Homework 2
– Solving TSP using Search Algorithms

CMSC 421 Spring 2024

Due Date: 11:59 PM Sept 30th, 2024

Welcome to your first programming assignment. In this assignment you are going to explore algorithms to solve the Traveling Salesman Problem (TSP). The goal of this assignment is for you to explore various mathematical and algorithmic trade-offs. So you are encouraged to collaborate, discuss and compare results with your classmates. However, the final code and lab report submitted should be your own work.

Note: This is an open-ended assignment with no clear correct answers. The idea is to encourage you to infer results empirically, i.e., conducting several experiments to solve the TSP problem and drawing your own conclusions on the outcomes.

Guidelines:

- Pick the programming language you intend to use to conduct the assignment. This choice is up to you, so pick the one you either are strongest with, most comfortable with, most interested in learning, etc. The corollary of this is that the TAs and instructor will not be providing coding advice, debugging, etc. You are welcome to use Piazza and each other to debug matters.
- Familiarize yourself with the AI:MA code repository and make use of it as you see fit. Feel free to adapt/copy the code from this repository or any other online resources with proper citation. Hints, tips and findings from the code repository you are encouraged to share on Piazza.

What will you be submitting:

1. **Code:** You will submit your implementation of **7 algorithms**: *NN*, *NN2O*, *RNN*, *A_MST*, *hillClimbing*, *simuAnnealing*, and *genetic*. These functions should read in an $N * N$ adjacency matrix that defines an undirected N-node graph with up to N^2 edges, plus any relevant parameters. The functions should output the relevant performance metrics: best cost, number of nodes expanded, CPU runtime, and real runtime.
    - To enable grader testing, you will also implement a **main() function** which reads from the command line an adjacency matrix in the following format: `a.out < infile.txt`, where the `infile.txt` looks like Figure 1: Format of infile.txt. Take the TSP problem in Figure 2: TSP Graph as an example, Figure 3: Example of infile.txt is its corresponding infile matrix. As graphs will be undirected, matrices will be symmetric. For consistency, the full matrix will be provided although all that is needed is the "upper triangle". Write your outputs to csv files, where the name of the file is the function name. For example, for the

hill-climbing algorithm, the output file should be hillClimbing.csv. The file should contain 4 entries in one line: total cost of the best solution, number of nodes expanded, CPU run time, and real-world run time. You must also include a **written explanation of how to run your code from the command line** in your report.

- You may need to write additional helper functions to perform the experiments, such as code to generate random adjacency matrixes of different sizes, and code to generate the graphs for the discussion. You do not need to submit this code.

```
n     /* integer # of rows/columns , all matrices will be square */
M11, M12, ..., M1n  /* each row of the matrix separated by a new line */
M21, M22, ..., M2n   /* you can assume these are integers */
...
Mn1, Mn2, ... ,Mnn
/* end of file */
```

Figure 1: Format of infile.txt

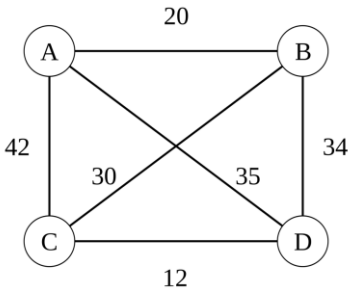
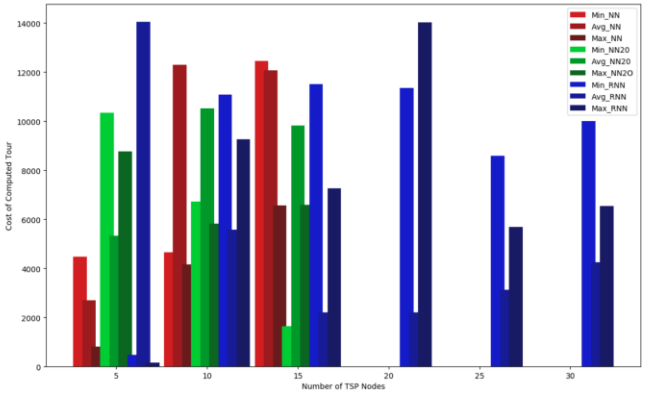


Figure 2: TSP Graph

```
4
0, 20, 42, 35
20, 0, 30, 34
42, 30, 0, 12
35, 34, 12, 0
```

Figure 3: Example of infile.txt

2. **Report:** You will perform experiments and conduct some empirical experimentation to explore various mathematical and algorithmic tradeoffs. Your write-up will include graphs/plots and written discussion. Refer to the figures below (Figure 4: Example Plots) for an intuition as to how the graphs/plots need to look. Feel free to get creative about how you want to represent data, and what can give you insights into how different solutions to TSP work. The graphs above are purely for representational purposes to give you an idea about what output is needed. Your graphs may look different.



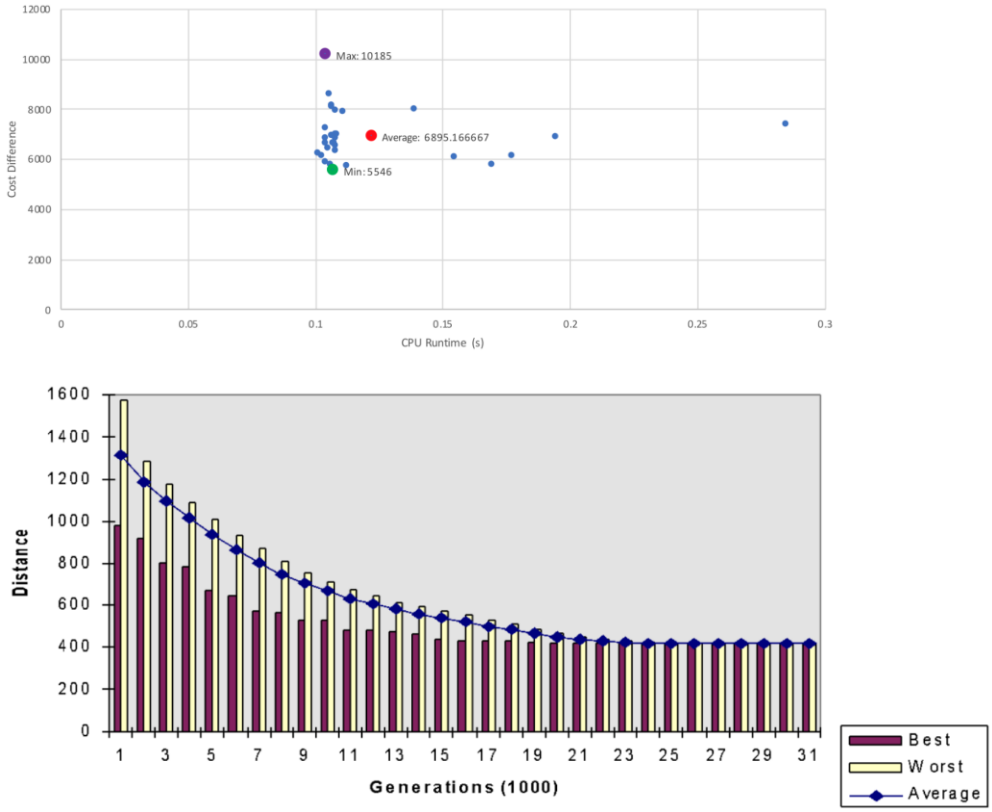


Figure 4: Example Plots

3. **Screen-recording of your code running**: While graders reserve the right to test your code, we do not have time to test every person in the class. Please submit a screen recording of you running your code from the command line and the resulting creation of .csv files to demonstrate that your code runs.

Overall, you will submit (via GRADESCOPE) your code (the *NN*, *NN2O*, *RNN*, *A_MST*, *hillClimbing*, *simuAnnealing*, and *genetic* algorithms + a main() function to run from the command line), a written report with plots/figures, and a screen-recording of your code running.

Grading rubrics:

- Presentation of the results in graphical form. Several experiments are defined for each of which there is a plot or chart showing tradeoffs in solution quality and accuracy and measures of time/space used to reach that solution. Are these graphs illustrative of the phenomena expected?
- Discussion of the experiments. For each of the experiments a short narrative explaining the phenomena observed will be submitted. Does this narrative accurately describe the mathematical and algorithmic phenomena behind the observations?

1 Part 1: Solve TSP Using Search Heuristics

You will utilize 3 different search heuristics to solve the TSP as follows:

1. **Nearest Neighbors** (*NN*): Here, you will keep expanding the closest node till the goal is reached.

2. **Nearest Neighbors with 2-Opt** (*NN2O*): Here, you will start with the NN solution, and iteratively optimize the graph using the 2-opt algorithm.

3. **Repeated Randomized Nearest Neighbors with 2-Opt** (*RNN*): This is an extension to the NN2O approach, with two additions - 1) instead of choosing the closest node, we randomly select one among the 'n' nearest nodes, and 2) repeat the procedure with different starting nodes and select the best one.

For the experiment:

1. Randomly create a family of 30 TSP graphs/matrices for each size 5, 10, 15, 20, ...
2. Run the above algorithms on each of them.
3. For RNN, experiment with different values of 'n', and report the one that gives the best value.
4. For each family you'll compute the AVERAGE/MIN/MAX of total cost, number of nodes, CPU and real-world runtime.
5. Use those data to plot 4 graphs – one each for total cost, number of nodes, CPU runtime, and real-world runtime. The x-axis of the plot is the size of the graphs/matrices (size 5, 10, 15, ...). The y-axis is the performance metric: total cost and number of nodes, and CPU and real-world runtime respectively. Use different colors, shapes, etc. to represent each search algorithm.

Compare and discuss the results in a short paragraph. Questions you can explore are for example:

- Which algorithm provides solution with the lowest cost? What's the difference between their best solutions, and how does that change when the size of the graph increases?
- Is there a difference between CPU and real-world runtime?
- Does changing the 'n' value in RNN change the outcome? Why do you think this might be?

What to submit:

Code: Your implementation of three functions: `NN()`, `NN2O()` and `RNN()`. Each function should first read in the graph/matrix from `infile.txt`. Then perform the algorithm and finally return the cost of the best solution.

Report: Two graph plots of your experiment results and your discussion.

2 Part 2: Solve TSP with A* and MST Heuristic

The traveling salesperson problem (TSP) can be solved using the minimum-spanning-tree (MST) heuristic, which estimates the cost of completing a tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link costs of any tree that connects all the cities.

Note: There are several algorithms for finding the minimum spanning tree of a graph. You are expected to perform your own literature study of these and use any of the algorithms for this purpose.

1. Define the *initial state*, *goal state*, *successor function* and *edge cost* ($g(n)$ for using A*).
2. Compute the heuristic $h(n)$ utilizing the MST value.
3. Implement the A* algorithm.
4. Compare the result of A* with MST with results from Part 1.
   For the experiment:
   a) Take the same family of 30 TSP graphs/matrices for each size 5, 10, 15, 20, ... from Part 1.
   b) Run A* with MST on each of these family of size 5, of size 10, etc. Please note that A_MST will likely slow down on larger adjacency graphs. If your A_MST implementation will not run on large graphs, you can instead compare over smaller intervals (ex: 30 graphs each of size 5, 6, 7, 8, 9, 10).
   c) For each graph/matrix compute the difference between the total cost from each part 1 algorithm and the optimal cost from A* with MST. Then for each family of 30 graphs/matrices you'll compute the AVERAGE/MIN/MAX of those differences.
   d) Plot a graph that displays the difference of performance in terms of total cost. The x-axis is still the size of the graphs/matrices, while the y-axis is the AVERAGE/MIN/MAX of differences of best solution's costs between algorithms in part 1 to A* with MST. Use different colors, shapes, etc. to distinguish searching algorithms from Part 1.
   e) Repeat step 3 and 4 for total number of nodes expanded.

What to submit:

Code: Your implementation of the function: `A_MST()`.

Report: Two graph plots and your discussion in two short paragraphs – one for cost, one for nodes expanded.

3 Part 3: Explore Local Search Algorithms

In this question, we explore the use of local search methods to approach TSPs.

1. Implement and test a **hill-climbing** method to approach TSPs.
2. Implement and test a **simulated annealing** method to approach TSPs.
3. Implement and test a **genetic algorithm** method to approach TSPs. (see section 4.3 of Larranaga et al. 1999)

Compare these results with each other and with the optimal solutions obtained from Part 1 and Part 2. What relationship do you find between each of these results, in terms of similarity and differences? Which algorithms seem to work better on larger TSPs and which work best on smaller ones? Compare not just the quality of the results but the time it takes to obtain them.

For the experiment:

1. Randomly create a family of 30 TSP graphs/matrices (or you can experiment with the size).
2. Run the A∗ algorithm with MST on those 30 TSP graphs to get the optimal solutions.
3. Run the above algorithms on those graphs.
4. Try different assignments to parameters like:
    - number of restarts for hill-climbing
    - number of restarts, initial temperature, cooling ratio α (you can assume the cooling function is $g(T) = \alpha * T$) for simulated annealing
    - number of generations, selection approach, probability of crossover, length of crossover, and mutation rate for genetic algorithm
5. Fix your parameters. Then for each graph/matrix you'll compute the total cost, number of nodes, CPU and real-world runtime.
6. Use those data to plot three graphs – one for each searching algorithm. For each graph, the x-axis is the CPU runtime, while the y-axis is the difference of total cost between the local searching algorithm and the optimal from A* with MST. In the case that using your CPU runtime produces bad graphs, you may use real runtime (please include a comment in the discussion about why you believe real runtime was a better metric in this case).
7. Maybe repeat the experiment a couple times for different sizes of TSP graphs.

What to submit:

Code: Your implementation of three functions: `hillClimbing()`, `simuAnnealing()` and `genetic()`.

Report: At least three graph plots of your experiment results and a short paragraph of your discussion.

4 Extra Credit

The TSP problem can be generalized as a Vehicle Routing Problem (VRP), where given a set of vehicles and locations, the task is to find the minimum cost path for vehicles to reach every location. This problem, just like the TSP, is NP-hard. How would you apply the algorithms you have used to solve TSP to find a solution to the VRP problem?

5 Submission Instructions

Submit a single zip file of the code to Gradescope by September 30th. This should include your implementations of the following functions:

- NN()
- NN2O()
- RNN()
- A_MST()
- hillClimbing()
- simuAnnealing()
- genetic()
- It should also include any additional code necessary for running these functions.

Submit your report and screen recording to Gradescope as well. **Do not forget to include instructions for running your code** in the report. If you are using edited versions of the AIMA code, please upload the changed files. If the code is tested, assume that the user has followed the official Instilation Instructions in the readme.