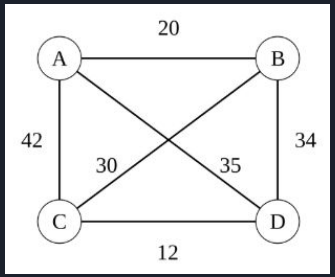
Traveling Salesman Problem

By: Alec Dean, Ben Tanger, and Logan Page

What is the Traveling Salesman Problem

The Traveling Salesman Problem is a graph problem that takes in an undirected, weighted, complete graph.

The solution to the problem is the path that hits every vertex in the graph and returns to the starting point with the lowest total weight (the cheapest cycle).



A practical application of the problem is a salesman going to various cities to promote a product, and wants to know what would be the quickest way to go to every city and return back to the office. The cities would be the vertices and the edges would be the length of the drive between cities.

Input and Output of TSP

Input:

Line 1: V vertices and E edges

Followed by n lines representing each edge in ‘u v w’ format

Example:

3 3

a b 3

b c 4

a c 5

Output:

First line is the total cost of the path

Second line is the ordered vertices in the path.

Example:

12

a b c a

Polynomial Runtime Modification

A modification of the problem could be to feed that algorithm a connected circle of vertices

This would cause the solution to just be a traversal through all the vertices and return to its starting point

The runtime of the algorithm would then be O(V)

Reduction to Prove NP Completeness

Converting the Hamiltonian Cycle into a Traveling Salesman Problem

- Create an undirected graph G of n vertices and m edges
- From that graph: create a new graph G` that sets each already existing edge to weight 1 and make new edges to weight 2 such that the new graph is weighted and complete. This takes $O(V^2)$
- Set a threshold B equal to n

Proof

- If G has a Hamiltonian Cycle, then there is a TSP solution = B in G` as all edges from G to G` have a weight of 1
- If G` has a TSP solution = B , then there is a Hamiltonian Cycle in G because the preexisting edges were set to weight 1

Exact Solution

Brute Force Approach

Solving TSP through brute force requires you to make every permutation possible within the graph and return the cheapest cost with the path at the end. This can be minimized by failing permutation generations early if the best known cost has already been exceeded.

- Reading in the complete graph is $(E + 1)$
- Creating a matrix of the vertices/edges is both (V^2)
- Recursively creating every permutation is $(V \cdot (V - 1)!)$
- Therefore, the overall runtime of the algorithm is $O(V!)$

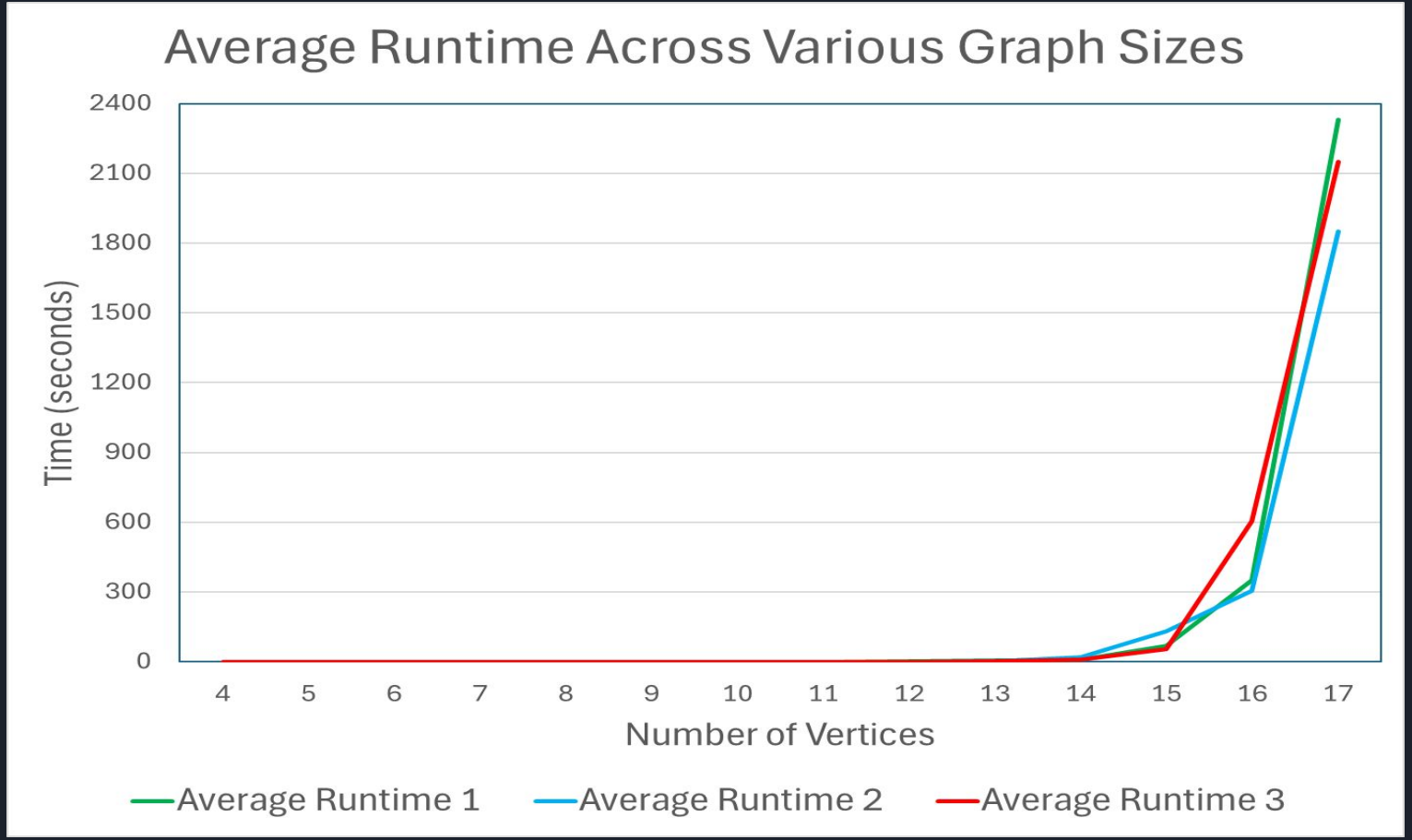
Input Order Effectiveness

Having the graph set up in such a way that the vertices are ordered by closeness relative to the other vertices could reduce the runtime

There is a higher probability of finding the lower costs quicker and thus making pruning more efficient

Being able to determine that a permutation is no longer good more often would drastically improve the runtime as opposed to a graph that has to generate more total permutations

Exact Solution Runtime



Greedy Approximation:

Approach

I approached the Traveling Salesman Problem by implementing a greedy heuristics solution.

- Start at a random vertex
- Find the top k nearest vertices
- Travel to one of those k vertices randomly
- Repeat until you get a completed tour and save it
- Repeat this process i times
- Return the lowest costing tour

The greedy choice is to pick the smallest edge weights or the nearest neighbor but introducing randomness by choosing one of the multiple nearest neighbors.

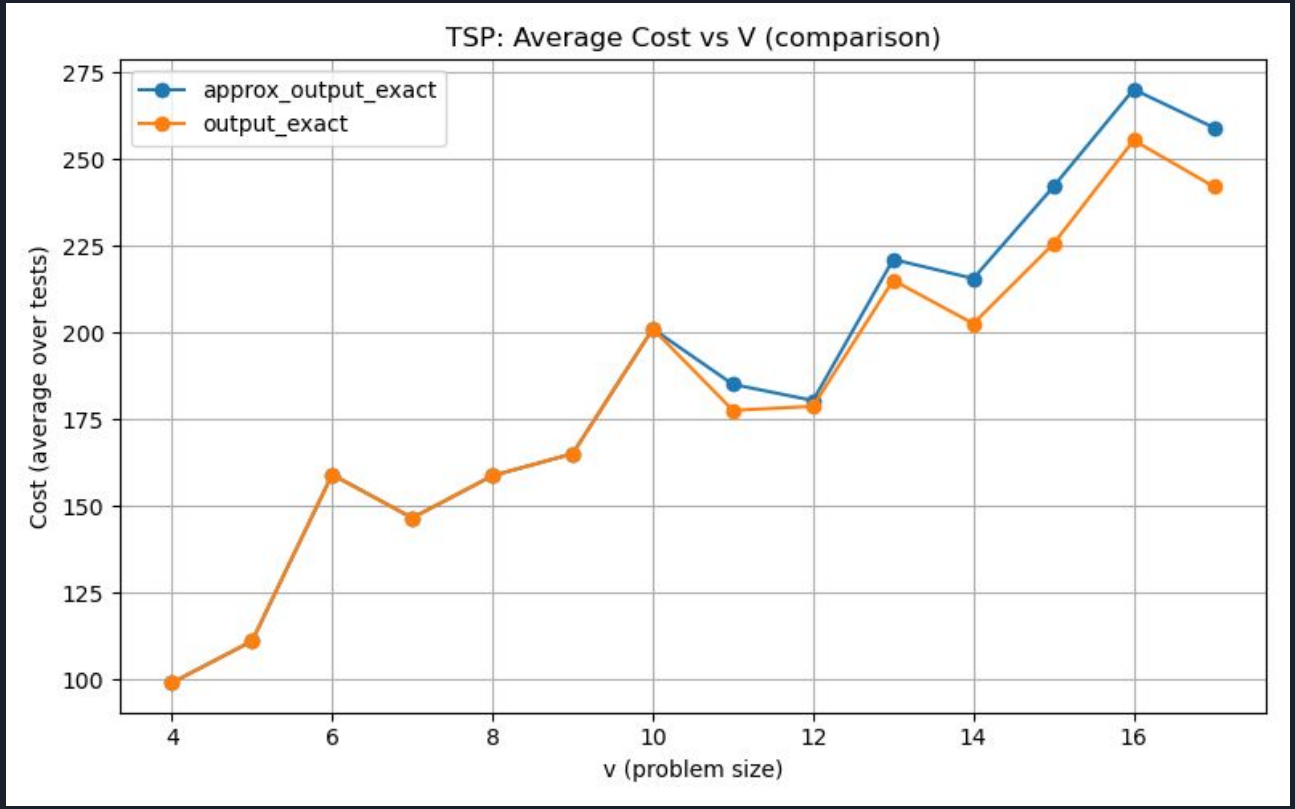
Runtime

The runtime of each part of the program is:

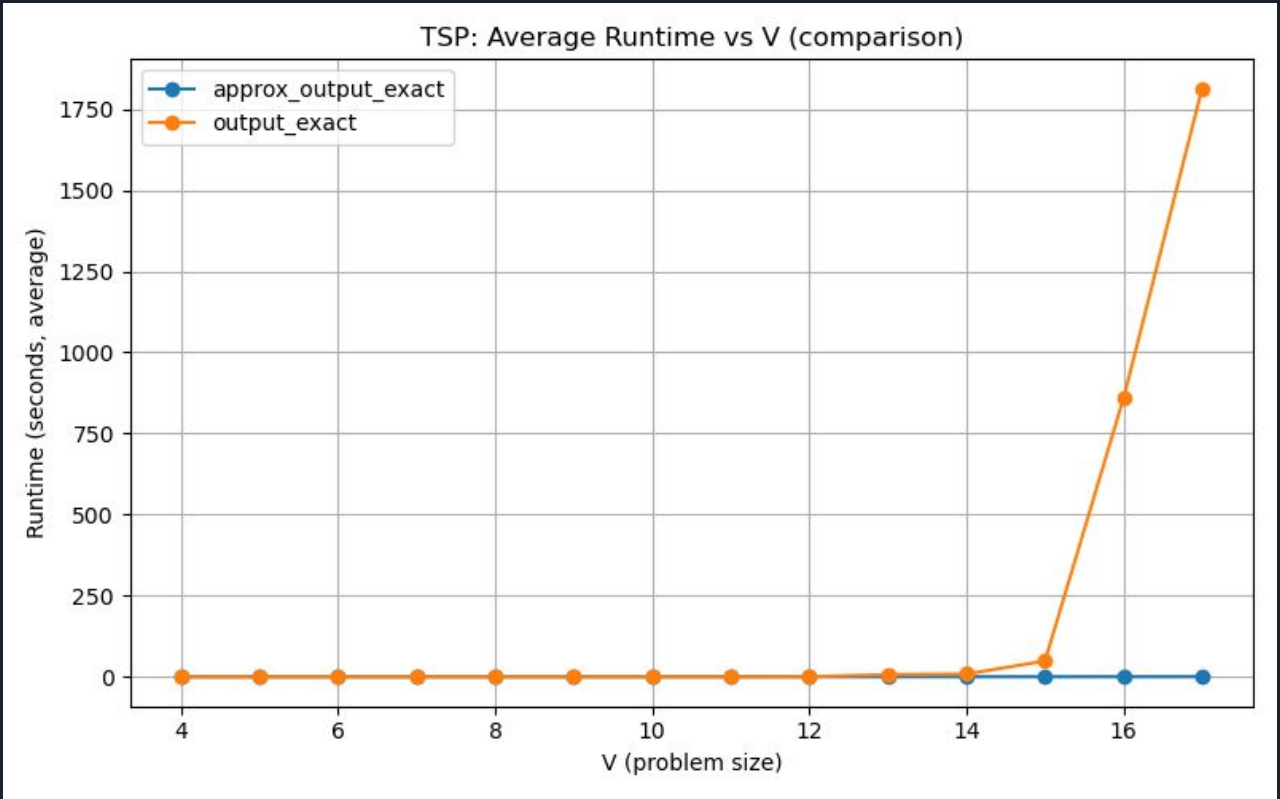
- Reading input: $O(E)$
- Index mapping: $O(V)$
- Distance_matrix creation: $O(V^2)$
- Single greedy tour construction: $O(V^2)$
- Main iteration loop: $O(I \times V^2)$ - runs I iterations, each performing the $O(V^2)$ greedy tour construction

Overall: The dominant term is $O(I \times V^2)$, which simplifies to $O(V^2)$ since we treat the number of iterations as a constant.

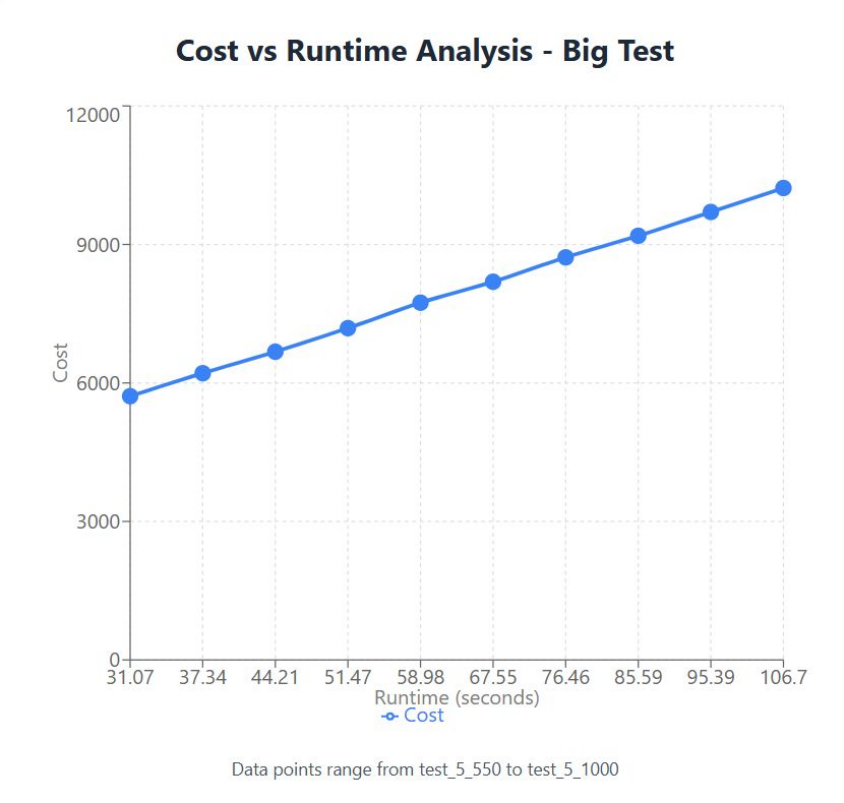
Approximation vs. exact solution tour cost



Approximation vs. exact solution runtime



Approximation of Big Graphs Runtime Vs Costs



Test Number	Cost	Runtime
test_5_550	5713	31.0650408
test_5_600	6211	37.3385155
test_5_650	6676	44.2082605
test_5_700	7187	51.4686761
test_5_750	7739	58.9843056
test_5_800	8192	67.5471699
test_5_850	8720	76.4598329
test_5_900	9188	85.5913746
test_5_950	9704	95.3872499
test_5_1000	10223	106.6952355

Random Restart approximation:

Approach

- Choose a random order of vertices as a tour and compute length
- Swap two random vertices
- Calculate new tour length
- If new tour length is shorter than original length keep it,
- Repeat swaps until local minimum is reached
- Restart with another random order of vertices

Runtime

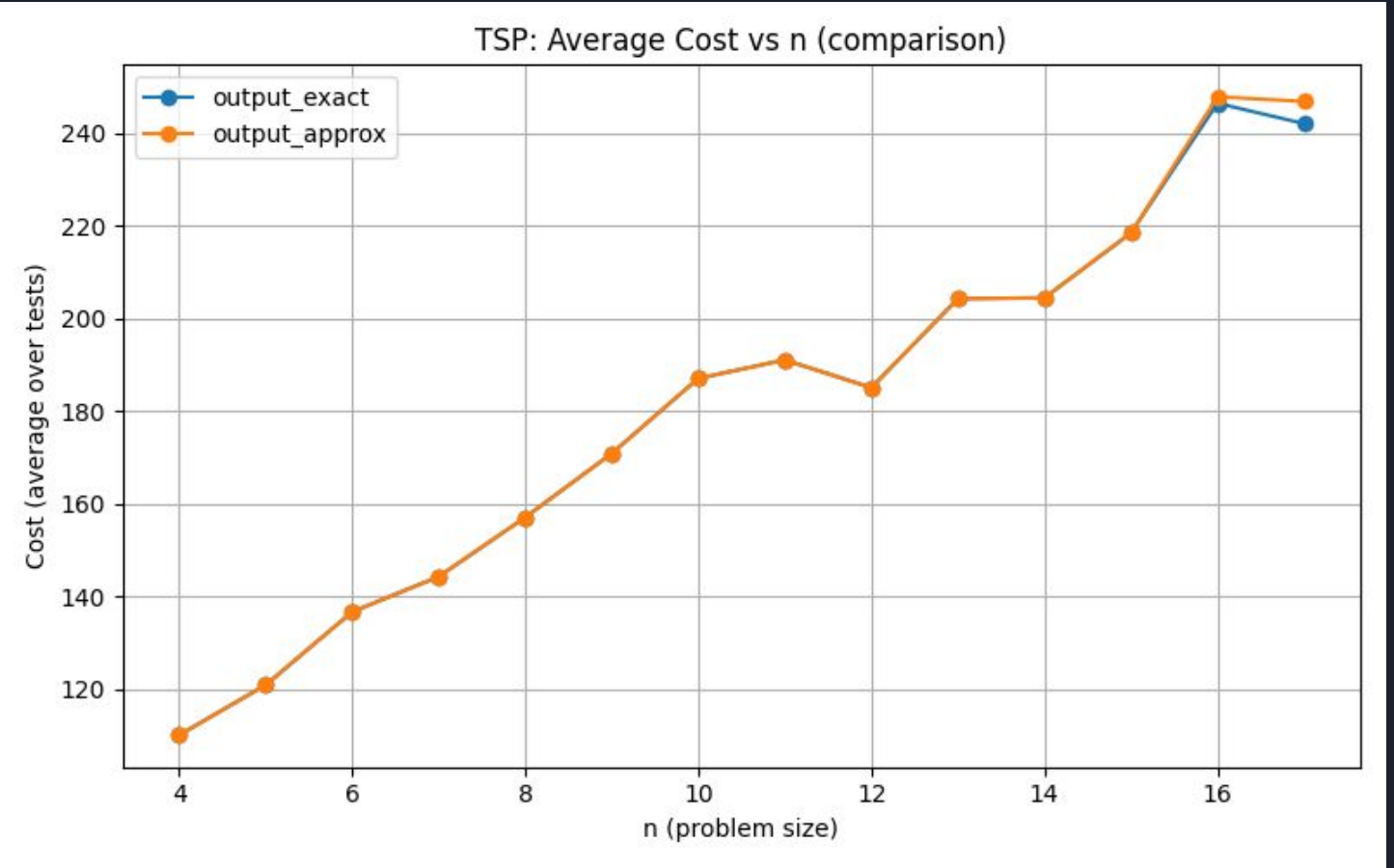
The Runtime is:

The runtime is O(V^3). The runtime of each part of the program is:

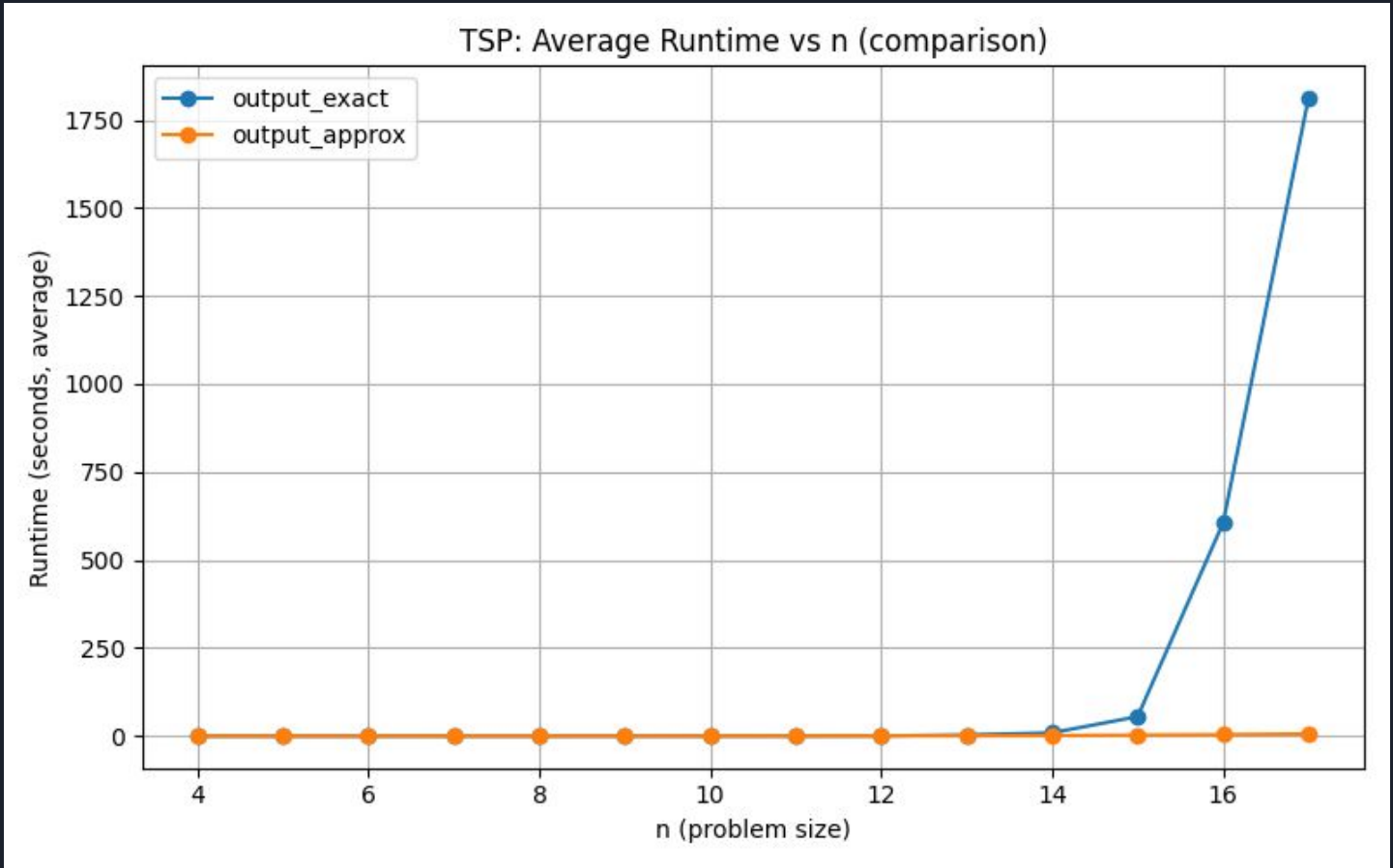
- index mapping: O(V)
- Distance_matrix: O(V^2)
- Calculate_tour_cost: O(V)
- swap: O(V)
- random_restart O(V^2)
- main loop O(V)

Each restart in the main loop calls random_restart therefore we get O(V^3)

Approximation vs. exact solution tour cost



Approximation vs. exact solution runtime



Parallelism: