Project 2 Report

Option B (AI Code)

Vaughn Cox

Problem Formulation:

A*:
Starting with the A* algorithm, much of it was straightforward between the pseudocode and the UCS implementation from last project. Additionally, grid.py & node.py were also quite like the last project, with a few minor adjustments to node.py to account for calculating cost + heuristic. The addition of a heuristic function was the primary difference between UCS and A*. As suggested, I implemented the Manhattan distance heuristic. The function used was f(n) = g(n) + h(n). In this, g(n) is the accumulated path cost from the start node and h(n) is the Manhattan distance from the current node to the goal. Under these conditions, A* finds an optimal path. This part illustrates more of the difference between UCS and A*. However, the overall structure of the search remained quite similar.

TSP:

For the TSP, I approached with a local search using 2-opt hill climbing with random restarts, as discussed in class. It begins by generating city coordinates randomly within a range. A random tour is first made as the initial solution, and the cost is calculated with Euclidean distance. The core mechanic used, as discussed, is the 2-opt operator, which allows for the reversal of segments in the tour (essentially). The algorithm continues iterating through neighbors until no further improvements are possible, which is a sign of a local minima. This then involves the random restarts. For example, the algorithm was ran with 10 restarts for experimenting purposes, as outlined by the AI option B instructions.

Testing Summary:

During development, I tested individual components quite often to ensure correctness before integrating everything together. For A*, this included verifying that the Manhattan heuristic returned correct values, confirming that neighbor generation stayed within grid bounds, and checking that path reconstruction produced a valid sequence from start to goal. I also tested small grid sizes manually to confirm that the returned path cost matched expected values.

For the TSP portion, I tested the tour cost calculation independently to try to make sure the Euclidean distance computation was correct and that the tour properly returned to the starting city. The 2-opt neighbor generation was also tested separately to confirm that segments were being reversed correctly and that no cities were lost or duplicated during swaps. Small city counts were used at first to visually inspect improvements and verify that the algorithm reduced cost over iterations.

Experiments:

After verifying individual components, full experiments were executed using the required configurations. For A*, three grid sizes (10×10, 25×25, and 50×50) were tested across 10 different seeds, resulting in 30 total runs. For TSP, three problem sizes (20, 30, and 50 cities) were tested with 10 seeds and 10 random restarts per seed, resulting in 300 total runs. These experiments were executed using a separate analysis.py script to make analysis easier and to keep main.py clean. Below are the tables of the results. I used AI to organize the information.

A* Experimental Results

10×10 Grid (Start: [0,0], Goal: [9,9], Cost Range: 1-5)

Seed	Steps	Total Cost	Expanded States	Max Frontier Size	Runtime (ms)
0	18	33	81	129	0.31
1	18	35	93	101	0.29
2	18	29	62	92	0.18
3	18	28	72	126	0.22

Seed	Steps	Total Cost	Expanded States	Max Frontier Size	Runtime (ms)
4	18	32	90	97	0.29
5	18	32	87	108	0.28
6	18	30	86	108	0.27
7	18	39	99	99	0.96
8	20	32	94	110	0.26
9	18	34	97	110	0.31
Mean	18.2	32.4	86.1	108.0	0.34
Std Dev	0.6	3.1	10.5	11.7	0.21

25×25 Grid (Start: [0,0], Goal: [24,24], Cost Range: 1-5)

Seed	Steps	Total Cost	Expanded States	Max Frontier Size	Runtime (ms)
0	48	92	624	336	2.19
1	50	85	586	347	2.74
2	48	85	615	402	2.28
3	48	87	623	363	2.99

Seed	Steps	Total Cost	Expanded States	Max Frontier Size	Runtime (ms)
4	48	93	624	330	2.27
5	48	88	620	332	2.80
6	50	83	598	358	2.07
7	48	85	614	373	3.09
8	48	88	620	313	2.21
9	48	85	608	318	2.91
Mean	48.4	87.1	613.2	347.2	2.56
Std Dev	0.8	3.1	12.1	26.8	0.35

50×50 Grid (Start: [0,0], Goal: [49,49], Cost Range: 1-5)

Seed	Steps	Total Cost	Expanded States	Max Frontier Size	Runtime (ms)
0	98	173	2495	847	11.42
1	98	179	2496	779	10.41
2	100	169	2453	750	10.18
3	98	188	2499	735	11.53

Seed	Steps	Total Cost	Expanded States	Max Frontier Size	Runtime (ms)
4	98	174	2496	807	11.37
5	98	182	2497	737	11.29
6	98	169	2486	824	10.87
7	102	179	2495	783	11.41
8	98	175	2495	759	9.89
9	100	182	2499	820	10.03
Mean	98.8	177.0	2491.1	784.1	10.84
Std Dev	1.3	5.9	12.4	36.0	0.58

TSP Experimental Results

Summary Statistics by City Count

Metric	20 Cities	30 Cities	50 Cities
Best Cost - Mean	379.56	486.35	594.78
Best Cost - Min	297.28	406.01	467.99
Best Cost - Max	484.01	657.79	657.79

Metric	20 Cities	30 Cities	50 Cities
Best Cost - Std Dev	37.62	74.18	50.97
Initial Cost - Mean	1028.74	1544.70	2646.50
Initial Cost - Std Dev	118.52	172.62	179.20
Improvement (Initial-Best)	649.18	1058.35	2051.72
Improvement %	63.1%	68.5%	77.5%
Iterations - Mean	36.1	75.9	160.4
Iterations - Std Dev	6.6	23.2	17.8

Detailed Results by City Count and Seed

20 Cities (10 seeds × 10 restarts = 100 runs)

Seed	Best Cost (Min)	Best Cost (Mean)	Best Cost (Max)	Initial Cost (Mean)	Iterations (Mean)
0	297.28	362.25	413.06	997.44	36.2
1	345.91	371.48	484.01	1007.67	35.7
2	345.91	371.33	484.01	1007.22	36.9
3	345.91	375.19	430.99	1063.14	37.0
4	345.91	371.48	484.01	1007.67	35.7

Seed	Best Cost (Min)	Best Cost (Mean)	Best Cost (Max)	Initial Cost (Mean)	Iterations (Mean)
5	345.77	374.79	413.29	1002.66	36.8
6	345.91	370.38	484.01	1000.87	37.5
7	345.91	374.48	484.01	1013.27	36.6
8	345.91	372.99	484.01	1019.30	36.2
9	345.77	377.65	430.99	1011.98	35.8
Overall	297.28	379.56	484.01	1028.74	36.1

30 Cities (10 seeds × 10 restarts = 100 runs)

Seed	Best Cost (Min)	Best Cost (Mean)	Best Cost (Max)	Initial Cost (Mean)	Iterations (Mean)
0	406.01	457.68	529.82	1607.33	72.7
1	406.01	461.17	507.75	1594.66	75.3
2	430.15	464.52	529.82	1599.70	73.4
3	406.01	464.72	507.75	1594.70	75.1
4	406.01	460.18	507.75	1594.66	75.3
5	406.01	463.01	507.75	1594.66	75.3

Seed	Best Cost (Min)	Best Cost (Mean)	Best Cost (Max)	Initial Cost (Mean)	Iterations (Mean)
6	406.01	461.95	507.75	1594.66	76.1
7	406.01	465.07	507.75	1594.66	76.0
8	406.01	464.44	507.75	1594.66	75.9
9	394.96	462.76	507.75	1594.66	76.2
Overall	394.96	486.35	657.79	1544.70	75.9

50 Cities (10 seeds × 10 restarts = 100 runs)

Seed	Best Cost (Min)	Best Cost (Mean)	Best Cost (Max)	Initial Cost (Mean)	Iterations (Mean)
0	523.48	579.50	647.68	2652.78	158.1
1	552.65	593.60	657.79	2650.58	159.7
2	552.65	591.46	647.68	2650.58	156.7
3	552.65	594.44	652.48	2650.58	160.6
4	523.48	580.96	652.48	2650.58	157.9
5	523.48	584.80	652.48	2650.58	158.6
6	523.48	584.73	652.48	2650.58	160.4

Seed	Best Cost (Min)	Best Cost (Mean)	Best Cost (Max)	Initial Cost (Mean)	Iterations (Mean)
7	523.48	588.41	647.68	2650.58	160.8
8	523.48	587.74	647.68	2650.58	160.9
9	523.48	587.69	647.68	2650.58	160.4
Overall	467.99	594.78	657.79	2646.50	160.4

Final Analysis and AI Disclosure:

A* expands fewer states than Uniform Cost Search (UCS) because it uses heuristic guidance to focus the search toward the goal. It adds h(n) into the function, which obviously changes how the algorithm runs. A* expands nodes that reduce the remaining distance to the goal, while UCS expands nodes in all directions equally based solely on path cost so far. Although A* is more efficient than UCS in practice, it is still exponential in the worst case. In such cases, A* behaves similarly to UCS and may need to explore an exponential number of nodes relative to the depth of the solution.

As for exact search being infeasible for TSP, this is due to the size of the problem. Solving TSP optimally for moderate or large *n* is computationally infeasible, which motivates the use of heuristic or approximation methods such as local search. There is simply too many combinations for exact search to be convenient in any way.

Local search methods like 2-opt hill climbing improve a solution by repeatedly moving to a better neighboring solution. Because each move strictly reduces the tour cost, convergence is typically fast. The algorithm quickly eliminates obvious inefficiencies in a random tour. However, once the algorithm reaches a configuration where no single 2-opt swap reduces cost, it terminates. This configuration is a local minimum, not necessarily the global minimum. The algorithm has no mechanism to accept temporarily worse solutions in order to escape these local minima. This is why random restarts are necessary. By beginning from different random initial tours, the algorithm explores different regions of the search space, increasing the likelihood of finding better solutions. The chosen neighborhood operator, which was 2-opt, reverses a segment between two edges in the tour. This operator is nice because it removes crossing edges and significantly improves tour structure early in the search, but it also has its drawbacks in other scenarios. Since 2-opt only considers swaps of two edges at a time, it limits its ability to be fully optimal. As such, 2-opt offers a good balance between computational efficiency and solution quality, but it does not guarantee the best possible local minimum.

From the experimental data, here's a concrete example from the 20-city runs with seed 0:

Run 1 (Restart Index 0):

Best Cost: 297.28

Best Tour: [14, 8, 9, 15, 16, 6, 10, 13, 4, 2, 1, 12, 17, 7, 3, 18, 5, 11, 19, 0]

Initial Cost: 858.11

Iterations: 50

Run 2 (Restart Index 1):

Best Cost: 413.06

Best Tour: [1, 11, 5, 18, 3, 7, 8, 0, 10, 13, 4, 17, 15, 12, 16, 14, 2, 19, 6, 9]

Initial Cost: 1033.61

Iterations: 28

Analysis of the two**:**

These two runs started from different random initial tours and converged to significantly different local optima:

Cost difference: 297.28 vs. 413.06 (a 39% difference)

Tour structure: The better tour (297.28) likely has better properties, like fewer crossings, more logical city clustering

Convergence speed: Interestingly, the worse local minimum took fewer iterations (28-50), suggesting it was easier to find but lower quality

This shows that the cost landscape for TSP is highly rugged. Run 1 started in a region of the search space that, through 2-opt moves, led to a high-quality result. Run 2 started in a different region that funneled into a lower-quality result. Because hill climbing is deterministic once initialized, different random starting tours lead the search into different regions of the solution landscape, which shows how the random restarts change the results greatly, and why these restarts are important to find more efficient routes.

As for AI use in this project, most of the use was ChatGPT. I ran a lot of my code through it to discover any flaws, and I also used it to help debug. Some example prompts are me directly pasting error messages in, to help me source the issue. Another use of ChatGPT was helping me organize the code. I asked, for example, “what .py files should exist to help keep the code tidy for these tests?”, when handling my testing code. The idea here is to help me determine how modular to make the code. Lastly, I used Deepseek AI to organize the experiment results. I did this because Deepseek allowed me to submit the JSON files and it organized information into tables for me, saving a lot of time and busy work. My prompt read “here are the experiment results. could you put this in a table to match these instructions?” I then listed the table instructions from the project instructions provided. To make sure the AI information was right, I overviewed the code and ran tests. Additionally, I asked a classmate about parts of the code which his AI developed, to make sure the work was consistent between different projects.