Dijkstra's Algorithm

Different implementations Analysis

Nathan, Shun Kah, Huai Zhi

A

Implementation 1

Dijkstra with adjacency matrix and array priority queue

B

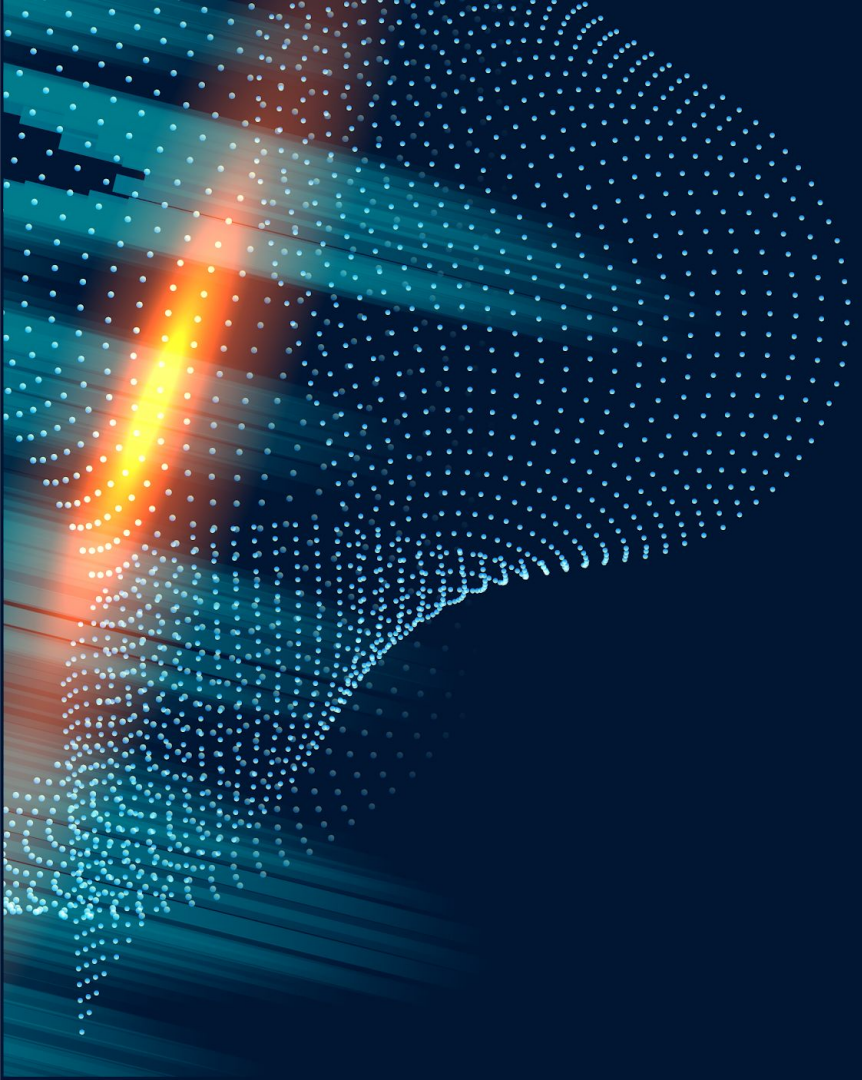
Implementation 2

Dijkstra with adjacency list and minimising heap priority queue

C

Which is better

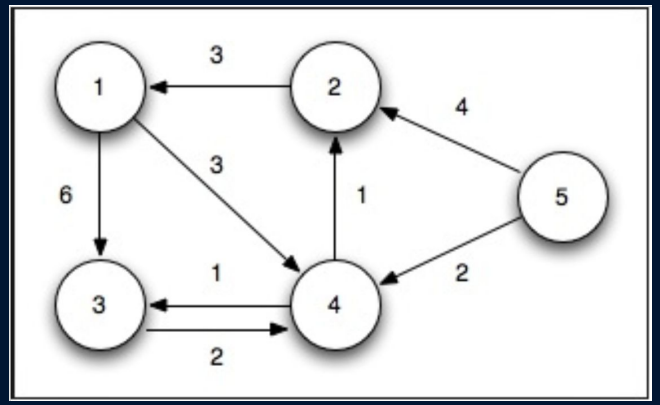
Comparison with both implementation



Part A

Adjacency Matrix with Array as Priority Queue

Explanation (General Concept)



Graph

0	0	6	3	0
3	0	0	0	0
0	0	0	2	0
0	1	1	0	0
0	4	0	2	0

Adjacency Matrix

```
[(8.0, 1), (8.0, 2), (1.0, 6)]
[(8.0, 1), (8.0, 2), (2.0, 1), (7.0, 3)]
[(7.0, 3), (8.0, 1), (8.0, 2), (10.0, 4), (4.0, 5), (12.0, 7)]
[(7.0, 3), (8.0, 1), (8.0, 2), (10.0, 4), (12.0, 7), (5.0, 2), (14.0, 9)]
[(7.0, 3), (8.0, 1), (8.0, 2), (10.0, 4), (12.0, 7), (14.0, 9)]
[(8.0, 1), (8.0, 2), (10.0, 4), (12.0, 7), (14.0, 9), (17.0, 8)]
[(8.0, 2), (10.0, 4), (12.0, 7), (14.0, 9), (17.0, 8)]
[(10.0, 4), (12.0, 7), (14.0, 9), (17.0, 8)]
[(12.0, 7), (14.0, 9), (17.0, 8)]
[(14.0, 9), (17.0, 8), (13.0, 8)]
[(14.0, 9), (17.0, 8)]
[(17.0, 8)]
```

Priority Queue

Explanation (Pseudocode)

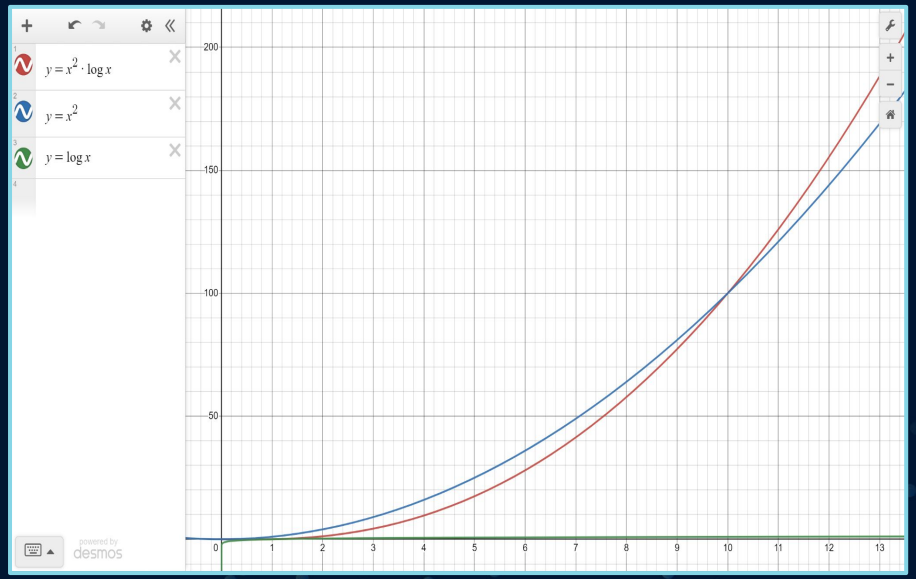
- For each vertex v
  - Set distance[v] = infinity
- Set distance[source] = 0
- Put all vertices into priority queue Q, sorted by the distance of v in ascending order
- While Q is not empty
  - Set u to be cheapest of Q
  - For each vertex v adjacent to u
    - If distance of v is more than distance of u + weight of u to v
      - Update distance of v to be distance of u + weight of u to v
      - Insert v into Q according to its d[v]

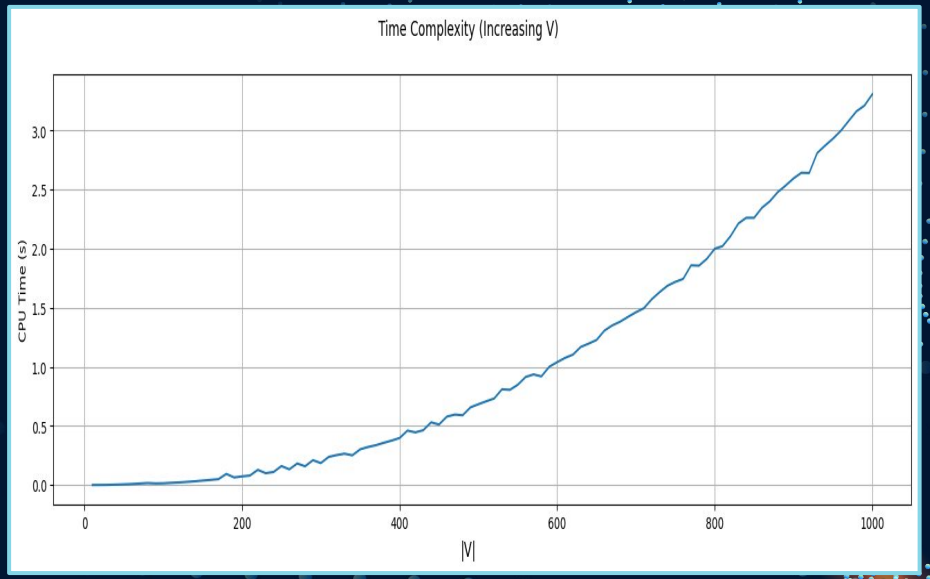
Time Complexity (Theoretical)

- Adding all vertices to Q → $O(|V|)$
- Looping through priority queue → $O(|V|)$
  - Sorting Q in ascending distance → $O(|V| * \log(|V|))$
  - Removing node with minimal distance → $O(1)$
  - Looping through adjacent nodes → $O(|V|)$
  - Recalculate distance and update Q → $O(1)$

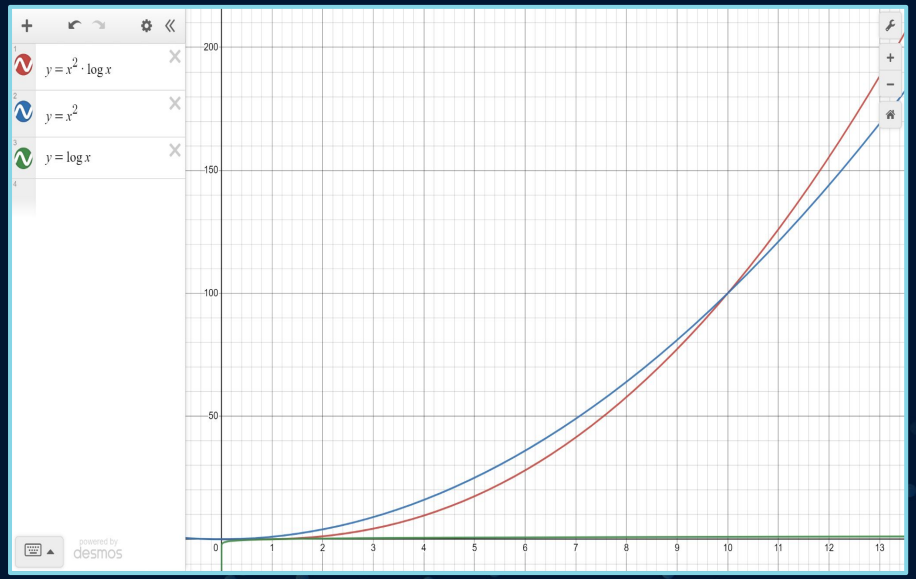
Total time complexity → $O(|V|) + O(|V|) * (O(|V| * \log(|V|)) + O(|V|)) = O(|V|^2 * \log(|V|))$

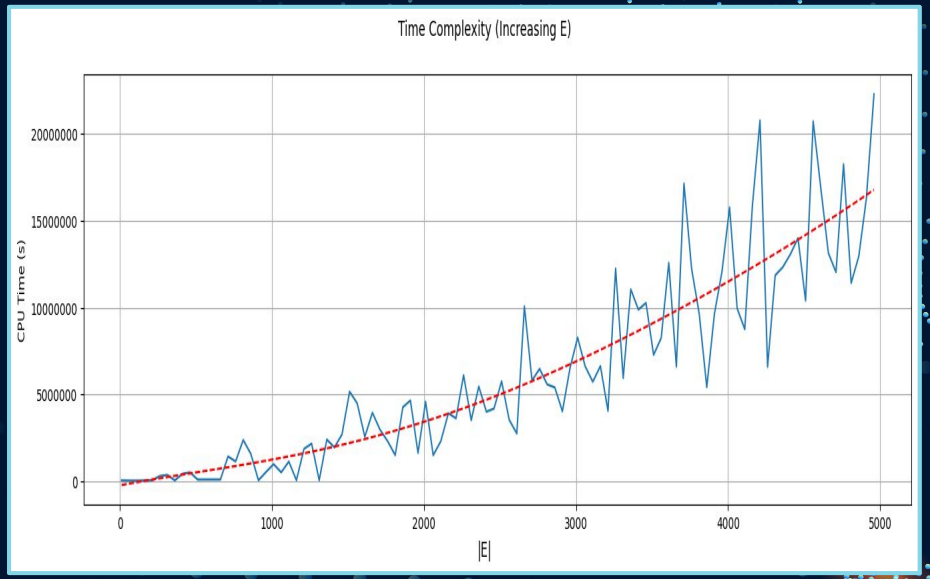
Time Complexity (Empirical)

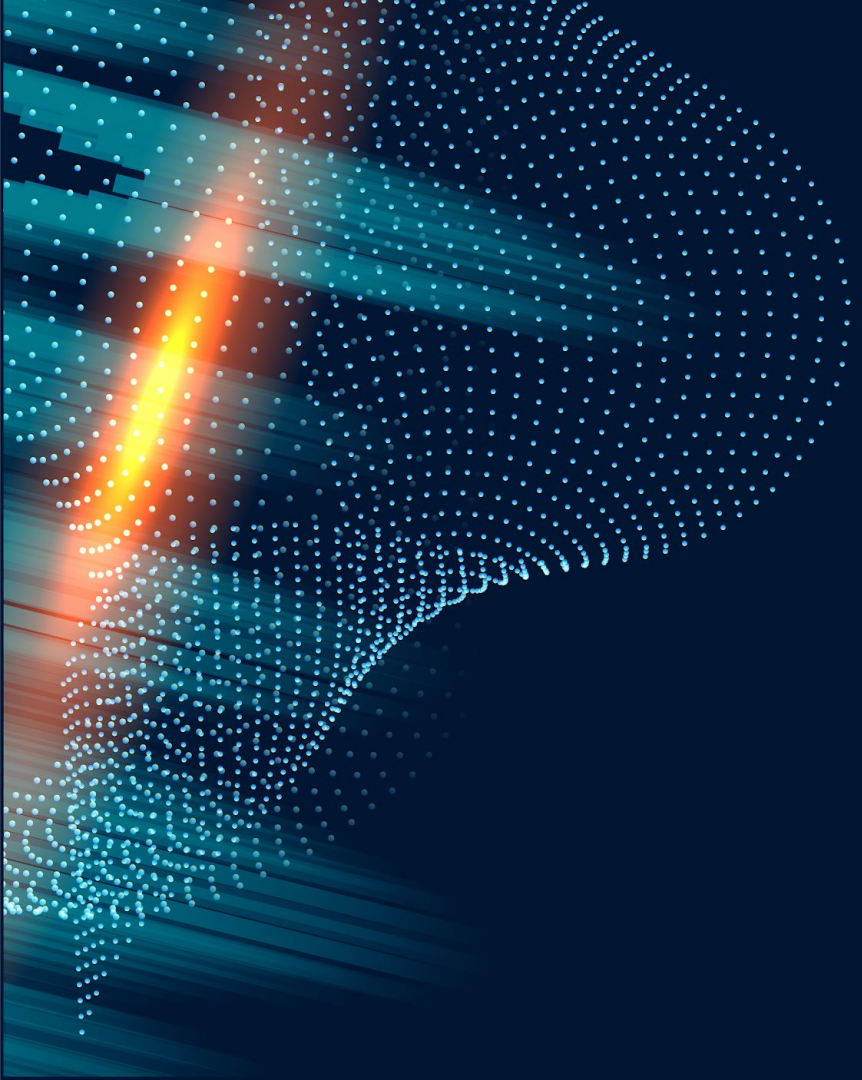




Time Complexity (Empirical)



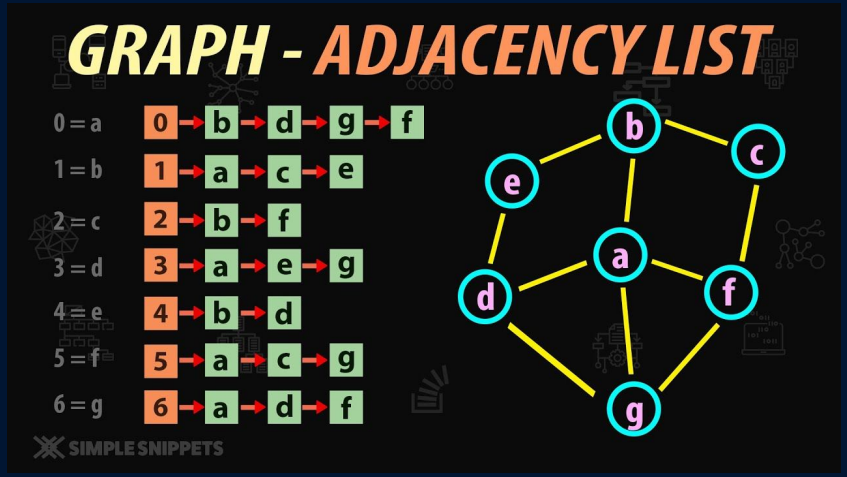




Part B

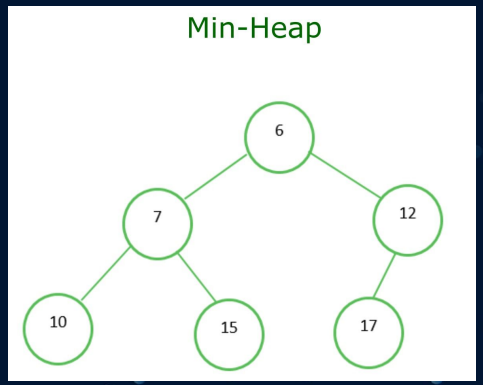
Adjacency List with Min Heap Priority Queue

Explanation (General Concept)




```
Sample graph:
0 -> [1]
1 -> [0, 2]
2 -> [1, 3]
3 -> [2, 4]
4 -> [3, 5]
5 -> [4, 6]
6 -> [5, 7]
7 -> [6, 8]
8 -> [7, 9]
9 -> [8]
[[(1, 11)], [(0, 11), (2, 6)], [(1, 6), (3, 2)],
```

(vertex, weight)



Sorted based on:

Explanation (Pseudocode)

- Initialise a min heap for the priority queue
- Push source vertex with distance 0 into the priority queue
- Create distance array based on number of vertices in the graph, all initialised to infinity
- Initialise distance from source vertex as 0
- While Priority Queue is not empty:
    - Pop first vertex from priority queue (the item with the shortest distance to source vertex)
    - FOR all the adjacent vertices to that popped vertex:
        - Check if Distance from an adjacent vertex to source vertex is smaller than currently recorded distance in distance array + source vertex is in min heap priority queue
        - If so, update distance with smaller amount
        - Push the adjacent vertex and new distance into priority queue

Time Complexity (Theoretical)

- Initialise a min heap for the priority queue O(VlogV)
- Add source vertex with distance 0 into the priority queue O(1)
- Create distance array based on number of vertices in the graph, all initialised to infinity O(V)
- Initialise distance from source vertex as 0 O(1)
- While Priority Queue is not empty: O(V)
    - Pop first item from priority queue (the item with the shortest distance to source vertex) O(logV)
    - FOR all the adjacent vertices: O(E)
        - Check if Distance from said item to source vertex is smaller than currently recorded distance in distance array + source vertex is in min heap priority queue
        - If so, update distance with smaller amount
        - Add the vertex and new distance into priority queue

Time Complexity (Theoretical)

- Initialise a min heap for the priority queue O(VlogV)
- Add source vertex with distance 0 into the priority queue O(1)
- Create distance array based on number of vertices in the graph, all initialised to infinity O(V)
- Initialise distance from source vertex as 0 O(1)
- While Priority Queue is not empty: O(V)
    - Pop first item from priority queue (the item with the shortest distance to source vertex) O(logV)
    - FOR all the adjacent vertices: O(E)
        - Check if Distance from said item to source vertex is smaller than currently recorded distance in distance array + source vertex is in min heap priority queue
        - If so, update distance with smaller amount
        - Add the vertex and new distance into priority queue

Time Complexity:

```
O(V log V) + O(E log V) + O(V)
```

```
= O((V+E)logV) = O(ElogV)
```

Our Sample Graph:

```
64 -> [4, 84, 43, 29]
65 -> [13]
66 -> [93]
67 -> [27]
68 -> [0, 41, 33]
69 -> [54, 9]
70 -> [10]
71 -> [32, 8]
72 -> [83, 56]
73 -> []
74 -> [14, 57, 50]
75 -> []
76 -> [87]
77 -> [83]
78 -> [10, 91, 15, 10]
79 -> [32]
80 -> [51]
81 -> [25, 4, 53, 41, 91]
82 -> []
83 -> [51, 20, 95, 72, 77]
84 -> [64, 56]
85 -> [60]
86 -> [89]
87 -> [76]
88 -> [42, 95]
89 -> [46, 86]
90 -> [2, 57, 95]
91 -> [78, 81]
92 -> []
93 -> [66, 28]
94 -> []
95 -> [83, 90, 59, 22, 88]
96 -> [18, 98]
97 -> [48, 49]
98 -> [16, 17, 36, 96]
99 -> [27]
[[(68, 17), (21, 5)], [(15, 12)], [(90, 16), (18, 15), (16
```

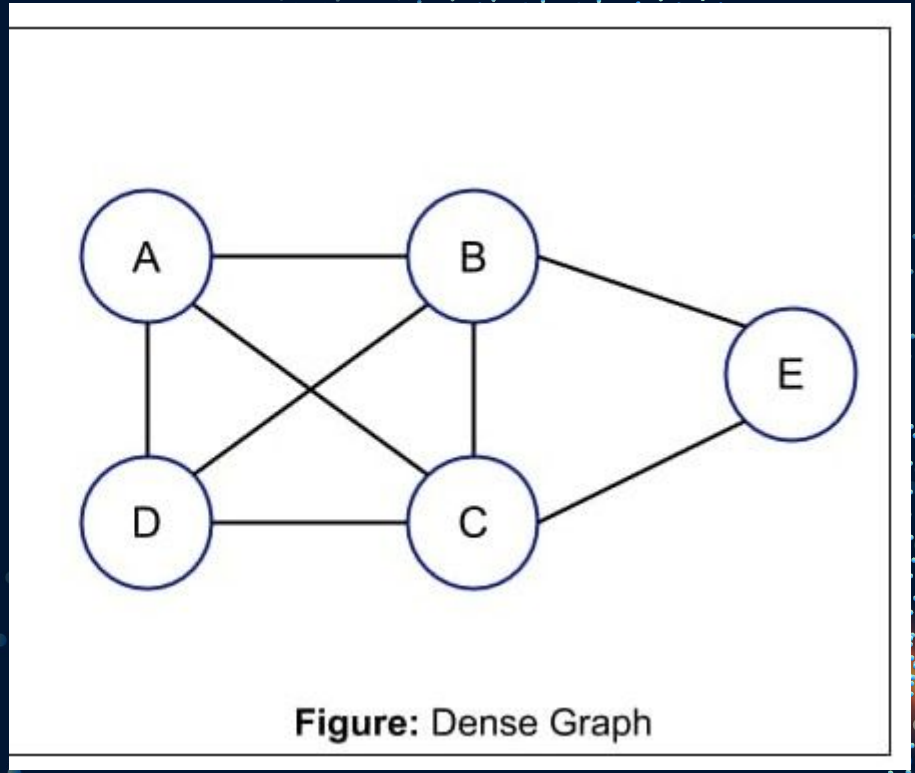
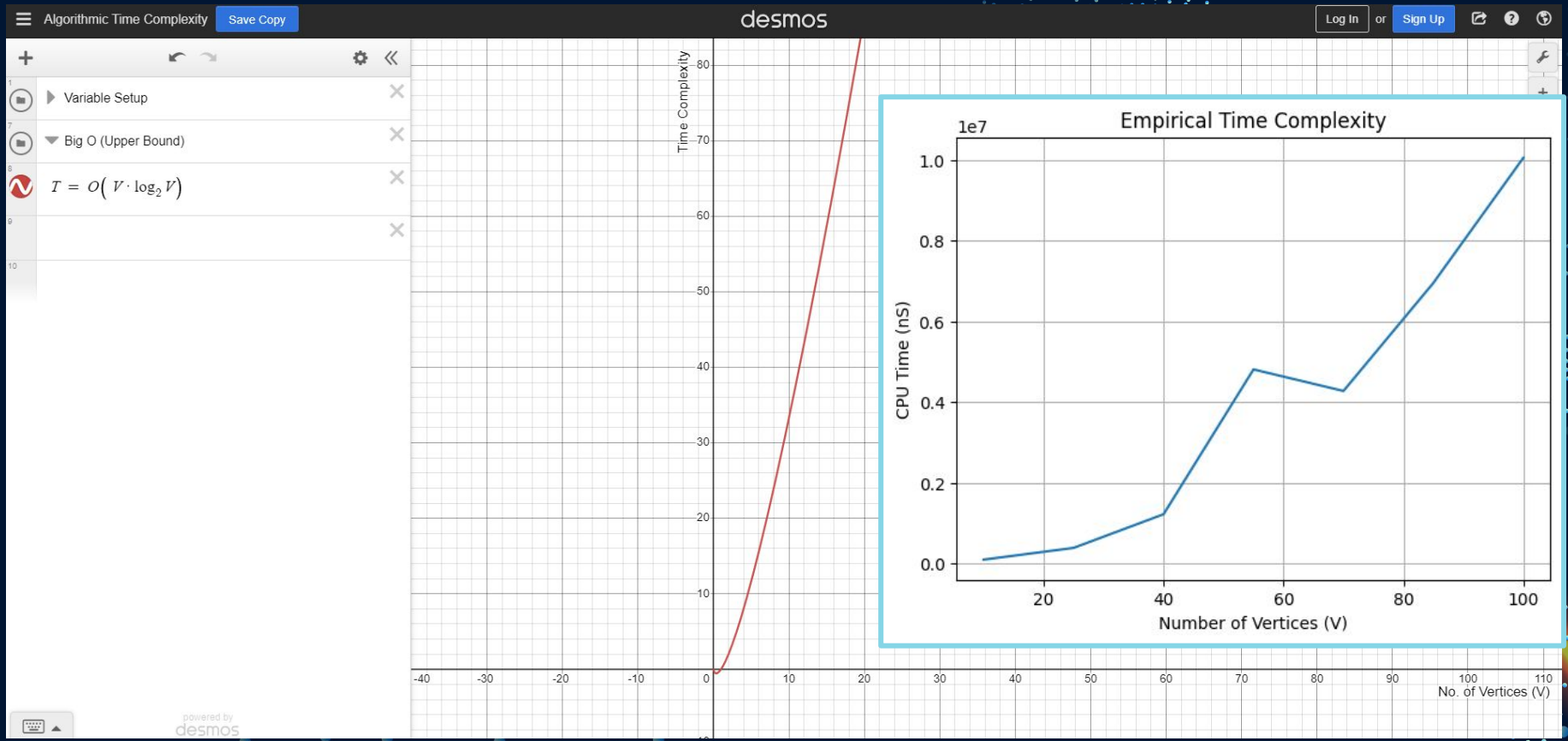


Figure: Dense Graph

Time Complexity (Theoretical, with increasing V)



Sample Graph

```
Sample graph:
0 -> [1]
1 -> [0, 2]
2 -> [1, 3]
3 -> [2, 4]
4 -> [3, 5]
5 -> [4, 6]
6 -> [5, 7]
7 -> [6, 8]
8 -> [7, 9]
9 -> [8]
[[(1, 11)], [(0, 11), (2, 6)], [(1, 6), (3, 2)],
```

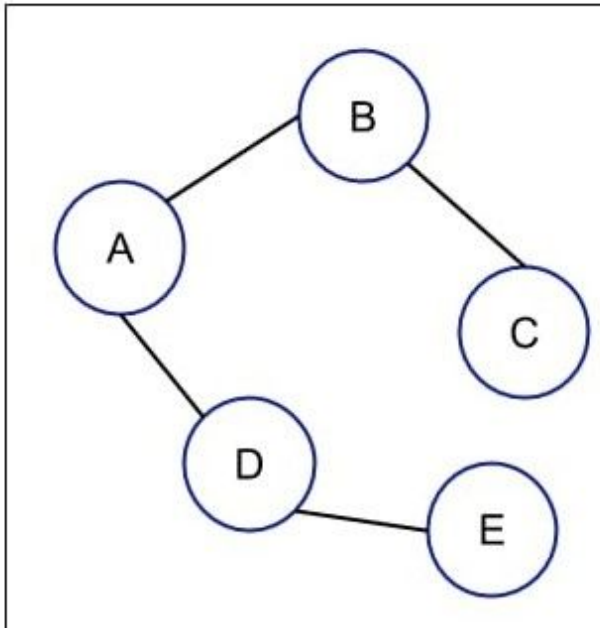
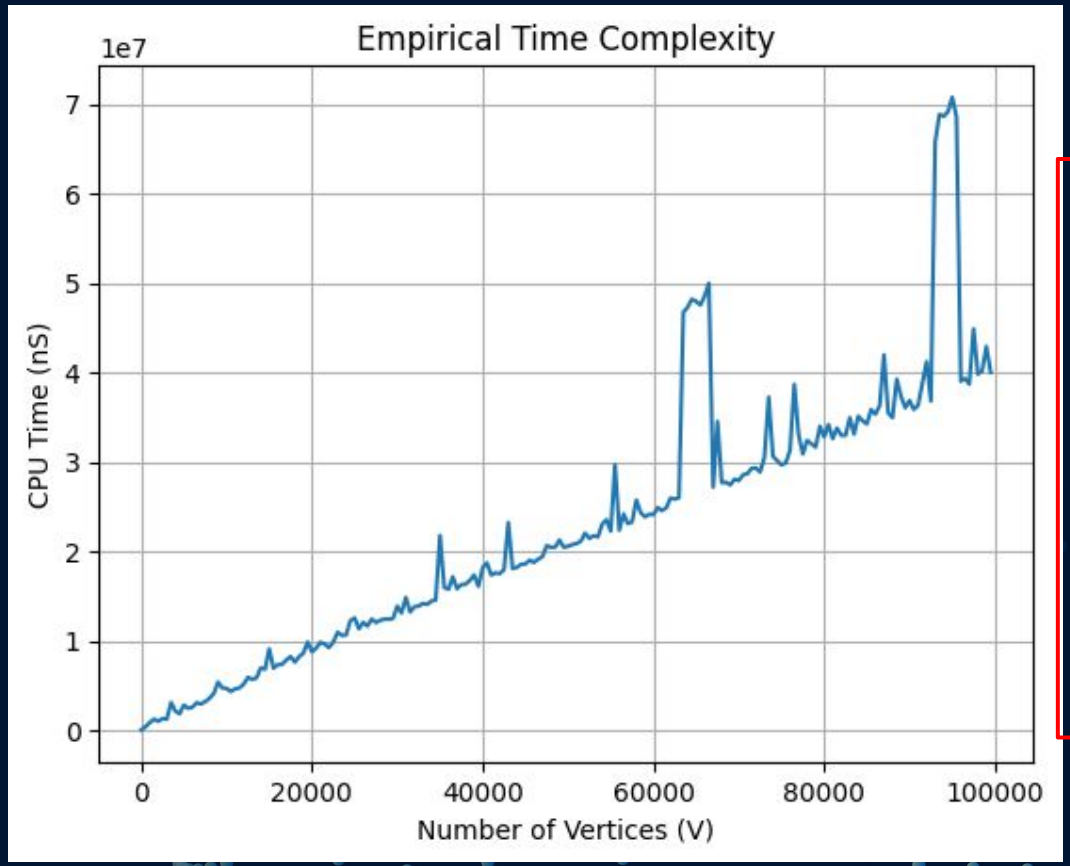


Figure: Sparse Graph

Time Complexity (Theoretical, with increasing V)



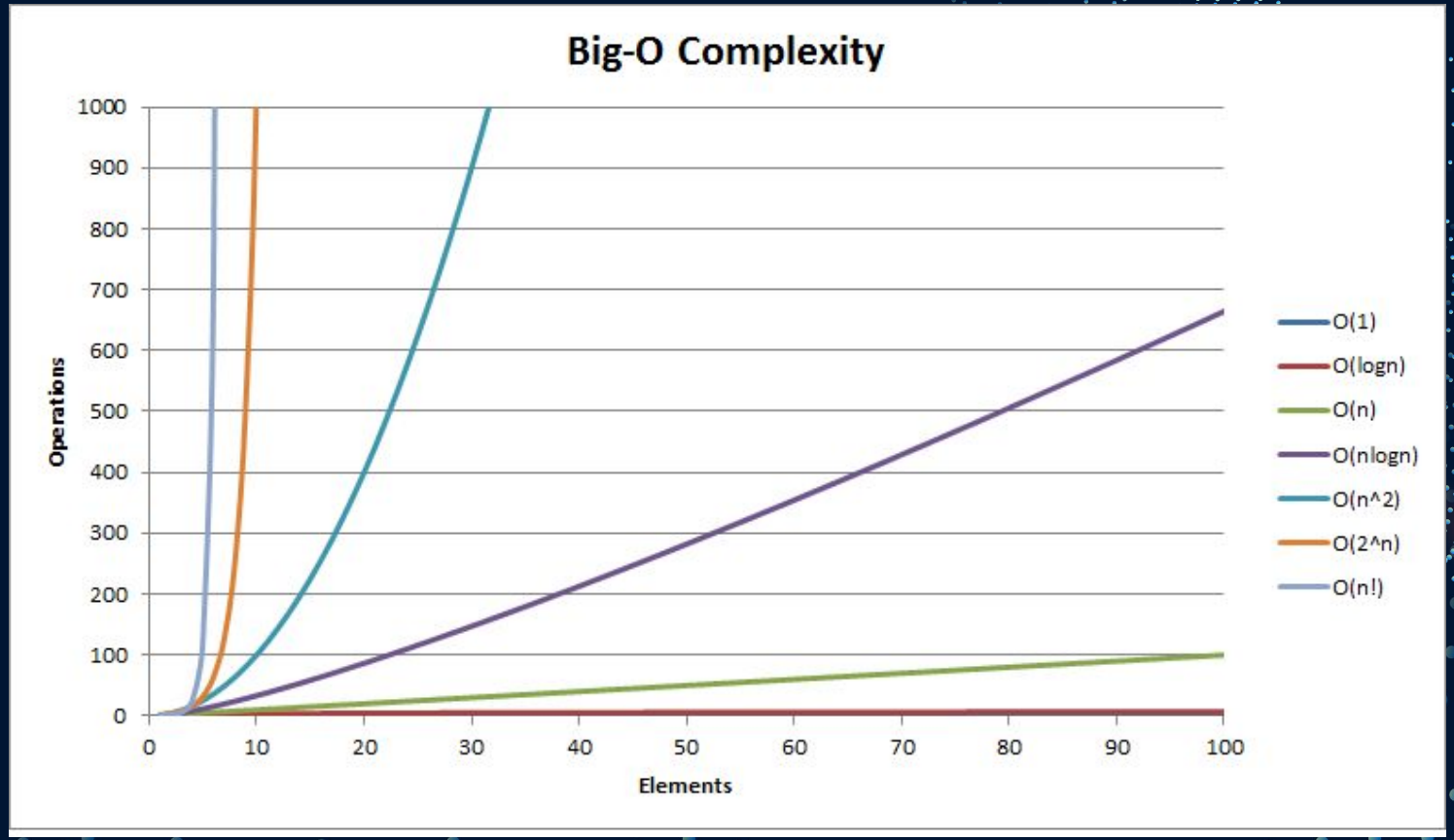
Relaxing the adjacent vertices: O(1) instead of O(E)

Time Complexity:

`O(V log V) + O(log V) + O(V)`

`= O((V+1)logV) = O(VlogV)`

Time Complexity (Theoretical, with increasing V)



Sample Graph: (Increasing E)

```
setNumOfEdges(300)
```

```
29 -> [13, 73, 41, 78, 28, 50, 59, 50, 39, 90]
30 -> [61, 91, 31, 57, 22, 75, 16]
31 -> [24, 49, 39, 18, 30, 73, 47, 78]
32 -> [89, 69, 12, 15, 78, 90]
33 -> [37, 7, 97, 83, 89, 55, 55, 61]
34 -> [48, 54, 15, 80]
35 -> [62, 40, 90]
36 -> [96, 63]
37 -> [60, 33, 40, 3]
38 -> [90, 87, 14, 45, 14]
39 -> [65, 31, 84, 79, 29, 78, 0]
40 -> [53, 60, 35, 22, 37, 70, 19, 12]
41 -> [19, 87, 29, 3, 2, 67, 51, 92, 60]
42 -> [28, 86, 49, 85, 16, 13]
43 -> [55, 10, 94, 63, 74]
44 -> [70, 19, 71]
45 -> [74, 76, 38]
46 -> [7, 77, 77, 47, 75, 23, 66]
47 -> [16, 64, 46, 73, 28, 9, 96, 28, 98, 31, 28]
48 -> [34, 93, 88, 19]
49 -> [69, 23, 31, 97, 42, 78, 80, 24, 79]
50 -> [68, 10, 29, 79, 29, 16, 90]
51 -> [77, 97, 85, 65, 75, 41]
52 -> [26, 71]
53 -> [40, 83, 64]
54 -> [84, 4, 89, 34, 97, 0, 57]
55 -> [43, 9, 4, 79, 33, 12, 33]
56 -> [3, 6, 71, 74, 81]
57 -> [21, 54, 30, 83]
58 -> [61, 16]
59 -> [29, 83, 18, 70, 21]
60 -> [13, 22, 37, 40, 41]
61 -> [74, 58, 30, 24, 9, 33]
62 -> [35, 28, 8, 69, 76, 91]
```

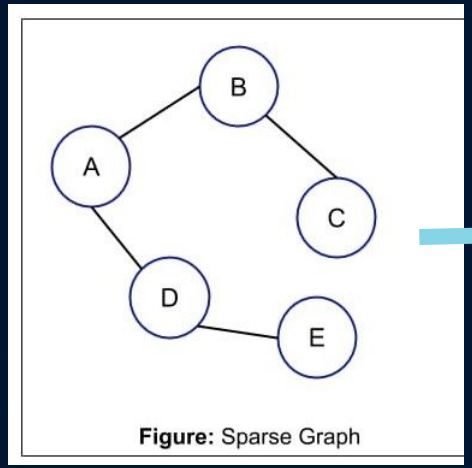


Figure: Sparse Graph

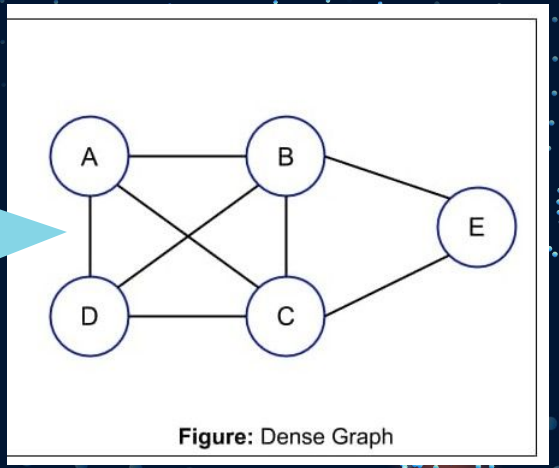
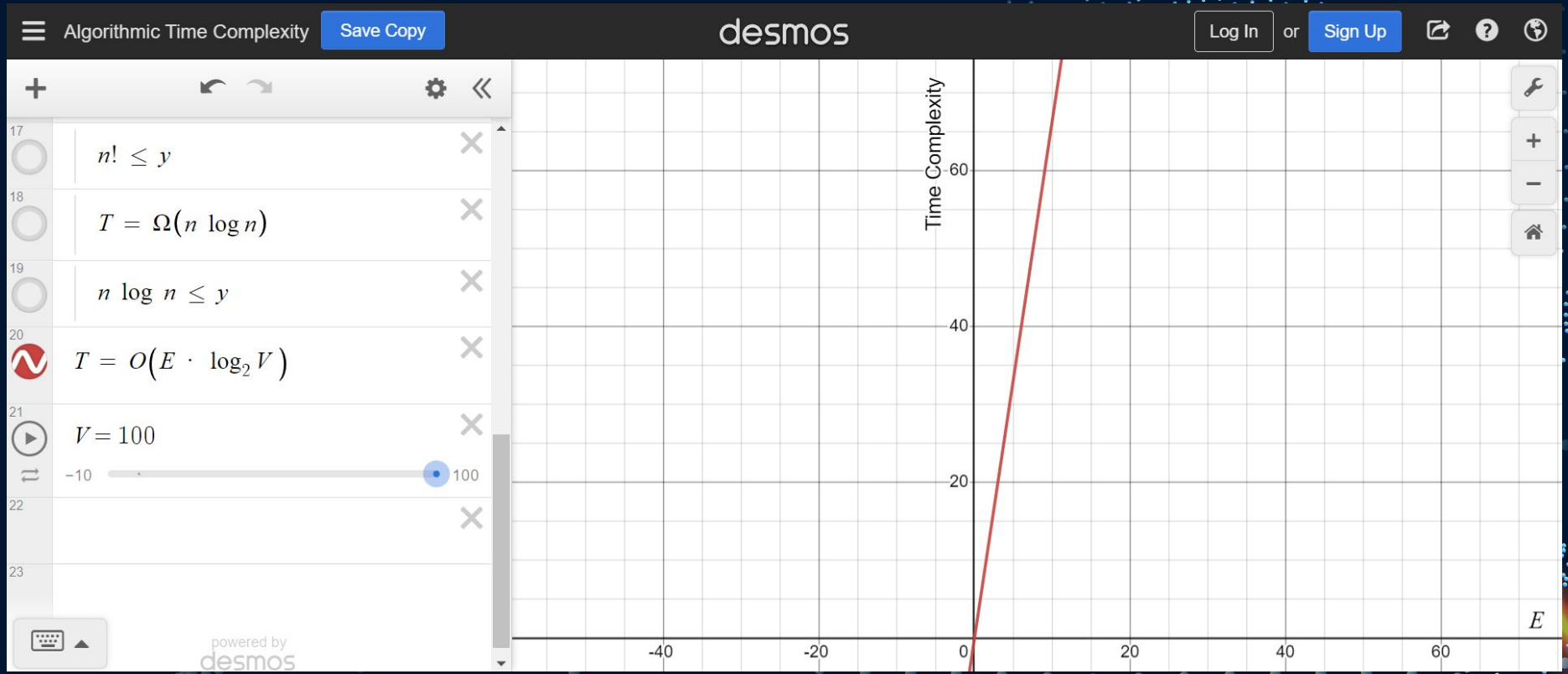
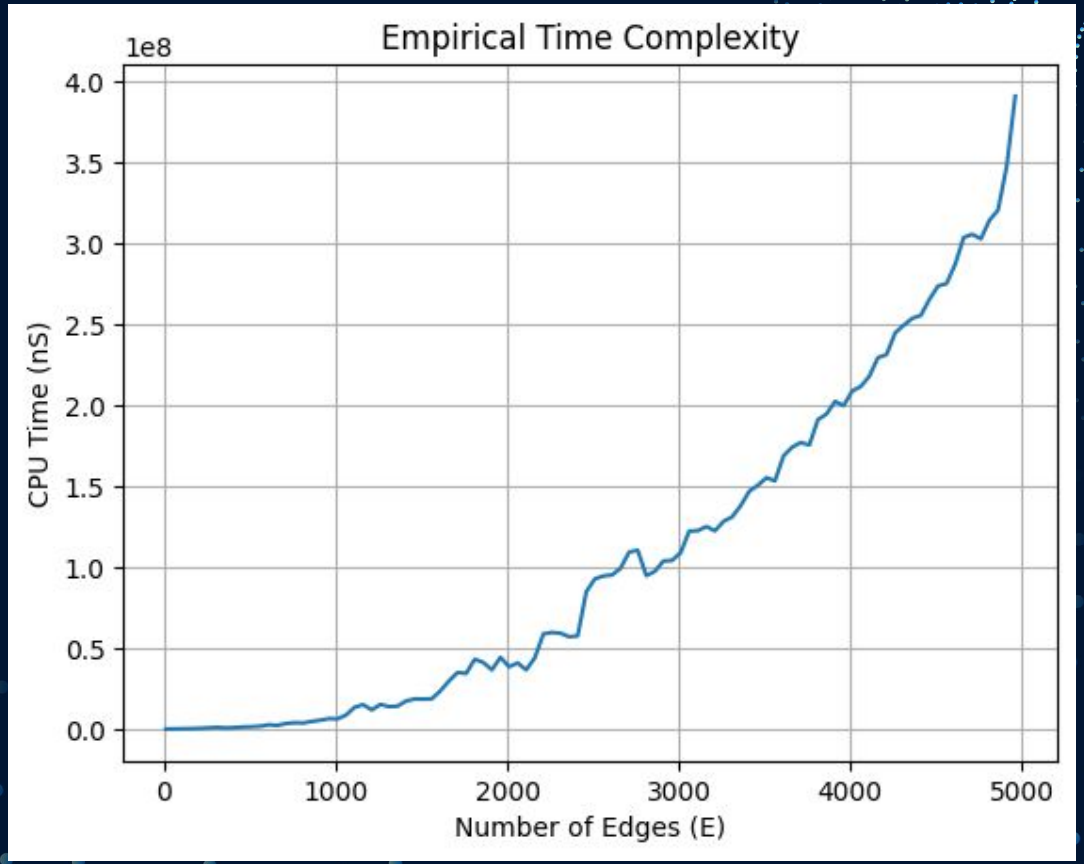


Figure: Dense Graph

Time Complexity (Theoretical, with increasing E)



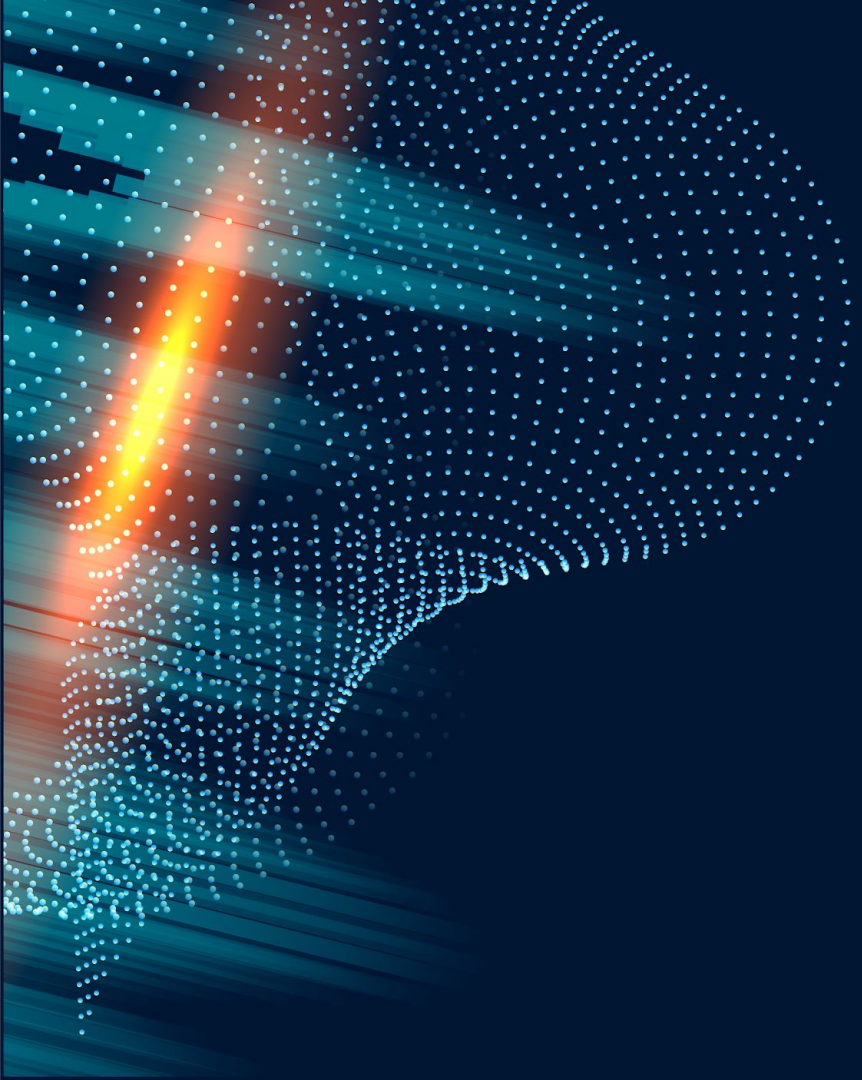
Time Complexity (Empirical, increasing E)



Interpretation for difference

For a dense graph, E ~= V^2

This means that the time complexity of Dijkstra's algorithm for a dense graph will be closer to O(V^2 log V) than O(E log V).

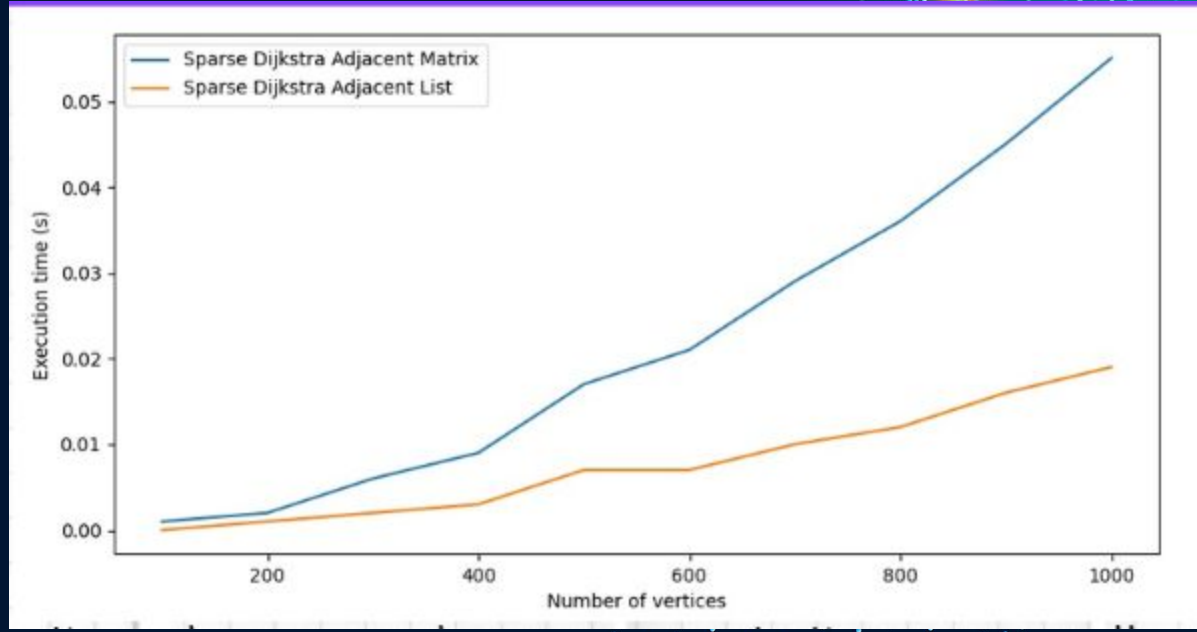


Part C

Which implementation is better?

In general

The second implementation with the graph stored in an array of adjacency list and uses a minimising heap as the priority queue would be better.



Storing graph in an array of adjacency list compared to a adjacency matrix

- More space efficient than adjacency matrix as it only stores the edges that actually exist in the graph, compared to the adjacency matrix which stores the weight of all possible edges
- Space complexity of adjacency matrix is O(v^2) while space complexity of adjacency list is O(V+E), hence it will be better to use the array of adjacency list as it uses less space.

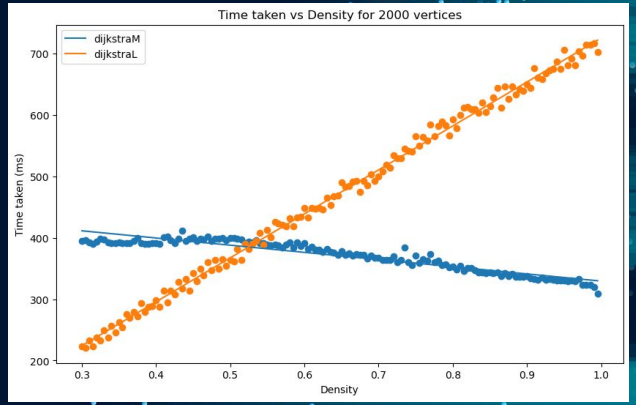
Using a minimising heap compared to an array for priority queue.

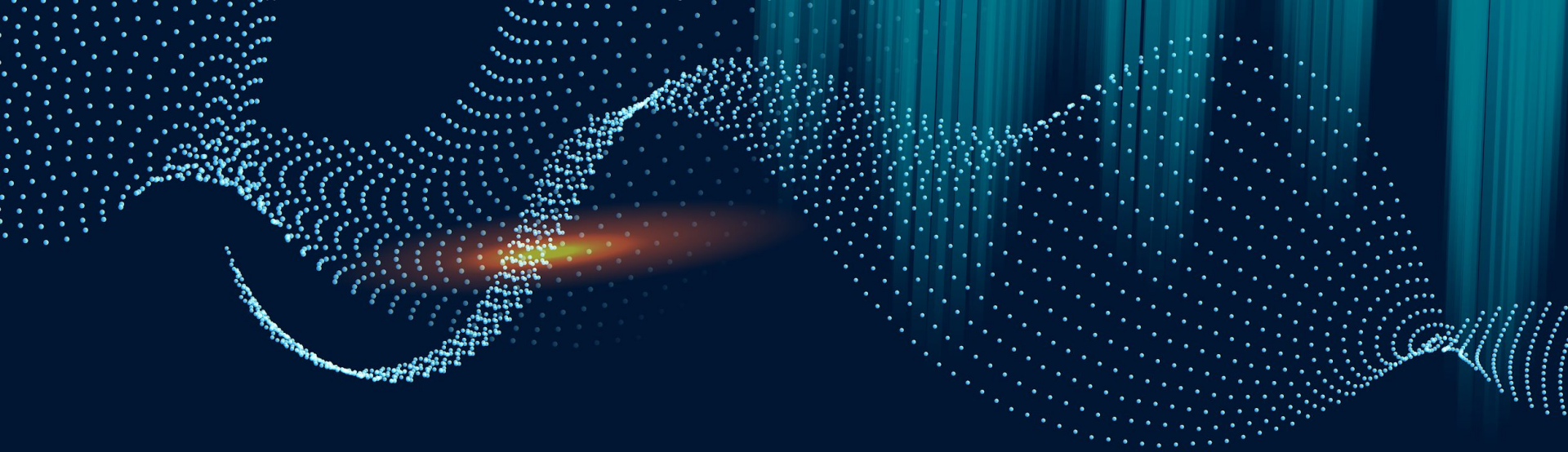
- Heap structure uses a time of O(logV) for dijkstra algorithm to go through it, while using an array for priority queue will use a time of O(V), a much longer time than a minimising heap.

However, when E is large

$$D = \frac{|E|}{|V|(|V|-1)},$$

- The adjacency matrix representation is more efficient for **DENSE** graphs because it stores all the nodes and weights of the graph.
- Allows us to just directly check the connection using the matrix
- The adjacency list representation needs to iterate over the list of edges adjacent to each vertex, not efficient for dense graph.
- For denser graphs, the heap would essentially have the same if not slower time than using an array.
- Time complexity for heap essentially becomes O(V^2), but will be too complex to implement.





Thank you!