Scheduling Problem

By Sarah Sheikh

Introduction:

CPU Scheduling in Operating Systems is an important problem to achieve "good performance". In this problem, there is a list of tasks, each task will have different entry time, processing time and deadline. The task of the algorithm is to find an order of performing the tasks in order to minimize the number of tasks which doesn't meet the deadline (missed deadlines). There are many algorithms to solve this problem(Tabu search, ant algorithm, simulated annealing). In here, we use stochastic hill climbing and genetic algorithm.

➔ The program will have some inputs:
- 1st line: n (n is the number of task)
- Next n lines: <name_task> <entry_time> <processing_time> <deadline>

(When program read these inputs, it stores data in an array, each element respect to a task)

➔ The program will have outputs:
- Each line: <name_task> <start_time> <end_time>

(The order of tasks is as the order is written on the file, does the tasks from line 1 to the last line)

Stochastic Hill Climbing (SHC)

The Algorithm

The algorithms includes 5 steps:

Step 1: Initialize the starting solution
Step 2: Generate different neighbors, just generate a certain number of neighbors, not all possible neighbors(solutions)
Step 3: Choose the neighbors that have the number of missed tasks is less than or equal the current solution (best neighbors)
Step 4: Move to the best neighbor(solution)
Step 5: If there is no better solution after finite iterations, return the current best solution, otherwise move to step 2

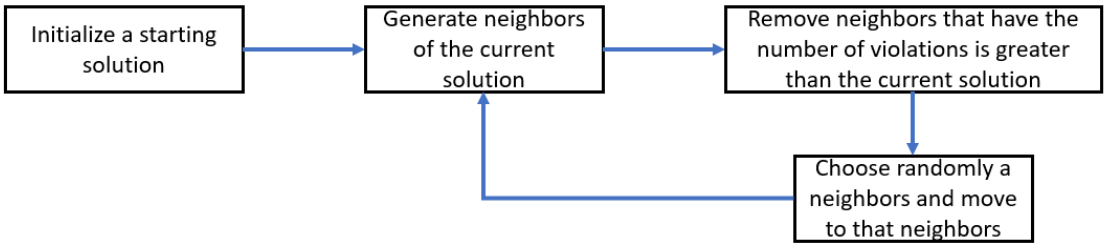

Figure 1 SHC Algorithm as a diagram

Key Points:

- Problem Statement:
  - If the program has a list of n task, it creates a list of n tasks. Each permutation of this array is a feasible solution. The number of violations of a solution is the number of missed tasks. When the violation reaches 0, it means that the algorithm gets the perfect solution (there is no missed task)
  - To calculate the violation of a solution, program does tasks from left to right (in the list of tasks), whenever it meets a task and does not have enough time to do this task, the number of missed tasks increases by 1.
- Initializing step (Initialize function):
  - In this step, program initializes a random solution as the starting solution
- Generate neighbors:
  - Program has function called neighbor. To generate a neighbor from the current solution, algorithm random 2 positions i and j (I < j), move element from position j^{th} to position i^{th}
  - Example:
    - solution [4, 1, 5, 2, 3]
    - random i = 1, j = 3 (index start from 0)
    - neighbor [4, 2, 1, 5, 3]
    - Since the number of possible neighbors is very large, so program generate only a fixed number of neighbors (e.g., 10, 20, ...). Via practice, I realize that increasing this parameter does not help too much in improving solution. It should be from 60 to 120 (n <= 50).
- Choosing the next solution:
  - Instead of the best solution (the solution that have smallest number of violations) program choose randomly from solutions that have the quality is greater than the quality of the current solution.
- Termination condition:
  - After a given number of iterations, the current solution that has the least number of violations does not change, the program terminates and return solution, otherwise continues the searching process. However, it may loop forever in plateau or ridge situation. In plateau case, when program try to look around its neighbors, and all these neighbors has the number of violations, thus program won't know what the right direction is to get better solution. (the below image is a simplified search space; the horizontal axis is the solution and the vertical axis is the number of violation)

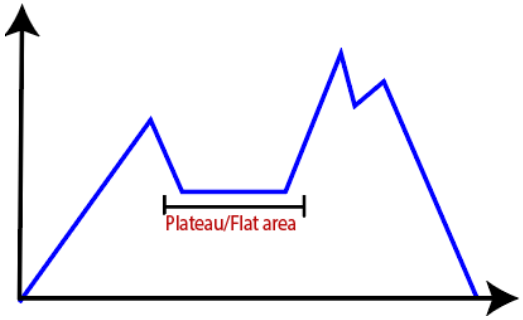

Figure 2 A plateau

- In the ridge case, since program does not consider all neighbors, it can lead to circumstances that program will miss the solution that has smaller number of violations and be stuck in there.

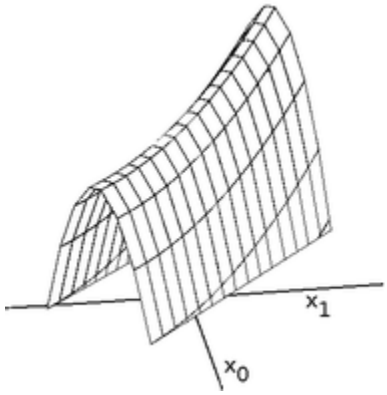

Figure 3 A Ridge

- Therefore, program is still set a limitation of the number of iterations to make sure that program will be terminated. Increasing this parameter will decrease a lot of violations.

The Scripts

The scripts of SHC includes 4 functions mainly:

- Violation: return the number of violations (the number of missed tasks)
- Neighbor: generate neighbors
- Initialize: initialize the starting solution
- sHill: It will call initialize function to create a random solution, next it will go to a loop, in each iteration, it will call neighbor function and then do the choosing the next solution step, assign the current solution to the selected neighbor. This loop will end when it meets the termination condition.

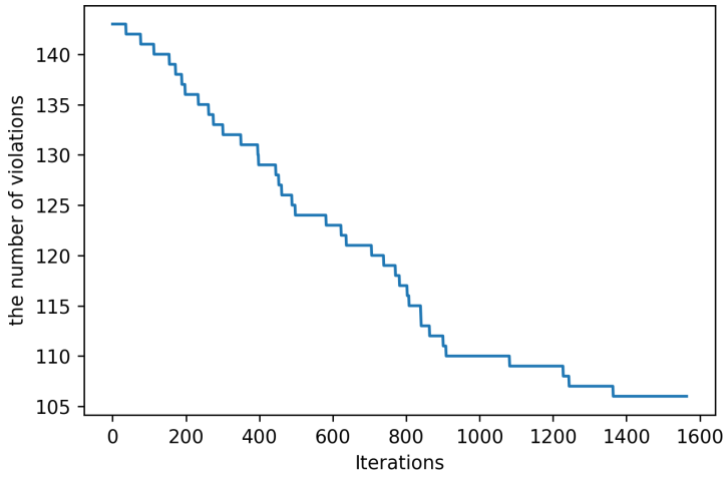

Figure 4 The number of violations after each iteration

The Time and Space Complexity

The time complexity of SHC is polynomial and its space complexity is linear. However, SHC cannot run forever even the number of iterations is set a very large number. Since in each step, SHC will chose the better solution (the solution that has the number of violations is less than the current solution) to move, thus if program does not see any better solutions, SHC will be stuck at local optimal (the solution that has number of violations is smaller than or equal any its neighbor) and therefore, it cannot be loop forever.

2. Genetic Algorithm(GA)

The Algorithm

The algorithm includes 5 steps:

Step 1: Initialize a population
Step 2: Select parents for mating
Step 3: Recombination of parents to get new children
Step 4: Mutate new children
Step 5: Select the best individuals for the next generations
Step 6: If the number of generations does not exceed the limitation, then turn back to step 2, otherwise, stop algorithm

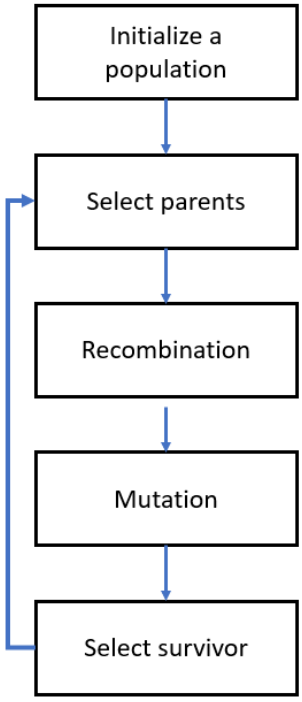

Figure 5 GA Algorithm as a diagram

➔ Problem Statement: If program has a list of n task, it creates a list of n tasks. Each permutation of this array is a feasible solution. The number of violations of a solution is the number of missed tasks. When the violation reaches 0, it means that the algorithm gets the perfect solution (there is no missed task). To calculate the violation of a solution, program does tasks from left to right (in the list of tasks), whenever it meets a task and does not have enough time to do this task, the number of missed tasks increases by 1.

➔ Initialize population:
Since each permutation of the list of tasks is a solution, thus at the beginning of program. It will initialize a list of random permutations by shuffling tasks in the list.
Ex: The size of population is 3, then a random population of 5 tasks is:
[1, 2, 3, 4, 5]
[3, 4, 5, 1, 2]
[2, 4, 3, 1, 5]

➔ Select parents:
In this step, program will calculate the probability of individuals based on violations.

$$P = \frac{\mu * 2 - \text{violations}}{total\ of\ violations}$$

Where μ is the mean of violations?

In here, we use roulette wheel algorithm to choose parents. The algorithm will help program choose the parents by their probability, thus the individuals that have large probability will be chosen many than other individuals. The more fit the parent will be the better chances of selection. The idea is

inspired from nature, the strong individual will occupy many opportunities in mating (The number of parents twice the size of populations):

Algorithm: ROULETTEWHEELSELECTION()

```
r := random number, where 0 ≤ r < 1;
sum := 0;
for each individual i
{   sum := sum + P(choice = i);
    if r < sum
    {   return i;
    }
}
```

Figure 6 The Roulette Wheel Selection

Example: the population has 3 solution a1, a2 and a3, the numbers of violations are 4, 4, 2, respectively.
Mean of the number of violations is 3.33((4+4+2)/3)
Thus, the probability of a1, a2, a3 are 0.266, 0.266, 0.468, respectively (These results have been rounded off). Thus, the parents list may be [a1, a3, a3], [a2, a3, a3], [a1, a2, a3], …
The distribution of parent list just needs to be approximate as the distribution of a1, a2, a3.

➔ Mating:

After choosing parents, each two consecutive elements in parents are recombined to create two new children. To create two new permutations, program create the first child is the first half of the first parent and elements in the second that do not appear in the first half of the first parent. The second child is the remaining elements of parent 1 and parent 2. In this step, the number of children is twice as the size of current population.
Example: Parent1 = [3, 1, 2, 4, 5, 6]
Parent 2 = [1, 2, 3, 6, 4, 5]
Step 1: Child1 = [3,1, 2], Child2 = []
Step 2: Child1 = [3, 1, 2, 6, 4, 5], Child2 = [1, 2, 3]
Step 3: Child1 = [3, 1, 2, 6, 4, 5], Child2 = [1, 2, 3, 4, 5, 6]

➔ Mutation

A small percentage (default is 0.2) of new children will be mutated, the mutating operator is the same as the generating neighbor in stochastic hill climbing.
To mutate a neighbor from the current solution, algorithm random 2 positions i and j (I < j), move element from position j^{th} to position i^{th}

- Example:
  solution [4, 1, 5, 2, 3]
  random i = 1, j = 3 (index start from 0)
  neighbor [4, 2, 1, 5, 3]

➔ Selection

In this step, the current population and the new children are combined and form new population. However, the size of population is fixed, thus elements that has smallest violations will be kept for the next generation
Example:
The population includes a1, a2, a3

The new children are a4, a5, a6
Fitness: a1: 7, a2:6, a3: 2, a4:6, a5:3, a6:10
Thus, to keep the size of the population, program will choose 3 best individuals,
The new population is (a3, a5, a4) or (a3, a5, a2)

The Scripts

The scripts of SHC includes 8 functions mainly:

- Initialize: Initialize a random population
- Fitness: return the number of violations
- getBest(data, population): traverse the population and return the individual that has the least number of violations
- getParents: do selecting parents
- mate: Mating step
- Mutation: mutation step
- Selection: selection step
- GA: at first, it calls initialize function to create a random population (random means its individual are created randomly, without any oriented direction), next it goes to a loop (each iteration respect to a generation). In each generation, it calls getParents function to choose parents for mating step, next, it puts these parents to mate function to get new children, and then these children will be mutated in mutation function.

  This process will be repeated until the number of generations is over the given limit.

By practice, we realize that increasing the size of population is more effective than increasing the number of generations. Because the size of population is very large, increasing the number of generations will increase a lot of computation (Since program will have to do all the below process in the new generation), but mating and mutation are two random process, there is nothing to guarantee that a better solution (the solution that has the number of violations is less that the current best solution) will appear in the new generation. In contrast, increasing the size of population will increase search area, it means that program will have high chance to discover new solutions, escape from the local optima and thus it can get closer to the optimal solution (the solution that has the least number of violations in all possible solutions)
The time and space complexity of GA are polynomial. The algorithms can run and improve the solution if it does not exceed CPU time limit or meet the optimal solution.

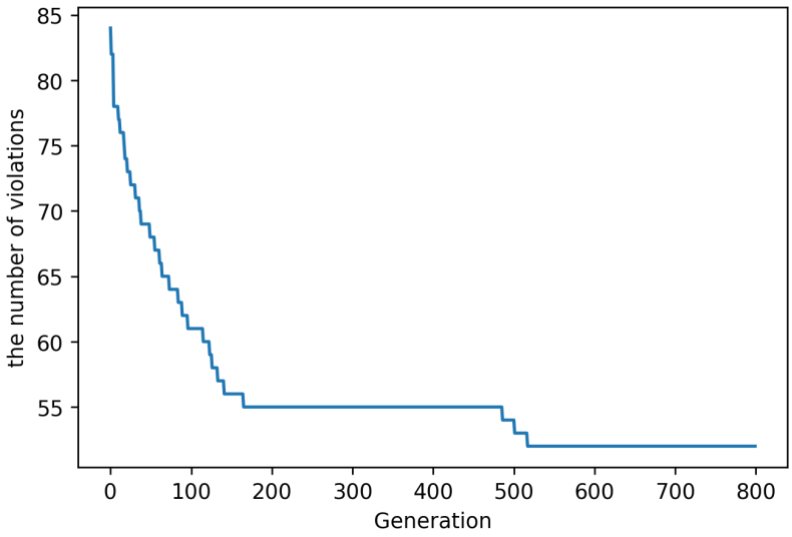

Figure 7 The number of violations after each generation

Input and Output

Input.txt file include the first row to be the number of rows and then the following each row has the <name_task> <entry_time> <processing_time> <deadline>

File results_GA.txt and results_SHC.txt are output for Genetic Algorithm and Stochastic Hill Climbing respectively! Each line in these output files give the <name_of_task><start_time><end_time> in turn showing the order in which the tasks should be performed.

The number of tasks which doesn't meet the deadline is displayed as output in the Jupyter Notebook!

```
the number of violations:  154
[55, 8, 11, 1, 26, 18, 21, 13, 2, 9, 0, 27, 5,
```

Comparison Between GA and SHC

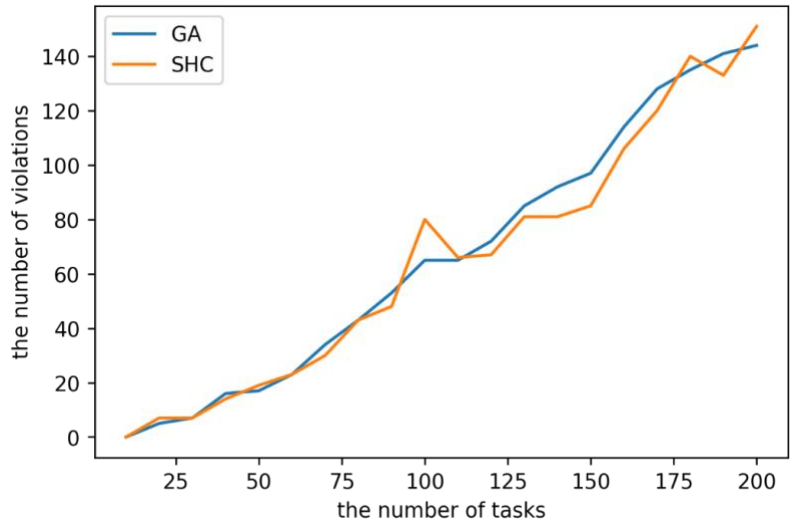

Figure 8 The number of violations of GA and SHC algorithms when the number of tasks increase from 10 to 200

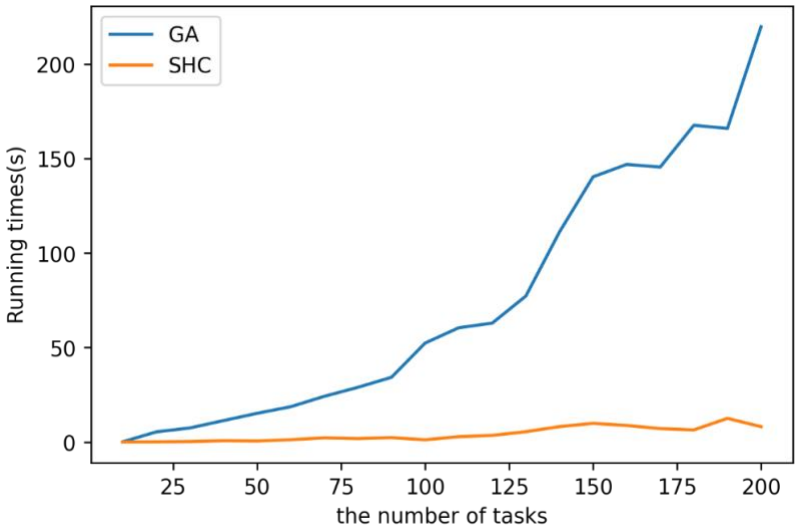

Figure 9 The running time of GA and SHC algorithms

Hence, in this problem, SHC is more efficient than GA algorithm. Their violations are quite similar, but SHC takes much less time than GA algorithm.
Both GA and SHC are non-optimal algorithms, and they just return good enough solution. However, from practice, the result of GA is relatively stable and more reliable than SHC algorithms.