YOUR LOGO HERE

Project 2: Dijkstra Algorithm

SDDB Group 3

Chiang Qin Zhi

Donovan Goh Jun Heng

Gao Wen Jie

INDEX.HTML

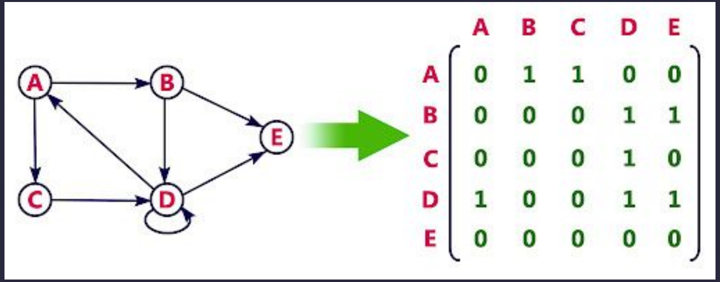
YOUR LOGO HERE

/TABLE OF CONTENTS

/01

Adjacency Matrix (Array)

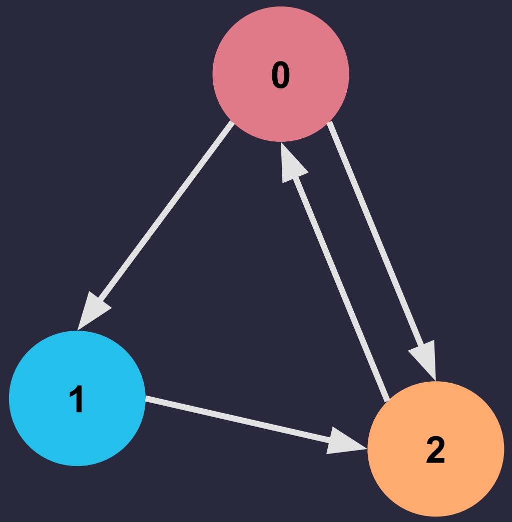
Part (a)



Adjacency Matrix

- A square matrix used to represent a finite graph.
- Elements in the matrix indicate whether the pair of vertices are adjacent in the graph
- Binary representation in matrix
  0 represent not adjacent
  1 represent adjacent

Adjacency Matrix



Node	Adjacent To
0	1, 2
1	2
2	0

Adjacency Matrix:

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Dijkstra's Algorithm

- An algorithm to find the shortest paths form a single source vertex to all other vertices in in a **weighted, directed** graph

- All weights must be **non-negative**

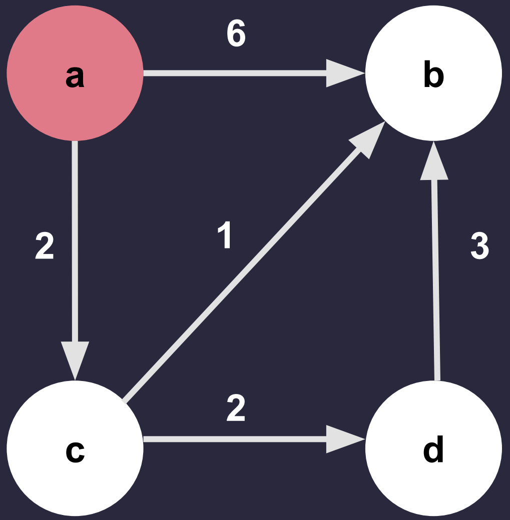
```
def dijkstra_array(self, start):
    visited = [False] * self.size
    distance = [float('inf')] * self.size
    visited[start] = True
    distance[start] = 0

    for i in range(self.size):
        # Finds closest vertex to current node that is not yet visited.
        nearest = min(range(self.size), key = lambda x: float('inf') if visited[x] else distance[x])
        visited[nearest] = True

        for node in range(self.size):
            # 1. Edge exists && 2. Node not visited && 3. Total distance from start to chosen node is < current distance to node
            if self.matrix[nearest][node] > 0 and not visited[node] and (distance[nearest] + self.matrix[nearest][node] < distance[node]):
                distance[node] = distance[nearest] + self.matrix[nearest][node]

    return distance
```

Dijkstra's Algorithm



Step 1:

- Initialising visited and distance arrays

Step 2:

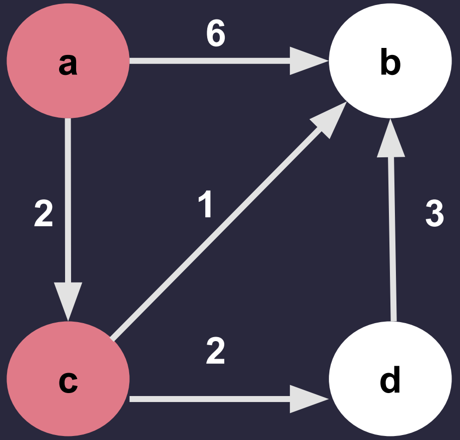
- Start from node 0 and mark as visited

- Update distance array for a

distance

a	b	c	d
0	inf	inf	inf

Dijkstra's Algorithm



Step 3:

- Find the nearest vertex to current vertex that has not been visited
- Mark the nearest vertex as visited
- Update nearest array for all adjacent vertices
- Update distance array if new calculated total distance from start to chosen node < current stored distance.

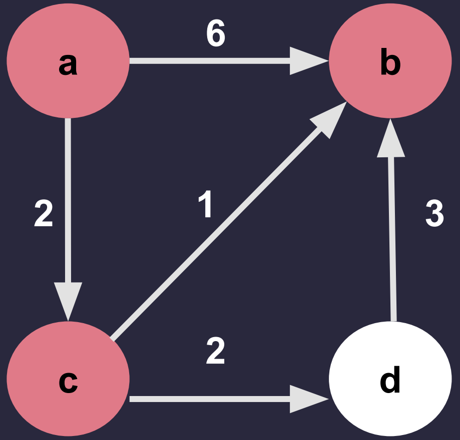
distance

a	b	c	d
0	6	2	inf

nearest

a	b	c	d
null	a	a	null

Dijkstra's Algorithm



Step 4:

- Keep repeating previous steps until all vertices are visited

distance

a	b	c	d
0	3	2	4

nearest

a	b	c	d
null	c	a	c

Implementation of Random Graphs and Time-measurement

```python
import time
import random
import matplotlib.pyplot as plt

random.seed(1234)

def generate_random_graph_adjMatrix(num_vertices, num_edges):
    graph = GraphMatrix(num_vertices)

    for _ in range(num_edges):
        startNode = random.randint(0, num_vertices- 1)
        endNode = random.randint(0, num_vertices - 1)
        cost = random.randint(1, 10)
        graph.addEdge(startNode, endNode, cost)

    return graph

def measure_time_adjMatrix(graph, start_vertex=0):
    start_time = time.time()
    graph.dijkstra_array(start_vertex)
    end_time = time.time()
    return end_time - start_time
```

```python
def empirical_test_adjMatrix(num_runs):
    vertices = [10, 50, 100, 200, 400, 600, 800, 1000, 5000]  # Different graph sizes to test
    edge_factors = [0.1, 0.2, 0.4, 0.8, 1]  # Different edge factors to test
    average_times = {factor: [0] * len(vertices) for factor in edge_factors}

    # Summing times each run
    for run in range(num_runs):
        random.seed(run)
        for factor in edge_factors:
            for i, vertex_no in enumerate(vertices):
                max_edges = int(factor * (vertex_no * (vertex_no - 1)))
                graph = generate_random_graph_adjMatrix(vertex_no, max_edges)
                execution_time = measure_time_adjMatrix(graph)
                average_times[factor][i] += execution_time
                #print(f"Vertices: {vertex_no}, Edge Factor: {num_edges}, Time: {execution_time:.4f} seconds")

    # Calculating the average times
    for factor in edge_factors:
        for i in range(len(vertices)):
            average_times[factor][i] /= num_runs

    # Saving results to csv.
    with open("Average Results (Adj Matrix).csv", 'w') as file:
        header = 'Vertices,' + ','.join([str(factor) for factor in edge_factors]) + '\n'
        file.write(header)
        for i, v in enumerate(vertices):
            row = f"{v}," + ','.join([str(average_times[factor][i]) for factor in edge_factors]) + '\n'
            file.write(row)

    # Plotting
    for factor, execution_times in average_times.items():
        plt.plot(vertices, execution_times, label=f'Edge Factor: {factor}')
    plt.xlabel('Number of Vertices')
    plt.ylabel('Execution Time (seconds)')
    plt.title('Dijkstra Algorithm (Adj Matrix) Execution Time')
    plt.legend()
    plt.show()

    return average_times
```

Graphs with varying numbers of vertices and edge factors (percentage of max possible edges) were generated, while keeping one factor fixed

Time Complexity of Dijkstra algorithm

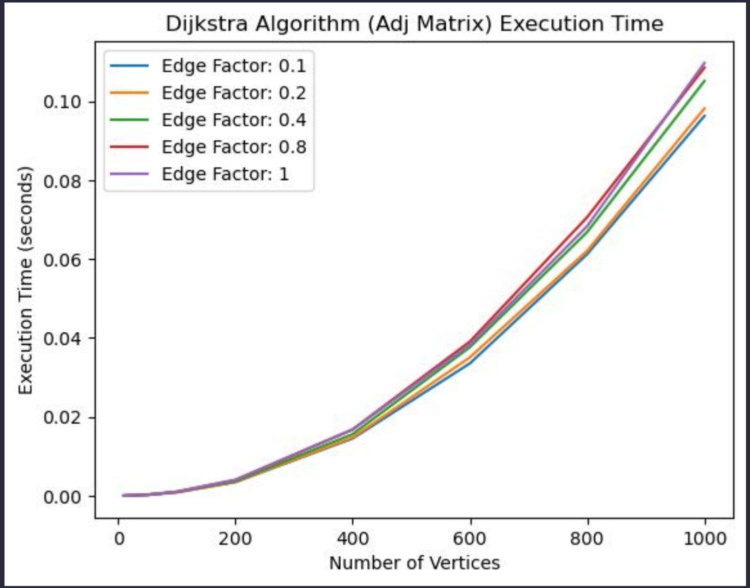
Theoretically

We need to check each vertice to find the unvisited vertex with the shortest path which requires O(V) time.

We need to check every vertex's neighbours, and since each vertex can only have maximum V-1 neighbours, it takes [O(V) * O(1)] = O(V) time to update every vertex's neighbours

O(V) * O(V) = O(V^2)

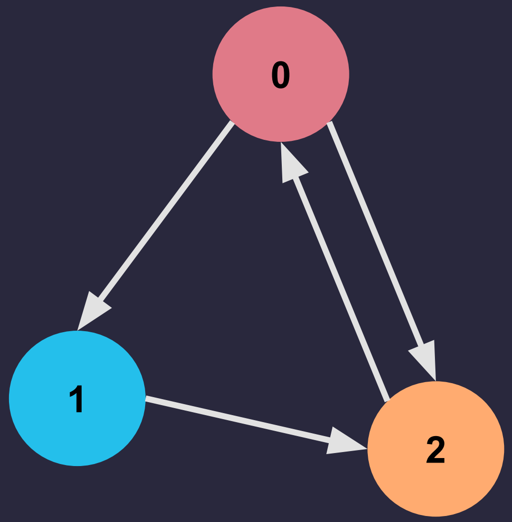
Empirically



/02

Adjacency List (Minimizing Heap)

Part (B)



Adjacency List

- A collection of unordered lists used to represent a finite graph.

Node	Adjacent To
0	1, 2
1	2
2	0

Adjacency List:

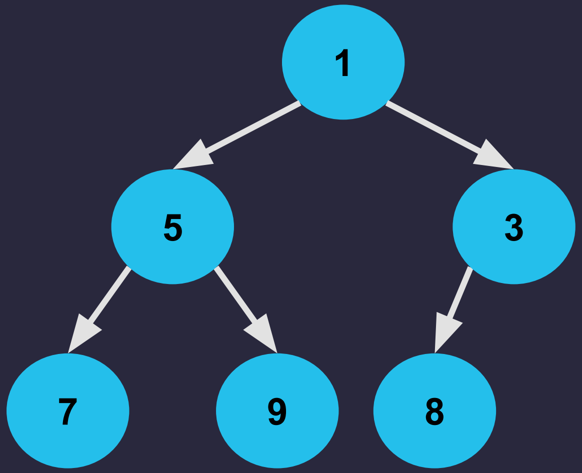
[[1,2] , [2] , [0]]

Minimizing Heap

- A complete binary tree where value in each parent node ⩽ values in its child nodes.

- Typically represented as an array with root at Arr[0].

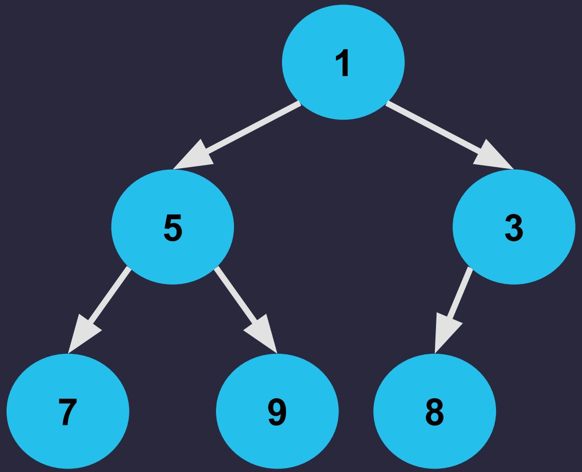
- For ith node, i.e. Arr[i]
    - Parent: Arr[(i - 1) // 2]
    - Left Child: Arr[2i + 1]
    - Right Child: Arr[2i + 2]



Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Minimizing Heap

```
Insert         : O(log N)
Extract Min    : O(log N)
Find Min    : O(1)
Decrease Key: O(log N)
Build Heap     : O(N)
```



Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Dijkstra's Algorithm: Adj List & Min Heap

1. Create a minimising heap for priority queue and initialise with source vertex & distance = 0
2. While minimum heap is not empty, pop the root vertex to get node with minimum distance/highest priority
3. For each adjacent vertex of popped vertex, calculate the new distance by adding the weight of edge between the 2 vertices
    a. If new distance < current, update the distance array and update heap
4. Repeat until all vertices are popped or destination vertex attained

```
    def dijkstra minheap (self, start):
        distances = [float('inf')] * self.size
        distances[ start] = 0
        priority queue = [(0, start)]
        while priority queue:
            current dist, current vertex = heapq.heappop(priority_queue)
            if current dist > distances[current_vertex]:
                continue
            for connectedNode, cost in self.graph[current vertex].items():
                if current dist + cost < distances[connectedNode]:
                    distances[connectedNode] = current dist + cost
                    heapq.heappush(priority_queue, (distances[connectedNode],
connectedNode))
        return distances
```

Theoretical Time Complexity

Time taken to visit all vertices with Adjacency List: O(V+E)

Time taken to process vertex using MinHeap as priority queue (extract min distance): O(logV)

Total time required

O(V+E) * O(logV) = O((V+E)logV)

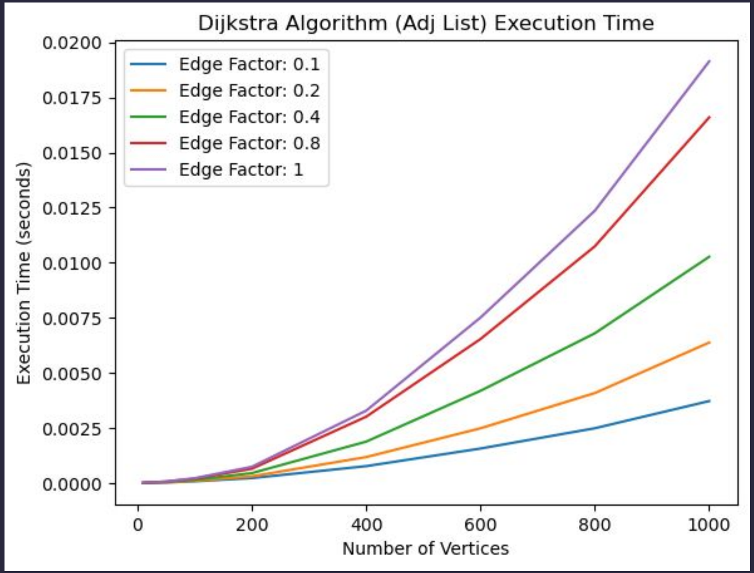
Empirical Time Complexity

Edge factor is number of edges as percentage of max edges.

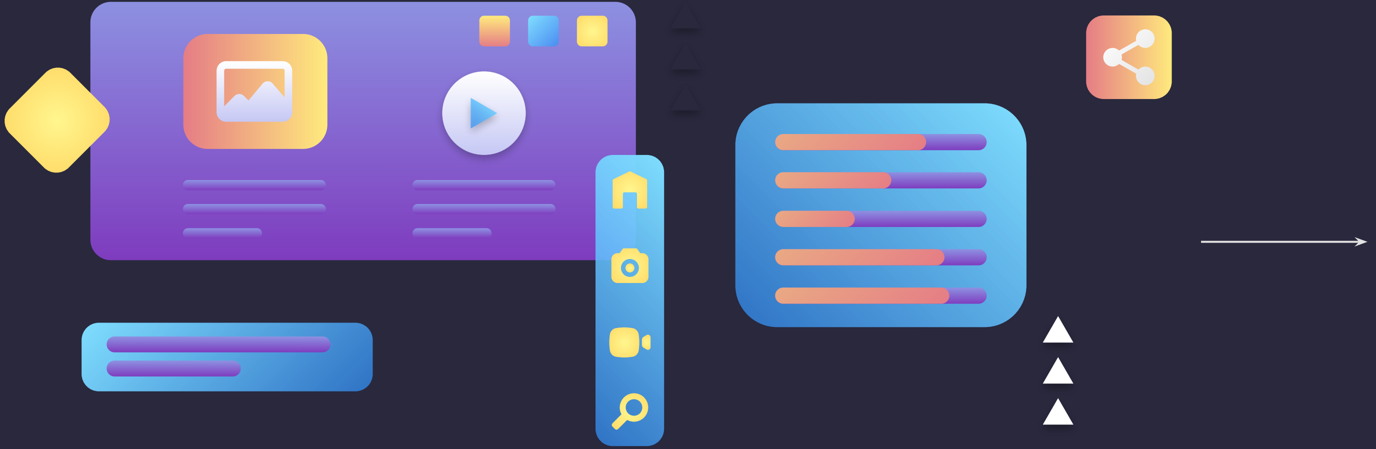
- Maximum edges in graph: **n*(n-1)**

Execution time increases as number of vertices increases. Rate of increase also increases when edges increase.

This is due to O((V+E)logV), where increases in edges and vertices increases time taken.



Average of 20 runs



/03

Comparison between Implementations

Part (c)

Adjacency Matrix

$O(n^2)$ space complexity

O(n) checking of presence of nodes
- O(1) required to check each neighbouring node

However,

➔ Difficult to add or remove the graph's vertices as there is a need to change the entire adjacency matrix

Overall, better for dense graphs

Adjacency List

O(|V|+|E|) space complexity

- space taken up will depend on the number of edges

O(E) time complexity at maximum
- Check any neighbours
- Able to dynamically grow list

Time complexity increases w |E|

Better for sparse graphs

Array vs Minimising heap (for priority queue)

Space is **equal** ≈ n

ConstructHeap() - O(n)
insert() and extractMin() - O(lgn)
getMin- O(1)

Construction - O(n)
Extract and update - O(1)
MinValue- O(n)

Minheap

Array

Conclusion

	Adjacency Matrix and Array	**Adjacency List and Minimizing Heap**
Time Complexity	$O(V^2)$	For **sparse** graphs: $O((\|V\|+\|E\|)\log\|V\|)$ For **dense** graphs, where $\|E\| = \|V\|\|V-1\|$: $O(\|V^2\|\log\|V\|)$
Space Complexity	$O(V^2) + O(V) = O(V^2)$	$O(\|V\|+\|E\|)+O(V)= O(\|V\|+\|E\|)$

For sparse graphs, we should use adjacency list and minimising heap.

For dense graphs, we should use adjacency matrix and array.

YOUR LOGO HERE

<Thank You>

INDEX.HTML

Analysis of Algo

Dijkstra_algorithm:

Similar to both O(n)

d[j]=0; pi[j]=inf, S[j]=0; initialization => heap or array

Matrix: $O(V^2)$
AdjList: O(|V|+|E|)

Heap: O(1)
Array: O(n)

For all n in S[j]

Update: Heap: O(lgn)
Array: O(1)

v = getMin(pq);

Heap is better than array.

s(

Heap: O(1+lgn) = lgn
Array: O(n+1) = n

)

distance[v] <- distance of [u, v] + s(u)