Dijkstra Implementation

By Ethan, Jordan, and Jared

A) Adjacency Matrix + Array Priority Queue

From Tutorial Wk6

$$
\begin{array}{c c}
\text{vertex} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{bmatrix}
0 & 4 & 2 & 6 & 8 \\
\infty & 0 & \infty & 4 & 3 \\
\infty & \infty & 0 & 1 & \infty \\
\infty & 1 & \infty & 0 & 3 \\
\infty & \infty & \infty & \infty & 0
\end{bmatrix}
\end{array}
$$

```python
# i = infinity
i = float('inf')

adj_matrix = [
    [0, 4, 2, 8, 6],
    [i, 0, i, 4, 3],
    [i, i, 0, 1, i],
    [i, 1, i, 0, 3],
    [i, i, i, i, 0]
]
```

A) Adjacency Matrix + Array Priority Queue

Priority queue Q stores the vertices whose shortest paths have NOT been determined yet. (like a to-do list)

```
Q = list(range(n))
```

Q = [0,1,2,3,4]

```python
def dijkstra(graph, source):
    # Number of vertices n
    n = len(graph)

    d = [i] * n
    pi = [None] * n
    S = [False] * n
    d[source] = 0

    Q = list(range(n))

    while Q:

        u = Q[0]

        for vertex in Q:
            if d[vertex] < u:
                u = vertex

        Q.remove(u)

        S[u] = True

        for v in range(n):
            if S[v]!=True and d[v] > d[u] + graph[u][v]:
                d[v] = d[u] + graph[u][v]
                pi[v] = u
```

```
for each vertex v {
  d[v] = infinity;
  pi[v] = null pointer;
  S[v] = 0;
 }
 d[source] = 0;
 put all vertices in priority queue, Q, in d[v]'s increasing order;
 while not Empty(Q) {
   u = ExtractCheapest(Q);
   S[u] = 1;          /* Add u to S */
    for each vertex v adjacent to u {
      if (S[v] ≠ 1 and d[v] > d[u] + w[u, v]) {
      remove v from Q;
  d[v] = d[u] + w[u, v];
  pi[v] = u;
  insert v into Q according to its d[v];
               }
  } // end of while loop
}
```

```python
while Q:

    u = Q[0]

    for vertex in Q:
        if d[vertex] < u:
            u = vertex

    Q.remove(u)

    S[u] = True

    for v in range(n):
        if S[v]!=True and d[v] > d[u] + graph[u][v]:
            d[v] = d[u] + graph[u][v]
            pi[v] = u

return d
```

<u>O(|V| x |V|) = O(|V|^2)</u>

Finding closest neighbour iterates through priority queue
→ O(|V|)

Relaxing its neighbours
→ O(|V|)

A) Adjacency Matrix + Array Priority Queue

What about E?

- adjacency matrix always has |V × V| entries, regardless of number of edges E.

- When the number of edges E is increased, the algorithm still needs to iterate through all V vertices to identify neighbors even if some vertices are not connected by edges

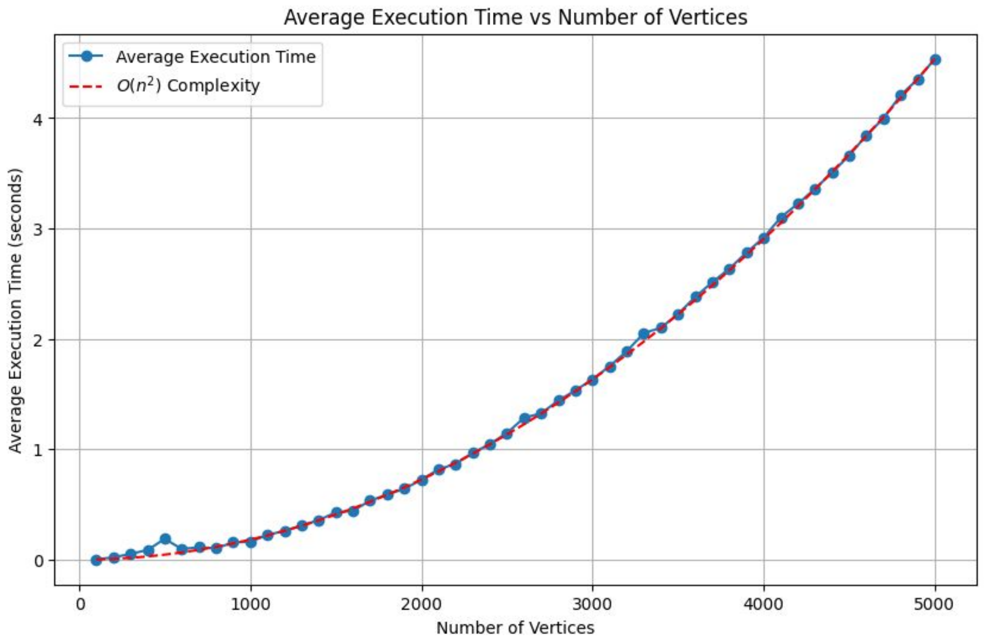
- increasing E does not significantly change the number of operations needed in Dijkstra’s algorithm when an adjacency matrix is used.

A) Adjacency Matrix + Array Priority Queue

Keep E constant at 10000, Vary V (start from 100, find the average execution time, step = 100), plot graph

Average execution time increases quadratically with the number of vertices

Thus the algorithm’s execution time follows O(V^2), which supports the theoretical time complexity of Dijkstra’s using Adjacency matrix and array priority queue

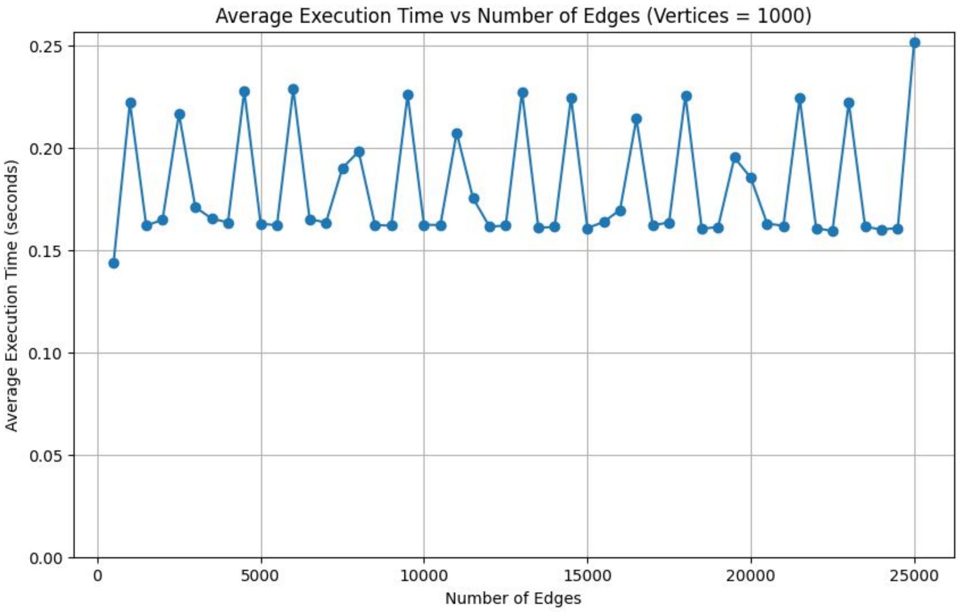


A) Adjacency Matrix + Array Priority Queue

Keep V constant at 1000, Vary E (start from 500, find average execution time, step = 500), plot graph

Average execution time **does not follow a well-defined relationship** as number of edges increases. Remains approximately constant at about 0.2 seconds

Thus the time complexity does not depend on the number of edges of the graph when we use this implementation on Dijkstra's algorithm

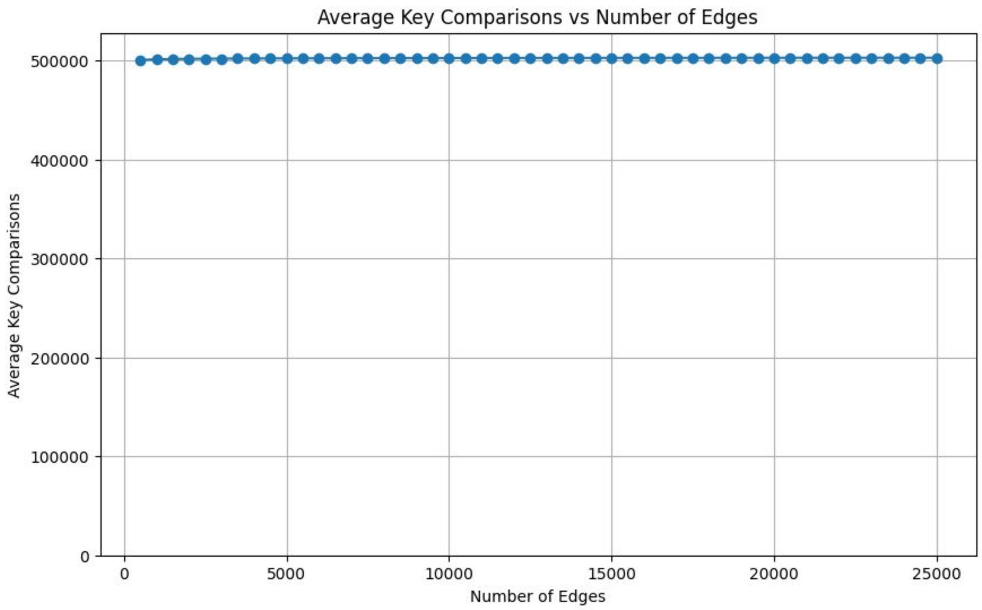


A) Adjacency Matrix + Array Priority Queue

Average key comparisons follows a horizontal line number of edges increases. Remains approximately constant at about 500,000

Thus the time complexity does not depend on the number of edges of the graph when we use this implementation on Dijkstra’s algorithm

<u>Overall: O(V^2) no E term</u>

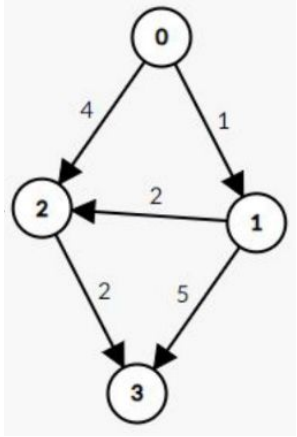


B) Using adjacency list and minimizing heap as priority queue

B) Using adjacency list

Adjacency list helps with storing in the graph

Example:




```
graph = {
    0: [(1, 1), (2, 4)],
    1: [(3, 5)],
    2: [(1, 2), (3, 2)],
    3: []
}
```

Min-Heap as priority queue

Efficiently extracts with the smallest distance in O(Log|V|) time and to update the distance of neighbouring vertices in the heap

1) Initially, the source vertex is pushed in with distance of 0, other vertices are set to infinity.

2) While priority queue is not empty:

   -Extract vertex u with smallest distance

   -For each neighbour v of u, if shorter path to v is found,update the distance v and either insert or update v in the heap.

Theoretical analysis

1)Heap operations:

-The operation to fix the heap after popping a vertex takes **O(Log|V|)**

-For all vertices V it will be **O(|V|LOG|V|)**

2)Edge relaxation:

-For V vertices, **|E|** max number of times where distance can be adjusted

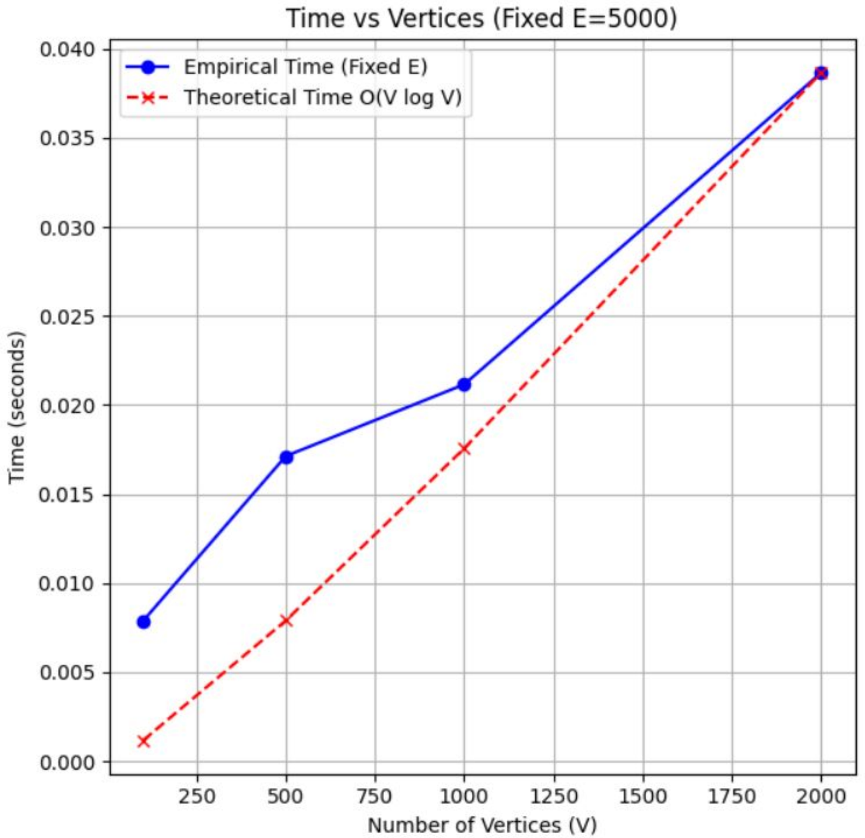
-When there is a shorter path and it needs to update the heap it takes **O(Log|v|)**, combining both you will get **O(|E|LOG|V|)**

3)Total:O((|V|+|E|)LOG|V|)

Empirical analysis against theoretical(fixed |E|, vary |V|)

-Fixed |E|= 5000

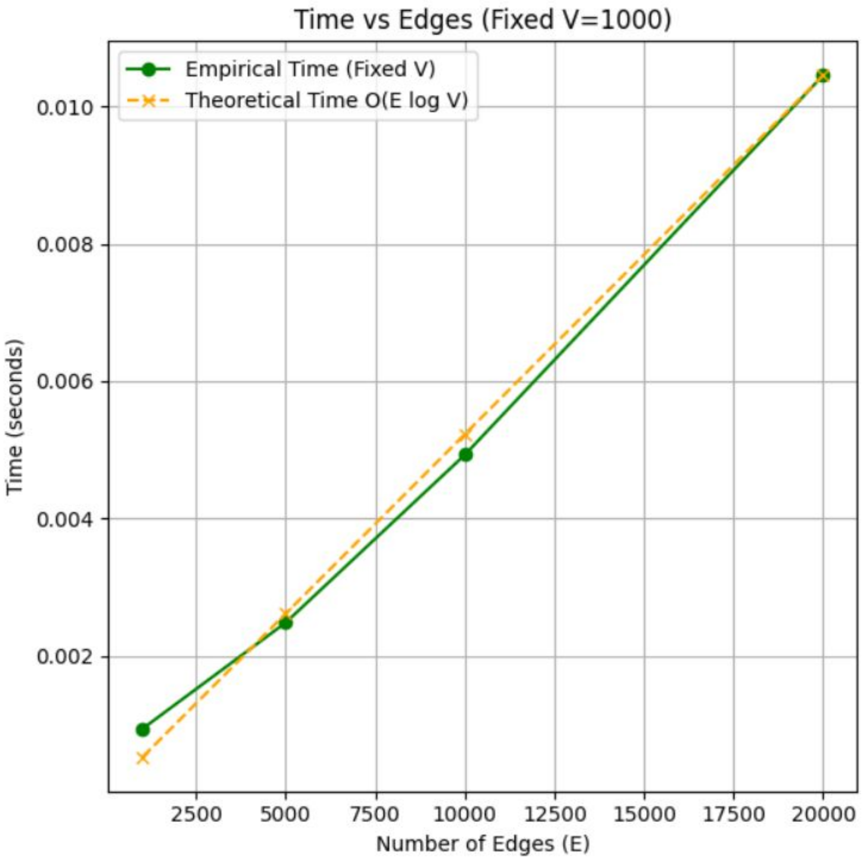
-Deviation at 500 possibly caused from overheads



Empirical analysis against theoretical(fixed |V|, vary |E|)

-Fixed |V| = 1000

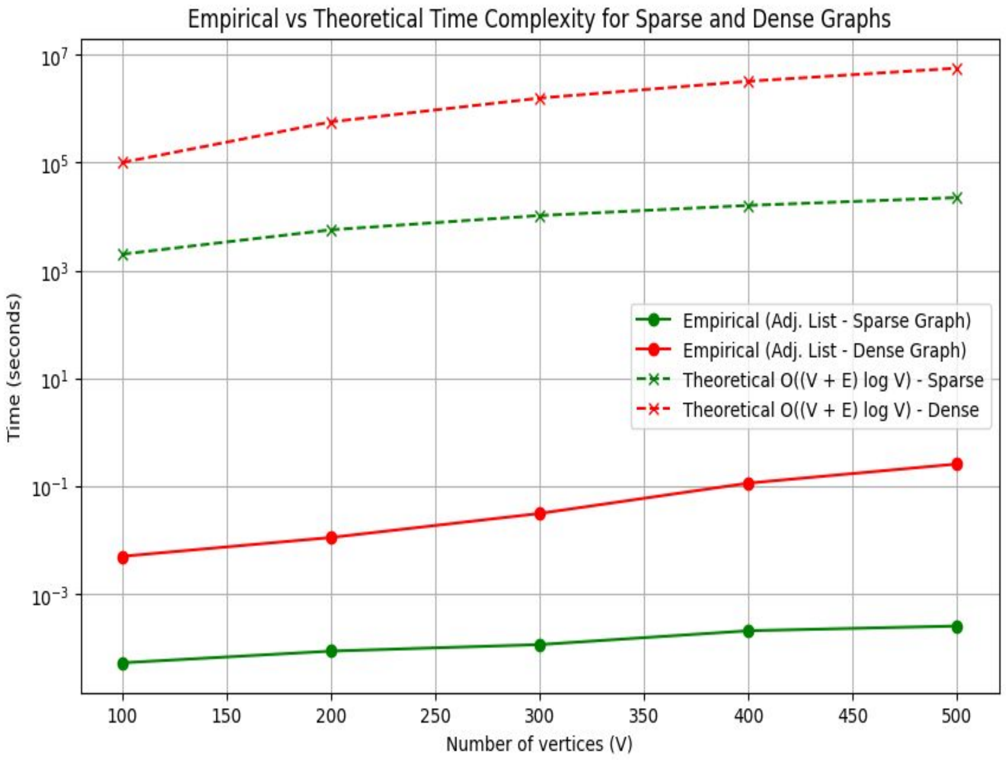
-Time increase as edges increase as naturally there are more pathing to check and more edge relaxation.



Empirical analysis against theoretical(sparse and dense graph)

-Sparse graph follows closely to theoretical time complexity

-Dense graph has a slight deviation towards the end



Conclusion

The algorithm is more efficient for sparse graphs,while dense graph introduces more computational complexity due to higher number of edges.

C) Comparison of the 2 Algorithms

Comparison of Time Complexity

Dijkstra's Algorithm w/ adjacency matrix & array: **$O(|V|^2)$**

Dijkstra’s Algorithm w/ adjacency list & minimizing heap: **$O((|V|+|E|)\log|V|)$**

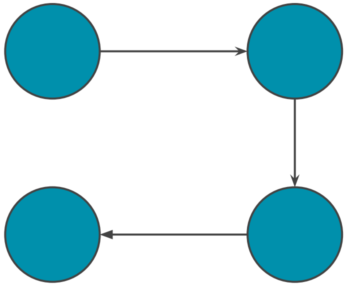
Scenario: |E| is small

When |E| is small:

- Minimum |E| for any graph is (|V|-1)

As |E| → (|V|-1),
(|V|+|E|)log|V| → (2|V|-1)log|V|

O(|V|log|V|) < O(|V|²)



Thus, **Part B's algorithm** (minimizing heap) is the optimal algorithm for **scarcely connected graphs**

Scenario: |E| is large

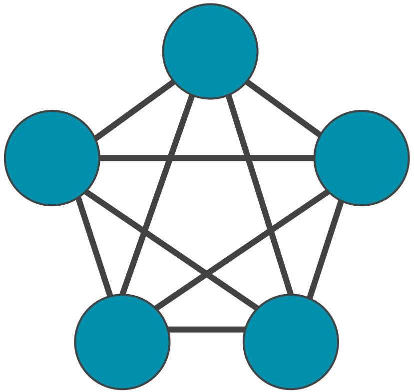
When |E| is large:

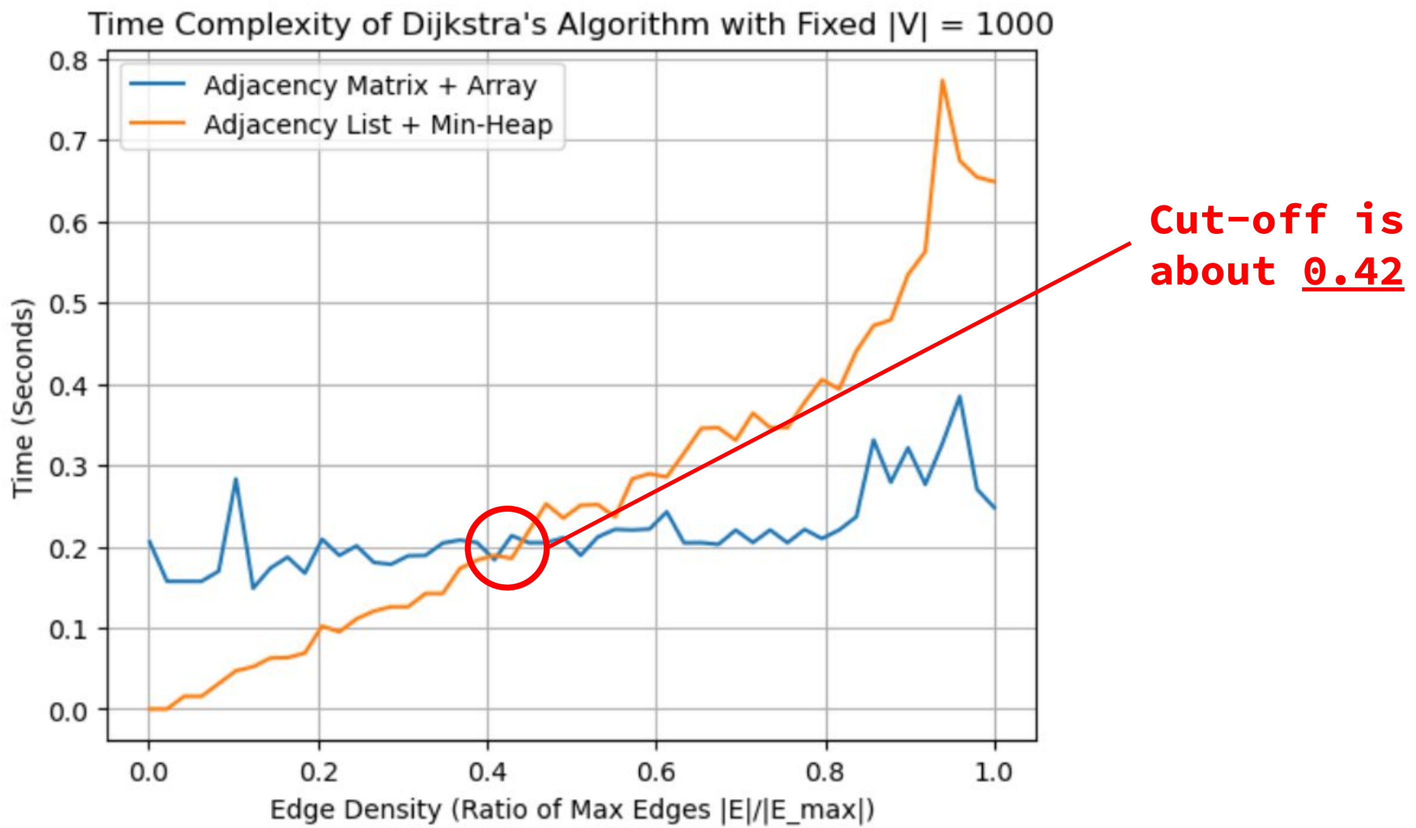
- Maximum |E| is $|V|(|V|-1)$

As $|E| \rightarrow |V|(|V|-1)$,
$(|V|+|E|)\log|V| \rightarrow |V|^2\log|V|$

$\mathbf{O(|V|^2\log|V|)} > \mathbf{O(|V|^2)}$

Thus, **Part A's algorithm** (Adj matrix + array) is the better algorithm for **dense graphs**.




Time Complexity of Dijkstra's Algorithm with Fixed |V| = 1000
Adjacency Matrix + Array
Adjacency List + Min-Heap
Time (Seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.0
0.2
0.4
0.6
0.8
1.0
Edge Density (Ratio of Max Edges |E|/|E_max|)
Cut-off is
about 0.42

THANK YOU