SC2001

Project 2

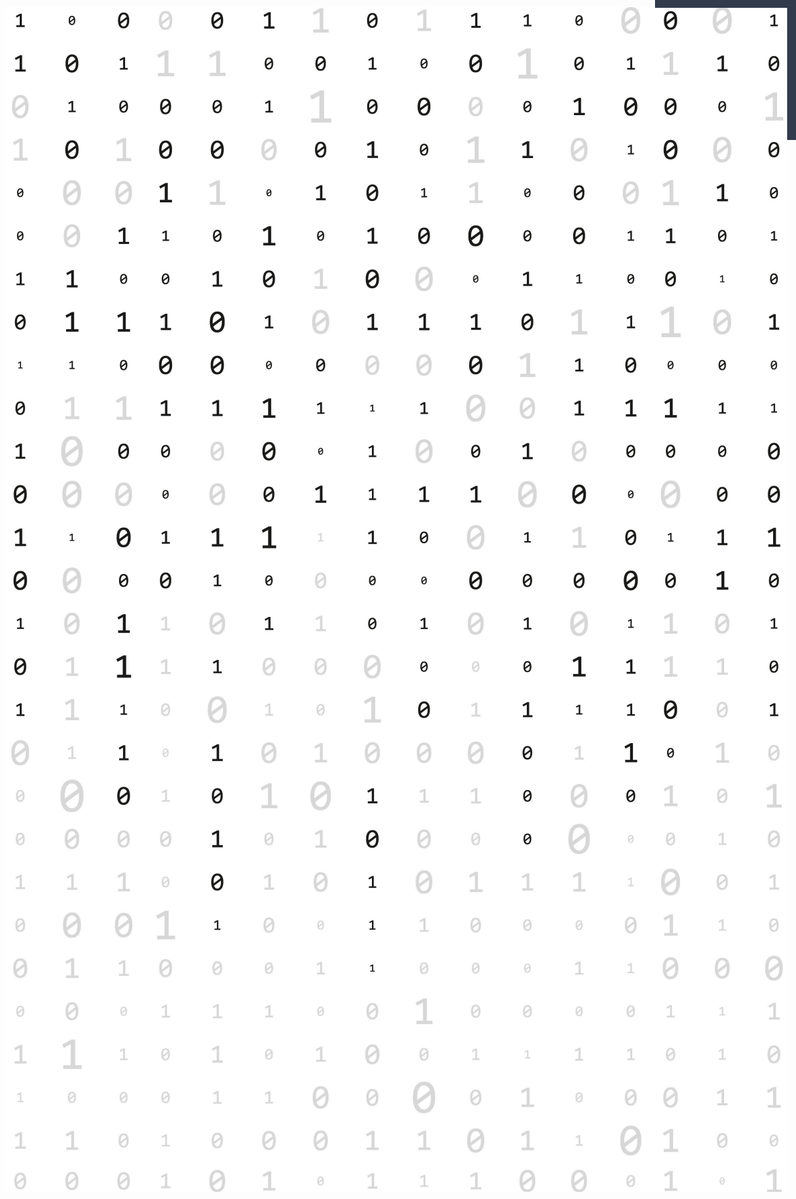
Dijkstra’s Algorithm Analysis

By:

Ming Kai

Mabel

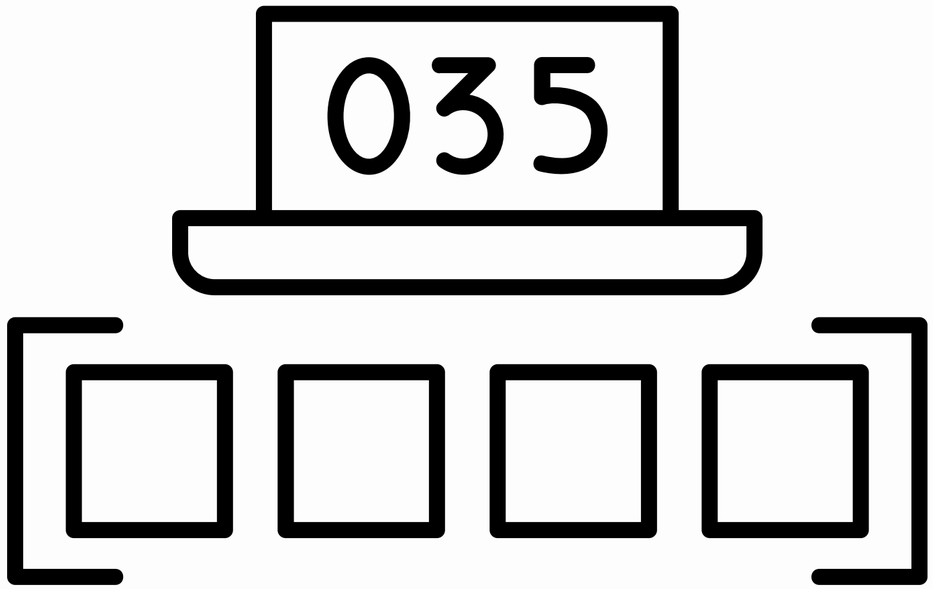
Aditi



Contents

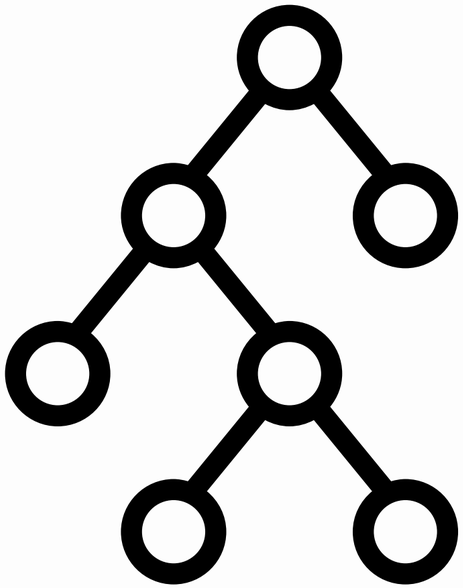
- Introduction to Dijkstra's Algorithm
- Arrays with Adjacency Matrix
- Min-Heap with Adjacency List
- Algorithm Comparison

Data Structure



Array

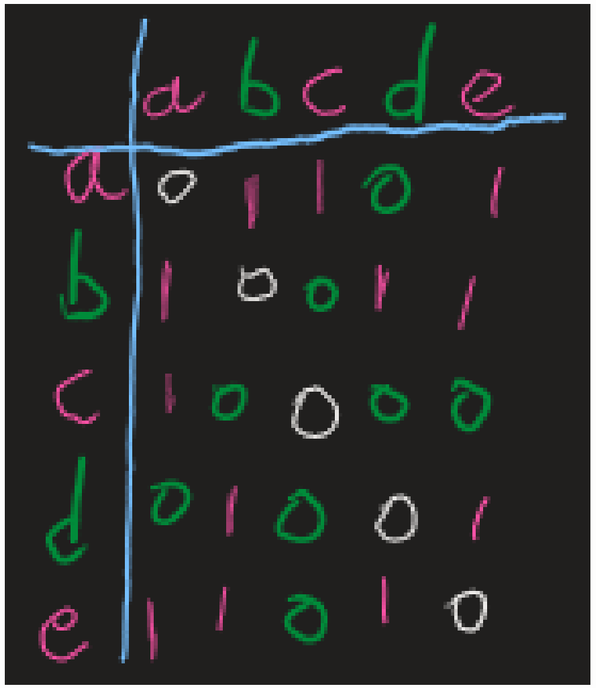
Stores the elements in a list such that each element can be accessed by its index



Heap

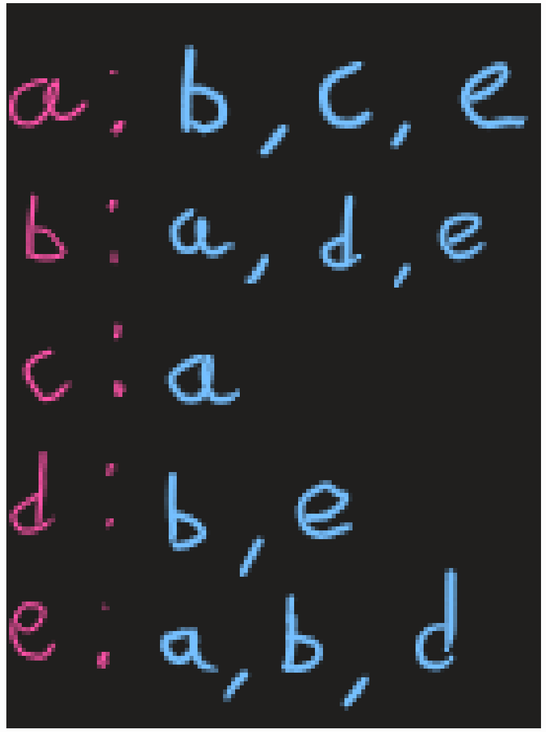
Stores the element in a tree satisfying the heap property (Parent > Children)

Data Structure Cont.



Adjacency Matrix

Stores the edges in a Matrix or in programming terminology a 2D array



Adjacency List

Stores the edges in a list such which can be contained in an array

Understanding Dijkstra’s Algorithm

```
dijkstra(graph, source):
dist = {v: float('inf') for v in graph}
dist[source] = 0
prev = {v: None for v in graph}
unvisited = set(graph.keys())

while unvisited:
    # Find the vertex in unvisited with th
    u = min(unvisited, key=lambda vertex:
    unvisited.remove(u)

    for neighbor, weight in graph[u].items
        alt = dist[u] + weight
        if alt < dist[neighbor]:
            dist[neighbor] = alt
            prev[neighbor] = u

return dist, prev
```

Adjacent Matrix with Array Priority Queue

Time Complexity

Time Complexity:
(v^2) + (v) -> O(v^2)

```python
def dijkstra_array(adj_matrix, src):
    n = len(adj_matrix)
    dist = [float('inf')] * n
    dist[src] = 0
    visited = [False] * n

    for _ in range(n):
        min_dist = float('inf')
        u = -1
        for i in range(n):
            if not visited[i] and dist[i] < min_dist:
                min_dist = dist[i]
                u = i

        if u == -1:
            break

        visited[u] = True

        # Update distances of adjacent vertices
        for v in range(n):
            if adj_matrix[u][v] != 0 and not visited[v]:
                new_dist = dist[u] + adj_matrix[u][v]
                if new_dist < dist[v]:
                    dist[v] = new_dist

    return dist
```

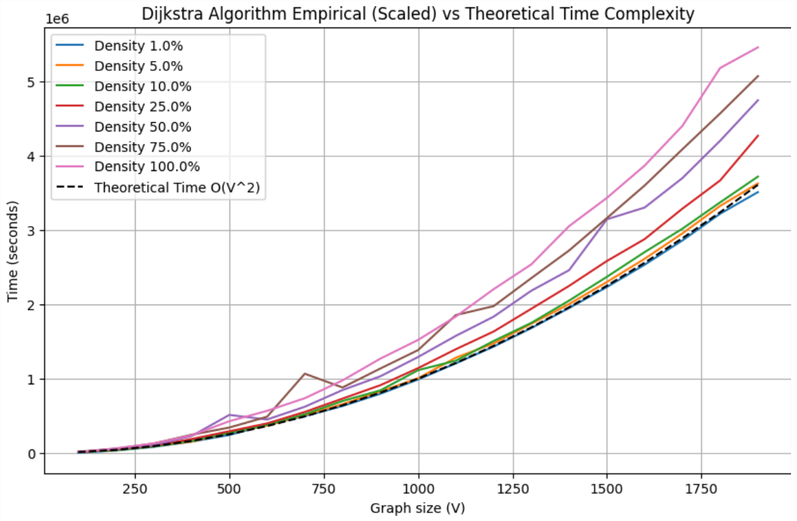
O(v)

O(v^2)

O(v)

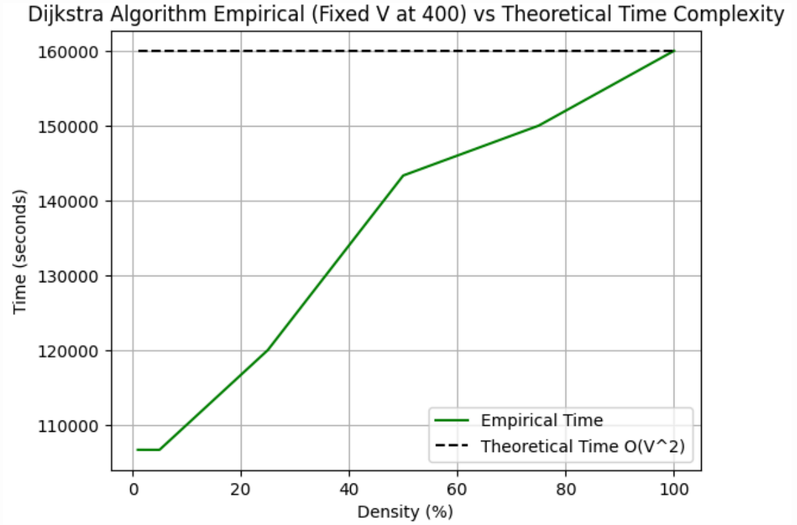
O(v)

Time Complexity (Array+Matrix)



Fixed Edges with Varying Vertex

The edges do not affect the time complexity largely



Fixed Vertex with Varying Edges

The overall empirical time is lower than theoretical time regardless of density of the graph

Adjacent List with Minheap Priority Queue

```python
def dijkstra_b(adj_list, source):
  V = len(adj_list)

  distance = [float('inf')] * V
  distance[source] = 0

  priorityQueue = [(0, source)]
  visited = [False] * V

  while priorityQueue:
    current_distance, u = heapq.heappop(priorityQueue)

    if visited[u]:
      continue
    visited[u] = True

    for v, weight in adj_list[u]:
      if not visited[v] and distance[u] + weight < distance[v]:
        distance[v] = distance[u] + weight
        heapq.heappush(priorityQueue, (distance[v], v))
  return distance
```

O(v)

O(v)log

O(E)logV

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

Time Complexity:

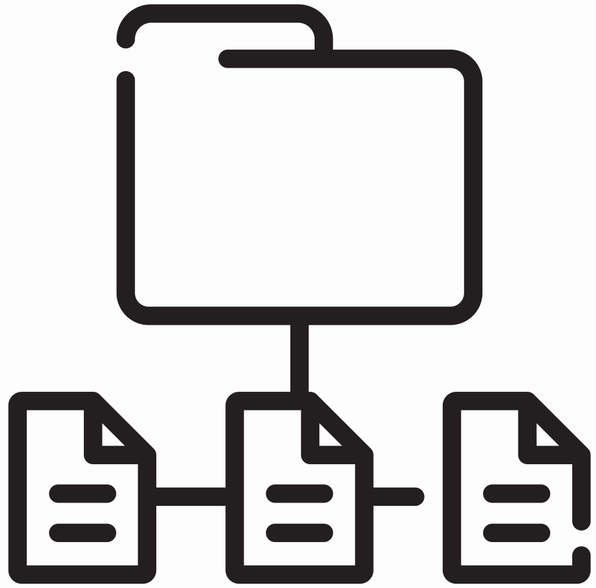
O[V + Vlog(V) + Olog(V)] ->

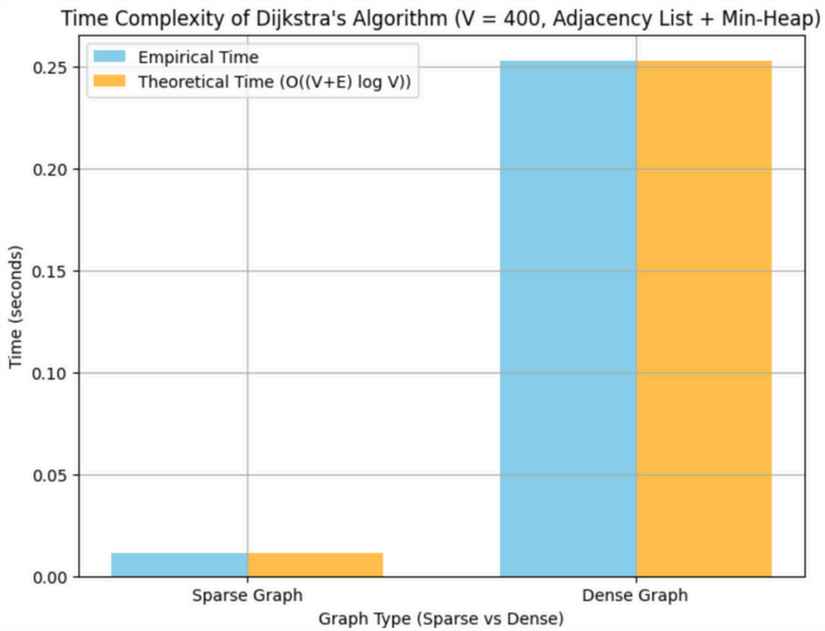
O((V+E)logV)

Generation of edges for Sparse and Dense Graphs

```
if edge == 'sparse':
    e = v * 10
elif edge == 'dense':
    e = v * ((v-1)//2)
```

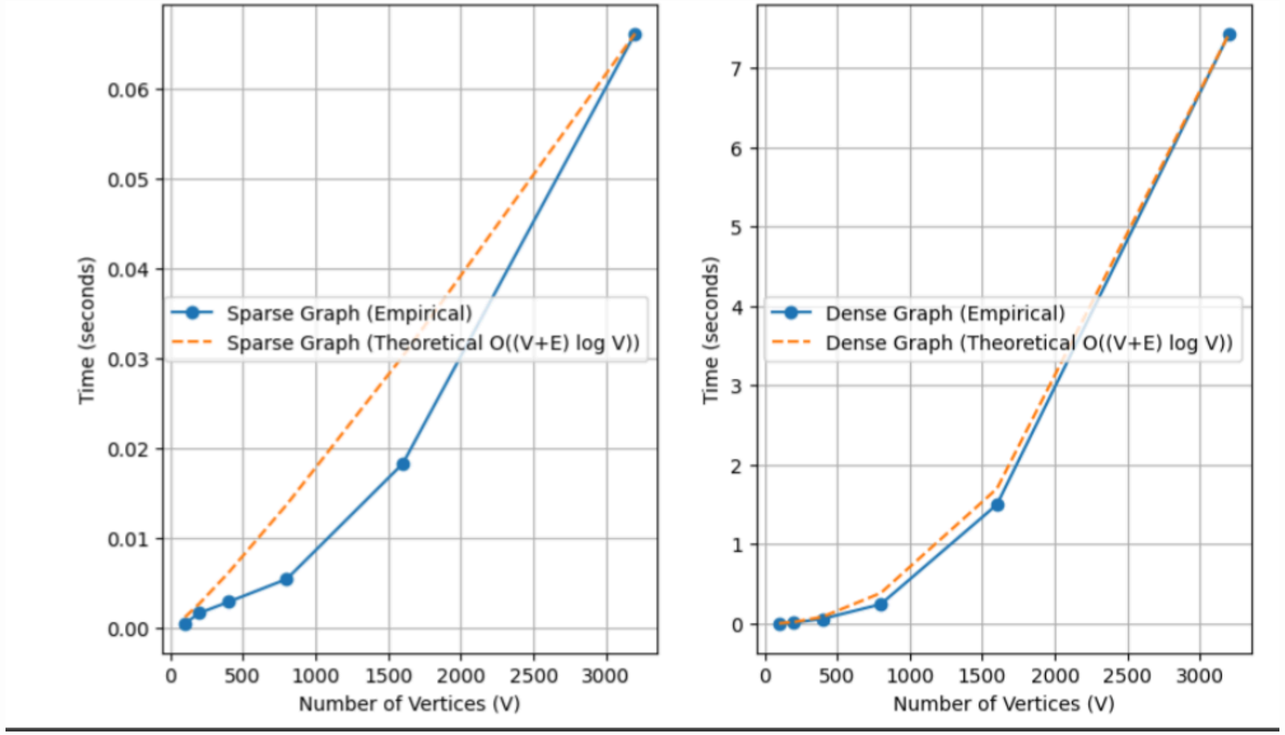
Time Complexity (Minheap+List)





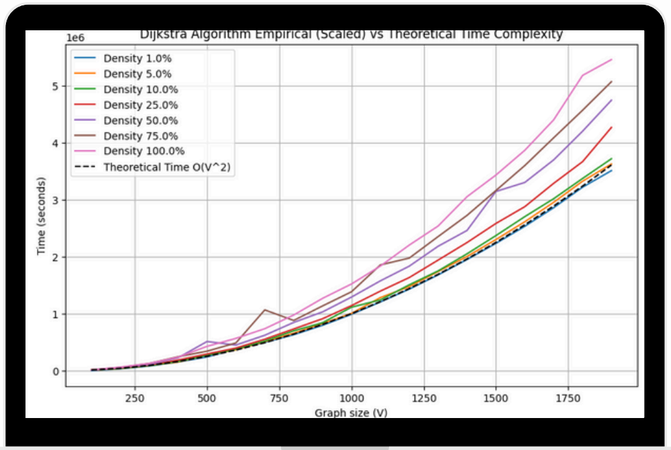
Different methods evaluate performance and complexity.

1. Fixed vertex = 400
2. Analysing time complexity with varying edges.

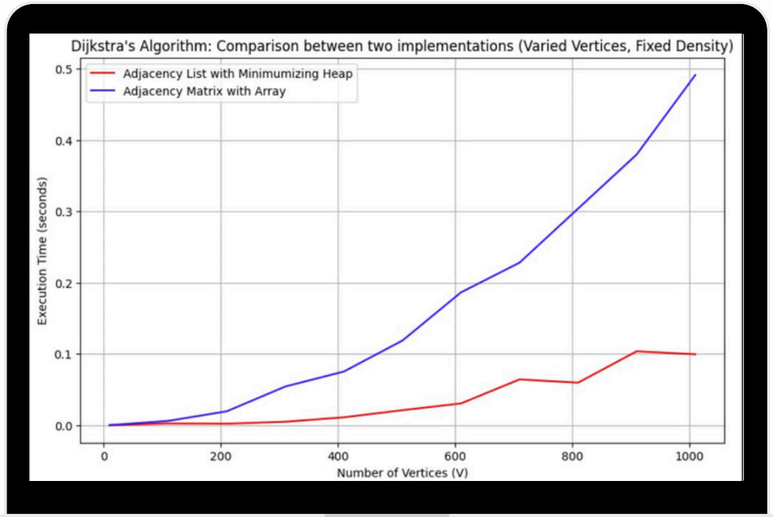


Comparing Time Complexity

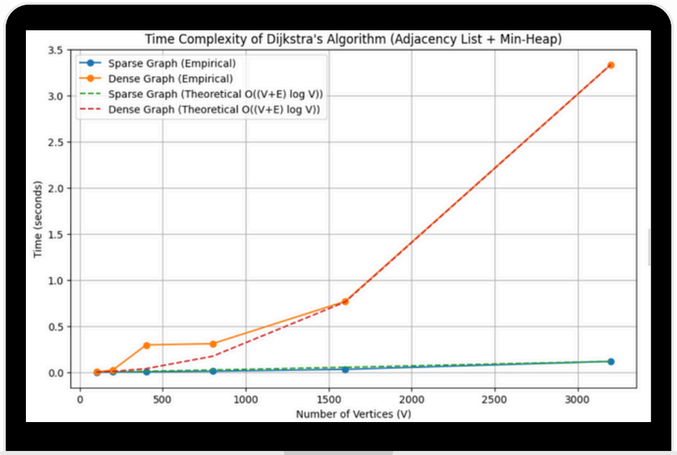
(Array+Matrix) vs (Minheap+List)



Density: Largely Unaffected
Time Complexity: O(V^2)



Comparing the time complexity:
Density (30%)



Density: Largely affected
Time Complexity: O((V+E)(logV))

Conclusion

01 Minimum Heap (Priority Queue)

- Good for sparse graph (Small |E|)
- Useful for social networks, friend’s list

02 Array (Priority Queue)

- Good for dense graph (Large |E|)
- Useful for mesh network