2021

ECE 457A: Assignment 2

BOSCO HAN, DAIVIK GOEL, LICHEN MA

Problem 1

Problem formulation:

State space: Encoded information of x_1 and x_2 and the corresponding Easom function value (x_1, $x_2 \in$ [-100, 100])

Initial state: Any pair of x_1 and x_2 values inside the range [-100, 100]

Goal state: Global minimum of -1 at $x = (\pi, \pi)^T$

Goal test: Position of x_1 and x_2 match the goal state

Sets of actions: Increase or decrease the value of x_1 or x_2 by a set constant

Concept of cost: Given that this is a SA problem where we are trying to minimize the value of the Easom function, cost can be represented by the value of the Easom function

Note: The goal state and goal test are if we know the expected value for the Easom function, given that we are performing Simulated Annealing algorithm for this problem it is more likely that our goal state is “the global minimum” and our goal test is letting the SA algorithm find a solution using its probability of acceptance

Neighborhood function:

A simple neighborhood function for this problem would simply be to increment or decrement x_1 or x_2 by a fixed constant (0.1 for example)

Cost function:

Given that this is a SA problem where we are trying to minimize the value of the Easom function, cost can be represented by the value of the Easom function:

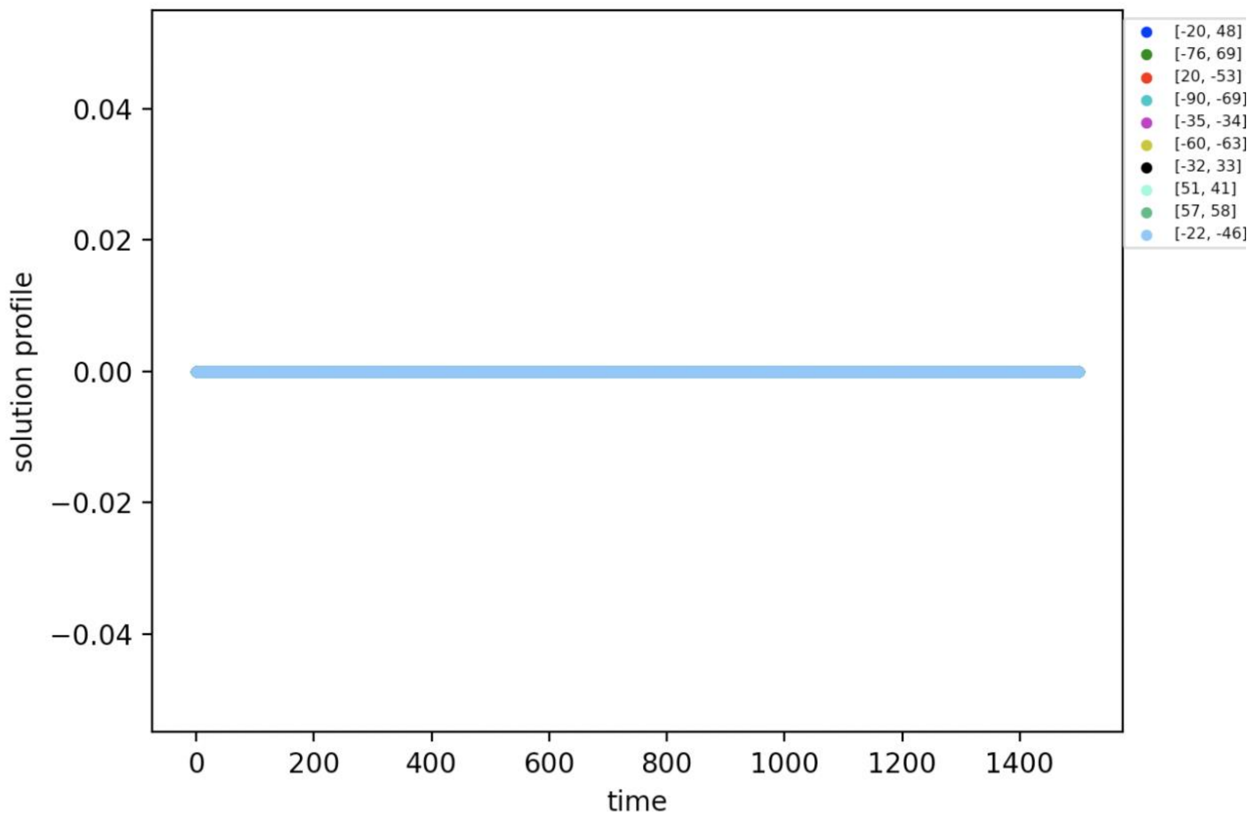
$$C = -\cos x_1 \cos x_2 \exp(-(x_1 - \pi)^2 - (x_2 - \pi)^2)$$

The solution description for this problem is simply to implement and execute a Simulated Annealing algorithm that modifies x_1 and x_2 and stops when a low enough temperature is reached or when the system is stuck (no better moves are found and no worse moves are accepted). The basic algorithm/pseudo code is as follows:

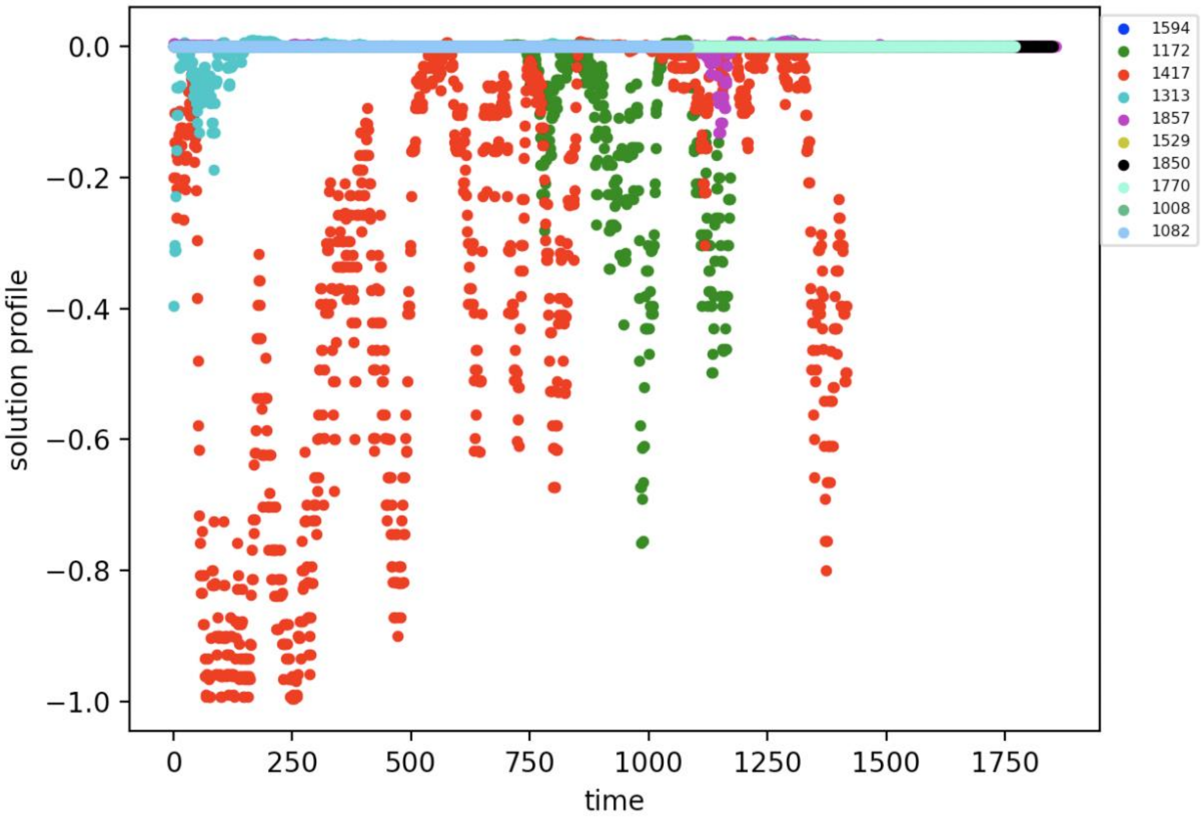
```
curr_solution = initial_point
curr_temperature = initial_temperature
curr_cost = easomFunction(curr_solution)
while curr_temperature > stopping_temperature:
    next_solution = generateNeighboringSolution(curr_solution)
    next_cost = easomFunction(next_solution)
    if next_cost - curr_cost < 0:
        # always accept improving move
        curr_solution = next_solution
        curr_cost = next_cost
        curr_iter_at_temp = max_iterations_at_temp
    else:
        acceptance_probability = math.exp((curr_cost-next_cost)/curr_temperature)
        if random.random() < acceptance_probability:
        # accept worse solution in this case otherwise reject the new solution
        curr_solution = next_solution
        curr_cost = next_cost
        curr_iter_at_temp = max_iterations_at_temp
         else:
        # don't accept worse solution
    curr_temperature = annealingSchedule(curr_temperature, annealing_rate)
```

The solution profile as a function of time for each of the following scenarios is provided below

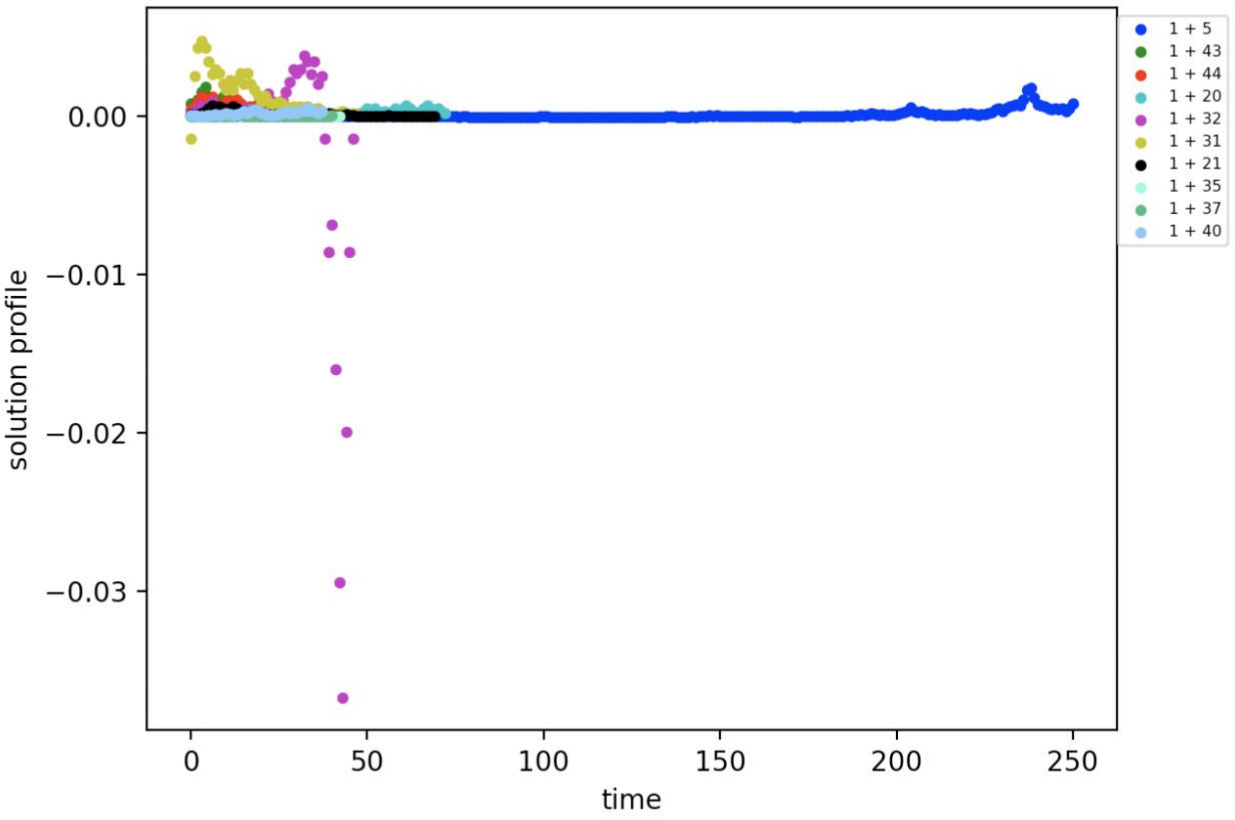
Experiment 1:



Experiment 2



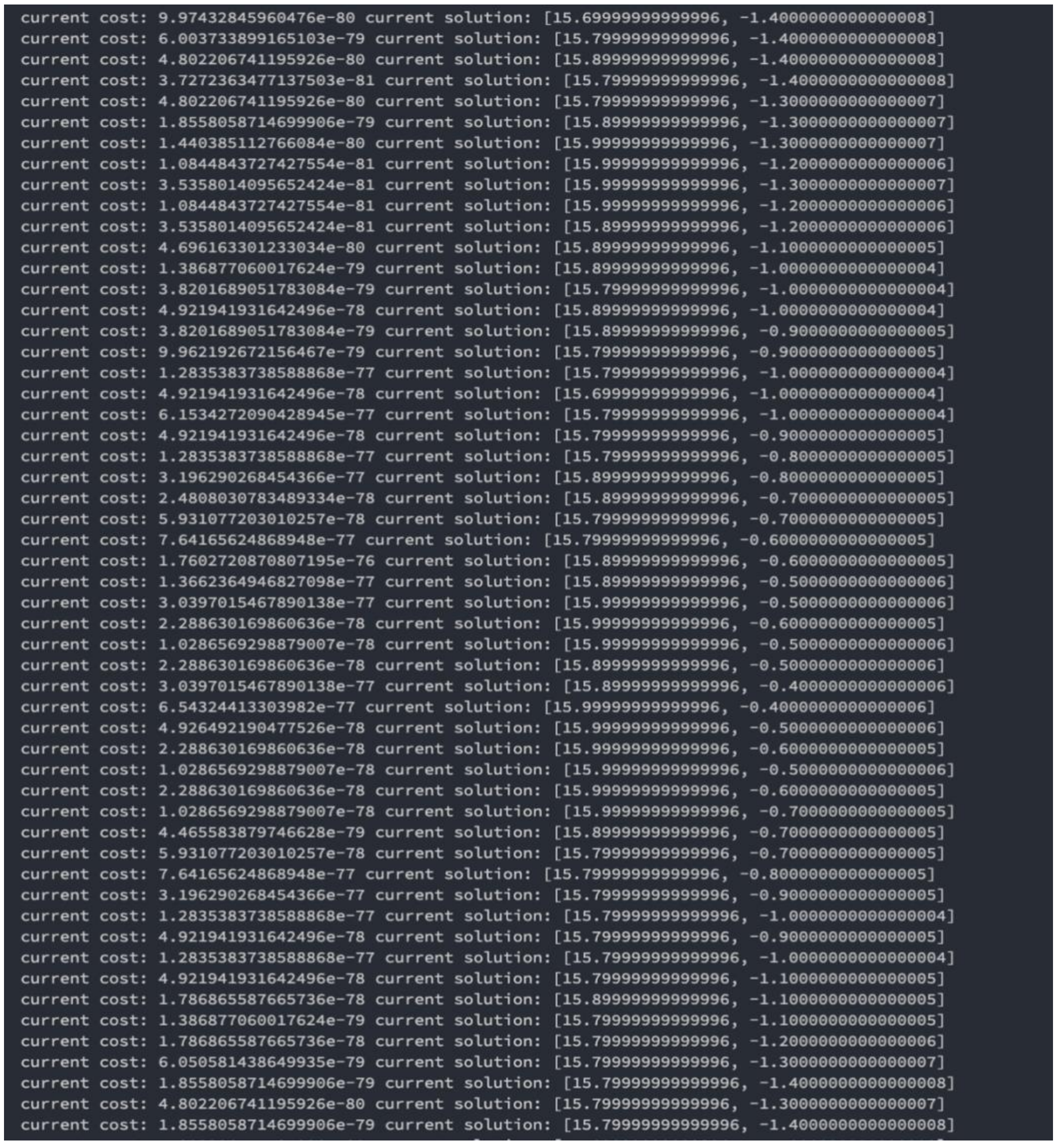
Experiment 3:



We can see from the solution profile vs time plots for the three experiments that selecting different initial temperatures and annealing schedules produced better solutions than selecting 10 different initial points randomly. This is due to the fact that for experiment 2 and 3, a fixed initial point of [0,0] was selected which happens to be close to the global minimum of [π, π]. During the execution of the simulated annealing algorithm, the program was able to reach a point close to the global minimum which allowed the algorithm to continue to find better and better solutions. The easom function which is used to determine cost for the SA algorithm, is difficult to work with since points far away from the global minimum all have values close to zero. These regions where every direction you go is flat and little information is given about whether you are moving in the right direction or not are extremely difficult for the SA algorithm to navigate. Hence, the first experiment which produced starting points such as [-20,48] found it difficult to get anywhere close to the optimal solution.

The best solution found after conducting the three experiments was [3.1, 3.1]. The setting of SA that achieved this solution was initial temperature: 1417, annealing function = current temperature - 1, start point: [0,0]. This setting was better than the other settings since we started at a location which is reasonably close to the global minimum. The high temperature and slow annealing rate also contributed to providing more time for the simulated annealing algorithm to explore the surroundings and find the global minimum.

Sample output:



```
current cost: 9.97432845960476e-80 current solution: [15.69999999999996, -1.4000000000000008]
current cost: 6.003733899165103e-79 current solution: [15.79999999999996, -1.4000000000000008]
current cost: 4.802206741195926e-80 current solution: [15.89999999999996, -1.4000000000000008]
current cost: 3.7272363477137503e-81 current solution: [15.79999999999996, -1.4000000000000008]
current cost: 4.802206741195926e-80 current solution: [15.79999999999996, -1.3000000000000007]
current cost: 1.8558058714699906e-79 current solution: [15.89999999999996, -1.3000000000000007]
current cost: 1.440385112766084e-80 current solution: [15.99999999999996, -1.3000000000000007]
current cost: 1.0844843727427554e-81 current solution: [15.99999999999996, -1.2000000000000006]
current cost: 3.5358014095652424e-81 current solution: [15.99999999999996, -1.3000000000000007]
current cost: 1.0844843727427554e-81 current solution: [15.99999999999996, -1.2000000000000006]
current cost: 3.5358014095652424e-81 current solution: [15.89999999999996, -1.2000000000000006]
current cost: 4.696163301233034e-80 current solution: [15.89999999999996, -1.1000000000000005]
current cost: 1.386877060017624e-79 current solution: [15.89999999999996, -1.0000000000000004]
current cost: 3.8201689051783084e-79 current solution: [15.79999999999996, -1.0000000000000004]
current cost: 4.921941931642496e-78 current solution: [15.89999999999996, -1.0000000000000004]
current cost: 3.8201689051783084e-79 current solution: [15.89999999999996, -0.9000000000000005]
current cost: 9.962192672156467e-79 current solution: [15.79999999999996, -0.9000000000000005]
current cost: 1.2835383738588868e-77 current solution: [15.79999999999996, -1.0000000000000004]
current cost: 4.921941931642496e-78 current solution: [15.69999999999996, -1.0000000000000004]
current cost: 6.1534272090428945e-77 current solution: [15.79999999999996, -1.0000000000000004]
current cost: 4.921941931642496e-78 current solution: [15.79999999999996, -0.9000000000000005]
current cost: 1.2835383738588868e-77 current solution: [15.79999999999996, -0.8000000000000005]
current cost: 3.196290268454366e-77 current solution: [15.89999999999996, -0.8000000000000005]
current cost: 2.4808030783489334e-78 current solution: [15.89999999999996, -0.7000000000000005]
current cost: 5.931077203010257e-78 current solution: [15.79999999999996, -0.7000000000000005]
current cost: 7.64165624868948e-77 current solution: [15.79999999999996, -0.6000000000000005]
current cost: 1.7602720870807195e-76 current solution: [15.89999999999996, -0.6000000000000005]
current cost: 1.3662364946827098e-77 current solution: [15.89999999999996, -0.5000000000000006]
current cost: 3.0397015467890138e-77 current solution: [15.99999999999996, -0.5000000000000006]
current cost: 2.288630169860636e-78 current solution: [15.99999999999996, -0.6000000000000005]
current cost: 1.0286569298879007e-78 current solution: [15.99999999999996, -0.5000000000000006]
current cost: 2.288630169860636e-78 current solution: [15.89999999999996, -0.5000000000000006]
current cost: 3.0397015467890138e-77 current solution: [15.89999999999996, -0.4000000000000006]
current cost: 6.54324413303982e-77 current solution: [15.99999999999996, -0.4000000000000006]
current cost: 4.926492190477526e-78 current solution: [15.99999999999996, -0.5000000000000006]
current cost: 2.288630169860636e-78 current solution: [15.99999999999996, -0.6000000000000005]
current cost: 1.0286569298879007e-78 current solution: [15.99999999999996, -0.5000000000000006]
current cost: 2.288630169860636e-78 current solution: [15.99999999999996, -0.6000000000000005]
current cost: 1.0286569298879007e-78 current solution: [15.99999999999996, -0.7000000000000005]
current cost: 4.465583879746628e-79 current solution: [15.89999999999996, -0.7000000000000005]
current cost: 5.931077203010257e-78 current solution: [15.79999999999996, -0.7000000000000005]
current cost: 7.64165624868948e-77 current solution: [15.79999999999996, -0.8000000000000005]
current cost: 3.196290268454366e-77 current solution: [15.79999999999996, -0.9000000000000005]
current cost: 1.2835383738588868e-77 current solution: [15.79999999999996, -1.0000000000000004]
current cost: 4.921941931642496e-78 current solution: [15.79999999999996, -0.9000000000000005]
current cost: 1.2835383738588868e-77 current solution: [15.79999999999996, -1.0000000000000004]
current cost: 4.921941931642496e-78 current solution: [15.79999999999996, -1.1000000000000005]
current cost: 1.786865587665736e-78 current solution: [15.89999999999996, -1.1000000000000005]
current cost: 1.386877060017624e-79 current solution: [15.79999999999996, -1.1000000000000005]
current cost: 1.786865587665736e-78 current solution: [15.79999999999996, -1.2000000000000006]
current cost: 6.050581438649935e-79 current solution: [15.79999999999996, -1.3000000000000007]
current cost: 1.8558058714699906e-79 current solution: [15.79999999999996, -1.4000000000000008]
current cost: 4.802206741195926e-80 current solution: [15.79999999999996, -1.3000000000000007]
current cost: 1.8558058714699906e-79 current solution: [15.79999999999996, -1.4000000000000008]
```

Time complexity and memory complexity:

Memory complexity of the implemented program is constant since we do not store any information except the cost of the previous solution (this is ignoring the graphing component, to produce graphs we store arrays containing the cost value for a particular execution and the timestamp which is O(n) but assuming this is not part of the base functionality, constant space is all that is required)

Time complexity of the implemented algorithm is O(T) where T is the size of the initial temperature provided. In the worst case the algorithm does not get stuck in a spot where no improving moves are found or worsening moves accepted and instead continues until a low enough temperature is reached. The annealing schedule in this implementation decreases the temperature by a constant value and a minimum temperature is set at one. Thus the execution time for the program in the worst case depends on the size of the input starting temperature

See code in zipped folder attached to assignment

Problem 2

Problem representation:

State space: Encoded information of the board and the location of each player’s pieces

Initial state: 10 player one pieces at (1,4) and 10 player two pieces at (4,1)

Goal state: Any configuration where the Minimax algorithm driven player wins (other player has no possible moves)

Goal test: Other player has no possible moves for any of their pieces

Sets of actions: Move any of the current player’s pieces up, down, left, right
or diagonally as long as the square being moved to is not
occupied by the other player

Concept of cost: Our evaluation function is based on how many moves the
current player can make from a given position vs how many moves the other player can make

Neighborhood function:

A simple neighborhood function for this problem would be all the possible moves
to adjacent squares from any of the player’s pieces (left, right, up, down, and
diagonally) such that the adjacent square is not occupied by an opponent piece

Cost function:

Cost is given by how many moves the current player can make from a given
position minus how many moves the opposing player can make from the same
position.

The basic algorithm and pseudo code for the minimax algorithm is as follows:

```
public static int minimax(Square[][] board, int depth, boolean isMaximizing, int alpha, int beta) {
        if (atDepth){
                return evaluationFunction(player, board);
        }

if (isMaximizing){
                int best = Integer.MIN_VALUE;
                for (int[] node : getOptimizedAgentSquares(board)){
                        for (int[] move : getAllowedMoves(node, player, board)){
                                Square[][] newBoard = makeMove(move, node, player, board);
                                best = minimax(newBoard, depth+1, false, alpha, beta, map);
                                alpha = Math.max(alpha, best);
                                if (beta <= alpha){
                                        break;
                                }
                        }
                }
                return best;
        } else {
                int best = Integer.MAX_VALUE;
                for (int[] node : getRandomAgentSquares(board)){
                        for (int[] move : getAllowedMoves(node, player, board)){
                                Square[][] newBoard = makeMove(move, node, player, board);
                                best = minimax(newBoard, depth+1, true, alpha, beta, map);
                                beta = Math.min(beta, best);
                                if (beta <= alpha){
                                        break;
                                }
                        }
                }
                return best;
        }
}

// make best move looks at the values from minimax traversal and decides based on best outcome
```

A handworked example explaining the solution strategy is as follows:

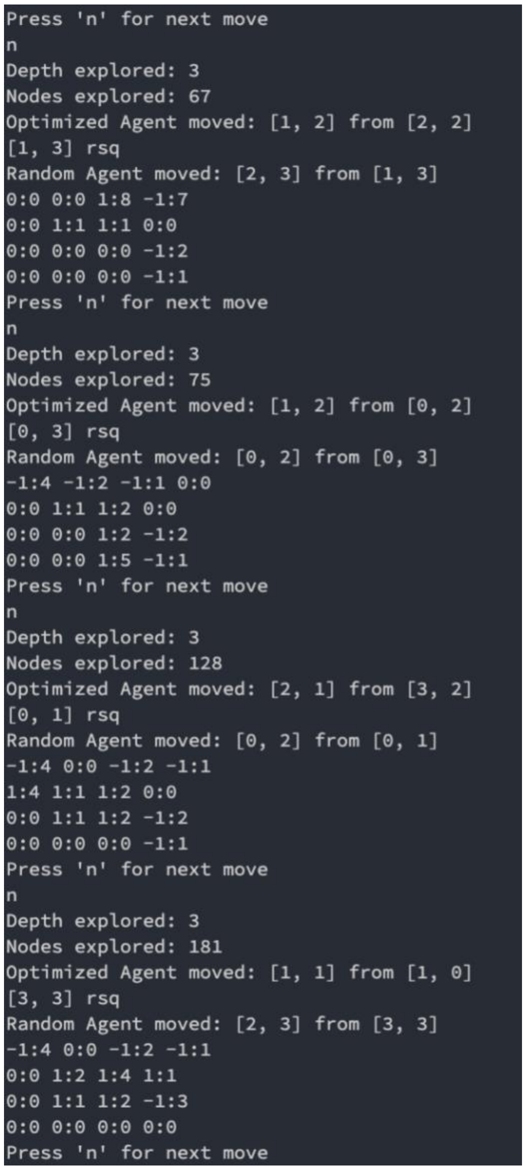
This example outlines a set of actions that can be made by black from a given state. In this case black is maximizing and wants to make the decision which gives it the highest score for the evaluation function. Our evaluation function is:

number of moves possible by current player - number of moves possible by other player

In our example the left most choices is the best since from that state black has 7 possible moves and white only has 2 possible moves giving us a evaluation function score of 5 which is higher than the other options. Depending on the depth we have for our minimax function both black and white players can use this information to decide what the best move would be.

The logic behind random agent is that it will check what available squares there are and randomly pick one. If there are available moves, it will pick one randomly. Otherwise it will continue to pick a new available square until it makes a move. This agent does poorly and loses five out of five games played against an optimized minimax agent set to a depth of two.

Sample output of program:



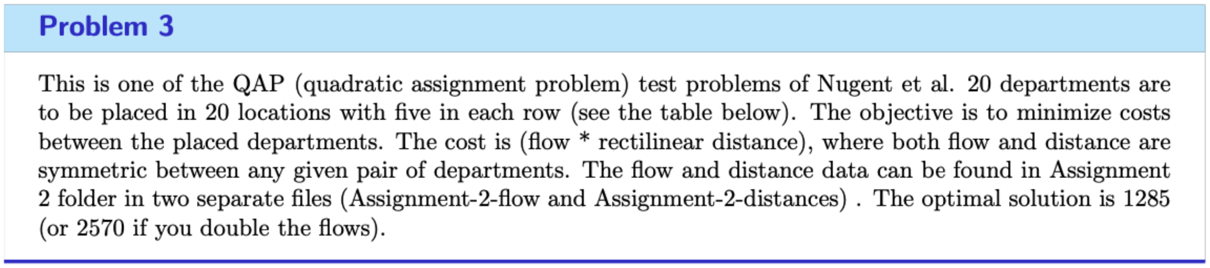
```
Press 'n' for next move
n
Depth explored: 3
Nodes explored: 67
Optimized Agent moved: [1, 2] from [2, 2]
[1, 3] rsq
Random Agent moved: [2, 3] from [1, 3]
0:0 0:0 1:8 -1:7
0:0 1:1 1:1 0:0
0:0 0:0 0:0 -1:2
0:0 0:0 0:0 -1:1
Press 'n' for next move
n
Depth explored: 3
Nodes explored: 75
Optimized Agent moved: [1, 2] from [0, 2]
[0, 3] rsq
Random Agent moved: [0, 2] from [0, 3]
-1:4 -1:2 -1:1 0:0
0:0 1:1 1:2 0:0
0:0 0:0 1:2 -1:2
0:0 0:0 1:5 -1:1
Press 'n' for next move
n
Depth explored: 3
Nodes explored: 128
Optimized Agent moved: [2, 1] from [3, 2]
[0, 1] rsq
Random Agent moved: [0, 2] from [0, 1]
-1:4 0:0 -1:2 -1:1
1:4 1:1 1:2 0:0
0:0 1:1 1:2 -1:2
0:0 0:0 0:0 -1:1
Press 'n' for next move
n
Depth explored: 3
Nodes explored: 181
Optimized Agent moved: [1, 1] from [1, 0]
[3, 3] rsq
Random Agent moved: [2, 3] from [3, 3]
-1:4 0:0 -1:2 -1:1
0:0 1:2 1:4 1:1
0:0 1:1 1:2 -1:3
0:0 0:0 0:0 0:0
Press 'n' for next move
```

General minimax/alpha-beta agent was developed for this assignment

On time and memory complexity:

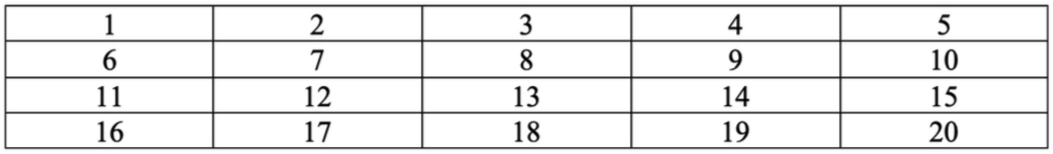
Time Complexity: Worst case for time complexity we need to explore all 8 possible directions for each node and there are no branches that we can prune so we would end up with O(m * n * 8^d) where m is width, n is height, d is the branching depth. For each square, of which there are m*n, there are 8 possible moves representing up, down, left, right, diagonal. After each move we may check future moves in accordance with the minimax algorithm thus we need to raise it to the power of the depth.

Memory complexity: For this group's implementation of the minimax algorithm, every time we recurse and move down the tree, a clone of the board is made to track the state associated with making the move. In the worst case we recurse down d times and we allocate new memory for a board clone at each step. This gives us a memory complexity of $O(d * m * n)$ where m is width, n is height, d is the branching depth.



Problem 3

This is one of the QAP (quadratic assignment problem) test problems of Nugent et al. 20 departments are to be placed in 20 locations with five in each row (see the table below). The objective is to minimize costs between the placed departments. The cost is (flow * rectilinear distance), where both flow and distance are symmetric between any given pair of departments. The flow and distance data can be found in Assignment 2 folder in two separate files (Assignment-2-flow and Assignment-2-distances) . The optimal solution is 1285 (or 2570 if you double the flows).



1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

Figure 6: Layout of department locations

The code for this problem can be found in the attached zip folder.

Optimization Problem Formulation

QAP is an optimization problem that will generate location of 20 departments with the objective of minimizing the cost dictated by (flow * rectilinear distance). The departments have several configurations for which they can be arranged in the 20 spots. In this problem, we are optimizing the total cost between the departments trying to find a configuration with the minimum cost

Simple Tabu Search

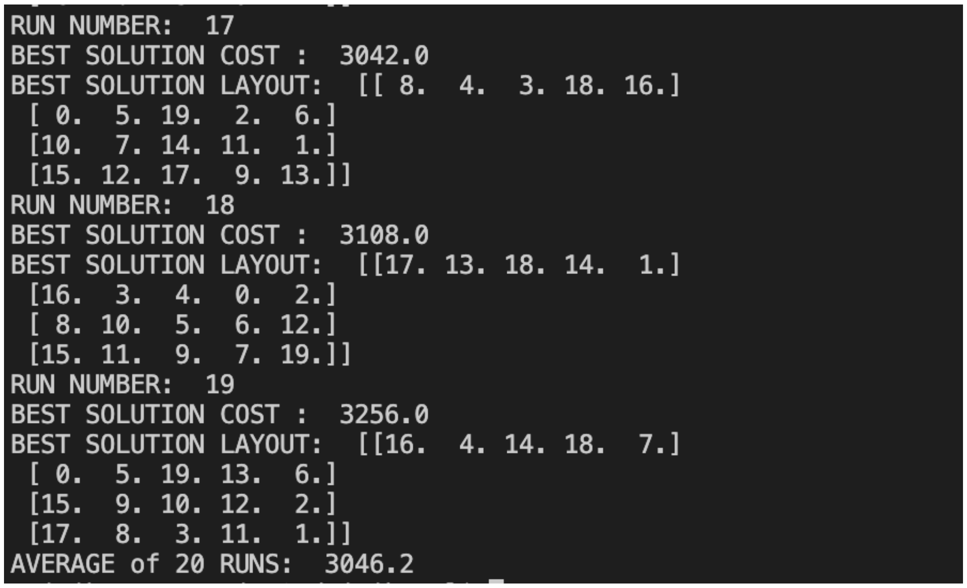
We developed a simple Tabu Search that creates a tabu list of size 100 and uses that to optimize. We first use created a random 4 X 5 matrix using the numpy and random libraries in Python. From there we take that initial solution and calculate the initial cost. We then proceed to the swapping function. This function will swap neighbours and calculate the cost of swapping the neighbours. If we find something that is lower than the current best cost, we assign it as the best layout. This will go on till we find the lowest cost of that neighbourhood. We then add that to the tabu list should it not be there before. If it is we cannot consider it as a solution (as we have already)

We keep repeating this until we hit the stopping criterion by iterations or optimal cost. From there we take the best solution and that is our final answer with the layout.

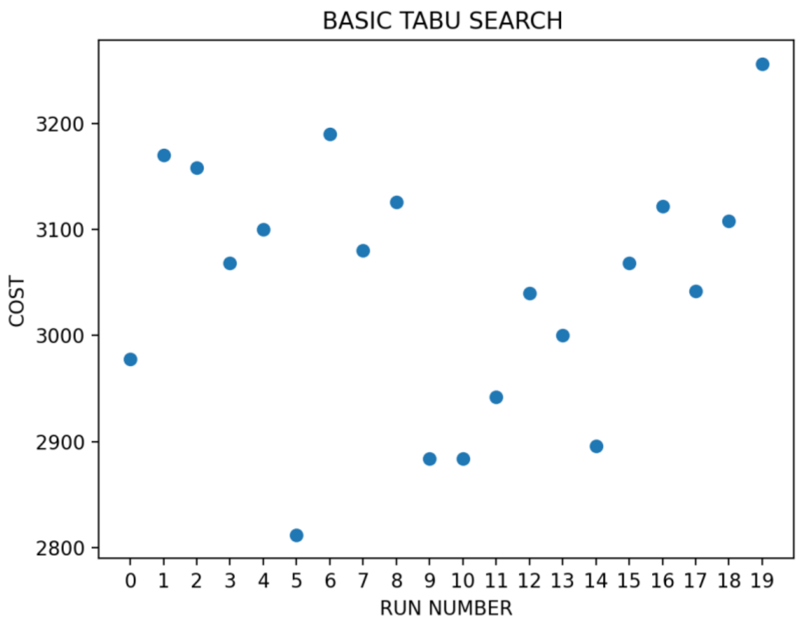
From a time, complexity point of view this search should be O(N^2 * (itr or opt) where N is the size of the matrix, itr is the iterations specified in the stopping criterion and opt is reaching the stopping condition. From a memory complexity we have a list of matrixes we have in a list. In that case the space complexity would O(N * M) where M is the size of the tabu list.

Experiment 1:

With experiment one we ran the simple tabu search 20 times and got different costs for each run. This likely is affected by the different initial conditions giving different costs. We got have an average cost of the 20 runs below.

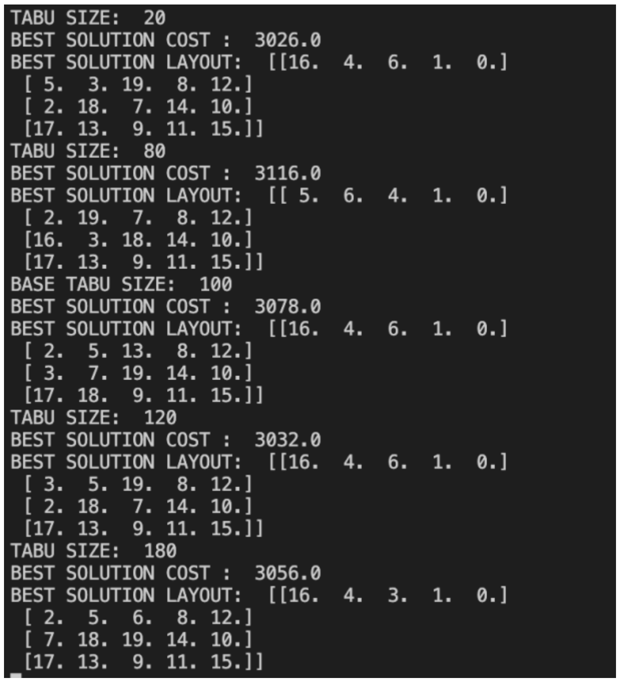


```
RUN NUMBER:  17
BEST SOLUTION COST :  3042.0
BEST SOLUTION LAYOUT:  [[ 8.  4.  3. 18. 16.]
 [ 0.  5. 19.  2.  6.]
 [10.  7. 14. 11.  1.]
 [15. 12. 17.  9. 13.]]
RUN NUMBER:  18
BEST SOLUTION COST :  3108.0
BEST SOLUTION LAYOUT:  [[17. 13. 18. 14.  1.]
 [16.  3.  4.  0.  2.]
 [ 8. 10.  5.  6. 12.]
 [15. 11.  9.  7. 19.]]
RUN NUMBER:  19
BEST SOLUTION COST :  3256.0
BEST SOLUTION LAYOUT:  [[16.  4. 14. 18.  7.]
 [ 0.  5. 19. 13.  6.]
 [15.  9. 10. 12.  2.]
 [17.  8.  3. 11.  1.]]
AVERAGE of 20 RUNS:  3046.2
```

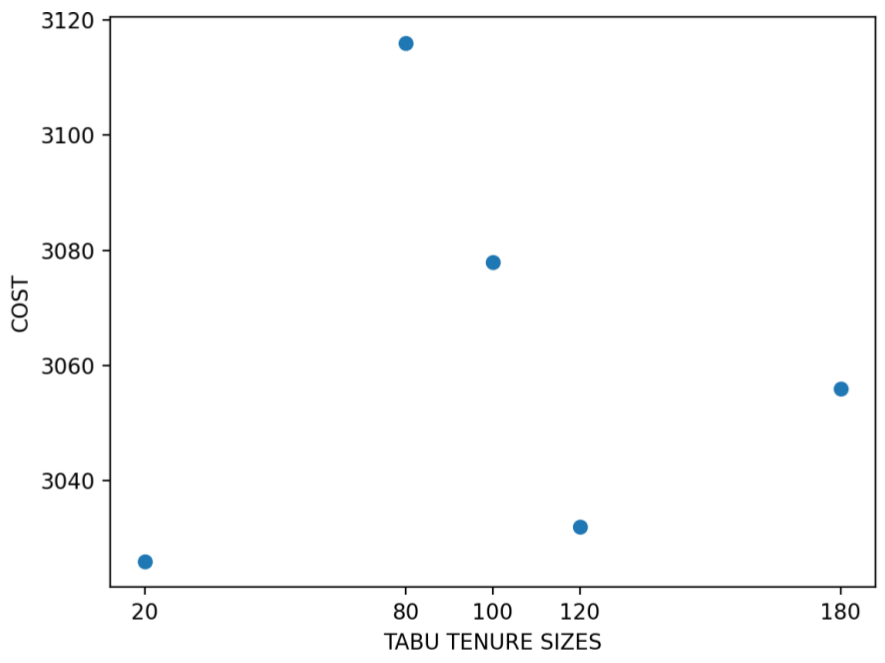


Experiment 2:

In this experiment we changed Tabu size to be magnitudes higher or lower than our initial value of 100. We have the results below. Our lowest cost is of Tabu Size 2. Of course, the sample size is lower for this experiment, but I was expecting the biggest tabu size to have the lowest cost. If we look at the average this does ring true where the higher our tabu size becomes the less cost, there is. As mentioned earlier with a smaller tabu list, our memory complexity should be lower with the sacrifice (theoretically being) having the possibility of a higher final cost

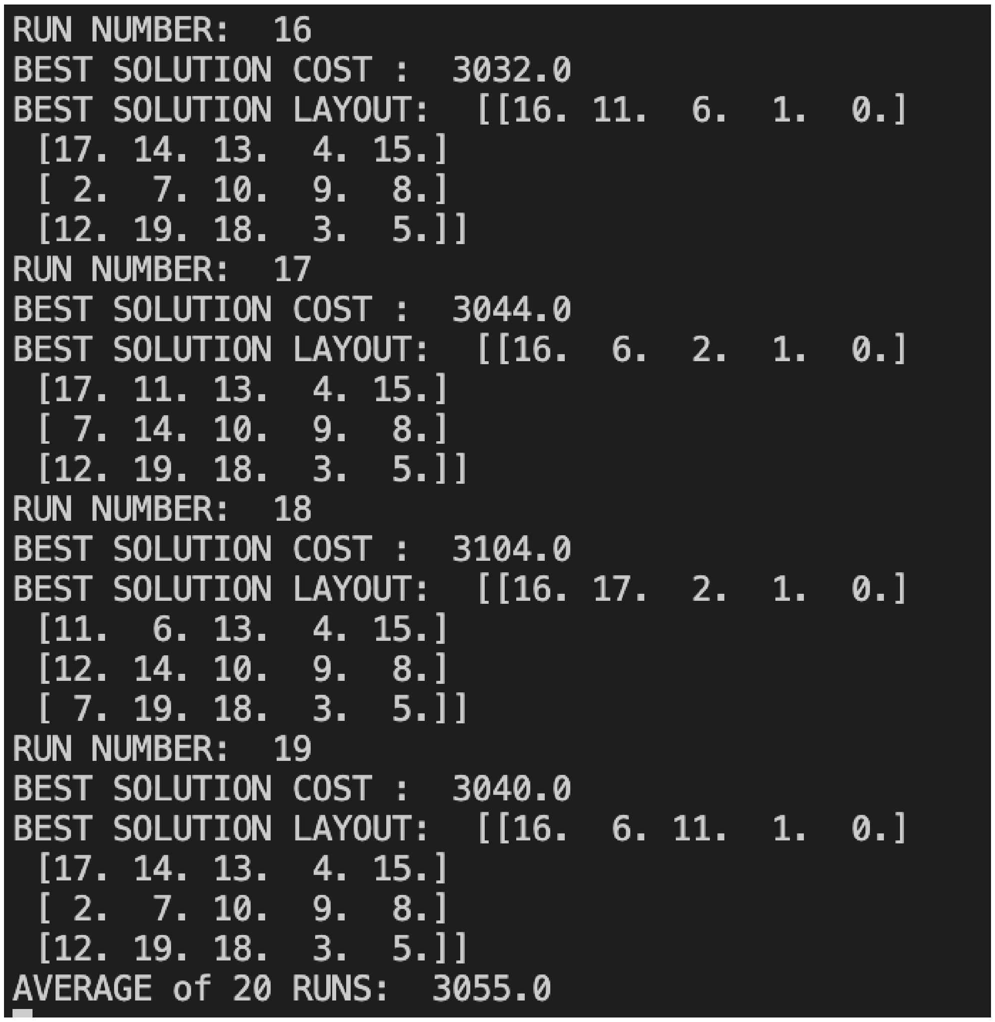


```
TABU SIZE:  20
BEST SOLUTION COST :  3026.0
BEST SOLUTION LAYOUT:  [[16.  4.  6.  1.  0.]
 [ 5.  3. 19.  8. 12.]
 [ 2. 18.  7. 14. 10.]
 [17. 13.  9. 11. 15.]]
TABU SIZE:  80
BEST SOLUTION COST :  3116.0
BEST SOLUTION LAYOUT:  [[ 5.  6.  4.  1.  0.]
 [ 2. 19.  7.  8. 12.]
 [16.  3. 18. 14. 10.]
 [17. 13.  9. 11. 15.]]
BASE TABU SIZE:  100
BEST SOLUTION COST :  3078.0
BEST SOLUTION LAYOUT:  [[16.  4.  6.  1.  0.]
 [ 2.  5. 13.  8. 12.]
 [ 3.  7. 19. 14. 10.]
 [17. 18.  9. 11. 15.]]
TABU SIZE:  120
BEST SOLUTION COST :  3032.0
BEST SOLUTION LAYOUT:  [[16.  4.  6.  1.  0.]
 [ 3.  5. 19.  8. 12.]
 [ 2. 18.  7. 14. 10.]
 [17. 13.  9. 11. 15.]]
TABU SIZE:  180
BEST SOLUTION COST :  3056.0
BEST SOLUTION LAYOUT:  [[16.  4.  3.  1.  0.]
 [ 2.  5.  6.  8. 12.]
 [ 7. 18. 19. 14. 10.]
 [17. 13.  9. 11. 15.]]
```

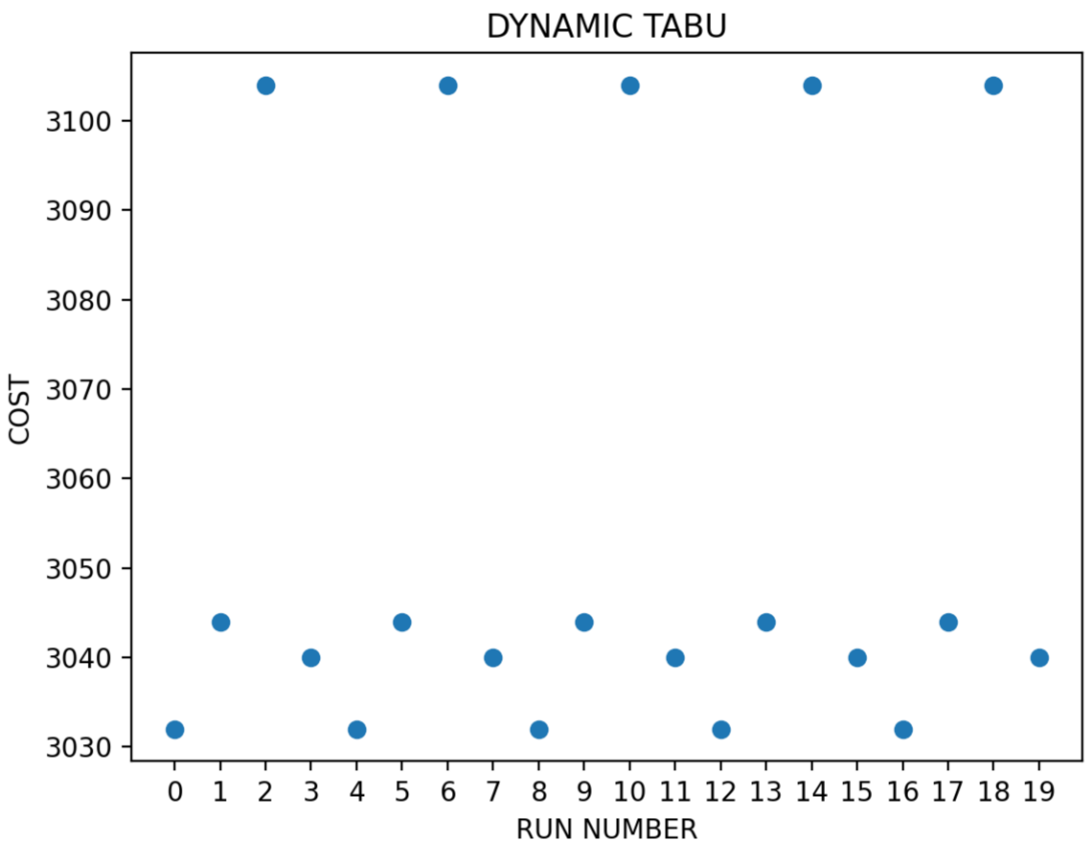


Experiment 3:

In this experiment we changed Tabu size to become dynamic. In our code we have it so every 45th iteration we pick a random variable between 1 and 300 to be our new tabu size. We then delete all elements that do not fit in the new tabu list by age in the list. With this change, we have the results below. Our average cost with the dynamic tabu list was 3055 which is slightly worse than our 20 our normal 20 runs. The explanation we can offer is that the random numbers may have been lower than 100 on average thus it was a lower average tabu size.

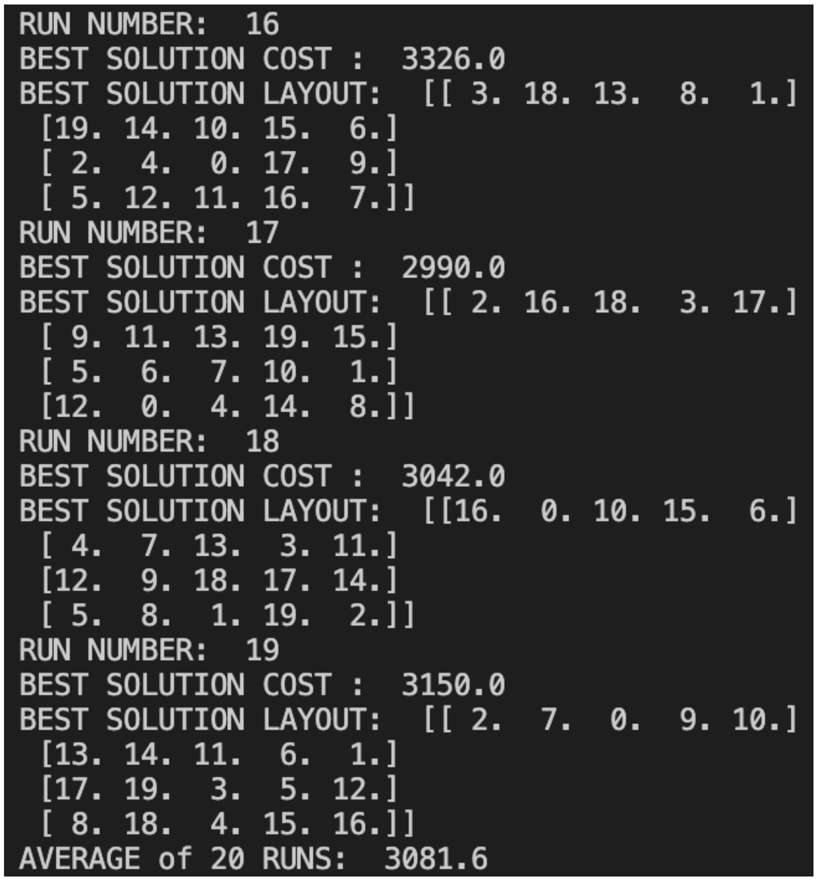


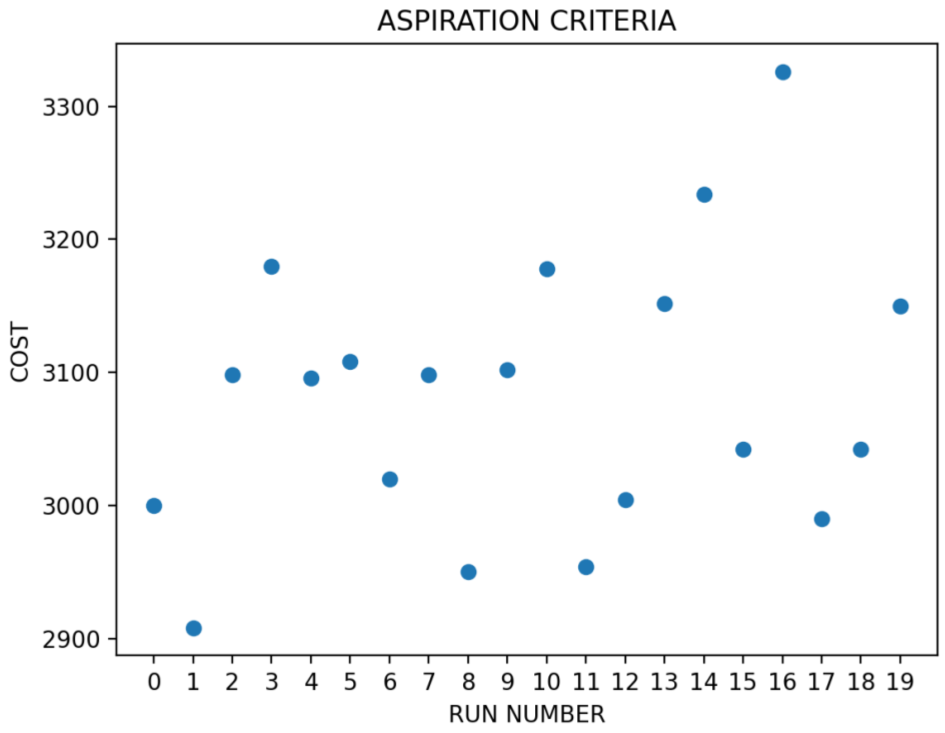
```
RUN NUMBER:  16
BEST SOLUTION COST :  3032.0
BEST SOLUTION LAYOUT:  [[16. 11.  6.  1.  0.]
 [17. 14. 13.  4. 15.]
 [ 2.  7. 10.  9.  8.]
 [12. 19. 18.  3.  5.]]
RUN NUMBER:  17
BEST SOLUTION COST :  3044.0
BEST SOLUTION LAYOUT:  [[16.  6.  2.  1.  0.]
 [17. 11. 13.  4. 15.]
 [ 7. 14. 10.  9.  8.]
 [12. 19. 18.  3.  5.]]
RUN NUMBER:  18
BEST SOLUTION COST :  3104.0
BEST SOLUTION LAYOUT:  [[16. 17.  2.  1.  0.]
 [11.  6. 13.  4. 15.]
 [12. 14. 10.  9.  8.]
 [ 7. 19. 18.  3.  5.]]
RUN NUMBER:  19
BEST SOLUTION COST :  3040.0
BEST SOLUTION LAYOUT:  [[16.  6. 11.  1.  0.]
 [17. 14. 13.  4. 15.]
 [ 2.  7. 10.  9.  8.]
 [12. 19. 18.  3.  5.]]
AVERAGE of 20 RUNS:  3055.0
```



Experiment 4

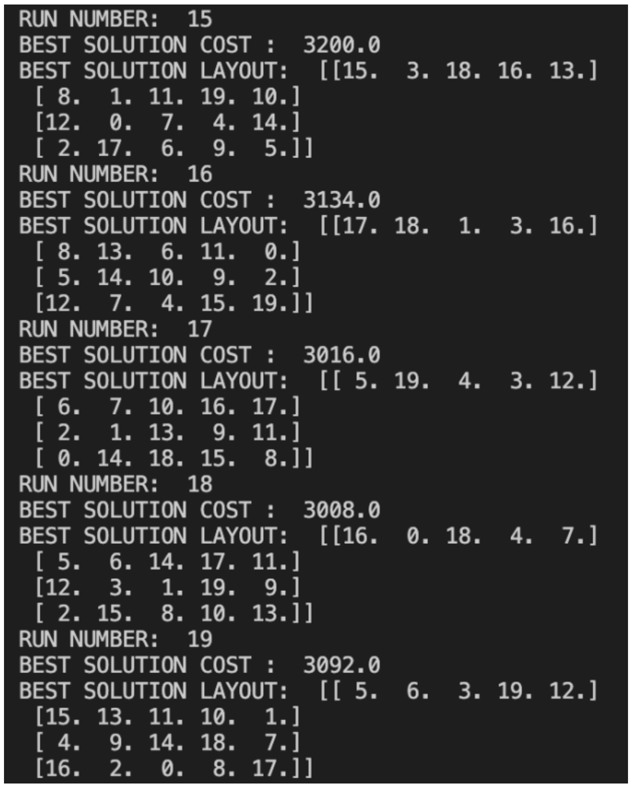
In experiment 4 we added an aspiration criterion by sorting our tabu list by cost. In this way the best solutions will stay in the tabu list longer. Surprisingly even with this aspiration criteria our average cost was higher than the initial 20 runs. We are not sure why this would occur as the aspiration criteria should help narrow it down, but we chalk it up again to the initial solution. The initial solution can have a big impact on what the cost is determined to be.


```
RUN NUMBER:  16
BEST SOLUTION COST :  3326.0
BEST SOLUTION LAYOUT:  [[ 3. 18. 13.  8.  1.]
 [19. 14. 10. 15.  6.]
 [ 2.  4.  0. 17.  9.]
 [ 5. 12. 11. 16.  7.]]
RUN NUMBER:  17
BEST SOLUTION COST :  2990.0
BEST SOLUTION LAYOUT:  [[ 2. 16. 18.  3. 17.]
 [ 9. 11. 13. 19. 15.]
 [ 5.  6.  7. 10.  1.]
 [12.  0.  4. 14.  8.]]
RUN NUMBER:  18
BEST SOLUTION COST :  3042.0
BEST SOLUTION LAYOUT:  [[16.  0. 10. 15.  6.]
 [ 4.  7. 13.  3. 11.]
 [12.  9. 18. 17. 14.]
 [ 5.  8.  1. 19.  2.]]
RUN NUMBER:  19
BEST SOLUTION COST :  3150.0
BEST SOLUTION LAYOUT:  [[ 2.  7.  0.  9. 10.]
 [13. 14. 11.  6.  1.]
 [17. 19.  3.  5. 12.]
 [ 8. 18.  4. 15. 16.]]
AVERAGE of 20 RUNS:  3081.6
```

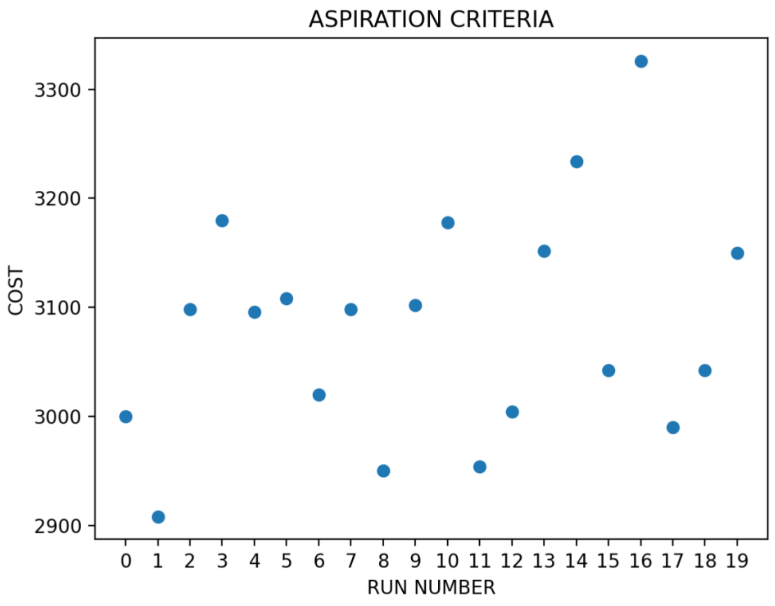


Experiment 5

In experiment 5 we added the clause to the aspiration to choose the layout at the top of the tabu list as the solution to focus on. We decided to go this route as swapping our best solution seemed like a good strategy to help us find equal or better solutions. Surprisingly this also had a higher average than the initial 20 runs. It seems like the first initial 20 runs may have been the outlier from the set of experiments.



```
RUN NUMBER:  15
BEST SOLUTION COST :  3200.0
BEST SOLUTION LAYOUT:  [[15.  3. 18. 16. 13.]
 [ 8.  1. 11. 19. 10.]
 [12.  0.  7.  4. 14.]
 [ 2. 17.  6.  9.  5.]]
RUN NUMBER:  16
BEST SOLUTION COST :  3134.0
BEST SOLUTION LAYOUT:  [[17. 18.  1.  3. 16.]
 [ 8. 13.  6. 11.  0.]
 [ 5. 14. 10.  9.  2.]
 [12.  7.  4. 15. 19.]]
RUN NUMBER:  17
BEST SOLUTION COST :  3016.0
BEST SOLUTION LAYOUT:  [[ 5. 19.  4.  3. 12.]
 [ 6.  7. 10. 16. 17.]
 [ 2.  1. 13.  9. 11.]
 [ 0. 14. 18. 15.  8.]]
RUN NUMBER:  18
BEST SOLUTION COST :  3008.0
BEST SOLUTION LAYOUT:  [[16.  0. 18.  4.  7.]
 [ 5.  6. 14. 17. 11.]
 [12.  3.  1. 19.  9.]
 [ 2. 15.  8. 10. 13.]]
RUN NUMBER:  19
BEST SOLUTION COST :  3092.0
BEST SOLUTION LAYOUT:  [[ 5.  6.  3. 19. 12.]
 [15. 13. 11. 10.  1.]
 [ 4.  9. 14. 18.  7.]
 [16.  2.  0.  8. 17.]]
```



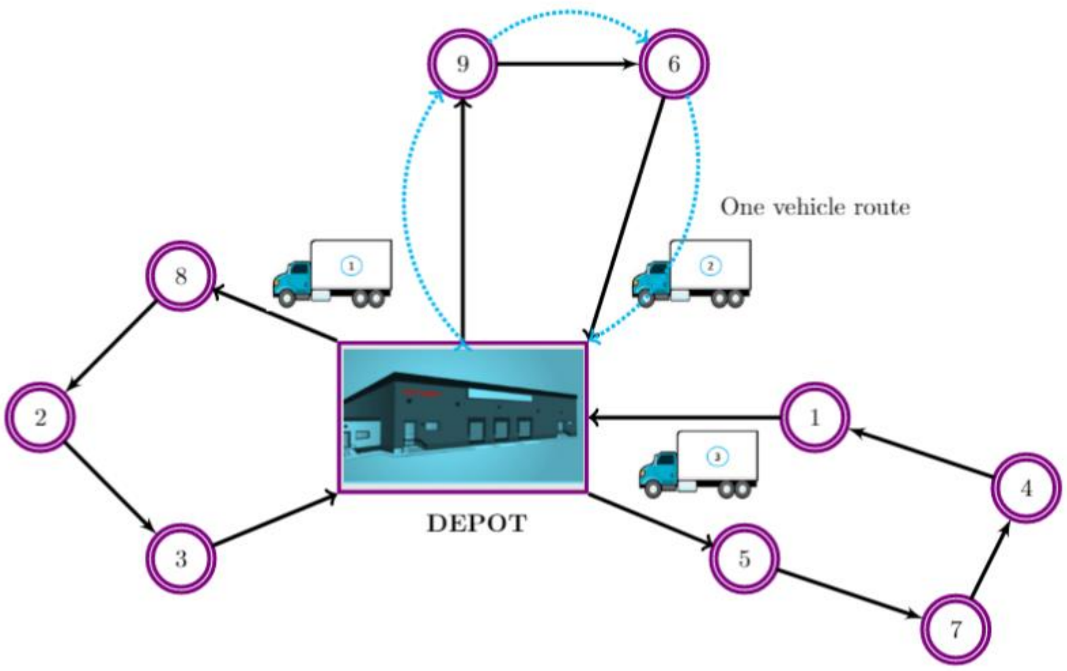
Problem 4

Simulated Annealing can be used to solve The Vehicle Routing Problem (VPR). VRP is defined as having m vehicles at a depot that need to service customers in c cities. The travel distances between every two cities are defined by a matrix D with element d_{ij} denoting distance between cities i and j. The travel distances from the depot to each of the cities are known. Each customer j has a service time s_j. The objective of the VPR is determining the routes to be taken by the set of m vehicles while satisfying the following conditions:

Ⓐ Starting and ending at the depot,

Ⓑ Having minimum cost (travel time +service time),

Ⓒ Each customer is visited exactly once by exactly one vehicle. The figure below illustrate possible routes for 9 customers and 3 vehicles problem



Optimization problem formulation:

VRP is an optimization problem that will generate trips of up to m vehicles with the objective of minimizing the total time spent on the journey (travel time + service time). The vehicle starts and ends at the depot and makes one single trip to visit a subset of cities. Each city should be only visited exactly once by one vehicle. In this problem, we are optimizing the total time taken by one vehicle.

Solution representation in the context of SA solution formulation

The vehicle routing problem is represented by a permutation of n customers/cities represented by {1,2,3,4…n}. the numbers representing customer nodes with the depot being node zero. The network involves a depot, with service time = 0, and a set of city nodes with service times as their heuristic.
The initial solution is made by implementing a simple graph traversal with randomized cities selected as next nodes to be visited.

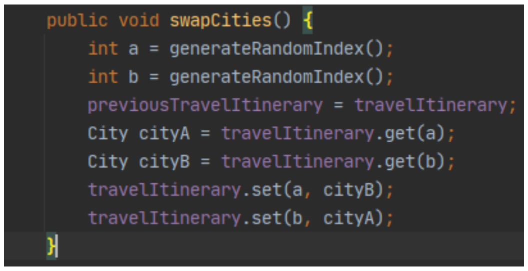
Objective function used to calculate cost

The cost can be calculated by adding the total accumulated time spent by the vehicles in consideration. In this implementation there is one vehicle under consideration. Cost is the total time which includes travel time of visiting every city exactly once + service time of each city plus time taken to return to the depot. When trying to minimize the total distance travelled, the cost is the total amount of KM travelled when the truck leaves the depot until the truck returns to the depot.

Neighborhood operator:

The neighborhood operators implemented are swap and revert-swap. The probability of selecting the revert operator depends on the current temperature and Delta E of the simulated annealing algorithm. Where Delta E = (cost of best solution found so far – cost of current solution) and the current temperature decreases at the cooling rate for each iteration. After taking each operation, the cost function is evaluated on the travel itinerary under consideration to determine if the current itinerary beats the best solution seen so far. Revert-swap, undoes the previous swap of cities. It is equivalent to taking potentially worsening moves to explore the search space. However, as temperature decreases, the chances of taking non improving actions decreases.

Swap: select two random cities in the shuffled list of cities, swap their positions.



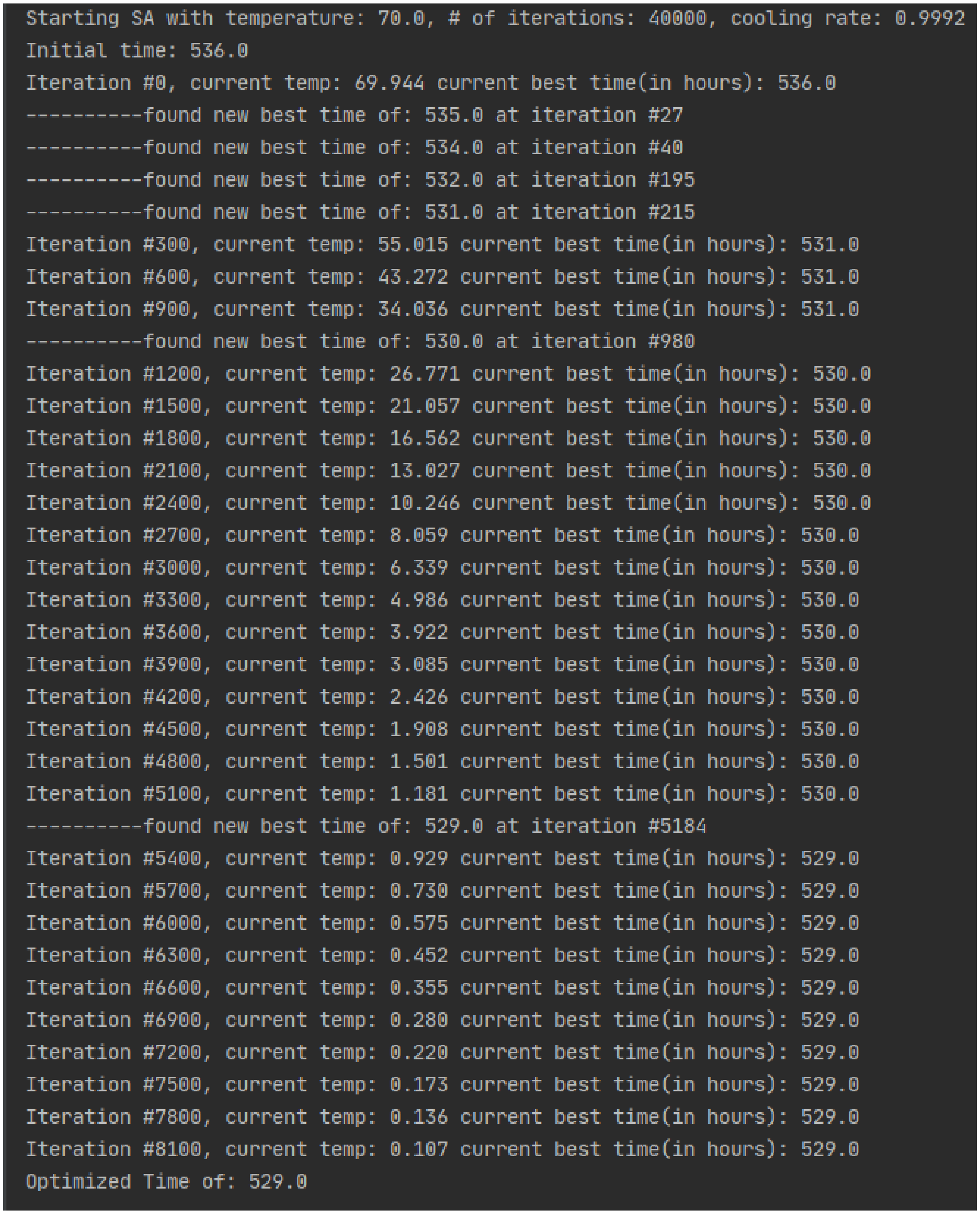
```java
public void swapCities() {
    int a = generateRandomIndex();
    int b = generateRandomIndex();
    previousTravelItinerary = travelItinerary;
    City cityA = travelItinerary.get(a);
    City cityB = travelItinerary.get(b);
    travelItinerary.set(a, cityB);
    travelItinerary.set(b, cityA);
}
```

Revert-swap: revert the itinerary from the previous swap.

How would the problem change if add the constraint that the total duration of any route shouldn't exceed a pre-set bound T?

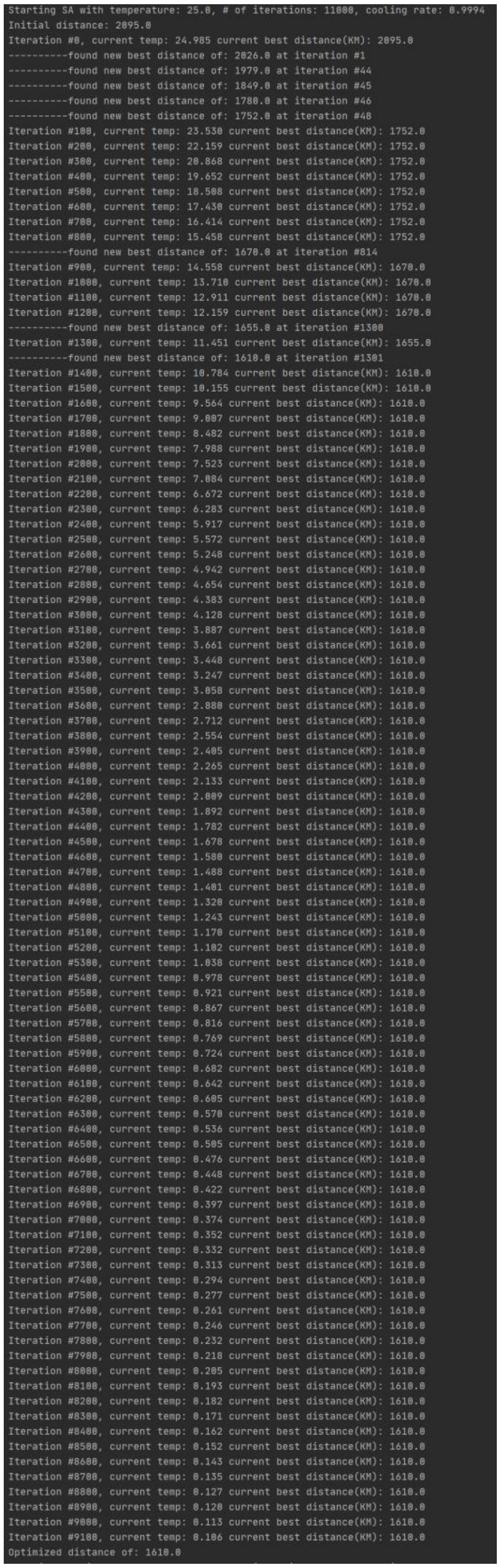
In that case. The optimization would only accept non-worsening solutions that fall in the category where the entire travel itinerary time is less than or equal to the pre-set bound T. For every itinerary that takes a longer time to complete, simulated annealing should be performed to see if those configurations could be improved upon to fit the new problem constraint. SA would run until the total duration of every route fall below the pre-set bound.

Running on instance of n-39.vrp optimizing for total time:



```
Starting SA with temperature: 70.0, # of iterations: 40000, cooling rate: 0.9992
Initial time: 536.0
Iteration #0, current temp: 69.944 current best time(in hours): 536.0
----------found new best time of: 535.0 at iteration #27
----------found new best time of: 534.0 at iteration #40
----------found new best time of: 532.0 at iteration #195
----------found new best time of: 531.0 at iteration #215
Iteration #300, current temp: 55.015 current best time(in hours): 531.0
Iteration #600, current temp: 43.272 current best time(in hours): 531.0
Iteration #900, current temp: 34.036 current best time(in hours): 531.0
----------found new best time of: 530.0 at iteration #980
Iteration #1200, current temp: 26.771 current best time(in hours): 530.0
Iteration #1500, current temp: 21.057 current best time(in hours): 530.0
Iteration #1800, current temp: 16.562 current best time(in hours): 530.0
Iteration #2100, current temp: 13.027 current best time(in hours): 530.0
Iteration #2400, current temp: 10.246 current best time(in hours): 530.0
Iteration #2700, current temp: 8.059 current best time(in hours): 530.0
Iteration #3000, current temp: 6.339 current best time(in hours): 530.0
Iteration #3300, current temp: 4.986 current best time(in hours): 530.0
Iteration #3600, current temp: 3.922 current best time(in hours): 530.0
Iteration #3900, current temp: 3.085 current best time(in hours): 530.0
Iteration #4200, current temp: 2.426 current best time(in hours): 530.0
Iteration #4500, current temp: 1.908 current best time(in hours): 530.0
Iteration #4800, current temp: 1.501 current best time(in hours): 530.0
Iteration #5100, current temp: 1.181 current best time(in hours): 530.0
----------found new best time of: 529.0 at iteration #5184
Iteration #5400, current temp: 0.929 current best time(in hours): 529.0
Iteration #5700, current temp: 0.730 current best time(in hours): 529.0
Iteration #6000, current temp: 0.575 current best time(in hours): 529.0
Iteration #6300, current temp: 0.452 current best time(in hours): 529.0
Iteration #6600, current temp: 0.355 current best time(in hours): 529.0
Iteration #6900, current temp: 0.280 current best time(in hours): 529.0
Iteration #7200, current temp: 0.220 current best time(in hours): 529.0
Iteration #7500, current temp: 0.173 current best time(in hours): 529.0
Iteration #7800, current temp: 0.136 current best time(in hours): 529.0
Iteration #8100, current temp: 0.107 current best time(in hours): 529.0
Optimized Time of: 529.0
```

Running on instance of n-39.vrp optimizing for shortest distance:



```
Starting SA with temperature: 25.0, # of iterations: 11000, cooling rate: 0.9994
Initial distance: 2095.0
Iteration #0, current temp: 24.985 current best distance(KM): 2095.0
----------found new best distance of: 2026.0 at iteration #1
----------found new best distance of: 1979.0 at iteration #44
----------found new best distance of: 1849.0 at iteration #45
----------found new best distance of: 1780.0 at iteration #46
----------found new best distance of: 1752.0 at iteration #48
Iteration #100, current temp: 23.530 current best distance(KM): 1752.0
Iteration #200, current temp: 22.159 current best distance(KM): 1752.0
Iteration #300, current temp: 20.868 current best distance(KM): 1752.0
Iteration #400, current temp: 19.652 current best distance(KM): 1752.0
Iteration #500, current temp: 18.508 current best distance(KM): 1752.0
Iteration #600, current temp: 17.430 current best distance(KM): 1752.0
Iteration #700, current temp: 16.414 current best distance(KM): 1752.0
Iteration #800, current temp: 15.458 current best distance(KM): 1752.0
----------found new best distance of: 1670.0 at iteration #814
Iteration #900, current temp: 14.558 current best distance(KM): 1670.0
Iteration #1000, current temp: 13.710 current best distance(KM): 1670.0
Iteration #1100, current temp: 12.911 current best distance(KM): 1670.0
Iteration #1200, current temp: 12.159 current best distance(KM): 1670.0
----------found new best distance of: 1655.0 at iteration #1300
Iteration #1300, current temp: 11.451 current best distance(KM): 1655.0
----------found new best distance of: 1610.0 at iteration #1301
Iteration #1400, current temp: 10.784 current best distance(KM): 1610.0
Iteration #1500, current temp: 10.155 current best distance(KM): 1610.0
Iteration #1600, current temp: 9.564 current best distance(KM): 1610.0
Iteration #1700, current temp: 9.007 current best distance(KM): 1610.0
Iteration #1800, current temp: 8.482 current best distance(KM): 1610.0
Iteration #1900, current temp: 7.988 current best distance(KM): 1610.0
Iteration #2000, current temp: 7.523 current best distance(KM): 1610.0
Iteration #2100, current temp: 7.084 current best distance(KM): 1610.0
Iteration #2200, current temp: 6.672 current best distance(KM): 1610.0
Iteration #2300, current temp: 6.283 current best distance(KM): 1610.0
Iteration #2400, current temp: 5.917 current best distance(KM): 1610.0
Iteration #2500, current temp: 5.572 current best distance(KM): 1610.0
Iteration #2600, current temp: 5.248 current best distance(KM): 1610.0
Iteration #2700, current temp: 4.942 current best distance(KM): 1610.0
Iteration #2800, current temp: 4.654 current best distance(KM): 1610.0
Iteration #2900, current temp: 4.383 current best distance(KM): 1610.0
Iteration #3000, current temp: 4.128 current best distance(KM): 1610.0
Iteration #3100, current temp: 3.887 current best distance(KM): 1610.0
Iteration #3200, current temp: 3.661 current best distance(KM): 1610.0
Iteration #3300, current temp: 3.448 current best distance(KM): 1610.0
Iteration #3400, current temp: 3.247 current best distance(KM): 1610.0
Iteration #3500, current temp: 3.058 current best distance(KM): 1610.0
Iteration #3600, current temp: 2.880 current best distance(KM): 1610.0
Iteration #3700, current temp: 2.712 current best distance(KM): 1610.0
Iteration #3800, current temp: 2.554 current best distance(KM): 1610.0
Iteration #3900, current temp: 2.405 current best distance(KM): 1610.0
Iteration #4000, current temp: 2.265 current best distance(KM): 1610.0
Iteration #4100, current temp: 2.133 current best distance(KM): 1610.0
Iteration #4200, current temp: 2.009 current best distance(KM): 1610.0
Iteration #4300, current temp: 1.892 current best distance(KM): 1610.0
Iteration #4400, current temp: 1.782 current best distance(KM): 1610.0
Iteration #4500, current temp: 1.678 current best distance(KM): 1610.0
Iteration #4600, current temp: 1.580 current best distance(KM): 1610.0
Iteration #4700, current temp: 1.488 current best distance(KM): 1610.0
Iteration #4800, current temp: 1.401 current best distance(KM): 1610.0
Iteration #4900, current temp: 1.320 current best distance(KM): 1610.0
Iteration #5000, current temp: 1.243 current best distance(KM): 1610.0
Iteration #5100, current temp: 1.170 current best distance(KM): 1610.0
Iteration #5200, current temp: 1.102 current best distance(KM): 1610.0
Iteration #5300, current temp: 1.038 current best distance(KM): 1610.0
Iteration #5400, current temp: 0.978 current best distance(KM): 1610.0
Iteration #5500, current temp: 0.921 current best distance(KM): 1610.0
Iteration #5600, current temp: 0.867 current best distance(KM): 1610.0
Iteration #5700, current temp: 0.816 current best distance(KM): 1610.0
Iteration #5800, current temp: 0.769 current best distance(KM): 1610.0
Iteration #5900, current temp: 0.724 current best distance(KM): 1610.0
Iteration #6000, current temp: 0.682 current best distance(KM): 1610.0
Iteration #6100, current temp: 0.642 current best distance(KM): 1610.0
Iteration #6200, current temp: 0.605 current best distance(KM): 1610.0
Iteration #6300, current temp: 0.570 current best distance(KM): 1610.0
Iteration #6400, current temp: 0.536 current best distance(KM): 1610.0
Iteration #6500, current temp: 0.505 current best distance(KM): 1610.0
Iteration #6600, current temp: 0.476 current best distance(KM): 1610.0
Iteration #6700, current temp: 0.448 current best distance(KM): 1610.0
Iteration #6800, current temp: 0.422 current best distance(KM): 1610.0
Iteration #6900, current temp: 0.397 current best distance(KM): 1610.0
Iteration #7000, current temp: 0.374 current best distance(KM): 1610.0
Iteration #7100, current temp: 0.352 current best distance(KM): 1610.0
Iteration #7200, current temp: 0.332 current best distance(KM): 1610.0
Iteration #7300, current temp: 0.313 current best distance(KM): 1610.0
Iteration #7400, current temp: 0.294 current best distance(KM): 1610.0
Iteration #7500, current temp: 0.277 current best distance(KM): 1610.0
Iteration #7600, current temp: 0.261 current best distance(KM): 1610.0
Iteration #7700, current temp: 0.246 current best distance(KM): 1610.0
Iteration #7800, current temp: 0.232 current best distance(KM): 1610.0
Iteration #7900, current temp: 0.218 current best distance(KM): 1610.0
Iteration #8000, current temp: 0.205 current best distance(KM): 1610.0
Iteration #8100, current temp: 0.193 current best distance(KM): 1610.0
Iteration #8200, current temp: 0.182 current best distance(KM): 1610.0
Iteration #8300, current temp: 0.171 current best distance(KM): 1610.0
Iteration #8400, current temp: 0.162 current best distance(KM): 1610.0
Iteration #8500, current temp: 0.152 current best distance(KM): 1610.0
Iteration #8600, current temp: 0.143 current best distance(KM): 1610.0
Iteration #8700, current temp: 0.135 current best distance(KM): 1610.0
Iteration #8800, current temp: 0.127 current best distance(KM): 1610.0
Iteration #8900, current temp: 0.120 current best distance(KM): 1610.0
Iteration #9000, current temp: 0.113 current best distance(KM): 1610.0
Iteration #9100, current temp: 0.106 current best distance(KM): 1610.0
Optimized distance of: 1610.0
```

Time and space complexity

The time complexity of this algorithm is O(**X** * # of cities)

Where **X** = Max(# of SA iterations, speed at which temperature cools to near 0)

Space complexity is O (n) where n = number of cities to the problem. In this implementation, the distances and times taken to reach each and every city is not stored in an adjacency list data structure, but rather calculated when the next destination city is selected.