Longest Path

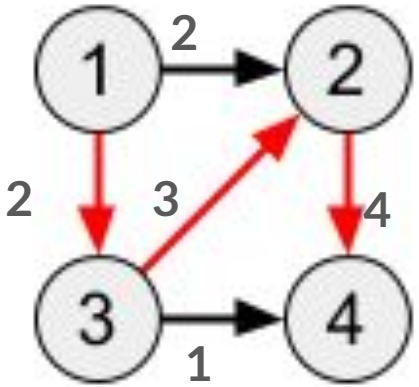
Nate Baker, Ellona Macmillan, Will Keppel, Ben Glass

Exact Solution

Ben Glass

Problem description and Practical Applications

- Given any weighted, directed graph, find the longest (heaviest) simple path in the graph.
- The path must be legal:
  - Each vertex connects to the last
  - No repeated vertices
- Practical applications:
  - Find the longest possible route on a map.
  - Find the way to use the most amount of a resource
  - Find the longest set of connections in a system:
    - DNA sequencing
    - Public transportation



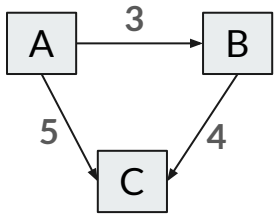
Details on Inputs/Outputs

Input:

- A graph
- First line: The number of vertices and edges in the graph (integer: number_V, integer: number_E)
- Every line after: An edge in the graph, (character: u, character: v, integer: w)

Example:

3 3
a b 3
b c 4
a c 5

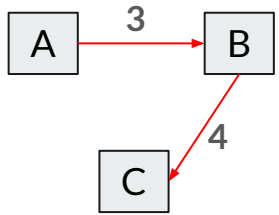


Output:

- First line: The integer weight of the longest path.
- Second Line: The longest path as a sequence of space-separated characters.

Example:

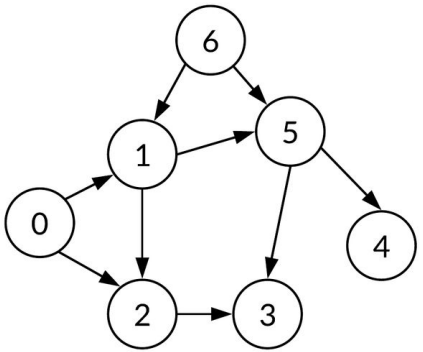
7
a b c



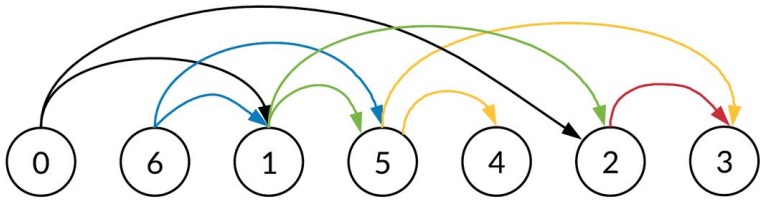
A restriction on the problem that makes it possible to compute the answer in polynomial time

- Restrict the input to weighted DAGs only
- A cycle-free graph can be sorted in order (topologically) O(V+E)
- For each vertex, track its current distance and predecessor O(V)
- Find the longest path to each vertex by relaxing tense edges (opposite of SSSP tense) O(VE)
    - If dist(v) < dist(u) + weight from u⟶v, then it is tense
- Remember the vertex with the longest distance, and its path backwards until it cannot go backwards O(E)
- Total: O(VE)

Unsorted graph



Topologically sorted graph



Approach for solving

High level algorithm steps:

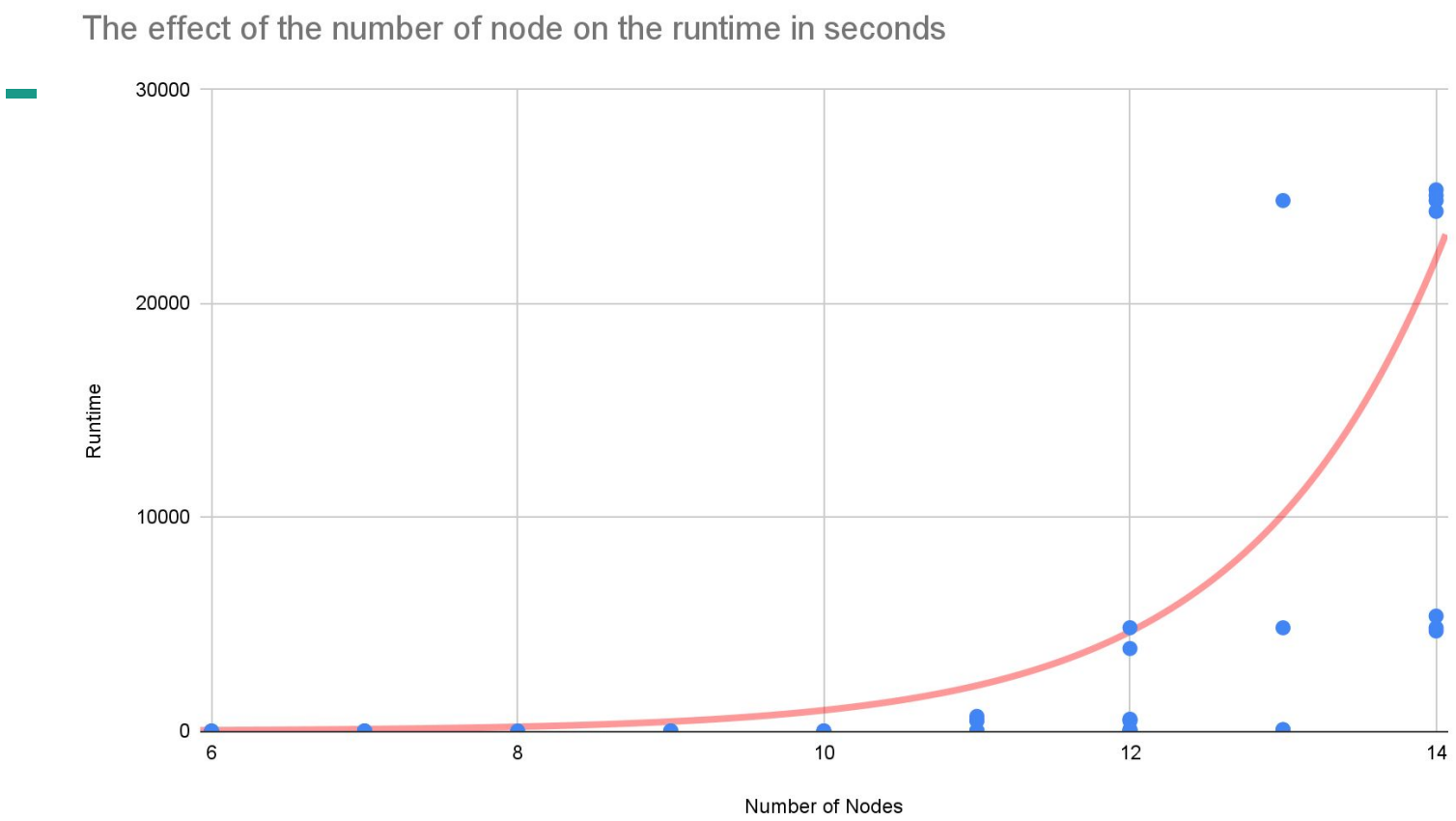
1. Read in the graph as an adjacency list
2. Use itertools to get a permutation of every vertex.
   a. Every possible length (2 to V).
3. Zip the permuted vertices into a list of edges. (pairwise function)
   a. [A, B, C, D] → [(A,B), (B,C), (C,D)]
4. Check every single path for legality:
   a. The vertices connect and are in sequential order
5. If the path is legal and bigger than the current maximum length path, it becomes the new max.
6. Traverse the resulting longest path and output its contents.

```python
def find_path():
    # longest path weight, longest path
    biggest = [float('-inf'), None]
    for path_length in range(2, len(graph) + 1):
        # Permuted vertices --> no repeated vertices --> no cycles
        for combo in itertools.permutations(graph.keys(), path_length):
            combo_weight = 0
            still_legal = True
            this_path = []

            for u, v in pairwise(combo):
                if debug:
                    print(f"testing ({u}, {v})")

                # Check for legality
                if v in graph[u]:
                    combo_weight += graph[u][v]
                    this_path.append((u, v))

                else:
                    still_legal = False
                    break

            # Maintain the biggest legal weight
            if still_legal and (combo_weight > biggest[0]):
                biggest[0] = combo_weight
                biggest[1] = this_path
    return biggest
```

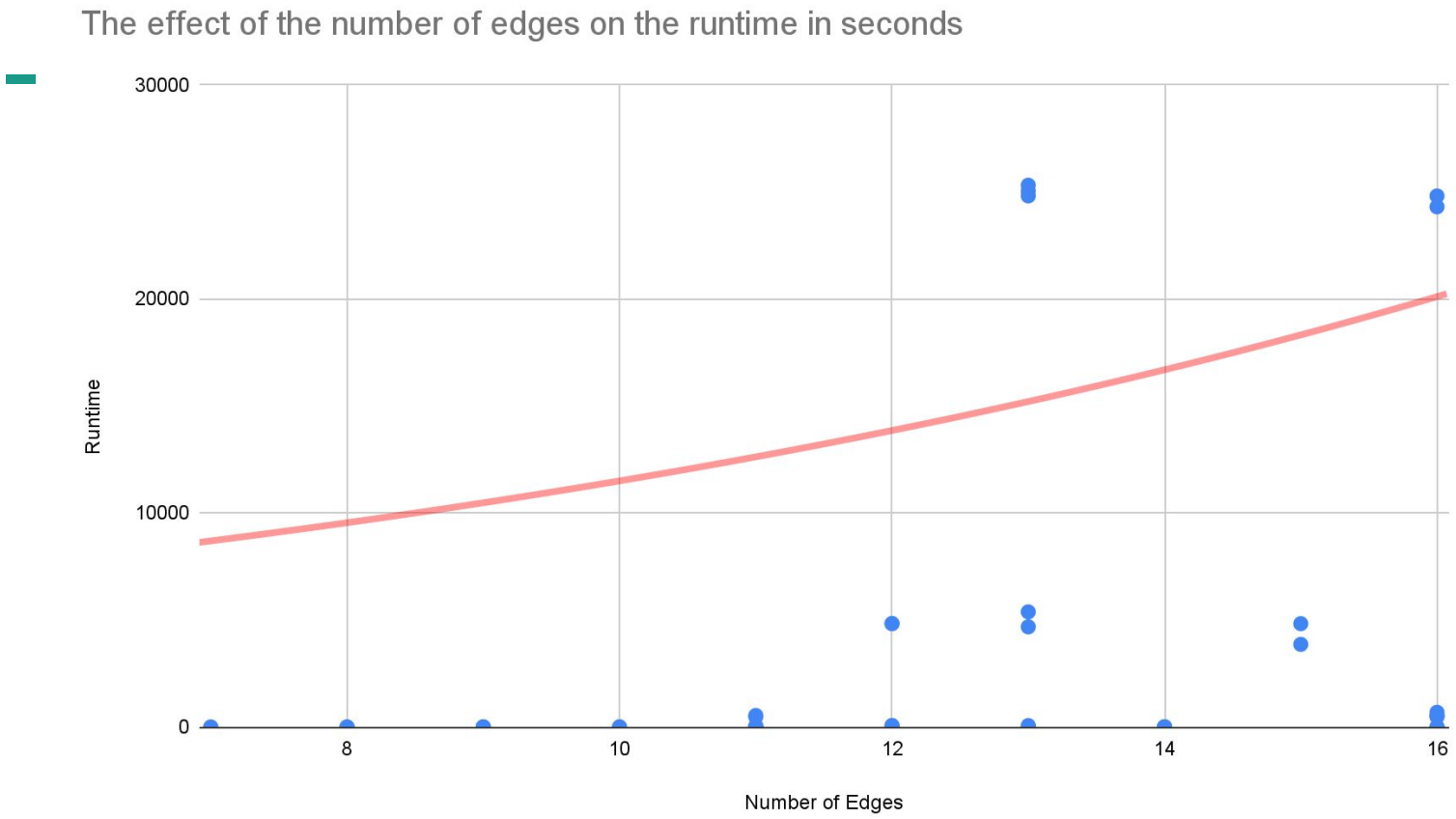
Analytical runtime

- Build the graph from input: O(V + E).
- **For every possible path length: O(V)**
  - **For every vertex permutation of that length O(V!)**
    - **Zip consecutive vertices into edges. O(V)**
    - **Check each path for legality. O(V)**
  - Update the longest path if necessary O(1).
- Traverse and output the longest path (could be up to V long):
- Total runtime: **O(V^2 * V!)**

Runtime plot: Number of Nodes



Runtime plot: Number of Edges



Can changing the input order make it run faster?

- Since every run permutes every length of the vertices vertices, it is very challenging to optimize the algorithm based on ordering.
- Since it always looks at every permutation before deciding the longest path, changing the input order would not make it run faster.
- The algorithm would have to be fundamentally changed for the input order to affect its runtime.

(These loops have to run all the way through, without breaking)

```
for path_length in range(2, len(graph) + 1):
    # Permuted vertices --> no repeated vertices --> no cycles
    for combo in itertools.permutations(graph.keys(), path_length):
```

Reduction from a known NP-Hard problem

Reduce the **Hamiltonian Path** problem to The **Longest Path** problem.

- Goal: Given a directed graph, determine if there is a hamiltonian path within it.
- Polynomial-time reduction:
  - Take in the graph.
  - Convert each undirected edge to two directed edges. O(E)
  - Set every edge weight to 1. O(E)
  - V ← Remember the number of vertices in the graph.
  - Feed the graph into the longest path algorithm.
  - If longest path returns a path with a weight **equal to V-1**, then there is a hamiltonian path within the graph. Return true.
    - Else return false
- This works because longest path and hamiltonian path both only allow simple paths.
- Total time for the reduction step is O(E)

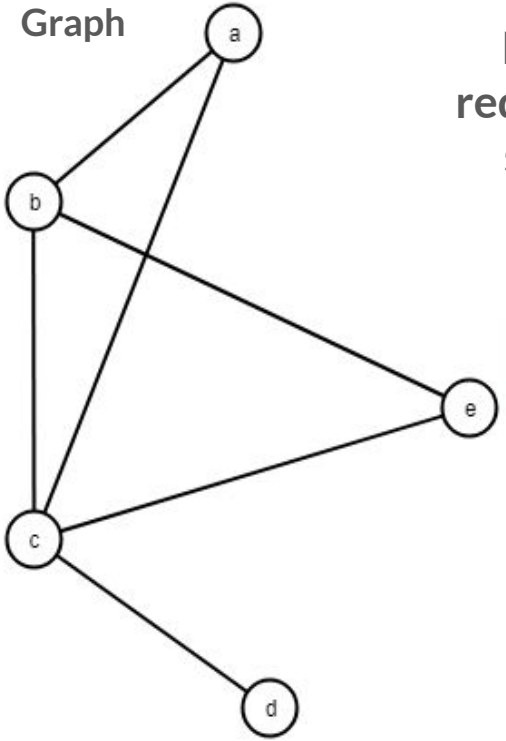
The Longest Path algorithm outputs a Hamiltonian path **IFF** one exists.

- **=> If G has a Hamiltonian path:**
  - A hamiltonian path has exactly V-1 edges.
  - This is the maximum possible weight of any simple path in G.
  - Weight = number of edges (all weighted 1).
  - **The LP algorithm must return a path of weight V-1**.
- **<= If the LP algorithm returns a weight of V-1:**
  - Weight = number of edges (all weighted 1).
  - Having V-1 edges means it visits all vertices and is by definition a Hamiltonian path.
  - Therefore, **the original graph has a hamiltonian path.**

Since, the Hamiltonian path problem is known to be NP-Hard, computing the longest simple path in a directed graph is NP-Hard.

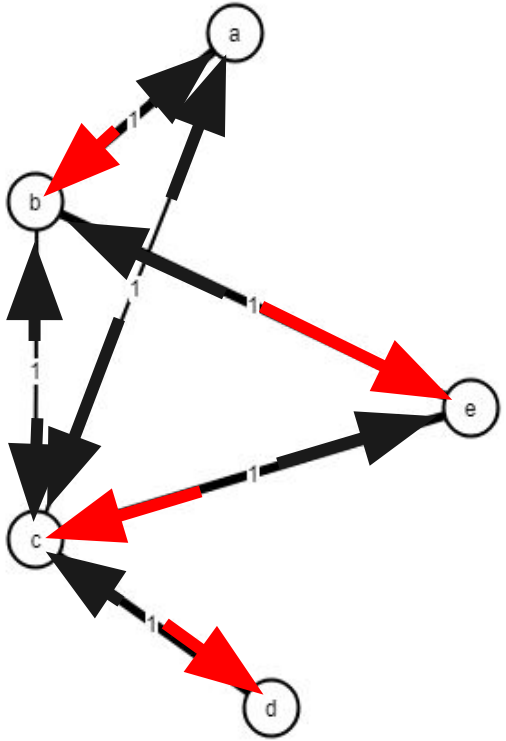
Reduction from a known NP-Hard problem visual

Original Graph



Main reduction step:

LP problem



V - 1 == 4
AND
LP == 4
Therefore
Hamiltonian
Path

Approximation 1

Will Keppel

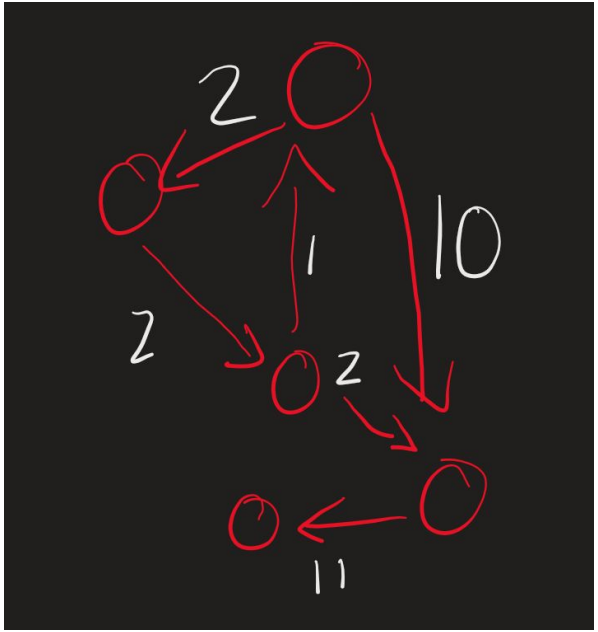
1st Basic Implementation -> $O(V^2)$ + some nonsense

1. Create Graph ***O(V + E)***
2. Sort Edges by weight ***O(ElogE)***
3. Make Max Spanning Tree using Modified Boruvka's ***O(EV)***
    a. Add the max weight that remains acyclic
    b. Could be really bad for dense graphs (O(V^3) worst case of every node connected(but almost never kinda like quicksort))
4. DFS with Max Weight Heuristic (Greedy) ***$O(V^2)$***
    a. For every node, check all of its paths
    b. Keep best overall
        i. random() < 0.5 Tie breaker
    c. If time runs too long, returns None, breaking the dfs stack

Why this didn't work

1. It sucked
2. It pruned too many edges because the goal of any MST is that it is exactly V-1 edges
3. Instead, I take the top % of edges that are unproblematic

Something that still is a problem

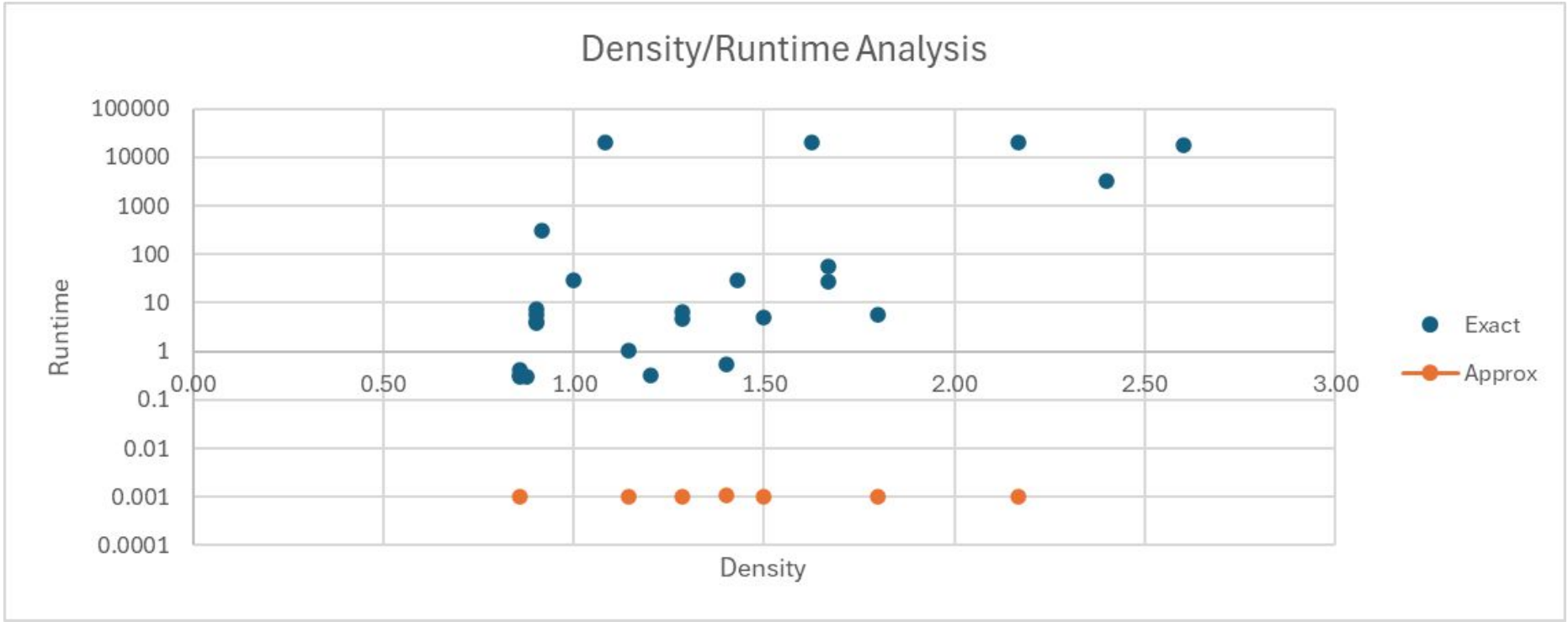


2nd Try -> O(E(E+V))

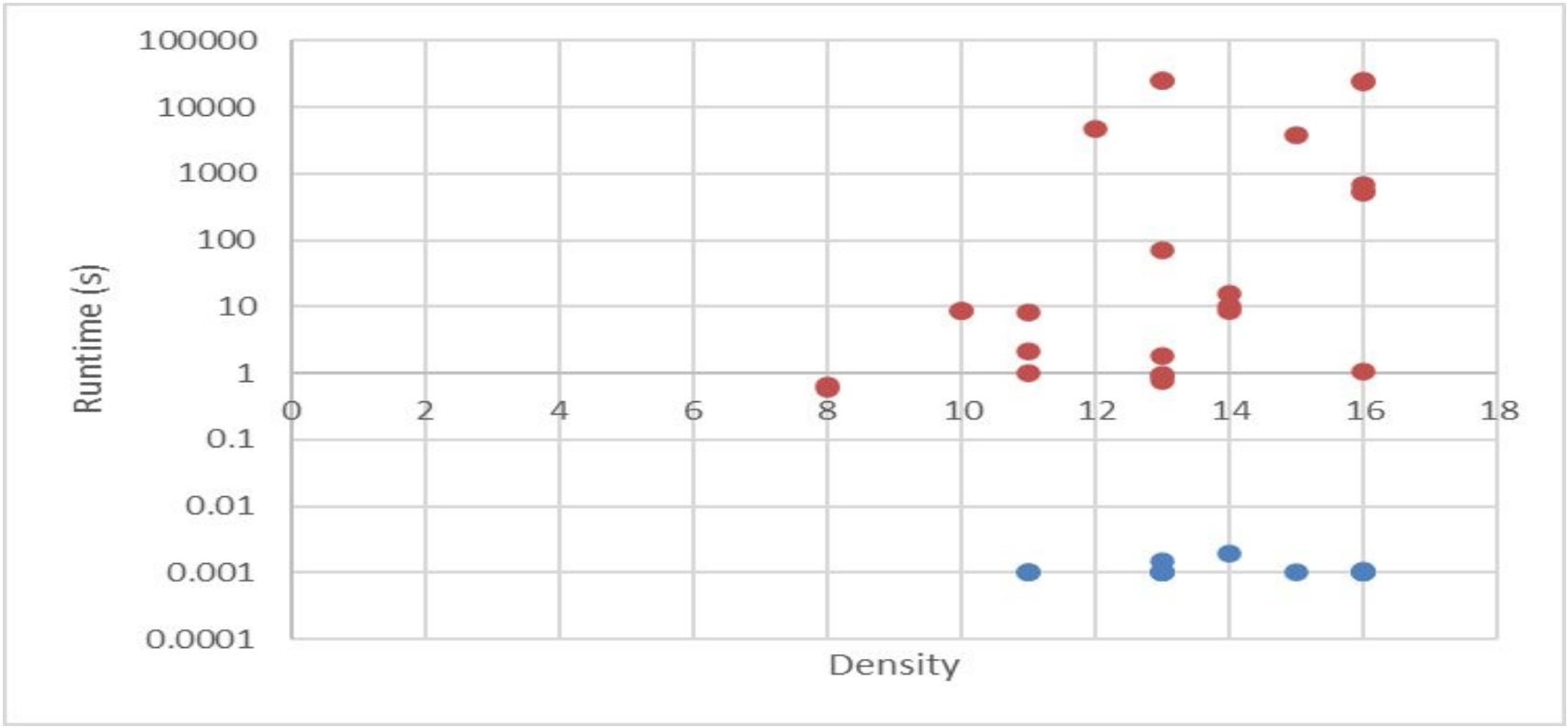
1. Create Graph from input O(E)
2. Make Directed Acyclic Subgragh O(E(E+V))
    a. Try to keep as many edges as possible while acyclic
    b. Detecting cycles = O(V+E) ->every edge could be a cycle
3. Beam search with width=15 O(B*V*D)
    a. Runtime = beam width * V * Average degree/density of graph (Almost certainly shorter than making graph)
    b. Works sort of like dfs but you keep a maximum number of paths you can evaluate at any time

Graphs

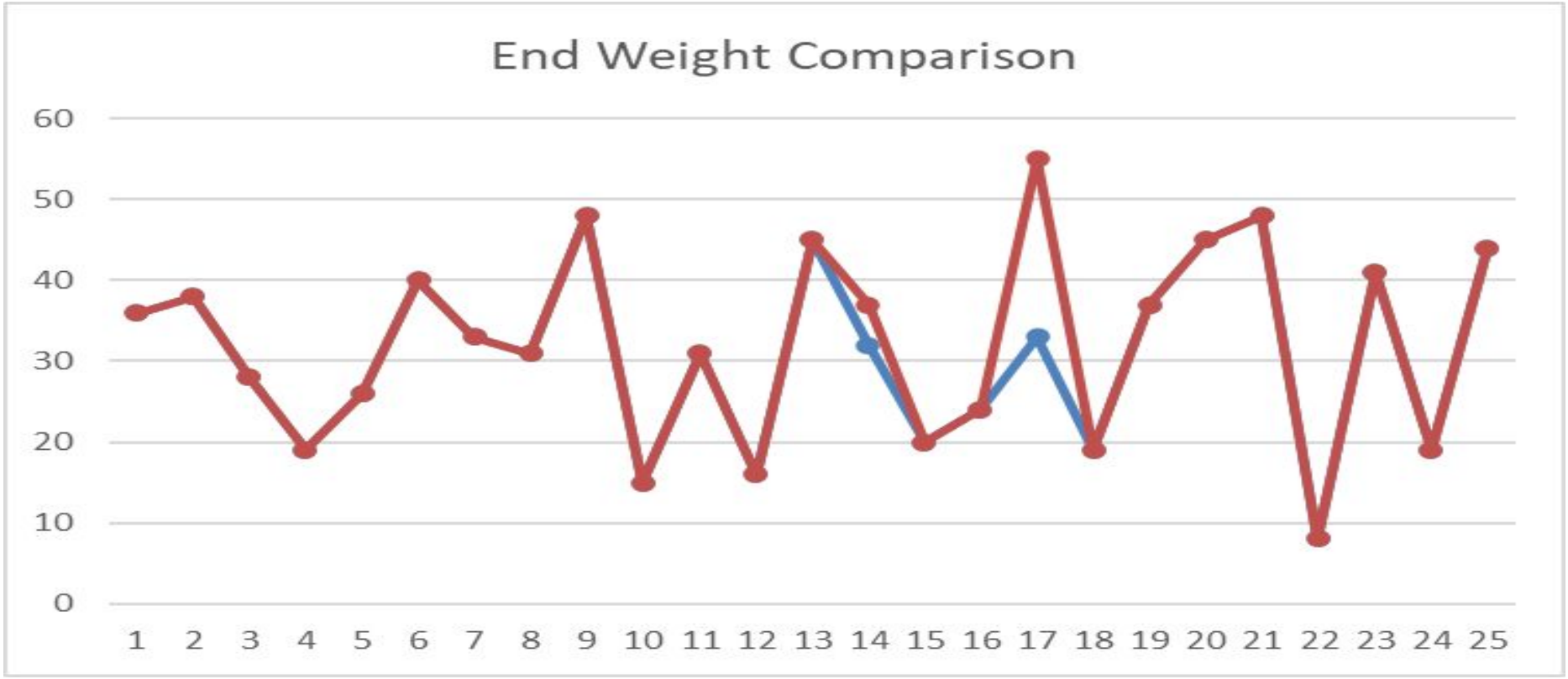
4/25 were worse than the exact



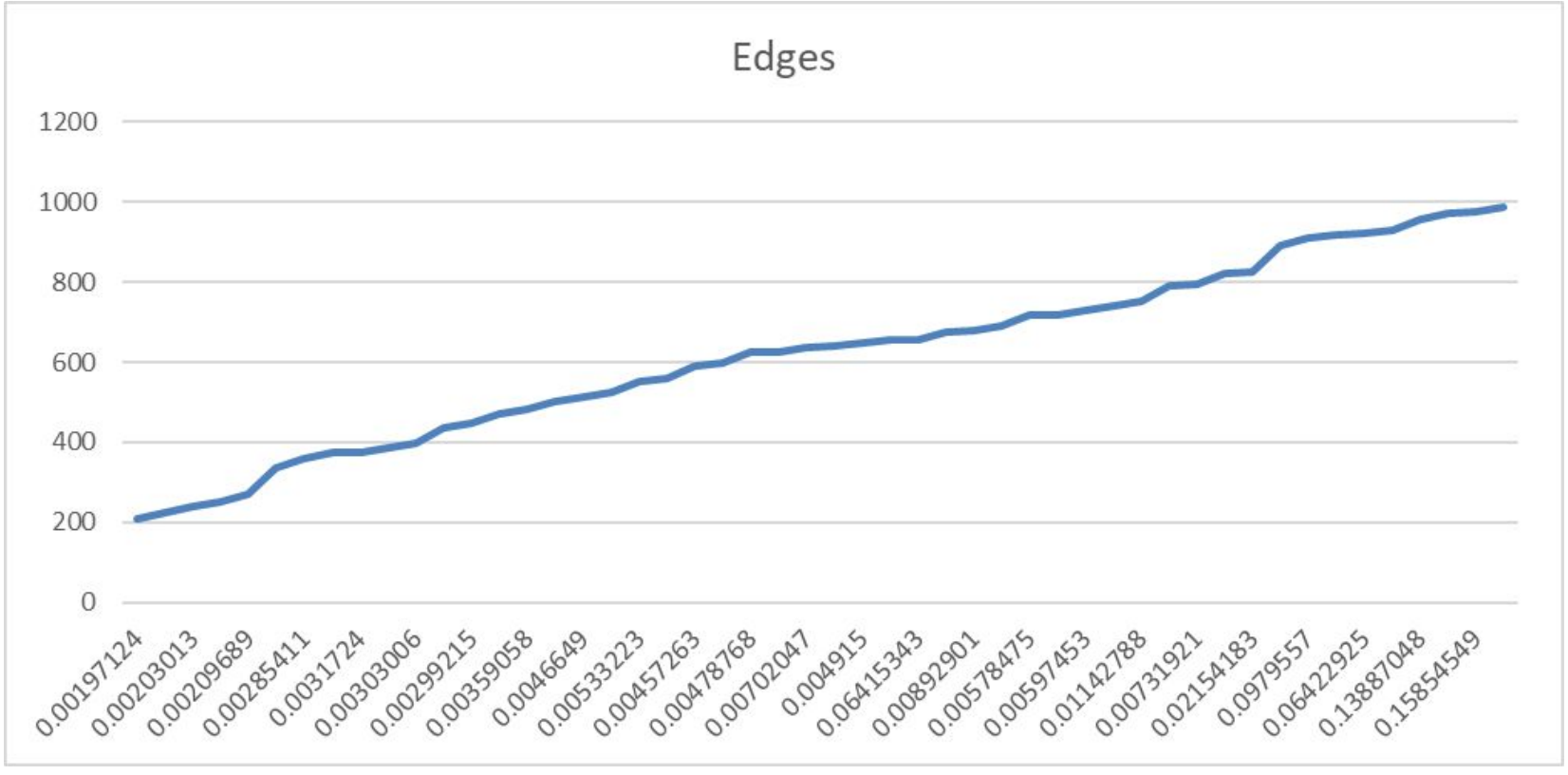
Graphs


100000
10000
1000
100
10
1
0.1
0.01
0.001
0.0001
Runtime (s)
0
2
4
6
8
10
12
14
16
18
Density

Graphs


End Weight Comparison
60
50
40
30
20
10
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Graphs


Edges
1200
1000
800
600
400
200
0
0.00197124
0.00203013
0.00209689
0.00285411
0.0031724
0.00303006
0.00299215
0.00359058
0.0046649
0.00533223
0.00457263
0.00478768
0.00702047
0.004915
0.06415343
0.00892901
0.00578475
0.00597453
0.01142788
0.00731921
0.02154183
0.0979557
0.06422925
0.13887048
0.15854549

My Approximation

Lower Bounds - The edge with the highest weight

+ The longest path must include the heaviest edge or something worse

Upper Bounds - The sum of the heaviest V-1 edges

+ The longest path super path should include all nodes, and all heaviest edges

Approximation 2

Nate Baker

Whatever-First-Then-Second-Then-Third-Then-So-On Search

- The idea is simple: guess.
  - Like really truly that simple

<u>The Algorithm:</u>

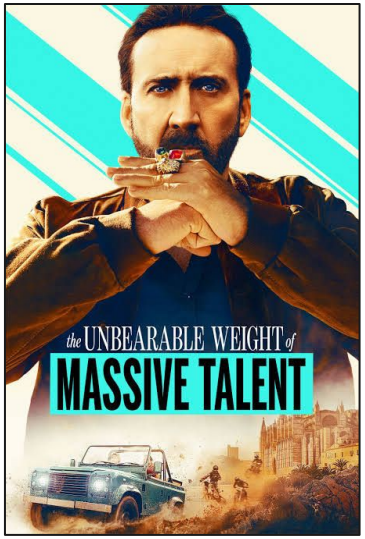
- Pick a random starting vertex
- Choose an edge to continue along (weighted randomness by edge weight, n<=0 has weight 1)
- While it is possible to continue (ie. at least one outgoing edge exists that doesn't lead to a visited vertex):
  - best = max(best, current path)
  - Choose an edge to follow (same randomness scheme)

Analysis

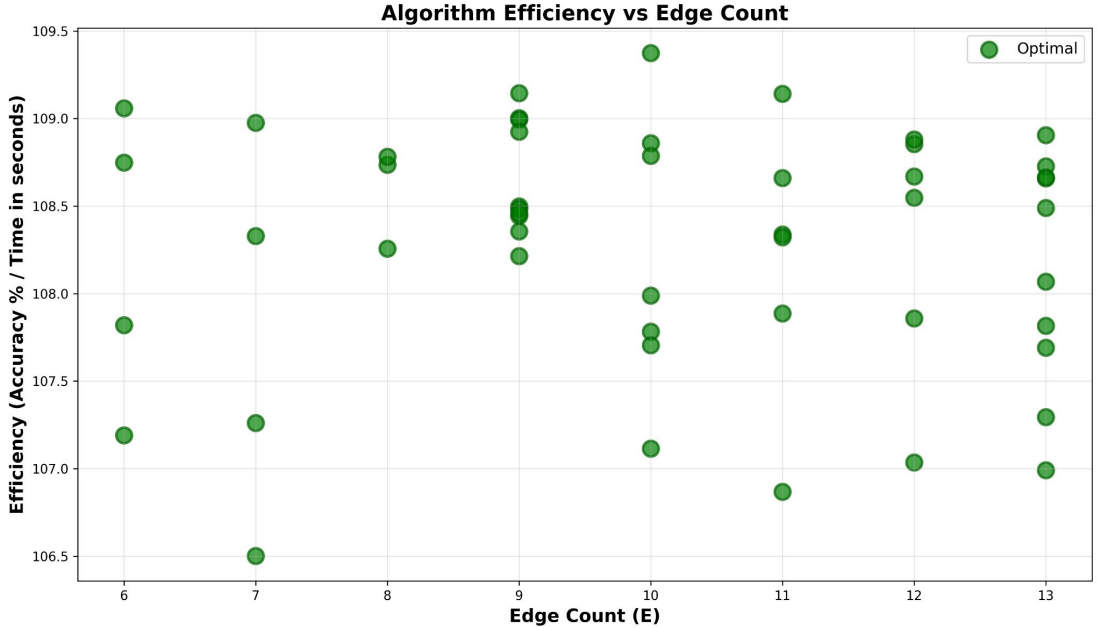
- This algorithm is just a modified traversal with no introduced asymptotic overhead, so O(V+E).
- By greedily utilizing the random edge weight heuristic, the choices made will tend to result in longer candidate paths
- The randomness guarantees that the ideal path will eventually be found (given infinite time ☝🤓)

Searching for Failure

- For all cases that the exact solver was run on (max 16 edges), this approximator would find the ideal when run for 1 second
- Updated the time parameter to accept floats
- The results:
  - .5s buffer for joining threads
  - ~.3s to start simulating (thread spinup and input parsing)
  - In **.1s**, the approximator finds an ideal solution for each of the “exact” test cases
  - I gave up here

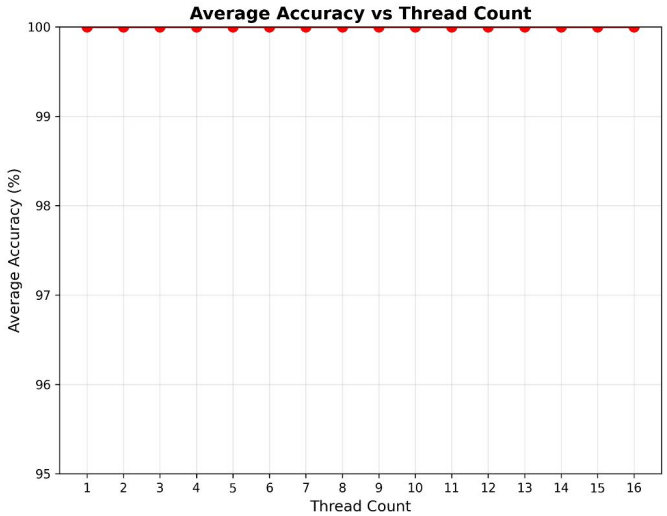


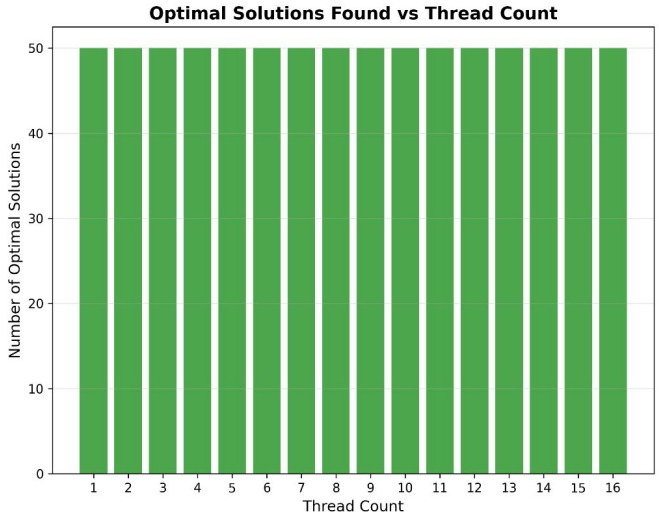
Accuracy Graph



Parallelization

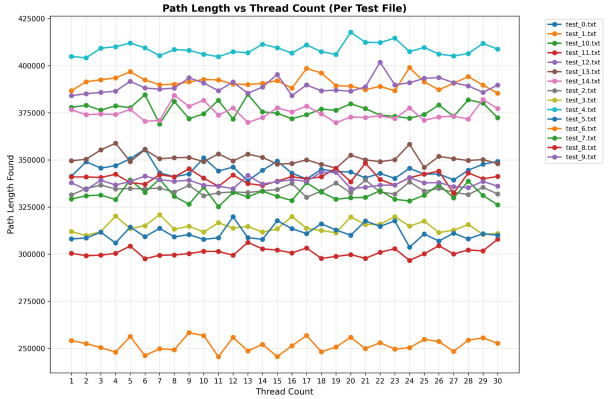
1 second of runtime

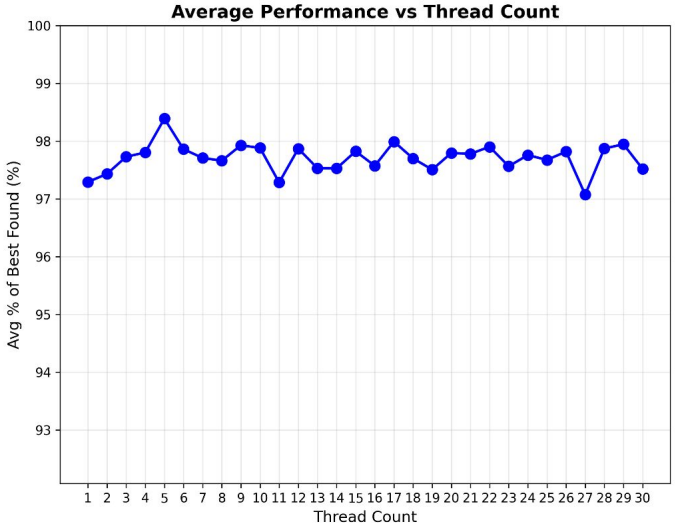


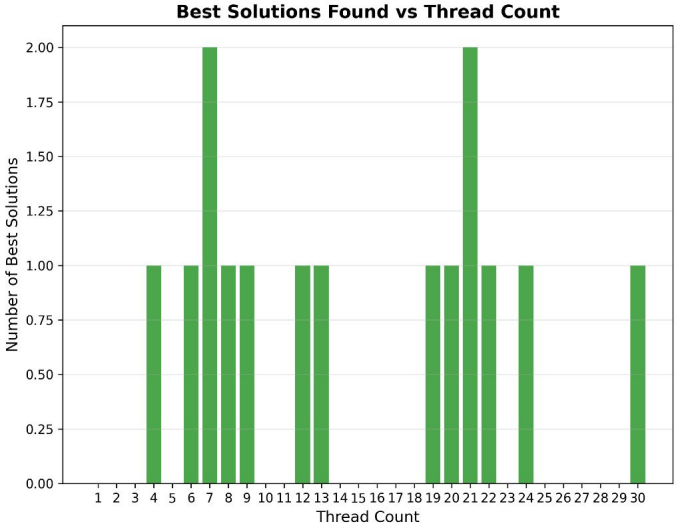


Graphs

1 secs, E in [5000,20000], 10 cases

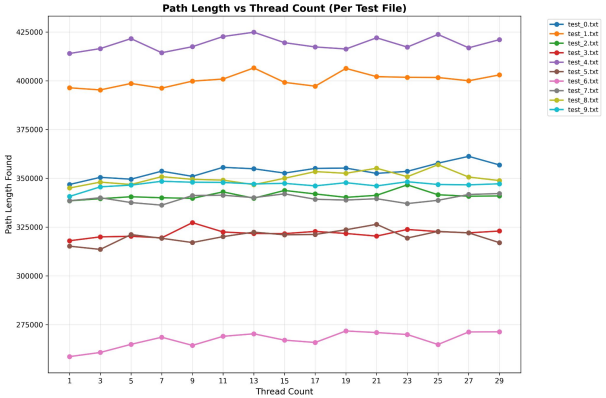


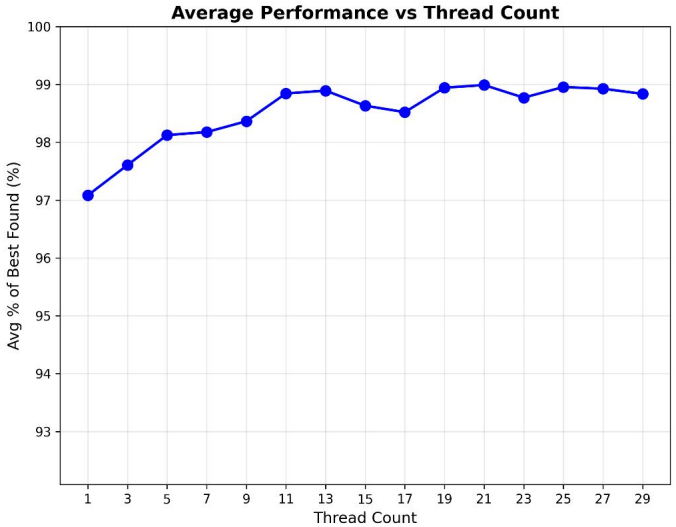




Graphs

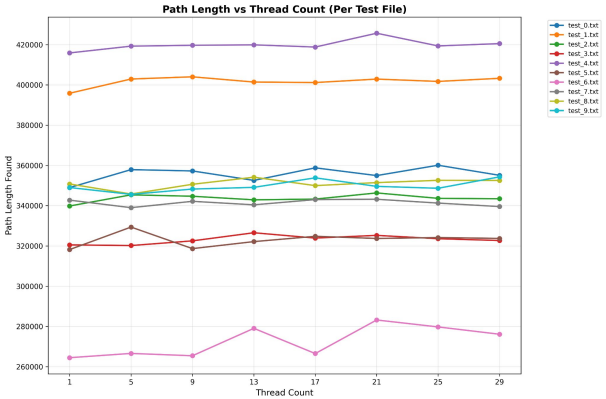
5 secs, E in [5000,20000], 10 cases

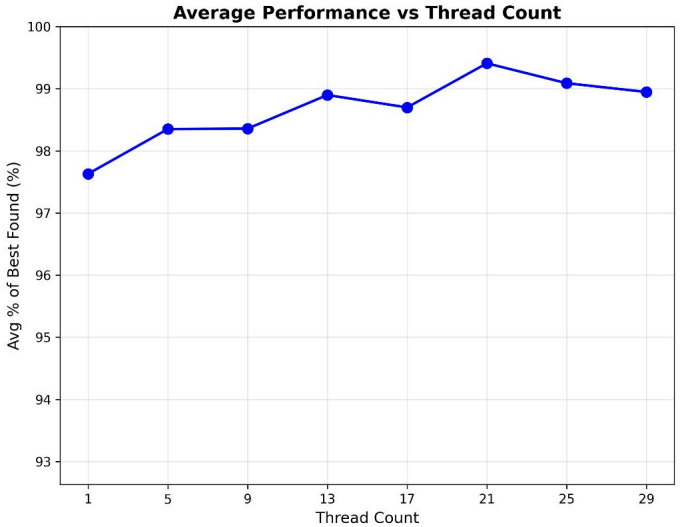


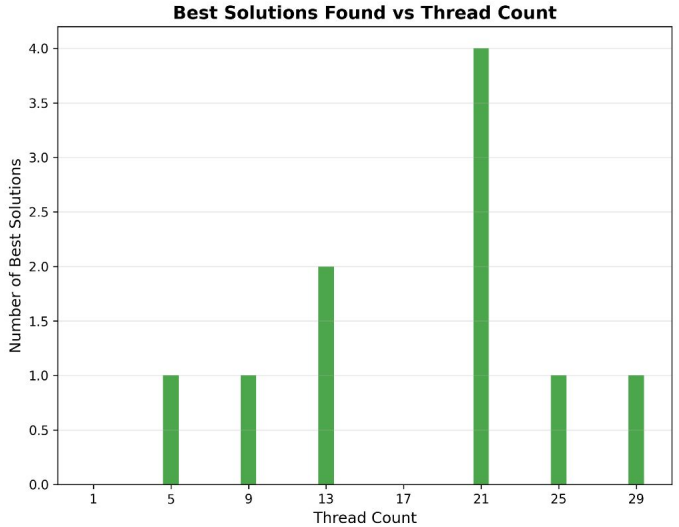


Graphs

10 secs, E in [5000,20000], 10 cases

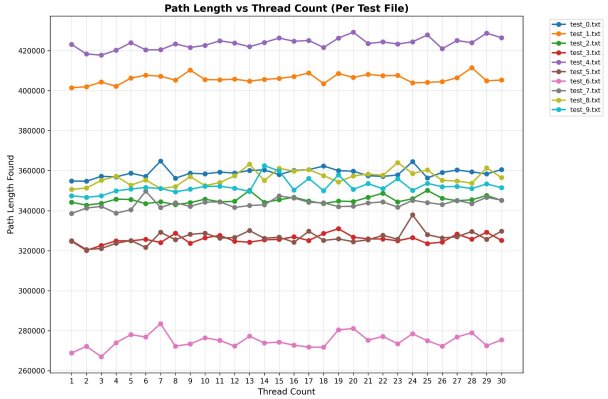


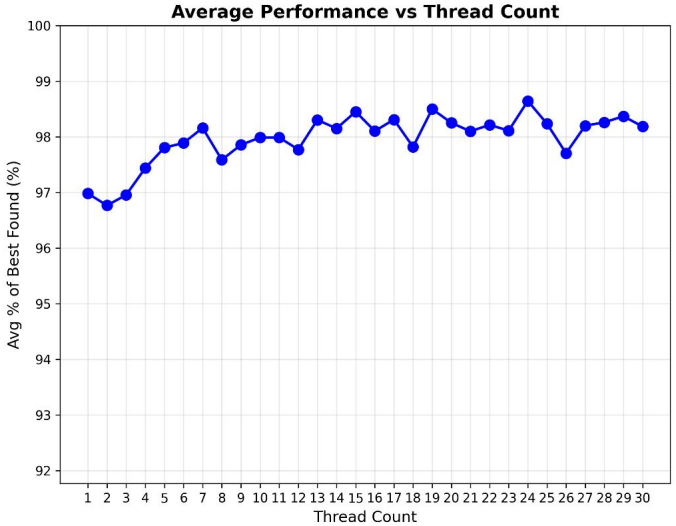


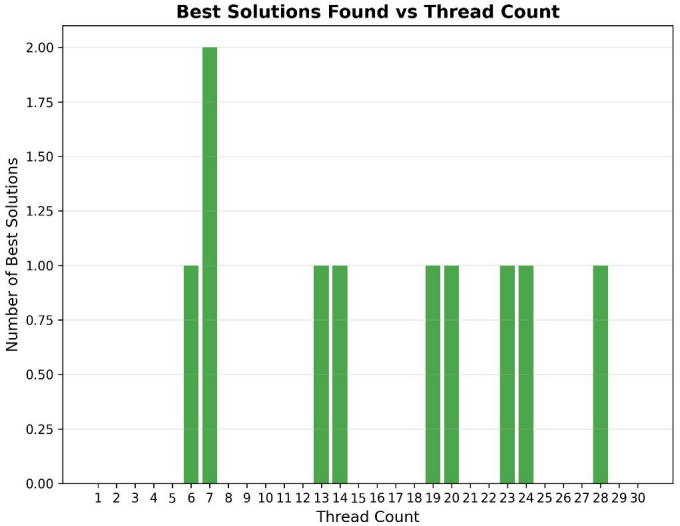


Graphs

45 secs, E in [5000,20000], 10 cases







Reduction and Bounds

Ellona Macmillan

Reduce Hamiltonian Path Prob. -> Longest Path Prob.

Input for HAM: Undirected, weighted graph

Input for LP: Directed, weighted graph

Reduction:

- Edit the input to be bidirectional (have each edge twice)
- Give each edge a weight of 1

Initial problem: finding a hamiltonian path

A Hamiltonian Path is a path that visits each vertex in a graph exactly once.

- A path that visits V vertices will contain V - 1 edges making the **length of the path V -1**

Weighted Graph Problems (TSP, Longest Path)

The input is a weighted graph specified by a line containing the number of vertices *n* and the number of edges *m* followed by *m* lines containing the edges given in *u v w* format, representing an edge between *u* and *v* of weight *w*. TSP graphs are undirected and edges will be listed only once and the graph will be a complete graph. LP graphs are directed.

The output contains two lines: the length of the path on one line (as an integer) followed by a list of vertices for the path/cycle on the second line.

Bounds

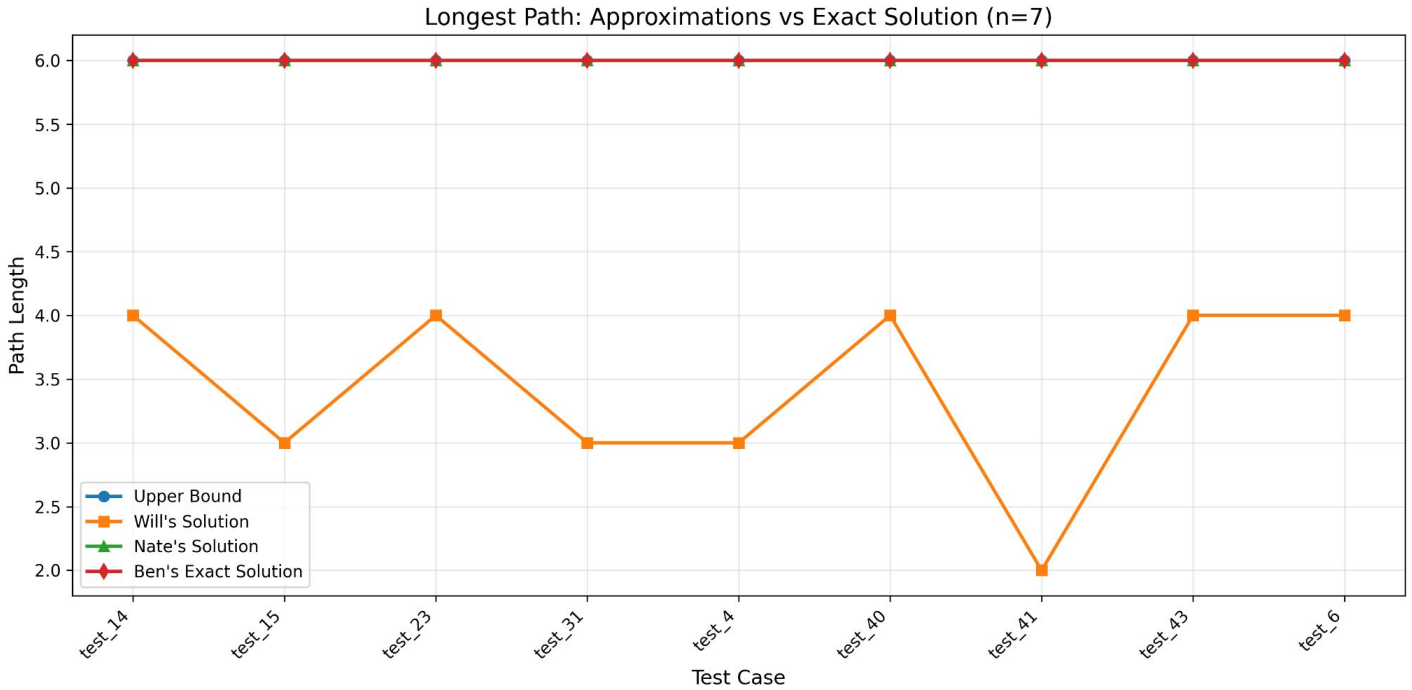
Approach I chose:

Use max number of edges and max edge weight

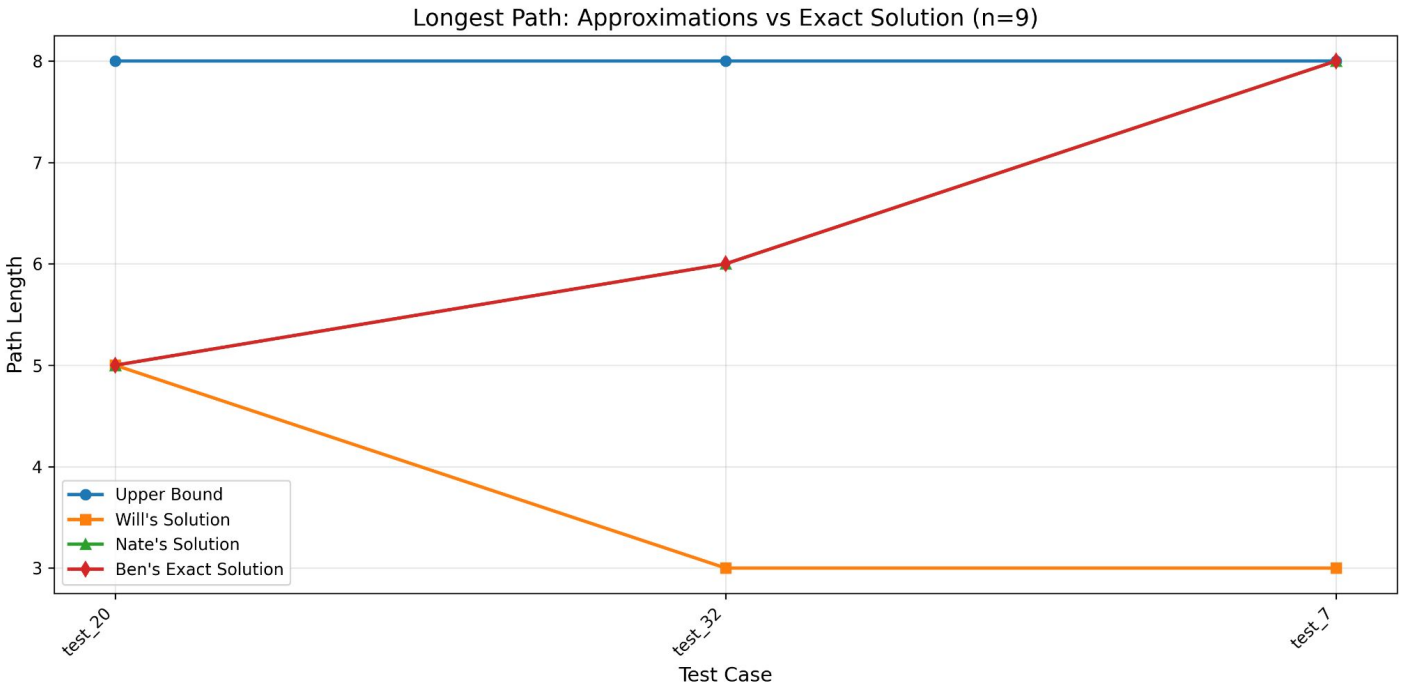
Upper bound:

(V - 1) x W (with W be the maximum weight of any edge in the graph)

~~Graph~~ Figures



Figures cont.



Figures cont. cont.