Name: Daniel Deutsch

Folder Contents:

- **tsp.py -** Implementation of TSPProblem class and A* search, along with “zero” heuristic, randomEdges heuristic, and cheapestEdges heuristic. The file is designed to be run from command line in the form Python3 tsp.py < infile.txt. **A.uniformCost(), A.randomEdge(), A.cheapestEdge() implementation is in this file.**
- **experiment.py -** Sets up and runs part 1 of the experiment, producing the graphs and writing data to respective .csv files for each Alg. The program randomly generates matrices and passes them to tsp.py.
- **tsp2.py -** Building on the TSPProblem class with TSPProblemLocalSearch class. This file also contains the mst_heuristic, along with hill climbing, simulated annealing, and genetic algorithm implementations. It runs in the same way tsp.py. **hillClimbing(), simuAnnealing(), and geneticAlgorithm() are in this file.**
- **experiment2.py -** Sets up and runs part 2 of the experiment. For part 2, MST was taking far too long to calculate solutions for large graphs, so the setup includes randomly generating matrices with a known optimal cost path, and running the algorithms on these matrices to compare results to actual.
- ***.png -** graphs - runtime and cost_nodes are for part 1 - all others for part 2
- ***.csv -** data about the experiments - not really needed I will summarize the results in my report

Part 1:

Implementation Notes:

My implementation for cheapestEdge is just going to the cheapest neighbor in each state and returning the cost of the final path. I found that using the MST implementation was too time-consuming regardless of the many ways I tried it (there were a few). Because of this, the heuristic is not admissible; it is still, however, guaranteed to be closer to the actual value than cheapestEdge (i.e. less than because both are too expensive).

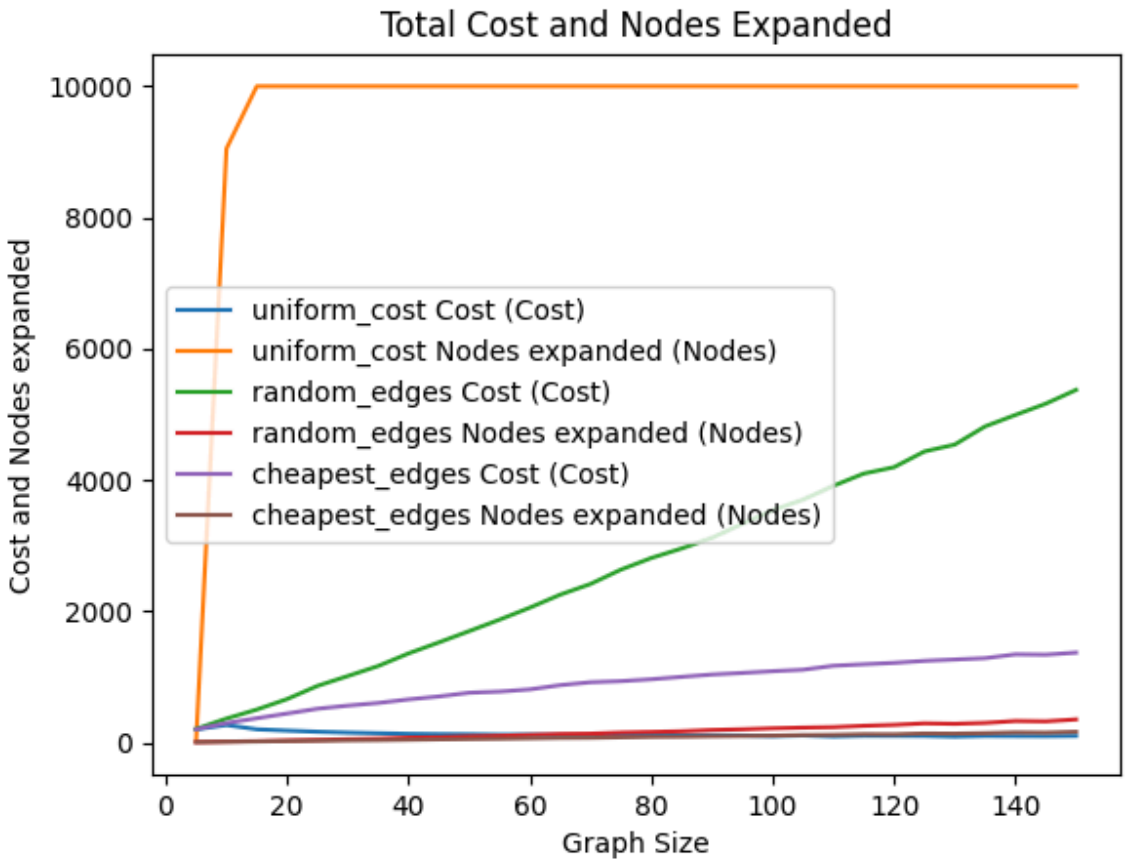
RandomEdges works by just grabbing a random remaining neighbor and going there, and calculating the cost of that path.

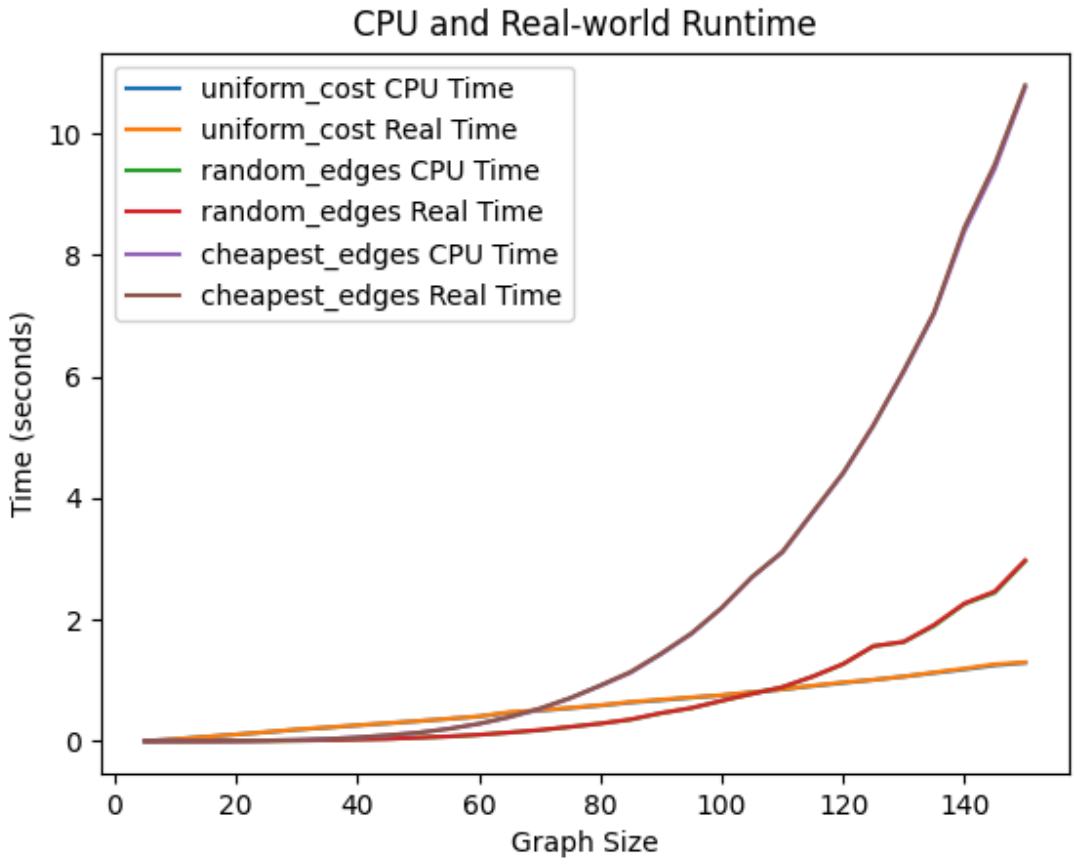
I looked pretty closely at the code provided in the repository when trying to do my implementation, but ultimately found that it left too much that I didn’t really need and wanted to build my own.

I began by building tsp.py and running it on random input of various sizes. Once this was working, I made experiment.py to build and run many examples using tsp.py. I liked this implementation a lot and wanted to closely mimic it in part two.

I had to cap the maximum nodes expanded in order to prevent uniform_cost from running forever and preventing the rest of the experiment from submitting, leading to the line for nodes expanded by uniform cost to plateau at 10000. The low costs are incorrect, as they represent the cost of the best path at this point, but this path does not visit all nodes.

Graphs:


Total Cost and Nodes Expanded
Cost and Nodes expanded
Graph Size
0
2000
4000
6000
8000
10000
0
20
40
60
80
100
120
140
uniform_cost Cost (Cost)
uniform_cost Nodes expanded (Nodes)
random_edges Cost (Cost)
random_edges Nodes expanded (Nodes)
cheapest_edges Cost (Cost)
cheapest_edges Nodes expanded (Nodes)



Graphs are also included in the file
There are also csv files with the data dumped from each algorithm.

Which algorithm provides the solution with the lowest cost? What's the difference of their best solutions, and how does that change when the size of the graph increases?

- The cheapestEdges heuristic provides the lowest cost solution for all sizes of graphs. This is because both this and randomEdges are inadmissible heuristics, meaning they both overestimate the true solution, but cheapestEdges is closer and more informative than randomEdges leading to better performance. Since each search terminates once the goal is found, cheapestEdges finds the lower cost solutions and does so in less time for all sizes. At the largest size, randomEdges average cost was around 5400, around 4000 more than the costs scored by cheapestEdges. UCS - proved unable to solve past n=20, and proved worse than the counterparts below this.

Which algorithm has the least runtime and how do their runtimes change with the size of the graph increases?

- The heuristic that runs the fastest is randomEdges, although UCS proves considerably worse than both other alternatives. This is to be expected, as the performance of A* search is directly correlated to how informed the heuristic is.

Is there a difference between GPU and real-world runtime?

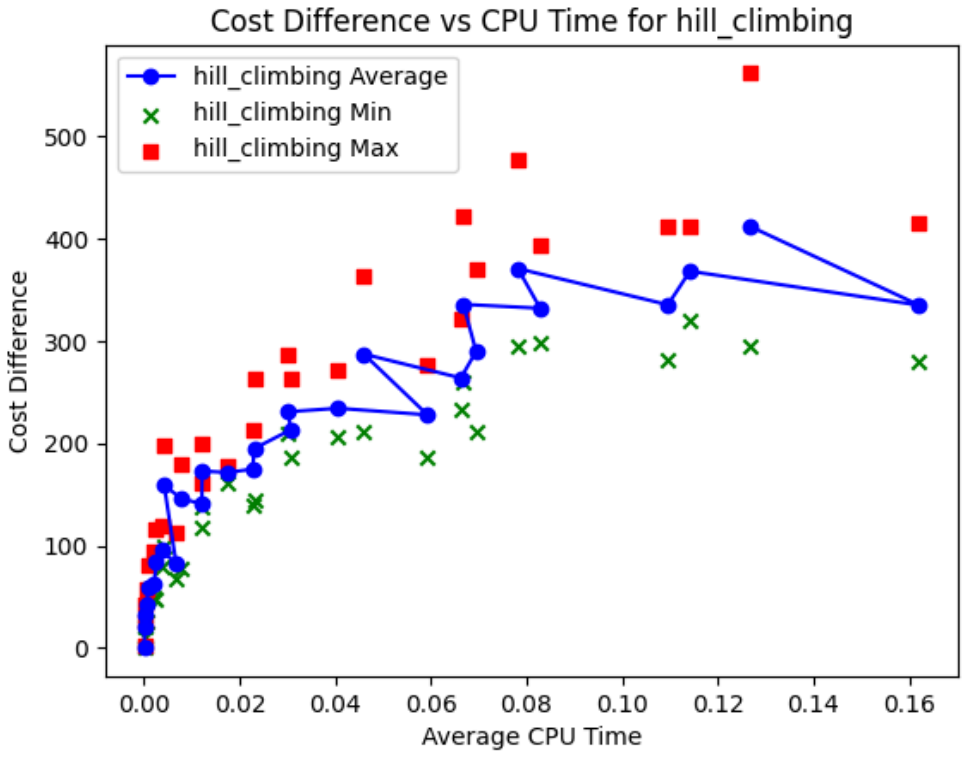
- In this case there was virtually no difference between the two run times. But depending on the implementation there would be a difference. If someone tried to implement it in a way that caused the CPU to rest more, or if they ran it on a machine with many other processes that had higher priority, there might be a bigger difference.

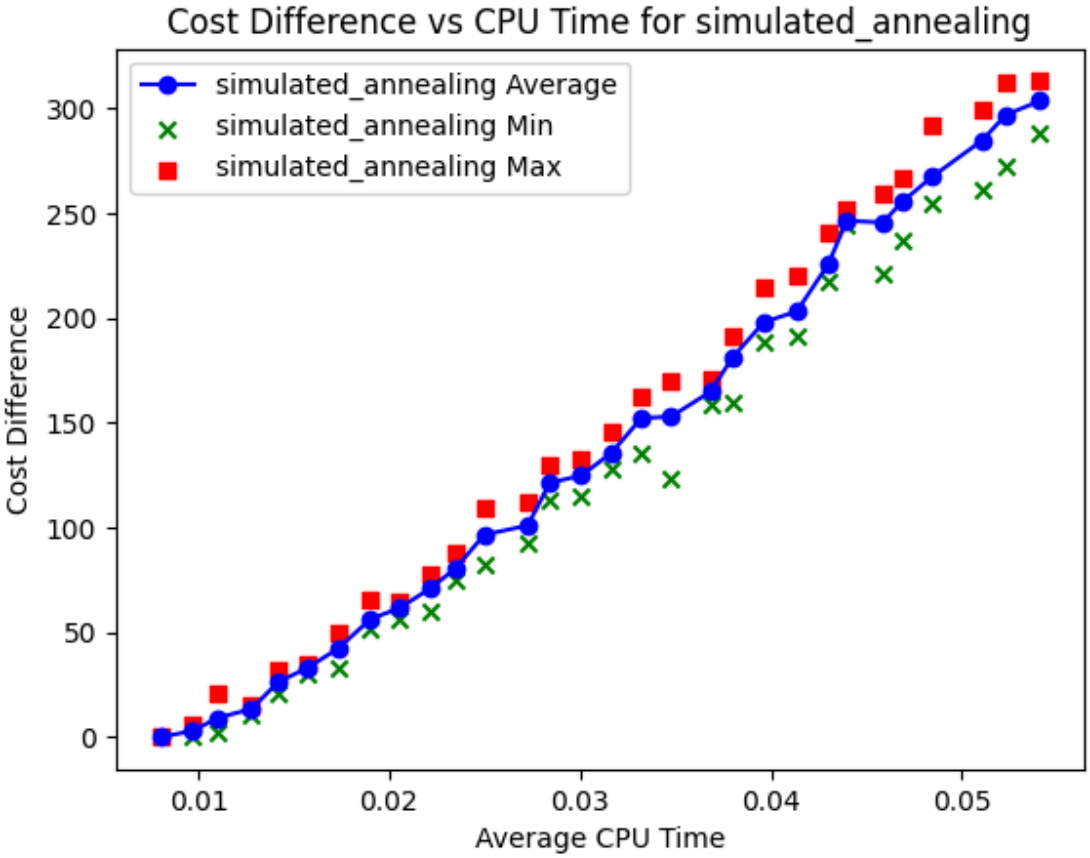
Part 2:

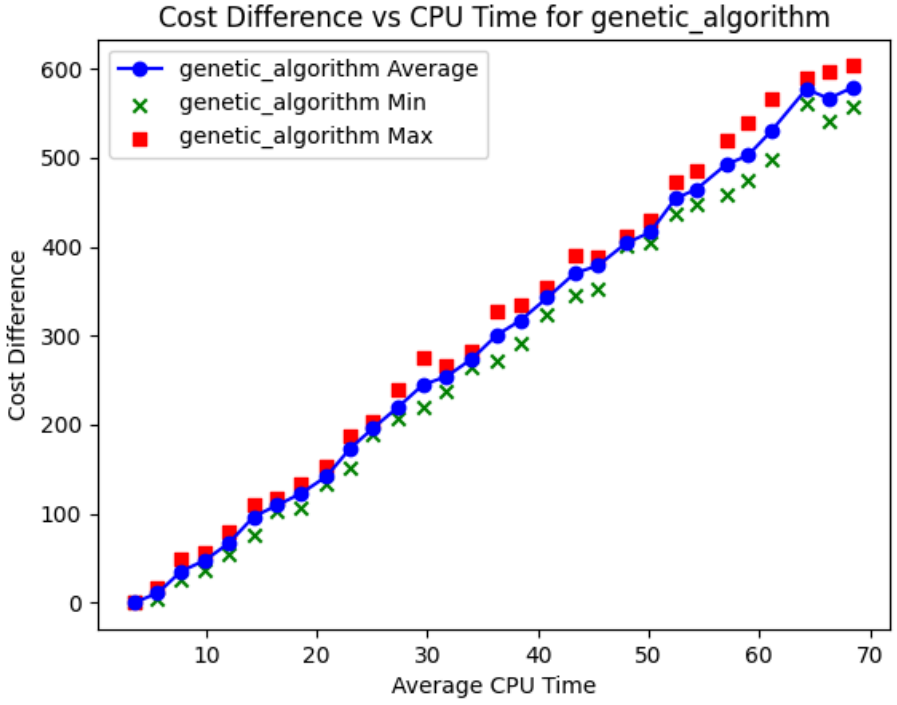
The main problem I had with this part was getting A* to run with MST heuristic on graphs that were pretty large, something I never ultimately figured out how to do. My solution was to randomly generate matrices with a known actual cost, by making sure a path existed in each graph that was size^2 length and no paths that were generated could be shorter. This worked well enough for me to collect data about differences between actual and correct results.

The hill climbing graph produced an odd shape, I can't really fix the full scope of the experiment because it takes three hours to complete. This was caused, I believe, by the addition of the restarts and the fact that the difference between local optima could be drastically different than the actual optimal causing the varying differences between actual and result and the odd shape.

Graphs




Cost Difference vs CPU Time for simulated_annealing
simulated_annealing Average
simulated_annealing Min
simulated_annealing Max
Cost Difference
Average CPU Time
0
50
100
150
200
250
300
0.01
0.02
0.03
0.04
0.05


Cost Difference vs CPU Time for genetic_algorithm
genetic_algorithm Average
genetic_algorithm Min
genetic_algorithm Max
Cost Difference
Average CPU Time
0
100
200
300
400
500
600
10
20
30
40
50
60
70

Compare these results with each other and with the optimal solutions obtained from the A∗ algorithm with the MST heuristic. Compare not just the quality of the results but the time it takes to obtain them. Give an assessment of the relationship between computation time and solution quality, i.e., if all I wanted was a 75% solution, which method is quickest? How about a 90% solution? Is there are general rule here?

- For smaller graphs - I think hill-climbing is best. It solves the matrices much quicker and with nearly the same results. Also the implementation I used allows for restarts to get multiple answers, I found that hill-climbing was by far the fastest, and really the only one that could handle these in reasonable time for large graphs.
- For larger graphs - I think simulated annealing is the best. The answers were much closer to the actual than either other option. I think if 75% was good enough then hill climbing might be better, considering it was still faster. But simulated annealing was fast enough that I think the tradeoff was worth it.
- I think genetic algorithms really struggled with the undirected graph because there were too many total paths for the larger values, that it took a long time and still couldn't really get close. An untested thought that I had is that these may perform better when the chromosome can be just 0 and 1 instead of range(n) options. As the graph shows, this algorithm took over 1000 times as long as the simulated annealing option, and still produced worse results. In hindsight, I could have added a better fitness test for selecting parents than what I had.