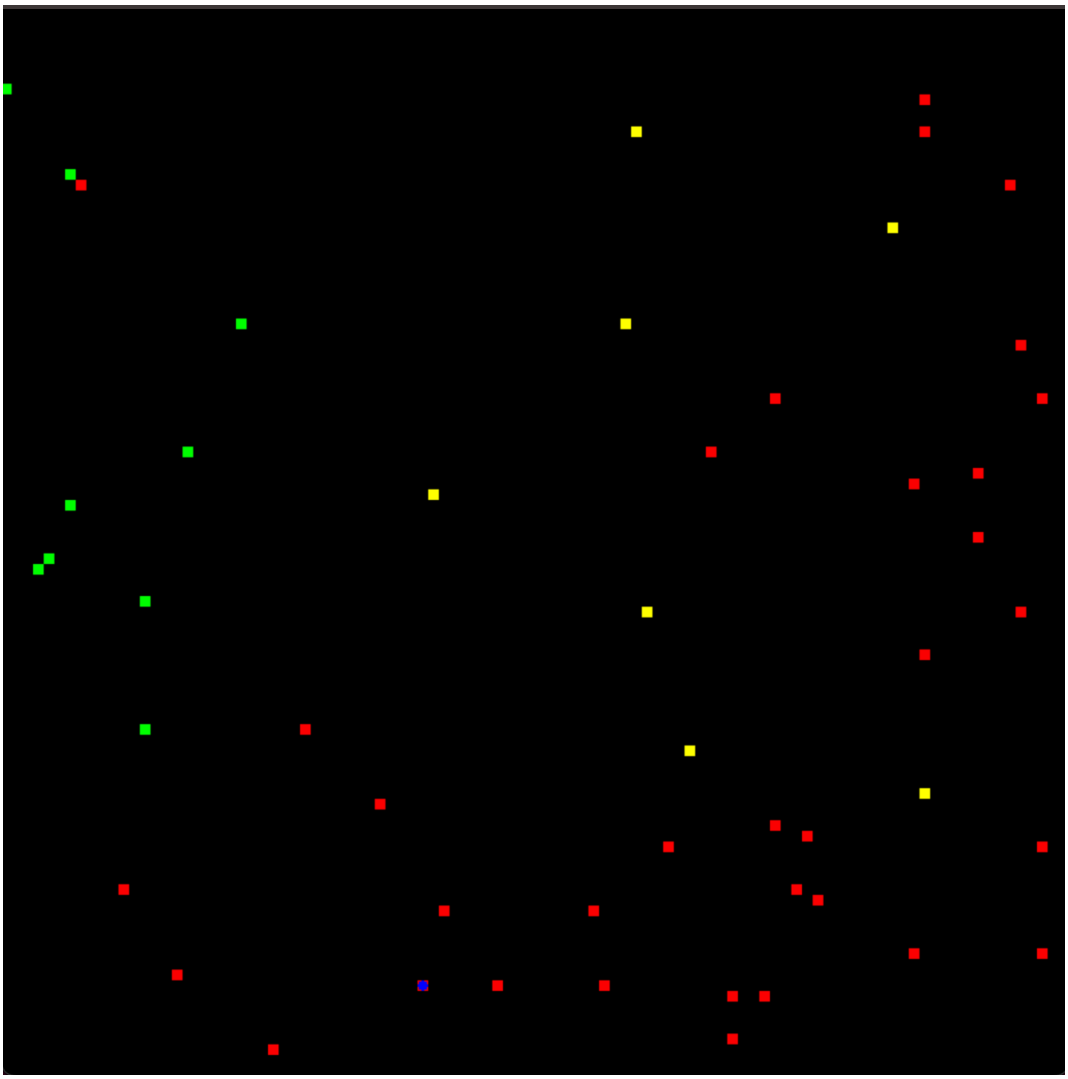

FIGURE 2. Visualization of the schedule

to go for relatively close tasks; however, it does sometimes deviate.

5. CONCLUSION

Overall, this solution works well for the problem. Simulated annealing is a very good optimization algorithm, especially when the number of constraints is lower. It is much faster than most non-metaheuristic algorithms and much simpler to implement than other metaheuristics. Often, one of the hardest parts of trying to find the optimal solution to a problem is finding an optimization algorithm that fits well. This means knowledge of which algorithms work with what types of problems is valuable. In conclusion, simulated annealing can generally solve optimal scheduling problems efficiently.

Appendix

Instead of copying all of my code here, I am just linking my GitHub repository: Generalized Optimal Scheduling