CS 326 – Project 2 Report

AI Option B: A* Search and TSP Local Search

1. Problem Formulation

1.1 A* Search (Grid World)

State

A state is a grid coordinate: (x, y), where
$0 \le x < m$ and $0 \le y < n$.

The start state is (0,0).
The goal state is (m−1, n−1).

Actions

From each state, four actions are possible:

- Move up
- Move down
- Move left
- Move right

Moves are restricted to valid in-bounds neighbors.

Goal

Reach the goal coordinate (m−1, n−1).

Cost Model

Each edge has a randomly assigned weight.
The total path cost is the sum of edge weights along the path.

Heuristics

Two admissible heuristics were implemented:

Manhattan distance: $|x1 - x2| + |y1 - y2|$

Euclidean distance: $\mathrm{sqrt}((x1 - x2)^2 + (y1 - y2)^2)$

Both heuristics never overestimate true path cost in a 4-direction grid, preserving optimality.

1.2 Traveling Salesman Problem (TSP)

State

A state is a complete permutation (tour) of n cities: [c0, c1, c2, ..., c(n−1)]

Cities are randomly generated within coordinate range (0.0, 100.0).

Moves (Neighborhood Operator)

The 2-opt operator:

- Selects two non-adjacent edges.
- Removes them.
- Reconnects the tour in the alternative way.

This produces an $O(n^2)$ neighborhood.

Objective

Minimize total Euclidean tour length, including the return to the starting city.

Cost Model

Distance between cities is Euclidean: $\mathrm{sqrt}((xi - xj)^2 + (yi - yj)^2)$

Total cost is the sum over consecutive edges.

2. Implementation Description

2.1 A* Implementation

The implementation followed the class pseudocode exactly:

1. Initialize frontier (priority queue) with start state.
2. Maintain g(n) values in a dictionary.
3. Use a priority queue ordered by $f(n) = g(n) + h(n)$.
4. Maintain a closed set to prevent revisiting expanded states.
5. Upon reaching goal, reconstruct path using parent pointers.

The only variable component was the heuristic function (Manhattan or Euclidean).

The algorithm guarantees optimality because:

- All edge costs are positive.
- The heuristic is admissible.

2.2 TSP Local Search Implementation

The TSP solution used hill climbing with multiple restarts:

For each restart:

1. Generate a random initial tour.
2. Evaluate all possible 2-opt swaps.
3. Apply the first improving swap found.
4. Repeat until no improving swap exists.
5. Record best solution for that restart.

After all restarts:

- Return the best tour found overall.

Metrics recorded:

- Initial tour cost
- Final tour cost
- Improvement
- Iterations until convergence
- Runtime per restart

This method does not guarantee global optimality but is computationally efficient.

---

3. Testing Summary

To ensure correctness and validity of both A* and TSP implementations, I developed automated unit tests using `pytest`. The tests verify correctness of outputs, legality of moves, cost calculations, and termination conditions.

3.1 A* Testing

The A* implementation was tested under both heuristics (Manhattan and Euclidean) using parameterized test cases.

1. Path Start and End Validation

For a 10×10 grid:

- Verified that the algorithm returns `"status" == "success"`.
- Confirmed the first state is the start (0,0).
- Confirmed the last state is the goal (9,9).

This ensures correct goal detection and path reconstruction.

---

2. Legal Moves and Bounds Checking

For every step in the returned path:

- Verified exactly one action per transition.
- Verified each action is one of the allowed 4 moves.
- Recomputed the next state using `move()` and ensured it matches the returned state.
- Confirmed the next state remains in bounds.
- Verified that the (state, action) pair exists in the cost map.

This guarantees:

- The algorithm does not generate illegal transitions.
- No out-of-bounds moves occur.
- All transitions correspond to valid cost entries.

---

3. Total Cost Verification

The total cost reported by A* was independently recomputed:

- Iterated over each (state, action) pair.
- Summed corresponding edge costs.
- Verified equality with `result["total_cost"]`.

This ensures the algorithm correctly accumulates path cost.

3.2 TSP Testing

The TSP implementation was tested for permutation validity, cycle correctness, and convergence behavior.

---

1. Tour Validity (Permutation Test)

Generated random tours were verified to:

- Contain exactly n cities.
- Include each city exactly once.
- Match `list(range(n))` when sorted.

This ensures that tours are valid permutations.

---

2. Cycle Cost Correctness

To verify that tour cost properly forms a cycle (returns to start):

- Used a 2-city example: (0,0) and (3,4).
- Distance is 5.
- Verified cost equals 10 (5 + 5).

This confirms:

- The implementation includes the return edge.
- The tour is treated as a closed cycle.

---

3. Hill Climbing Termination

For a randomly generated TSP instance:

- Ran hill climbing with 2-opt.

- Verified algorithm terminates in finite iterations.

- After termination, checked that for every possible 2-opt move:

  two_opt_delta ≥ 0

This confirms:

- The final solution is a local minimum.
- No improving neighbor exists.
- The stopping condition is correct.

Testing Conclusion

The test suite verifies:

For A*:

- Correct path reconstruction.

- Legal and bounded transitions.
- Accurate cost computation.
- Heuristic consistency across implementations.

For TSP:

- Valid permutation structure.
- Correct cycle cost calculation.
- Proper hill climbing convergence to local minima.

These tests collectively ensure both correctness and logical validity of the implementations before conducting experimental evaluation.

4. Experimental Results and Discussion

A* Metrics Table

m	n	algorithm	heuristic	num_runs	num_success	num_failure	steps_mean	total_cost_mean	expanded_states_mean	max_frontier_size_mean	runtime_ms_mean
10	10	astar	euclidean	10	10	0	18.4000	56.6000	304.0000	78.8000	0.4016
10	10	astar	manhattan	10	10	0	18.2000	56.6000	298.3000	85.6000	0.4154
25	25	astar	euclidean	10	10	0	48.2000	134.3000	2355.6000	246.3000	3.2609
25	25	astar	manhattan	10	10	0	48.2000	134.3000	2351.4000	291.0000	3.1621
50	50	astar	euclidean	10	10	0	99.4000	264.7000	9730.1000	529.6000	13.4580
50	50	astar	manhattan	10	10	0	99.4000	264.7000	9720.3000	642.7000	11.9480

4.1 A* Results

Path Length

Observed path lengths matched theoretical values:

- 10×10 → ~18
- 25×25 → ~48
- 50×50 → ~99

This confirms optimality.

Expanded States

Expanded nodes increased significantly with grid size:

- ~300 → ~2,300 → ~9,700

Manhattan expanded slightly fewer states than Euclidean due to better alignment with grid movement.

Frontier Size

Maximum frontier size increased with grid dimension, reflecting larger wavefront exploration.

Runtime

Runtime scaled proportionally with expansions:

- ~0.4 ms → ~3 ms → ~12 ms

Heuristic guidance dramatically reduced unnecessary exploration compared to UCS.

TSP Metrics Table

n_cities	algorithm	operator	num_runs	initial_cost_mean	best_cost_mean	improvement_mean	iterations_mean	runtime_ms_mean	overall_best_cost	best_cost_across_restarts_per_seed_mean
20	tsp local search	two_opt	100	1105.3343	406.1232	699.2110	14.9800	1.8500	362.2844	396.0458
30	tsp local search	two_opt	100	1601.0416	493.0356	1108.0060	24.0700	7.9500	437.5190	475.2905
50	tsp local search	two_opt	100	2606.9082	618.3194	1988.5887	44.7900	43.1900	559.7490	596.4401

4.2 TSP Results

Improvement from Random Tours

Large reduction observed:

- n=20: ~1105 → ~406
- n=30: ~1601 → ~493
- n=50: ~2606 → ~618

The larger the instance, the greater the total improvement.

Iterations

Iterations increased with n:

- ~15 → ~24 → ~45

Runtime

Runtime increased due to $O(n^2)$ neighborhood:

- ~1.85 ms → ~7.95 ms → ~43.19 ms

Despite growth, runtime remained manageable.

Restart Effect

Restarts improved robustness.
Overall best cost was lower than average best cost per run, showing variation across local minima.

---

5. Conceptual Analysis

5.1 Why A* Expands Fewer States Than UCS

Uniform-Cost Search (UCS) expands nodes in increasing order of g(n) only, without considering proximity to the goal.

A* expands nodes in increasing order of:

`f(n) = g(n) + h(n)`

If h(n) is admissible (never overestimates the true cost), then:

- A* preserves optimality.
- A* biases exploration toward the goal.
- Many nodes UCS would explore are never considered by A*.

Thus, A* reduces unnecessary expansions by incorporating heuristic guidance.

5.2 Why A* Is Still Exponential in the Worst Case

Even with a heuristic, the search space grows exponentially in branching factor and depth.

If:

- The heuristic provides little guidance (e.g., very weak heuristic),
- Or many paths have similar f-values,
- Or costs vary unpredictably,

then A* may still explore a very large portion of the state space.

Formally, in the worst case A* can still expand O(b^d) nodes, where:

- b = branching factor
- d = solution depth

The heuristic improves average-case performance but does not eliminate exponential worst-case complexity.

5.3 Why Exact Search Is Infeasible for TSP

The number of possible tours in TSP grows factorially:

(n − 1)! / 2

This growth is extremely fast. Even for 20 cities, the number of possible tours is astronomically large. Exact algorithms (like brute force or full search) require exponential or factorial time, making them impractical for moderate n. Therefore, approximate methods are necessary.

5.4 Why Local Search Converges Quickly

Local search begins with a complete solution and repeatedly applies small improvements (e.g., 2-opt swaps). Each step strictly improves cost until no better neighbor exists.

Because:

- Each move reduces cost,
- The neighborhood is efficiently searchable ($O(n^2)$ for 2-opt),
- Many crossings can be removed early,

the algorithm rapidly improves poor initial tours and reaches a local minimum in relatively few iterations.

5.5 Why Local Search Can Get Stuck in Local Minima

Local search only explores neighboring solutions. Once no improving move exists, it stops — even if a better global solution exists elsewhere.

This happens because:

- The search is greedy.
- It does not explore worse intermediate states.
- The landscape contains many local optima.

Thus, different initial tours may converge to different final solutions.

5.6 Effect of Neighborhood Operator

The neighborhood operator determines which solutions are reachable from the current state.

For 2-opt:

- Removes two edges and reconnects the tour.
- Eliminates crossings efficiently.
- Balances solution quality and computational cost.

Stronger operators (e.g., 3-opt) explore larger neighborhoods and may find better solutions but increase runtime.

5.7 Concrete Example: Same n (20), Different Local Minima

Algorithm: tsp local search
Operator: two_opt
Restarts: 10
Coord range: (0.0, 100.0)

Run A — Seed = 42

```
Restart 1: initial=1180.879 best=353.553 iters=17 runtime_ms=2
Restart 2: initial=1046.191 best=352.318 iters=17 runtime_ms=2
Restart 3: initial=942.242 best=353.553 iters=13 runtime_ms=2
Restart 4: initial=981.959 best=352.318 iters=16 runtime_ms=2
Restart 5: initial=1017.003 best=352.318 iters=14 runtime_ms=2
Restart 6: initial=1212.760 best=352.318 iters=17 runtime_ms=2
Restart 7: initial=984.068 best=352.318 iters=14 runtime_ms=2
Restart 8: initial=1044.884 best=352.318 iters=17 runtime_ms=2
Restart 9: initial=1054.750 best=352.318 iters=18 runtime_ms=2
Restart 10: initial=1106.740 best=352.318 iters=13 runtime_ms=2
```

Best overall cost: **352.318**
Best overall tour:
[12, 19, 14, 18, 10, 15, 2, 7, 16, 17, 8, 5, 6, 13, 1, 11, 4, 0, 9, 3]

Observation: Most restarts converge to the same local minimum (352.318), with a few slightly worse plateaus (353.553).

Run B — Seed = 43

```
Restart 1: initial=1065.967 best=394.139 iters=15 runtime_ms=2
Restart 2: initial=941.292 best=400.823 iters=14 runtime_ms=2
Restart 3: initial=986.403 best=419.546 iters=15 runtime_ms=2
Restart 4: initial=1086.355 best=392.198 iters=15 runtime_ms=2
Restart 5: initial=1004.345 best=392.198 iters=11 runtime_ms=2
Restart 6: initial=1109.210 best=383.370 iters=16 runtime_ms=2
Restart 7: initial=1147.561 best=393.508 iters=16 runtime_ms=2
Restart 8: initial=1035.798 best=386.124 iters=17 runtime_ms=2
Restart 9: initial=1018.944 best=392.198 iters=17 runtime_ms=2
Restart 10: initial=1088.344 best=394.139 iters=12 runtime_ms=1
```

Best overall cost: **383.370**
Best overall tour:
[14, 5, 18, 6, 2, 8, 11, 16, 15, 19, 13, 9, 7, 10, 12, 3, 1, 4, 0, 17]

Observation: The best solution here (383.370) is significantly worse than the best from Seed 42 (352.318), even with identical algorithm settings.

What this demonstrates

- Both runs use the same neighborhood operator (2-opt) and restart strategy, yet they converge to *different* locally optimal tours.
- Because 2-opt only accepts improving swaps, once a tour has no improving 2-opt move, it becomes "stuck" in that local minimum.
- Different random city layouts (different seeds) produce different cost landscapes, so the local minima reached (and their quality) can differ substantially.

6. Overall Conclusion

A* efficiently finds optimal solutions using heuristic guidance while remaining exponential in the worst case.

TSP local search scales to larger instances efficiently but sacrifices global optimality guarantees.

This project illustrates the fundamental AI tradeoff between:

- Exact optimal search
- Scalable heuristic optimization

Heuristics dramatically improve efficiency, but combinatorial complexity remains the underlying challenge.

Appendix A: AI Disclosure

AI tools were used to assist in structuring explanations and drafting written analysis. Tools used included ChatGPT and VSCode Github Copilot.

Example Prompts:

1. *Given my 2-opt TSP local search implementation, how can I reduce runtime without changing the algorithm's correctness? For example, are there ways to compute 2-opt deltas more efficiently or avoid recomputing full tour costs?"*
2. *"Based on my A* and TSP experiment outputs (expanded states, runtime, iterations, best cost, etc.), help me generate a structured metrics report that explains trends, scaling behavior, and comparisons between algorithms."*
3. *"Given these two seeded TSP runs (same n, same algorithm settings, different random seeds), show clearly how the runs converge to different local minima and explain why this happens in hill climbing with 2-opt."*

All algorithm design was pulled from class psuedocode slides.

Implementation, experimentation, debugging, and performance evaluation were completed independently with a structured idea in mind.