The Traveling Salesman Problem

Nick Hamilton, Luke Hine, Cal Fitzgerald

Traveling Salesman Exact Solution

This exact solution brute forces the optimization problem in order to return the shortest path that visits each city once and returns to the origin.

Decision Vs Optimization

The decision problem of the traveling salesman basically asks if there is a tour of the cities with a total length less than or equal to a given number and then gives a yes or no answer.

The optimization problem seeks to find the shortest path that visits each city and returns to the origin.

Example Input and Output

```
3 3
a b 10
a c 4
b c 9
```

```
23
a b c a
```

Real Life Uses of TSP

i. Drilling of printed circuit boards

ii. Overhauling gas turbine engines

iii. X-Ray crystallography

iv. Computer wiring

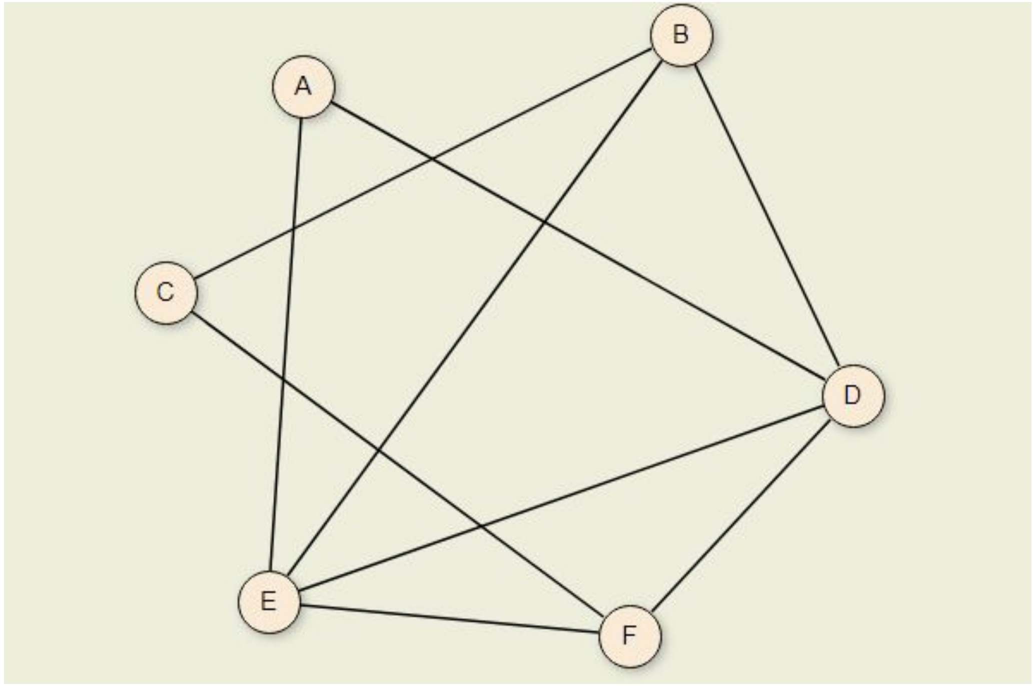
v. The order-picking problem in warehouses

https://cdn.intechopen.com/pdfs/12736/intechtraveling_salesman_problem_an_overview_of_applications_formulations_and_solution_approaches.pdf

Polynomial Certifier

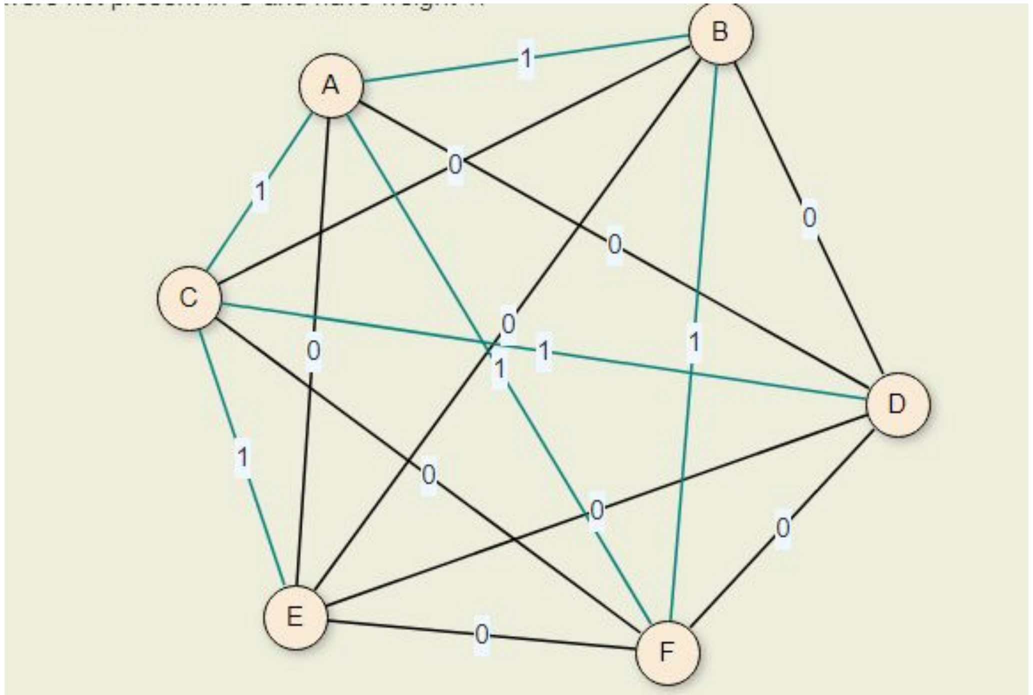
For the decision version of TSP, a polynomial certifier would involve checking if a given tour is less than or equal to a specified length.

Hamiltonian Cycle to TSP



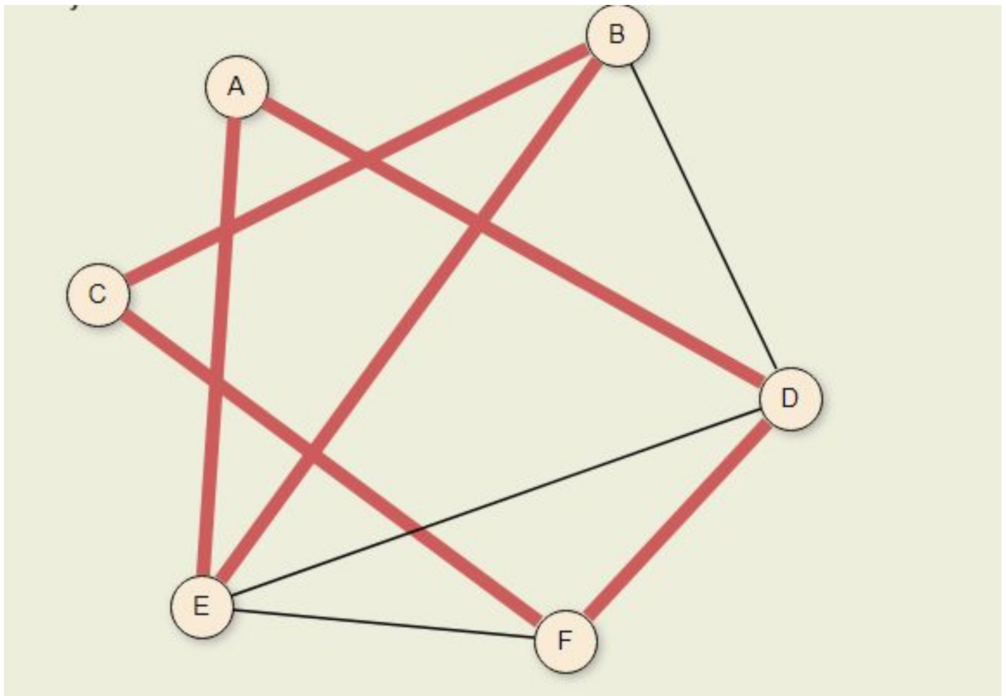
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/hamiltonianCycle_to_TSP.html

Hamiltonian Cycle to TSP



https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/hamiltonianCycle_to_TSP.html

Hamiltonian Cycle to TSP



https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/hamiltonianCycle_to_TSP.html

Code Explanation

It initializes an empty dictionary to represent the graph

Then iterates through each edge

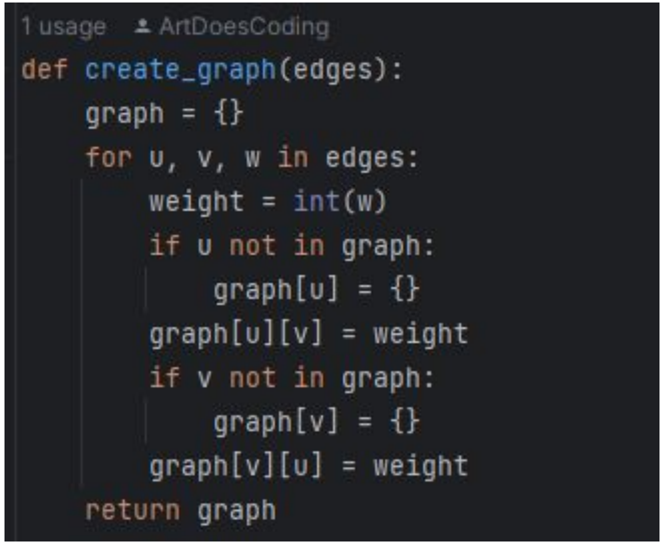
Checks if the starting node is already in the graph

This dictionary will hold adjacent nodes

and their respective weights.

Then adds the end node and the last bit ensures

That its an undirected graph

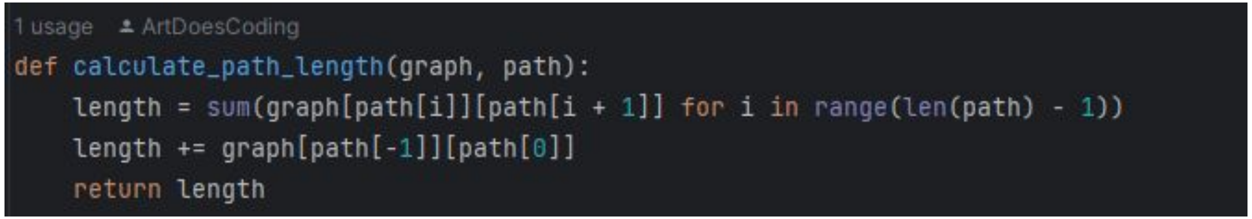


```python
def create_graph(edges):
    graph = {}
    for u, v, w in edges:
        weight = int(w)
        if u not in graph:
            graph[u] = {}
        graph[u][v] = weight
        if v not in graph:
            graph[v] = {}
        graph[v][u] = weight
    return graph
```

Code Explanation

The first line calculates the sum of the weights of the edges between consecutive nodes in the path.

It then completes the path by adding the weight from the last vertex to the origin



1 usage ArtDoesCoding

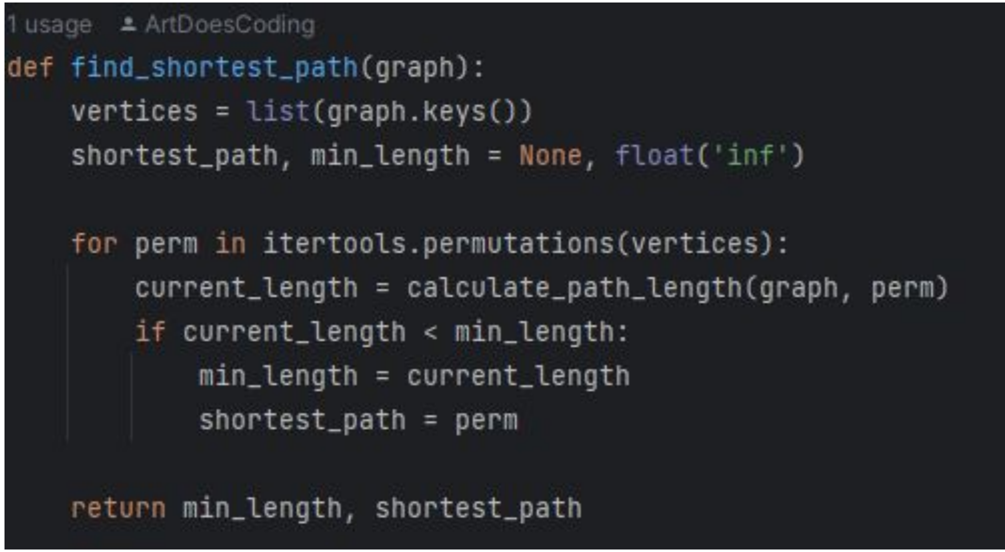
```python
def calculate_path_length(graph, path):
    length = sum(graph[path[i]][path[i + 1]] for i in range(len(path) - 1))
    length += graph[path[-1]][path[0]]
    return length
```

Code Explanation

Stores the nodes into a graph named vertices

Uses itertools to generate all permutations

Calculates the total length

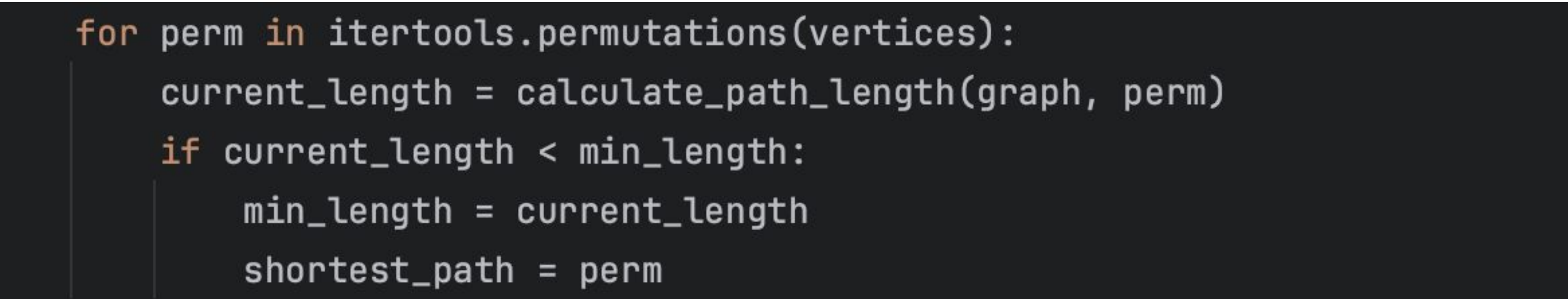


```python
def find_shortest_path(graph):
    vertices = list(graph.keys())
    shortest_path, min_length = None, float('inf')

    for perm in itertools.permutations(vertices):
        current_length = calculate_path_length(graph, perm)
        if current_length < min_length:
            min_length = current_length
            shortest_path = perm

    return min_length, shortest_path
```

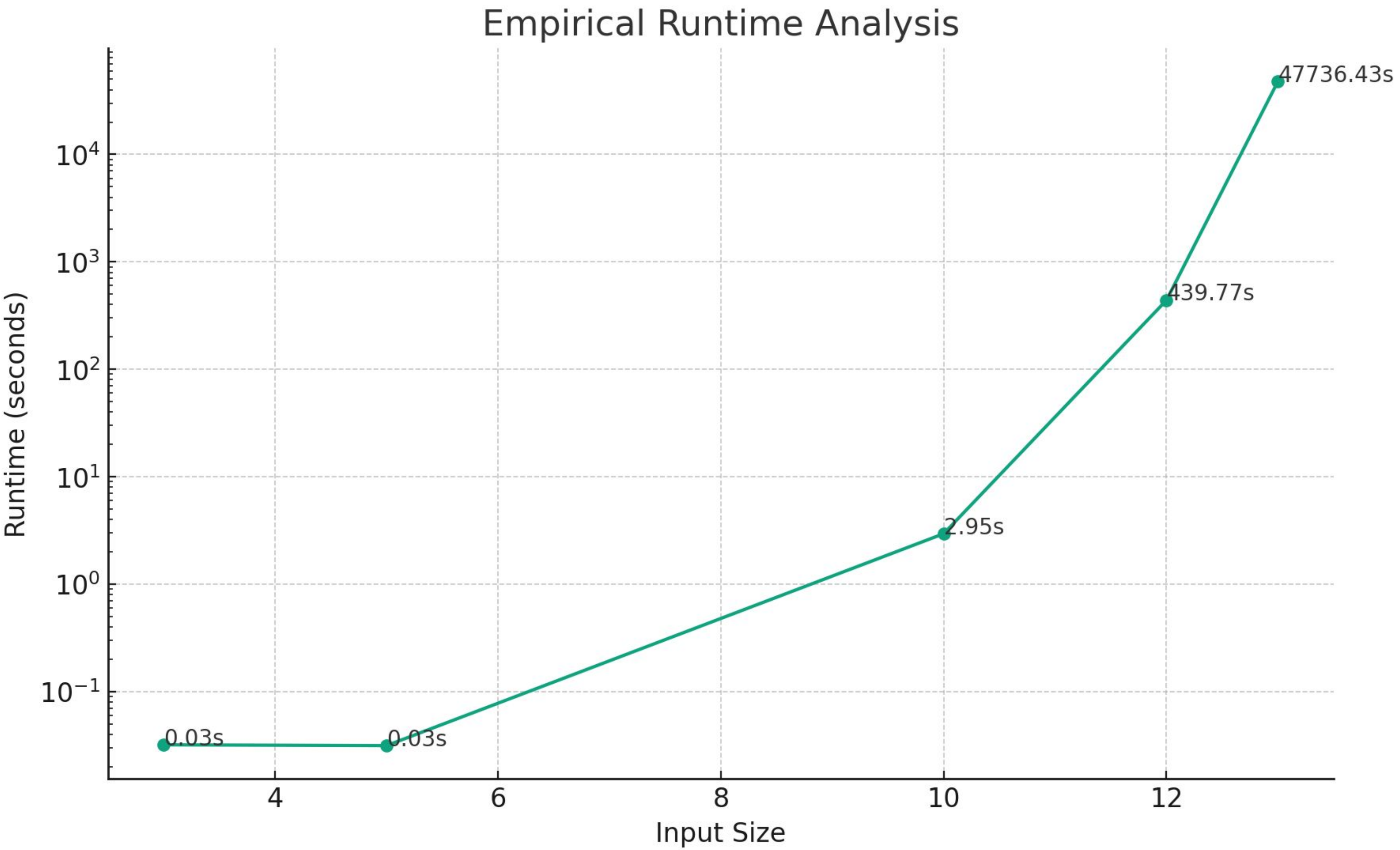
Big O runtime

The runtime complexity is O(n!).

itertools.permutations(vertices) generates all possible orderings of the vertices. If there are n vertices, there are n! (factorial) permutations thus it is the dominant term.



```
for perm in itertools.permutations(vertices):
    current_length = calculate_path_length(graph, perm)
    if current_length < min_length:
        min_length = current_length
        shortest_path = perm
```


Empirical Runtime Analysis
Runtime (seconds)
Input Size
10^{-1}
10^0
10^1
10^2
10^3
10^4
4
6
8
10
12
0.03s
0.03s
2.95s
439.77s
47736.43s

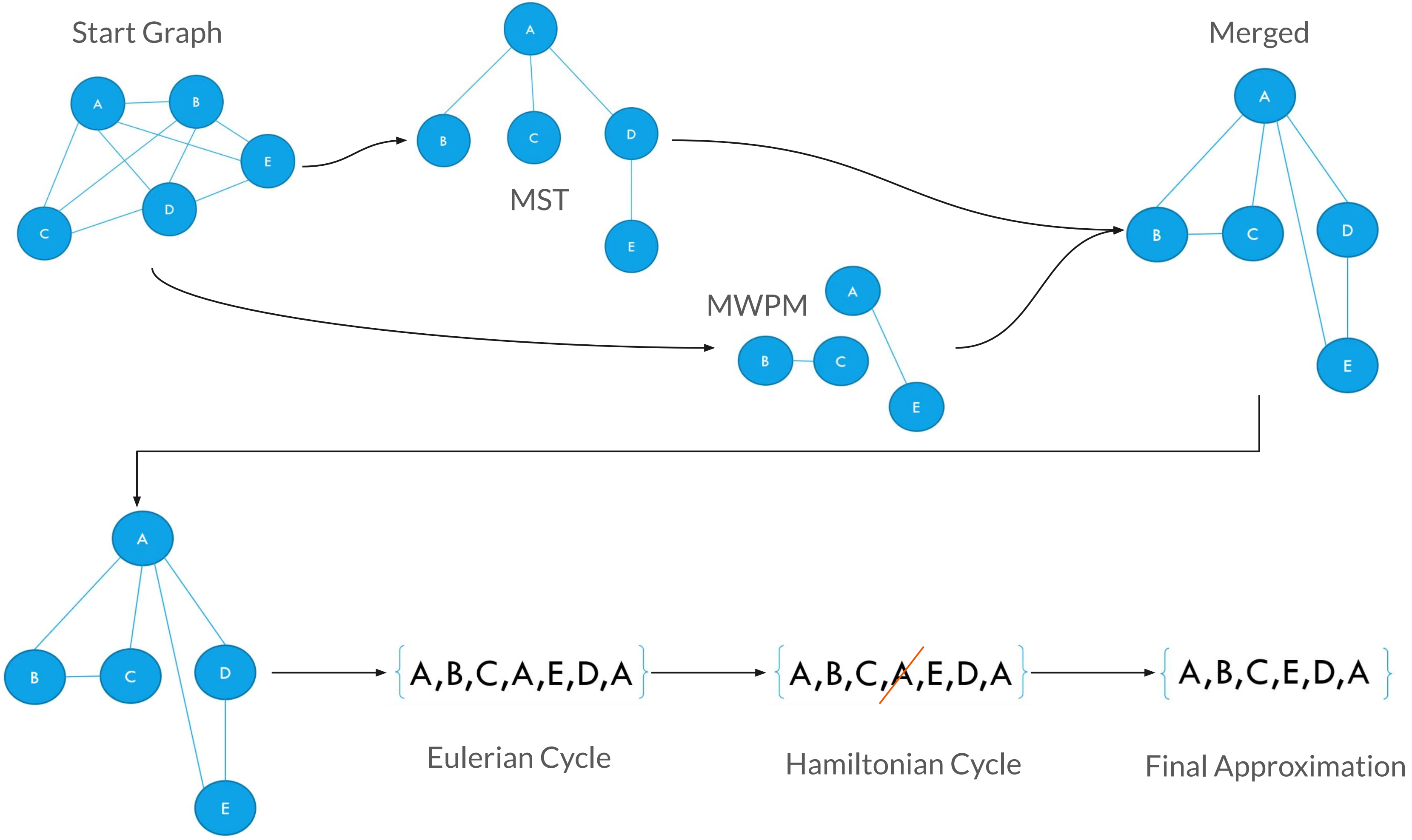
Bellman Ford Argument

The Longest Path Problem stipulates the path must be simple (no repeated vertices). However, after inverting weights, the Bellman-Ford algorithm might find a shortest path that is not simple in the original graph. Bellman-Ford can detect negative cycles, but it doesn't resolve them. Thus it could get stuck in them forever.

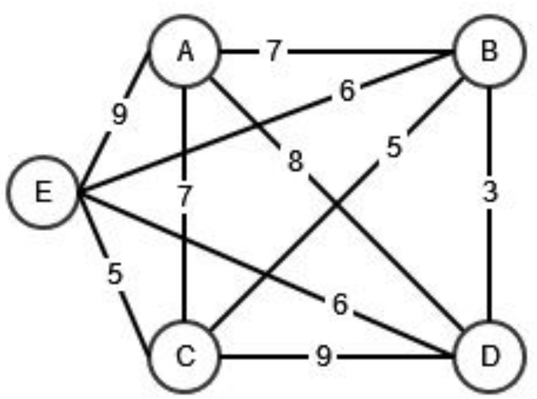
Christofides Algorithm

Discovered by Nico Christofides in 1976, this algorithm was designed to produce a result within 3/2 the optimal solution

```
Christofides Algorithm(Graph G):
    Find a minimum spanning tree T of G.
    Let O be the set of vertices with odd degree in T.
    Find a minimum weight perfect matching M in the induced subgraph of G on the vertices in O.
    Combine the edges of T and M to form a multigraph H.
    Find an Eulerian circuit in H.
    Convert the Eulerian circuit into a Hamiltonian circuit by skipping repeated vertices.
    Return the Hamiltonian circuit as the approximate solution to the TSP.
```


Start Graph
MST
MWPM
Merged
A
B
C
D
E
{A,B,C,A,E,D,A}
Eulerian Cycle
{A,B,C,~~A~~,E,D,A}
Hamiltonian Cycle
{A,B,C,E,D,A}
Final Approximation

Suboptimal Example



This graph will form a solution of [a b c e d a] with a weight of 31

The exact solution for this graph is [a b d e c a] with weight of 28

This is because Christofides makes some choices arbitrarily when faced with decisions that have the same weight locally, and therefore it may not always produce the same result.

Runtime Complexity Analysis

All components of Christofides run in linear time, except two

Minimum Spanning Tree Creation (MST)

- Runs in O(E log V) time when using Prim's or Kruskal's
- Other methods can be used, but these are most effective

Minimum Weight Perfect Matching (MWPM)

- Runs in O(V^3) time when using the blossom algorithm

Therefore the overall worst-case time complexity is O(V^3)

Runtime Comparisons

	Exact (seconds)	Approximation (seconds)
Test 1	0.03	0.70
Test 2	0.03	0.79
Test 3	2.95	0.79
Test 4	439.77	0.71
Test 5	47736.42	0.74

Result Comparisons

	Exact Result	Approximation Result
Test 1	23	23
Test 2	28	31
Test 3	19	33
Test 4	24	36
Test 5	33	53

Lower Bound Analysis

Performing lower bound analysis for TSP is as simple as getting the weight of the graph’s MST and comparing your solutions weight against that. For example, on test case 4:

	Weight	Delta
MST	16	n/a
Exact Solution	24	8
Christofides Solution	36	20

Nearest Neighbor Algorithm

Nearest Neighbor Algorithm chooses the closest vertex from the current vertex and continues to do so until no more unvisited vertices remain.

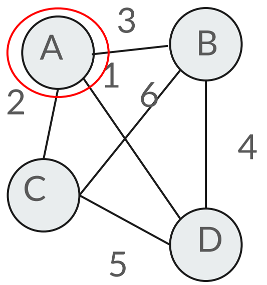
This is a greedy Algorithm for closest neighbor

We’ve implemented it to be both random and greedy. It is random with initial vertex selection.

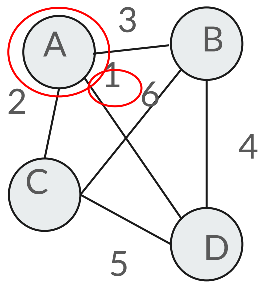
Nearest Neighbor Pseudocode

```
Add Current Vertex to Visited
Get smallest edge to another vertex not in Visited
Make that edge destination your current vertex and add edge weight to totalWeight
Repeat until no unvisited vertices remain and return to original vertex.
```

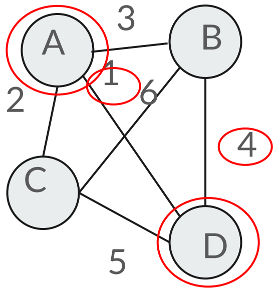
Nearest Neighbor Visualization

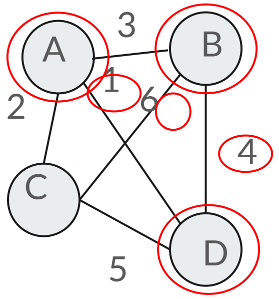


Pick Random Vertex

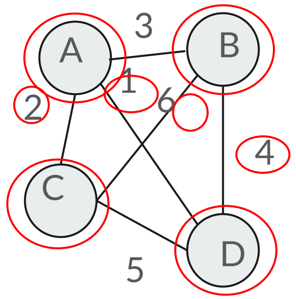


Find Shortest Edge





Repeat, ensuring no vertex visited twice



Final Vertex then takes the path to Starting Vertex

Visited List

A

A	D

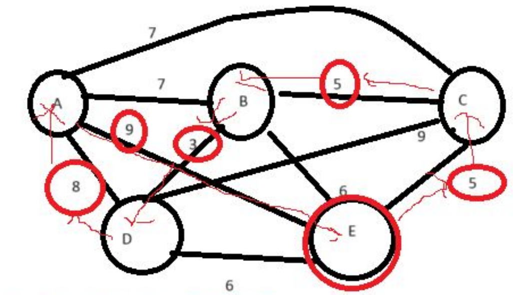
A	D	B

A	D	B	C	A

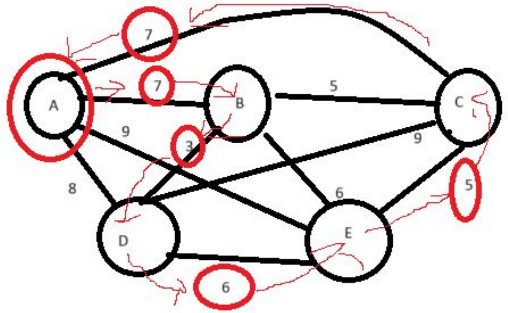
Total Length: 13

Visualization of Non Optimal Solution

Excuse the MS Paint, But as you can see, if the random initial vertex chooses a poor location, it will not return the optimal solution. This can be the case even if it chooses the correct starting location.



Not Optimal (Visited List = [E,C,B,D,A,E]
Total Length = 30,



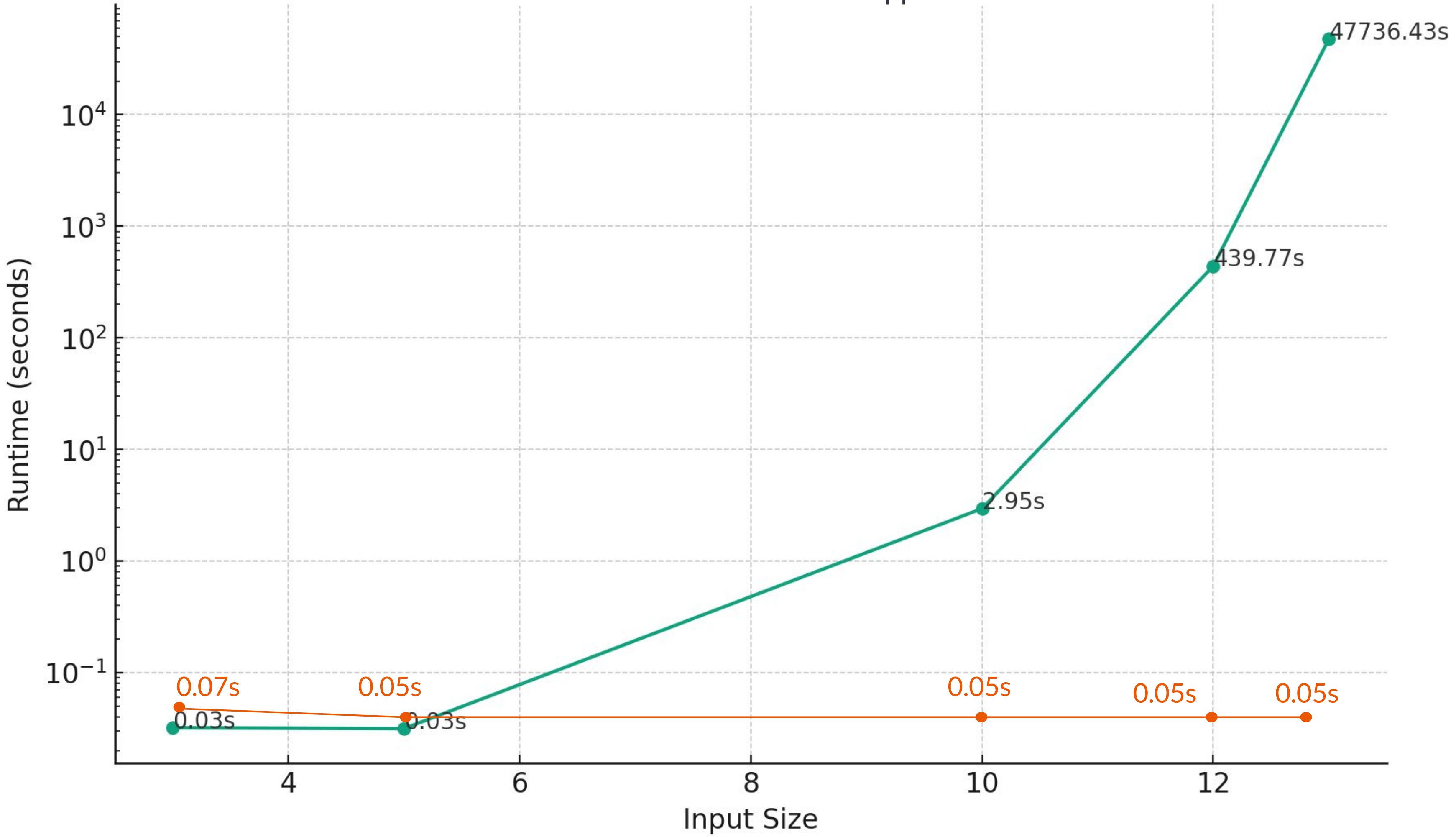
Optimal (Visited List = [A,B,D,E,C,A]
Total Length = 28

Nearest Neighbor Runtime Complexity

Nearest Neighbor has a worst case time class of $O(V^2)$

This is due to it going through all vertices and each edge from that vertex to find the smallest edge to a new vertex that is not in Visited.

Runtime Performance Exact and Approximate



Approx Solutions against Exact

	Exact Result	Approximation Result
Test 1	23	23
Test 2	28	(28,30) Can Vary
Test 3	19	(23,25,26,32,33) Variable
Test 4	24	(27,29,34,36,38,41) Varies
Test 5	33	(36,37,39,40,46,48) Varies

Approx Result Varies as a result of random initial vertex selection.

Lower Bound Analysis Nearest Neighbor

Lower Bound is the weight of the Graph's MST, and we can compare that to our own. We will analyze test case 5 with the MST, Exact, and Approx.

Test 5	Weight	Delta
MST	27	n/a
Exact Solution	33	6
Nearest Neighbor	(36,37,39,40,46,48)	(9,10,12,13,19,21)