# CMSC 421 Project 1: Informed Search Algorithms for Traveling Salesman Problems (1/29)

## 1 Introduction

In your first programming assignment, you will use 7 different algorithms to solve the Traveling Salesman Problem (TSP). This is a challenging assignment that asks you to implement the algorithms and understand them, so that you can discuss the trade-offs, usability, and hyperparameters of each algorithm. To do this, you are encouraged to discuss the project with your classmates or TAs and to use whichever programming language you are comfortable in.

Be mindful that this is an open-ended project without correct answers. You are encouraged to experiment with the algorithms and infer your own results. A* is the only algorithm that is theoretically guaranteed to find the correct solution, however, even a perfect implementation may struggle with problems with more than 10 cities.

### 1.1 Required Submission

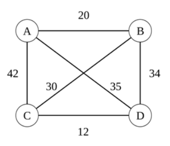
You will submit 3 items when you have finished the project:

1. Written report (`.pdf` **only!**): You will submit a written report that includes your findings and your answers to the questions in each section. Your report should be well written and include clear plots, like those shown in Figure 1. The figures can be made in any software of your choice, including Excel (by saving the results as a CSV and loading it), matplotlib, matlab, or plotly. On Gradescope, you can submit either one `.pdf` multiple times or multiple `.pdf`s.
2. Code file(s) (`.py`, `.java`, etc. – **NOT A** `.zip`): The other essential part of your project is the written code. You should submit all of the code you have written in separate files for each part of the project. The code must be runnable by the TAs and be your own work, but can be in any language you choose (so please include compile/environment setup instructions). **Each code file should be able to be run on a file provided as a command line argument (`sys.argv` in Python, `String args[]` in Java, etc.). The file format is discussed in Section 1.2. The file should either print out the runtime, CPU time, and cost or save it to a file**. Please limit the functions you use to the standard library, numpy, plotting libraries, scipy, and the AIMA repository.
3. Screen recording (short `.mp4` **only!**): A screen recording of you running each function once is required to help the TAs grade your assignment. From the command line, run each of the 7 algorithms on an adjacency matrix and show the output. You can use `Snipping Tools` on Windows or `CMD-Shift-5` on Mac, then convert to `.mp4`.

### 1.2 Input Files

The traveling salesman problems we provide are adjacency matrices, with rows delimitated by new lines and columns delimitated by spaces. This file format can be loaded by `mat = np.loadtxt(fname)`.

10 random adjacency matrices for sizes $5, 10 \ldots 25, 30$ are provided, which should be used in the experiments. Smaller matrices are provided for your A* experiments. Additionally, adjacency matrices with cities evenly distributed on the edge of a unit-side-length $n$-gon will be provided. These are to test your algorithm since you know that the shortest path is equal to $n$, **but should not be used for the experiments**. Download the matrices on Piazza



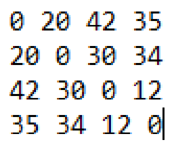

```
0 20 42 35
20 0 30 34
42 30 0 12
35 34 12 0
```

Figure 1: An example TSP problem and the corresponding adjacency matrix.

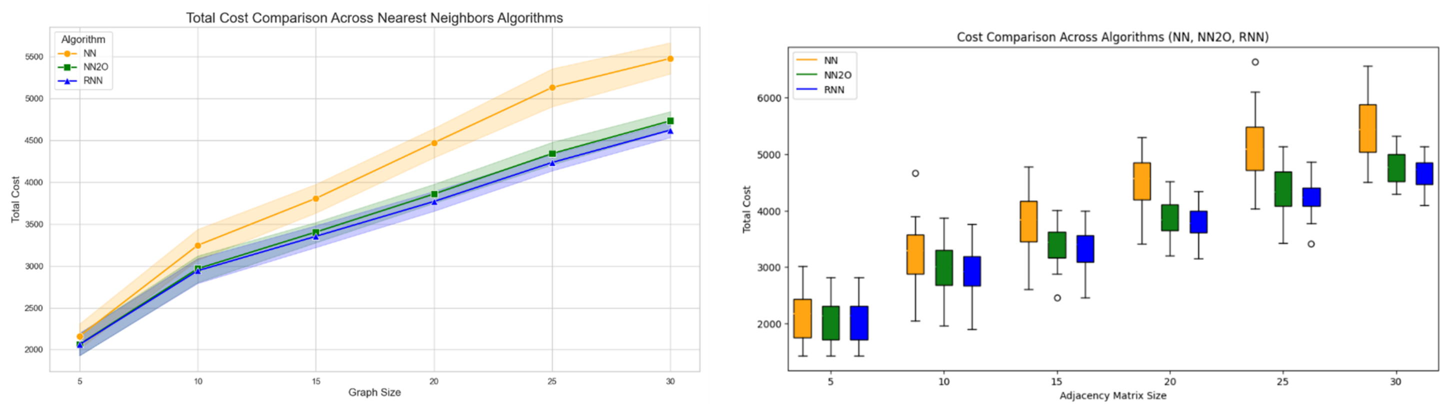


Figure 2: Examples of top-tier figures for your inspiration. When creating your figures, remember to include axis labels, a descriptive title and a legend. Use colors or shapes to differentiate different algorithms.

## 2 Part I: Nearest Neighbor, Nearest Neighbor 2-Opt, Repeated Randomness Nearest Neighbor

In this section, you will implement the three easiest algorithms. These are greedy algorithms that should run quickly for well over 50 cities, but, as you will see below, will generate less-optimal solutions than A*.

### 2.1 Algorithm Setup

You must submit (at least) three functions, described here:

- Nearest neighbor: This algorithm is the most simple. It should start with an arbitrary node, choose the next closest unvisited node until all are visited, then returning to the initial node.
- Nearest neighbor with 2-Opt: Starting with your nearest neighbor algorithm, add the 2-opt algorithm to refine your solution. This repeatedly tests all swaps of adjacent cities to see if the swap improves the cost. When a swap that improves the solution is found, perform the swap and restart by looking for new swaps from the beginning of the route. When all pairs of cities do not result in a beneficial swap, return the route.
- Repeated random nearest neighbor (RRNN): Write a similar algorithm to nearest neighbor with 2-opt with two differences: (1) instead of automatically choosing the nearest neighbor, randomly choose the next city from the $k$ closest cities and (2) repeat this process *num_repeats* times, choosing the best solution. Therefore, add the hyperparameters `k` and `num_repeats` to your function.

### 2.2 Experiment Setup

To test your algorithms, use the 10 random adjacency matrices of sizes $5, 10 \dots 25, 30$:

1. First, use these matrices to find optimal hyperparameters for your RRNN algorithm. Create two plots, one with different values of `k` on the $X$ axis and the median cost on the $Y$ axis, and another with different values of `num_repeats` on the $X$ axis and the median cost on the $Y$ axis. The cost should be the sum of the distances from the adjacency matrices of the route taken. Make sure to keep one parameter constant as you test the other parameter.
2. Then, compare the three algorithms to each other. To do this, run each algorithm on each of the matrices, finding the median real runtime required, median CPU time required, and median cost to traverse the cities. You can use `process_time_ns` and `time_ns` to track the time in Python. If the CPU time is zero, run the algorithm multiple times for each matrix before stopping the time and that value divided by the times ran.
3. Finally, use these results to create 3 plots, one for total time, one for CPU time, and one for the solutions' cost. For the first plot, put each algorithms' runtime on the $Y$ axis and the number of cities on the $X$ axis. Each tick on the $X$ axis should show size of the problem from $5, 10 \dots 25, 30$. On the second plot, put the algorithm's CPU time on the $Y$ axis. For the third plot, put each algorithms' cost on the $Y$ axis. Remember to label and title your figure, using color or shapes to distinguish between the algorithms.

### 2.3 Questions to Consider in Your Report

Please answer these question in the report, alongside your figures:

- Do the algorithms here find optimal paths to solve the problem? Why or why not?
- What values did you choose for `k`? What are the tradeoffs for lower or higher values?
- What values did you choose for `num_repeats`? What are the tradeoffs for lower or higher values?

- Which algorithm found a solution the quickest? Was this at the cost of solution accuracy?
- Using $n$, $k$, and *num_repeats*, what are the time complexities of the three algorithms? Do the time complexities effect the figures you generated?

## 3 Part II: A* with MST Heuristic

In this section, you will implement the most challenging algorithm, A*, using the MST heuristic. This algorithm will run much slower than the other algorithms but is guaranteed to find an optimal solution.

### 3.1 Algorithm Setup

Submit two functions, described here:

- Minimum-spanning tree heuristic: Using an algorithm like Prim's algorithm or Kruskal's algorithm, implement a function that calculates a heuristic cost for each city by summing the edges of a minimum spanning tree. The function should return the total cost of the minimum-spanning tree starting from the given city and connecting to every unvisited city. **Psuedocode for implementing Prim's Algorithm is available online, or you can use the scipy function.**
- A*: Implement the A* function, where $g(x)$ is the cost to get to each city and $h(x)$ is the minimum-spanning tree from the previous function. The function should input a matrix and return a complete route, beginning and ending at the same city. This is the hardest part of the project, but there are many resources that can assist you online, including AIMA. Try to optimize your algorithm, as it should run for matrices of $10-15$ cities. When writing the algorithm, consider that the current 'state' each iteration is the partial 'tour', and your success function tries to add another city to the 'tour' based on the cost to reach that city and the heuristic cost of completing the 'tour' from that city.

### 3.2 Experiment Setup

To test your algorithms, run A* on the matrices you are able to:

1. Start by recreating the same 3 plots comparing nearest neighbor, nearest neighbor with 2-opt, and RRNN. To compare the algorithms to A*, divide each value on the $Y$ axis by the corresponding value for A*. For example, for one figure, plot the CPU time for each algorithm divided by the CPU time for A* on the $Y$ axis and the size of the matrix on the $X$ axis. If the A* algorithm took 1 second for graphs of $n = 10$, 2 for $n = 20$, and 3 for $n = 30$, and RRNN took 0.5 seconds for $n = 10$, 1.5 for $n = 20$, and 3 for $n = 30$, you would plot $0.5,\ 0.75,\ 1.0$ for RRNN.
2. Then, on **an additional plot**, plot the median number of nodes expanded by A* on the $Y$ axis for each matrix size. A node is expanded when the heuristic is calculated on it and the number of nodes expanded should increase quicker than the number of cities.

### 3.3 Questions to Consider in Your Report

Answer these questions alongside providing the figures in your report:

- How are solutions represented by your version of A*? What does each node in the search space represent? What is a frontier node? Describe your evaluation function and stopping condition.
- What number of cities were you unable to run? What part of the algorithm, do you think, increased the time the most? How could you improve the algorithm?
- Discuss how the number of nodes expanded increases as the number of cities increases. Discuss how this relates to the previous question.
- How does A* compare to the other algorithms, in both time and solution cost? Is A* a practical algorithm for real world use?

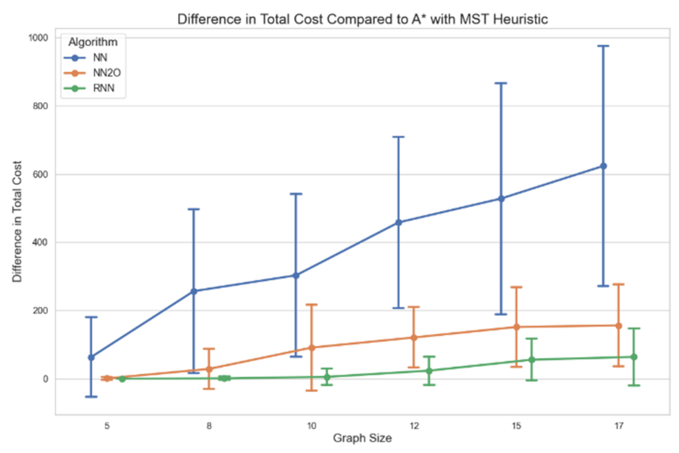
## 4 Part III: Hill Climbing, Simulated Annealing, Genetic Algorithms

In this section, you will implemented Hill Climbing, Simulated Annealing, and a Genetic Algorithm. Hill climbing, simulated annealing, and genetic algorithms are surprisingly strong algorithms despite their simplicity.

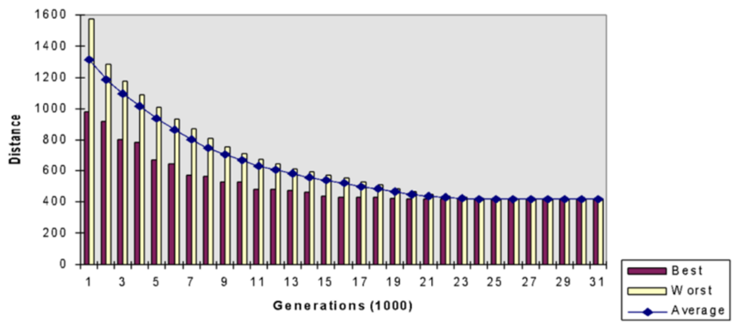
### 4.1 Algorithm Setup

You will write (at least) three functions:

- Hill climbing: Create a randomly-restarting hill climbing function. For *num_restarts*, start by generating a random solution to the problem. Then, generate random "neighboring" solutions by choosing two nodes to swap, and, if the solution improves the results, replace the source solution with it. After *num_restarts*, return the best solution found. Your function should take in both the adjacency matrix and `num_repeats`.



(a) This figure is an example of a plot that shows how the nearest neighbor-based algorithms or the Section 4 algorithms compared with A*, which can be accomplished by dividing the cost of each algorithm by A*'s respective cost. Your plot showing comparing your algorithms to A* stop be similar.



(b) This figure is an example of a plot that shows how the cost of the genetic algorithm's best solution decreases as the number of generations increases. Your plot showing your genetic algorithm in Section 4 should look similar.

- Simulated annealing: Your simulated annealing implementation should be similar to the implementation of hill climbing. Again, starting with a random solution and find a "neighboring" solution. Then, choose to replace the original solution with the new solution if it is less costly or with the probability

$$e^{\frac{score' - score}{t}}$$

  where $t$ is the temperature. Your algorithm should take in the adjacency matrix, `alpha` (how much the temperature decreases when a route is accepted), `initial_temperature`, and `max_iterations`.
- Genetic algorithm: the genetic algorithm starts with a population of randomly generated solutions, and, each iteration, pairs of "parents" are chosen to create "child" solutions. This crossover can be accomplished using any of the techniques in section 4.3 of Larrañaga et al., and the child should be mutated (cities randomly swapped) with *mutation_chance* probability. After all children are created, the combined list of children and parents should be sorted by score and the worse solutions should be discarded, which is called "elitism." The algorithm should take the adjacency matrix, `mutation_chance`, `population_size`, and `num_generations`.

## 4.2 Experiment Setup

1. First, create **at least one figure for each algorithm**, highlighting how **at least one** hyperparameter change the algorithms' solutions' cost.
2. Additionally, create **at least one figure for each algorithm** that shows how the cost of each algorithm improves over each generation or iteration. Do this by calculating the best cost that the algorithm has found on each iteration and plotting it on the $Y$ axis. Then, plot the generation or iteration on the $X$.
3. Then, like the figures previously created for Part II, create three figures that highlight the difference in performance between your hill climbing, simulated annealing, and genetic algorithms. These figures should show the runtime, CPU time, and cost divided by A*'s performance on the previous matrices.

## 4.3 Questions to Consider in Your Report

Please answer these question in the report, alongside your figures:

- How is the search state space represented differently in these local algorithms than nearest neighbor or A*? What does it mean to be a neighbor to one of these algorithm's solutions?
- What hyperparameters were the best for each algorithm? Discuss the tradeoffs between different values for at least one hyperparameter per algorithm.
- Compare the performance of these algorithms with the performance of the nearest neighbor algorithms. What makes these algorithms better or worse choices than the other algorithms?
- How does the performance of these algorithms compare with A*? When might you want to use A* over one of these algorithms? When might you use these algorithms over A*?
- How do these algorithms harness randomness to improve on deterministic algorithms like nearest neighbor or A*?

- Discuss four different real-world applications of the traveling salesman problem. Which algorithm would you chose (from all of the algorithms discussed) for each scenario? Try to choose setups and scenarios where different elements (time, cost, etc) are more relevant.

## 5 Part IV: Extra Credit

To participate in the extra-credit, generate a path for the provided `extra_credit.txt`, which is a really large matrix. Then, submit the path **and** add a paragraph to your report on the modifications made/algorithms run to find this solution. We will calculate the points you receive as

$$1/e^x * 10$$

where $x$ is the percentile ($0 \to 1$) of your solution compared to the other solutions. Even the worst solution will receive points under this model, so it makes sense to submit something even if you cannot run your advanced algorithms.

## 6 Figure Checklist

1. Two plots showing the effects of different hyperparameters on RRNN
2. Three figures showing the CPU time, runtime, and cost for NN, NN-2opt, and RRNN for various numbers of cities
3. Three figures, the same as above, but with the CPU time, runtime, and cost divided by A*'s respective statistics
4. One figure showing how the number of nodes expanded increases as the number of cities increases for A*
5. Three figures highlighting how different hyperparameters change hill climbing, simulated annealing, and the genetic algorithms
6. Three figures showing how the cost of the genetic algorithm's, hill-climbing's, and simulated annealing's solutions decreases over each iteration
7. Three figures, the same as above, but with the CPU time, runtime, and cost of **hill climbing, simulated annealing, and the genetic algorithm** divided by A*'s respective statistics

All figures should show the median or mean over multiple trials for reproducibility. Additionally, please make sure that you answer all of the questions provided, and discuss how you wrote the algorithms and ran the experiments.