Why Space-Filling Curve Heuristic Is a Great Choice

Handles Massive Inputs:

O(n log n) time is feasible for 50,000+ cities, which is crucial for your hard time limits.

Leverages Geometry:

Since it’s Euclidean TSP, ordering by space-filling curves clusters spatially close cities together, making an initial “tour” that is usually not terrible.

Easy to Combine With Local Optimization:

You can run a windowed 2-opt or simple improvement step in O(n) time per pass.

No Black-Box Libraries Needed:

All math and sorting is easy to code from scratch in C—meets the "no extensive libraries" rule.

Extremely Fast:

C will let you push the limits on big instances, and your method will work on the grader’s inputs.

A Few Notes About TSPwP (TSP with Penalties)

Your version is a "prize collecting" TSP:

You are allowed to skip some cities, but pay a penalty if you do.

Your goal: minimize total tour length + penalty for unvisited cities.

How This Impacts Your Approach

After constructing the tour, consider dropping some cities if their penalty is less than the extra length their inclusion would add.

This can be handled via a simple greedy post-processing step:

For each city, see if skipping it (and connecting its neighbors directly) reduces the sum “distance + penalties paid.”

Implementation Roadmap for Your Project

Step 1: Parse Input

Read penalty and city coordinates from file.

Step 2: Assign Space-Filling Curve Codes

For each city (x, y), assign a Morton or Hilbert code.

Morton is easier to code in C and fast; Hilbert is better but more complex—Morton is perfectly fine for this project.

Step 3: Sort Cities

Sort all cities by their Morton/Z-order code.

Step 4: Construct Initial Tour

Connect cities in sorted order to form an initial tour (circular path).

Step 5: Local Optimization (Optional but recommended)

For each city, try swapping its order with a neighbor in the tour (within a window, e.g., ±10).

Apply a 2-opt move if it shortens the tour.

Step 6: Penalty-Aware Pruning

For each city in the tour, check if skipping it (and paying the penalty) would reduce total cost (i.e., if penalty < extra distance caused by detouring to that city).

Remove city if it makes sense; repeat a few passes.

Step 7: Output the Solution

Output total cost and ordered city IDs of visited cities, as specified.

Summary Table: Heuristic Roadmap

| n (cities) | Approach | Execution time | Quality |

| ------------ | --------------------------------------------- | ----------------- | --------------- |

| < 5,000 | Morton + 2-opt (full pass) | Acceptable (secs) | Good |

| 5,000–20,000 | Morton + 2-opt (local only) | Fast (seconds) | Good/Very good |

| > 20,000 | Morton only, or 2-opt on local/random regions | Very fast | Acceptable–good |

For small n (≤5,000): Full 2-opt

For medium n (5,000 < n ≤ 20,000): Local 2-opt (limit window size)

For large n (>20,000): Local 2-opt on random regions (randomly select a few segments and do local 2-opt within them)

would this work just fine

ChatGPT said:

Your greedy penalty prune is the classic, correct, and practical approach for TSP with Penalties, especially for large n.

In fact, this method is what’s used in most research and contests for “TSP with node penalties” unless you want to go much deeper (with metaheuristics or dynamic programming, which are not practical for n = 50,000).

Why Greedy Prune Works Well

Simple: Removes cities only if it improves the solution.

Fast: One or two passes is typically enough.

Local-Optimal: No skipped city can be added back to improve the cost, and no visited city can be skipped to further improve.

Easy to implement and maintain.

Can it be improved?

Possible improvements (if you ever want to try, but not necessary for class projects):

Randomize order of pruning attempts (prevents some local traps, but helps only a little).

Multiple passes:

Already covered—you run until no more improvement, which is standard.

Re-visit skipped cities:

After pruning, sometimes re-inserting a previously skipped city (if it helps now) can slightly improve the solution, but this is rare and not usually worth the extra complexity.

Metaheuristics (Simulated Annealing, Tabu, etc):

Powerful but much more complex and not justified unless you’re competing at the research level.

Best Practice for Your Case

Apply the greedy prune (what you have).

Optionally rerun 2-opt after pruning to optimize the new, smaller tour.

Report: tour length, penalties paid, total cost.

A screenshot of a black screen

AI-generated content may be incorrect.

For 50k input size and 2-opt parameters with K = 3 full execution took Execution time: 56.49171800 seconds, I was initially using K = 1000 so it would take like 8 to 10 hours to optimize it please choose a moderate value for K especially for very large N.

# Refernces for some:

<https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/>

<https://en.wikipedia.org/wiki/Space-filling_curve>

<https://ashtl.sourceforge.net/morton_overview.html>

<https://en.wikipedia.org/wiki/Travelling_salesman_problem>

<https://en.wikipedia.org/wiki/Set_TSP_problem>

<https://en.wikipedia.org/wiki/Traveling_purchaser_problem>

<https://en.wikipedia.org/wiki/Heuristic_(computer_science)>

<https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/>