Man Travels in a Circle… Everybody Gasps

Artificial Intelligence Final Quarter Project

Chloe Nelson (24281821)

Eric Ng (63403259)

Brandon Wang (21002371)

CS271P Section 01, Dr. Kalev Kask, Department of Informatics

University of California, Irvine

Irvine, CA, 92697

Fall 2022

I. Problem

The traveling salesman problem is a well known graph problem that is NP-hard. The goal is to find the shortest path cost that traverses through every node once and returns to the start. To solve this problem, two different algorithms with heuristics will be implemented and explored. The two algorithms are the Branch and Bound Depth First Search (BnB-DFS) and the Stochastic Local Search (SLS).

II. Branch and Bound - Depth First Search: BnB-DFS

A. Approach

The first approach was to use BnB-DFS with a minimum remaining cost heuristic (greedy heuristic). To implement this, a stack was used that would contain the current visited nodes as well as each neighboring node (stack<([cur_visited_nodes], neighboring_node_1), ([cur_visited_nodes], neighboring_node_2) … >) and continue building the stack throughout the run. For each element popped from the stack, the current visited nodes and the neighboring node would be used to compute the heuristic by finding the minimum remaining cost of the current path. This is found by calculating the path cost for the current node if it were to take only the minimum cost paths until it reaches the original node. If the sum of the minimum cost path and the current cost of going from the last visited node to the neighboring node is larger than upper bound *U* then the branch will be pruned. In the event that the the cur_visited_node reaches the original node and it did not get pruned the new value *U* will be set to the sum of the minimum cost path and the current cost of going from the last visited node to the neighboring node and it will result to the best assignment (**see section II part D for further implementation details**).

B. Analysis

Two versions of the minimum remaining cost heuristic were tested, one was to set the upper bound *U* = final_path_cost + remaining_minimum_path_cost and one with *U* = final_path_cost + sum_minimum_path_cost. The hypothesis was that branches that were deeper down the tree should allow for more paths, thus potentially better paths to find. The idea was to see if the tradeoff for a greater amount of time spent searching deeper within the tree, would yield better path results. This was also tested to see if there was too little information from the remaining path cost.

Contrary to the initial hypothesis it was seen that using *U* = final_path_cost + remaining_minimum_path_cost yielded either similar or better results as seen in **Fig.1, Fig.2, Fig.3.** This means that finding paths that are deeper in the tree does not necessarily mean that they are better in terms of path cost, and pruning those branches earlier may be more effective. In

Fig.3 the difference between the two path costs is much more apparent when the standard deviation is higher, and it shows that the remaining minimum_path_cost performs substantially better.

In **Fig. 1** you can see that both heuristics produced the same results, but the remaining minimum cost was able to find more assignments where *U* improved. In terms of speed since there were only 5 paths, both heuristics were even.

In **Fig. 2** the remaining minimum cost's path found was `2.8312` while the second heuristic found a path cost of `3.642`. This shows that the tradeoff was not worth the extra time that it takes to move through the larger depth of the tree. In both cases the remaining minimum cost performed either better or equally as well as using the sum of the remaining cost.

While testing it was also noticed that lower standard deviation problems take longer to generate a path. This is because it uses the minimum remaining cost heuristic and there are more branches to check since values will be very close to *U*. This also leads to results that may not be optimal since every path cost is different by only a small margin. By contrast it was also found that the approach works better when standard deviation is higher since the difference between the remaining path costs will be much larger, thus leading to the heuristic being more effective (as seen in Fig.1). To improve the heuristics a normalization technique could be used when the standard deviation is low and the path costs are very similar which would make the standard deviation seem higher in order to distinguish different path costs.

C. Assessment of Algorithm

BnB-DFS using Nearest Neighbor Heuristic

Time Complexity - Exponential

- BnB uses DFS which has a exponential time complexity in terms of branching factor and max depth
- Nearest Neighbor has worst case time complexity of $O(N^2)$ since for each node it needs to find min() which is O(N) time complexity
- Neighbors will be worst case O(N) using set difference

Space Complexity - Linear

- Since the recursive call in Minimum_Remaining_Path will only go to depth at most N
- Stack is also a linear space complexity
- The sets used in Neighbors are also of space complexity O(N)

D. Description of Algorithms

BNB_TSP(cost_matrix, N) -

INPUT:

- cost_matrix - the adjacency matrix that contains all the cost traveling from one node to the next (generated from generate_travelling_salesman_problem.py)
- N - the number of nodes

OUTPUT:

- the best assignment found from running the BNB algorithm with the heuristic

NOTE: decided the algorithm will always start at Node 0

- Initially adds all the neighbors of node_0 (which will be all other nodes since it's a completed graph) along with a list of visited nodes (so far just node_0) to the stack, sets assignment to NULL (no assignments found to begin with), and sets upper bound variable 'U' to be INF (highest possible int)
- Begins DFS by popping the nodes off the stack and holds them in the variables 'cur_visited' (list of visited indices) and 'target' which is the node to move to
- 'cur_node' will always be the most recent index inserted into cur_visited
- initially no pruning happens, but later on if the sum of the cost from cur_node to target and the nearest neighbor heuristic is larger than upper bound variable 'U' then prune and skip the current assignment.
- If it manages to pass the pruning check and the length of the visited node contains all the nodes, then there is a new upper bound completed assignment, thus set 'U' to the new upper bound and 'best_assignment' is the new tour found.
- If passes pruning and full tour is not yet found, update the visited nodes and add to stack the new list of visited nodes as well as each neighbor.

Minimum_Remaining_Path(cost_matrix, cur_node, visited, N) -

INPUT:

- cost_matrix - the adjacency matrix that contains all the cost traveling from one node to the next (generated from generate_travelling_salesman_problem.py)
- cur_node - the current node to check for minimum cost of remaining path
- visited - the list of indices visited
- N - the number of nodes

OUTPUT:

- The summation of path costs from taking the minimum cost path until the goal is reached

Heuristic function for the Branch and Bound function

- First checks if all the possible nodes are already visited, if they are then just return the path cost of the last visited node to the initial node
- recursively calculates and returns the path cost from taking the minimum path of each node, until all the nodes are visited

Neighbors(cur_node, visited, N) -

INPUT:

- cur_node - the node to check the neighbor of
- visited - the list of indices visited
- N - the number of Nodes

Helper function for the Branch and Bound function

- Checks if the length of the visited nodes is the same size of the number of nodes (no more neighbors), in this case just send back the first node to complete the cycle
- Gets a list of unvisited nodes by doing the set difference of all the nodes against the nodes visited

Fig.1: Tested using 5_0.0_10.0.out file

1a. Remaining minimum cost

```
Current assignment: [0, 4, 3, 2, 1, 0], cost: 67.6261
Current assignment: [0, 4, 2, 3, 1, 0], cost: 52.8397
Current assignment: [0, 4, 2, 1, 3, 0], cost: 43.7968
Current assignment: [0, 4, 1, 2, 3, 0], cost: 37.589800000000004
Current assignment: [0, 3, 2, 1, 4, 0], cost: 34.8157
Best assignment:
[0, 3, 2, 1, 4, 0]
timer: 0.010574341s
```

1b. Sum remaining minimum cost

```
Current assignment: [0, 4, 3, 2, 1, 0], cost: 115.0893
Current assignment: [0, 4, 2, 3, 1, 0], cost: 92.6708
Current assignment: [0, 4, 2, 1, 3, 0], cost: 68.3637
Current assignment: [0, 3, 2, 1, 4, 0], cost: 40.6541
Best assignment:
[0, 3, 2, 1, 4, 0]
timer: 0.011291027s
```

Fig.2: Tested using 10_0.0_1.0.out file

2a. Remaining minimum cost

```
Current assignment: [0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0], cost: 8.8381
Current assignment: [0, 9, 8, 7, 6, 5, 1, 4, 3, 2, 0], cost: 7.0437
Current assignment: [0, 9, 8, 7, 6, 3, 2, 4, 5, 1, 0], cost: 6.8903
Current assignment: [0, 9, 8, 7, 5, 4, 2, 3, 6, 1, 0], cost: 6.485799999999999
Current assignment: [0, 9, 8, 7, 4, 5, 2, 3, 6, 1, 0], cost: 6.1152999999999995
Current assignment: [0, 9, 8, 7, 3, 6, 1, 4, 5, 2, 0], cost: 5.7059
Current assignment: [0, 9, 8, 6, 3, 7, 5, 4, 1, 2, 0], cost: 5.5694
Current assignment: [0, 9, 8, 6, 3, 7, 4, 5, 1, 2, 0], cost: 5.121
Current assignment: [0, 9, 8, 4, 7, 3, 6, 1, 5, 2, 0], cost: 4.7832
Current assignment: [0, 9, 7, 6, 3, 2, 8, 4, 5, 1, 0], cost: 4.5333000000000006
Current assignment: [0, 9, 7, 5, 4, 8, 2, 3, 6, 1, 0], cost: 4.1288
Current assignment: [0, 9, 7, 5, 4, 1, 6, 3, 8, 2, 0], cost: 4.0225
Current assignment: [0, 9, 7, 4, 5, 1, 6, 3, 8, 2, 0], cost: 3.5741
Current assignment: [0, 6, 1, 4, 5, 9, 7, 3, 8, 2, 0], cost: 3.5538999999999996
Current assignment: [0, 2, 8, 4, 5, 9, 7, 3, 6, 1, 0], cost: 3.3615
Best assignment:
[0, 2, 8, 4, 5, 9, 7, 3, 6, 1, 0]
timer: 0.565133810s
```

2b. Sum remaining minimum cost

```
Current assignment: [0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0], cost: 13.610799999999998
Current assignment: [0, 9, 8, 7, 6, 5, 4, 1, 3, 2, 0], cost: 13.215699999999998
Current assignment: [0, 9, 8, 7, 5, 4, 6, 3, 2, 1, 0], cost: 12.5627
Current assignment: [0, 9, 8, 7, 5, 4, 3, 6, 1, 2, 0], cost: 11.644100000000002
Current assignment: [0, 9, 8, 6, 5, 4, 7, 3, 2, 1, 0], cost: 11.552799999999998
Current assignment: [0, 9, 8, 4, 7, 3, 6, 5, 1, 2, 0], cost: 11.119799999999998
Current assignment: [0, 9, 8, 4, 5, 7, 3, 6, 1, 2, 0], cost: 10.9842
Current assignment: [0, 9, 5, 7, 3, 6, 8, 4, 1, 2, 0], cost: 10.943399999999999
Current assignment: [0, 9, 5, 4, 8, 7, 3, 6, 1, 2, 0], cost: 10.5922
Current assignment: [0, 9, 5, 4, 8, 6, 3, 7, 1, 2, 0], cost: 10.127799999999999
Current assignment: [0, 9, 5, 4, 1, 7, 3, 6, 8, 2, 0], cost: 9.2724
Best assignment:
[0, 9, 5, 4, 1, 7, 3, 6, 8, 2, 0]
timer: 2.460147619s
```

Fig. 3

File	File	Path	Path_cost
10_10.0_0.1	Remaining minimum cost	[0, 6, 3, 8, 2, 1, 7, 5, 4, 9, 0]	98.80089999999998

	Sum remaining minimum cost	[0, 8, 7, 6, 5, 4, 3, 2, 1, 9, 0]	99.65039999999999
8_5.0_1.0	Remaining minimum cost	[0, 6, 7, 1, 3, 2, 5, 4, 0]	37.820100000000004
	Sum remaining minimum cost	[0, 7, 5, 4, 3, 2, 1, 6, 0]	41.0499
6_1.0_0.5	Remaining minimum cost	[0, 3, 1, 2, 4, 5, 0]	3.8922
	Sum remaining minimum cost	[0, 5, 4, 2, 1, 3, 0]	3.8921999999999994
12_5.0_2.0	Remaining minimum cost	[0, 8, 1, 4, 9, 5, 7, 10, 6, 3, 2, 11, 0]	35.3678
	Sum remaining minimum cost	[0, 9, 8, 10, 6, 5, 4, 1, 7, 3, 2, 11, 0]	45.66029999999999

III. Stochastic Local Search: SLS

A. Approach

The simulated annealing is an SLS algorithm, where the goal is to find the global minima of some objective function. In the Traveling Salesman Problem, the objective function was defined to be the total path cost of a complete tour. To escape local minima, a temperature variable is used. When temperature is high, the probabilities of selecting a specific neighbor from the current node is roughly equal (blind algorithm), with no regards for cost. As temperature decreases over every iteration by a rate of alpha, these probabilities start favoring minimum cost

paths, until eventually the search becomes a greedy algorithm. Eventually, the algorithm stops once temperature is lower than a given stopping threshold.

The nearest neighbor and simulated annealing heuristics were selected to help maximize the simplicity of the challenge and to minimize the cost of the algorithm.

B. Analysis

There are a variety of factors in simulated annealing that affect the algorithm's efficiency, such as initial temperature, the rate of alpha, and the stop condition. It is necessary to see how the algorithm's average assignment cost in a random restart wrapper is changed when these factors are manipulated. To start with, the TSP problem should be large enough in nodes where a significant performance change would be easily detected. In this analysis, the test uses a randomly generated TSP problem with 100 nodes and the average costs are taken after running the SLS algorithm. Average cost is used instead of "best cost", because we are not only interested in getting to a solution as SLS is based on probabilities by nature. Therefore, average costs also account for instances where a suboptimal solution is returned, so any test we run with the lowest average cost is assumed to have found the "best cost" more often than others.

The first factor of interest is temperature. The likelihood that a "worse choice" is chosen is proportional to temperature and the difference in cost between the current tour and the new one. Therefore, higher initial temperatures should make the algorithm choose worse tours more often at the start, while lower initial temperatures should behave more similar to a greedy algorithm. A multi-step design is implemented to test which temperature values have the best average cost. To keep other factors constant during our test, assume an arbitrary alpha value of 0.9, with a stop value of 0.001. The procedure is as follows:

(1) Run SLS with the modified value of interest (temperature) through 10 iterations, and average the costs of each iteration's best assignment to find the average cost of the trial. Record the temperature that has the best average cost.
(2) Repeat step 1 multiple times (100), and count the number of times each value of interest (temperature) is chosen as having the best average cost. With these results, examine the cumulative count distribution of the temperatures to see which has the most probability of finding the best assignment.

Result:

```
SLS Tester: TSP Problem with 100 nodes
-------------------
Temp 0.9 -- Count: 15 , Prob: 0.15
Temp 0.8 -- Count: 14 , Prob: 0.14
Temp 0.7 -- Count: 7 , Prob: 0.07
Temp 0.6 -- Count: 14 , Prob: 0.14
Temp 0.5 -- Count: 11 , Prob: 0.11
Temp 0.4 -- Count: 6 , Prob: 0.06
Temp 0.3 -- Count: 15 , Prob: 0.15
Temp 0.2 -- Count: 4 , Prob: 0.04
Temp 0.1 -- Count: 14 , Prob: 0.14
```

```
SLS Tester: TSP Problem with 100 nodes
-------------------
Temp 0.1 -- Count: 5 , Prob: 0.05
Temp 0.5 -- Count: 11 , Prob: 0.11
Temp 1 -- Count: 16 , Prob: 0.16
Temp 10 -- Count: 11 , Prob: 0.11
Temp 50 -- Count: 10 , Prob: 0.1
Temp 100 -- Count: 19 , Prob: 0.19
Temp 500 -- Count: 14 , Prob: 0.14
Temp 1000 -- Count: 14 , Prob: 0.14
```

Interestingly, there doesn't seem to be any bias in the distribution even from temperatures as small as 0.1 all the way to temperatures of 1000. In our next test, the same procedure is followed but with our value of interest set as alpha, with temperature fixed at 0.9.

```
SLS Tester: TSP Problem with 100 nodes
-------------------
alpha 0.9 -- Count: 12 , Prob: 0.12
alpha 0.8 -- Count: 13 , Prob: 0.13
alpha 0.7 -- Count: 11 , Prob: 0.11
alpha 0.6 -- Count: 16 , Prob: 0.16
alpha 0.5 -- Count: 5 , Prob: 0.05
alpha 0.4 -- Count: 13 , Prob: 0.13
alpha 0.3 -- Count: 9 , Prob: 0.09
alpha 0.2 -- Count: 12 , Prob: 0.12
alpha 0.1 -- Count: 9 , Prob: 0.09
```

Once again, there doesn't seem to be any bias in the alpha selection. Alpha of 0.9 and 0.2 both have equal probability of producing the best average cost. To test this even further, select two arbitrary combinations of temperature and alpha to test against after many (1000) iterations.

```
Average 1 (Temp = 0.9, Alpha = 0.9) = 9981.95815289158
Average 2 (Temp = 0.4, Alpha = 0.4) = 9984.123402620726
```

Even with a large degree of change in temperature and alpha, the average value is still very close. How does this analysis work when done on a smaller TSP problem (e.g. 10 nodes).

```
SLS Tester: TSP Problem with 10 nodes
-------------------
alpha 0.9 -- Count: 11 , Prob: 0.11
alpha 0.8 -- Count: 12 , Prob: 0.12
alpha 0.7 -- Count: 16 , Prob: 0.16
alpha 0.6 -- Count: 6 , Prob: 0.06
alpha 0.5 -- Count: 11 , Prob: 0.11
alpha 0.4 -- Count: 9 , Prob: 0.09
alpha 0.3 -- Count: 12 , Prob: 0.12
alpha 0.2 -- Count: 7 , Prob: 0.07
alpha 0.1 -- Count: 16 , Prob: 0.16
```

```
SLS Tester: TSP Problem with 10 nodes
-------------------
Temp 0.9 -- Count: 11 , Prob: 0.11
Temp 0.8 -- Count: 13 , Prob: 0.13
Temp 0.7 -- Count: 6 , Prob: 0.06
Temp 0.6 -- Count: 10 , Prob: 0.1
Temp 0.5 -- Count: 10 , Prob: 0.1
Temp 0.4 -- Count: 9 , Prob: 0.09
Temp 0.3 -- Count: 15 , Prob: 0.15
Temp 0.2 -- Count: 15 , Prob: 0.15
Temp 0.1 -- Count: 11 , Prob: 0.11
```

Yet again, no visible patterns or bias are seen in the distribution. With TSP problems, it is clear that temperature and alpha do have much control over the average cost. We hypothesize this is because the TSP problems are all complete graphs with every node having a link to another. Simulated annealing uses temperature to escape local minima when performing local search for the best path. When every node has a link to the other however, then bad paths are easily reversed once temperature starts to decrease. Thus, over time average costs end up balancing out once the algorithm starts to become more greedy.

The only noticeable difference with low alphas is that the runtime is very quick (since it approaches the stopping condition faster). Since the temperature and alpha doesn't seem to have a significant effect on the result, it could be argued that a low alpha to quicken runtime would be just as efficient as a high alpha introducing more random change.

This SLS approach would work better on a non-complete graph where the path selection is limited. In such a scenario, the first path that is chosen would have a significant effect on the rest of the tour, and simulated annealing might be able to escape local minima in such a situation. When the graph is complete like the above TSP problems, perhaps a better implementation of the simulated annealing would be to calculate the cost from the current node to the next node, and using temperature to calculate the probability it will choose the next node based on the path cost (low temperatures) or semi-randomly (high temperatures).

The source code attached executes only a single run of the simulated annealing algorithm for grading and testing purposes, for real-life application purposes this should be put into a random restart wrapper to have the best chance of finding the optimal tour.

C. Assessment of Algorithm

Simulated Annealing

Time Complexity - Exponential

- Simulated annealing can be thought of as the physical annealing that is applied to the properties of a metal. Once the metal is at a bendable temperature, then it

starts to cool based on its environment. This is an exponential process in any environment. [1]

- Like the annealing of different metals, simulated annealing exponential decay depends on the temperature. When the temperature is high, it allows the algorithm to explore more, and when it is low, the algorithm will exploit more.
- Time complexity can be thought of as $O(N^d)$ where d is an integer that describes the 'cooling' rate of the temperature.

Space Complexity - Linear

- A local search algorithm does not need a data structure to keep track of a frontier or visited nodes. The implementation only uses sets of numbers to represent a complete tour path.

D. <u>Description</u>

<u>Simulated Annealing</u>

simulated_annealing(cost_matrix, N, T, alpha)

INPUT:

- **cost_matrix** - the adjacency matrix that contains all the cost traveling from one node to the next (generated from generate_travelling_salesman_problem.py)
- **N** - number of nodes
- **T** - initial temperature
- **alpha** - value between 0 and 1 that represents the rate at which T decreases exponentially

OUTPUT:

- Current total path after stopping condition is met

The main simulated annealing looping function

- Starts off by generating a random tour of the cities
- While temperature is above the stopping condition, the algorithm will randomly change two cities in the tour, and calculate the new total path cost to compare with the current total path.
- If the new tour has a total cost that is less than the current total cost, then the algorithm uses the new tour as its current assignment.
- Repeats until stopping condition is met

GENERATE_RANDOM_TOUR(N)

INPUT:

- N - number of nodes

OUTPUT:

- tour - a set of numbers from 1 to N-1

Helper function that generates a random tour

- Starts by creating a set of numbers from 1 to N-1
- Randomizes the tour by shuffling the numbers
- Returns the random tour as a set of numbers

CHANGE_PATH(curr_tour)

INPUT:

- **curr_tour** - The current tour order as a set of numbers representing nodes

OUTPUT:

- The new tour after randomly swapping two nodes (excludes the starting node)

Helper function that generates a new tour after having two random nodes swapped

- Randomly generates two numbers from 2 to N-1 (excludes 1 as starting node cannot change)
- Swaps the location of the two randomly selected nodes
- Returns the new tour with the two locations swapped

TOTAL_PATH_COST(cost_matrix, current_tour)

INPUT:

- **cost_matrix** - the adjacency matrix that contains all the cost traveling from one node to the next (generated from generate_travelling_salesman_problem.py)

- **current_tour** - The current tour order as a set of numbers representing nodes

OUTPUT:

- The total path cost of the given tour

Helper function that calculates the total path cost of the given tour

- Takes the current tour order and traverses it, keeping track of the path costs
- Returns the total path cost of the tour after finishing

IV. References

[1] Liang, F. (2020, April 20). *Optimization techniques — Simulated Annealing*. Towards Data Science. Retrieved November 21, 2022, from https://towardsdatascience.com/optimization-techniques-simulated-annealing-d6a4785a1de7

[2] Meghanathan, N. (2020, April 15). *Nearest Neighbor Heuristic for Traveling Salesman Problem*. Retrieved November 19, 2022, from https://www.youtube.com/watch?v=SiZ7eLVn7BI

[3] Wikimedia Foundation. (2021, February 28). *Nearest neighbour algorithm*. Wikipedia. Retrieved November 19, 2022, from https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

[4] McCaffrey12/01/2021, J. (n.d.). *Simulated annealing optimization using C# or python*. Visual Studio Magazine. Retrieved November 20, 2022, from https://visualstudiomagazine.com/articles/2021/12/01/traveling-salesman.aspx