Project 2

Implementation of
Dijkstra’s Algorithm using
List and Matrices

Team 9 - SS1

Generating Data for Adjacency Matrix

```python
def sparse_mat_gen(n):
  arr = np.array(abs(rand(n,n,format="csr",random_state=69, density=0.30).todense()))
  for i in range(n):
    if arr[i][i]!=0:
      arr[i][i]=0
  return arr
```

```python
matrix = sparse_mat_gen(4)
matrix
```

```
array([[0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.25002847],
       [0.01938707, 0.        , 0.        , 0.        ],
       [0.62663939, 0.        , 0.        , 0.        ]])
```

```python
def dense_mat_gen(n):
  arr = np.array(abs(rand(n,n,format="csr",random_state=69, density=0.89).todense()))
  for i in range(n):
    if arr[i][i]!=0:
      arr[i][i]=0
  return arr
```

```python
matrix = dense_mat_gen(6)
print(matrix)
```

```
[[0.         0.59180129 0.6421361  0.17644921 0.20846031 0.03057339]
 [0.48314755 0.         0.48453604 0.1275217  0.55669823 0.7378527 ]
 [0.47607226 0.45830668 0.         0.         0.         0.72664798]
 [0.01128194 0.         0.23755617 0.         0.56398497 0.06094189]
 [0.37712284 0.56972019 0.03687652 0.37961615 0.         0.06265923]
 [0.98089937 0.         0.20017419 0.33515206 0.67630761 0.        ]]
```

Array Algorithm

```
class Graph():
    def __init__(self,vertices):
      '''intialise graph constructor using various attributes'''
      self.vertices = vertices

      # intialise the adjacency matrix to all 0s
      self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

      # False for not visited, True for visited
      self.visited = [False for i in range(vertices)]

      # initialise all distances from source to INFINITY
      self.d = [sys.maxsize for i in range(vertices)]

      # pi = array of predecessors for each vertex
      self.pi = [None for i in range(vertices)] #set all piecessor to None

      # Q = priority queue to store which vertex to visit next
      self.Q = []
```

Array Algorithm

```python
def dijkstra(self,source):
    '''source refers to the source node
      function dijkstra is to traverse the directed weighted graph
      using the Dijkstra's algorithm which is a greedy algorithm,
      along with the implementation of a priority queue, as specified
      in the question, we have used an array for the same'''

    # set the distance of the source from source as 0
    self.d[source] = 0

    # insert source into the priority queue
    self.enqueue(source)

    # while the priority queue Q is not empty
    while (len(self.Q) > 0):

            # find vertex with min distance, i.e. top of the queue
            u = self.dequeue()

            for j in range(self.vertices):
              if self.d[j] > self.d[u] + self.graph[u][j] and self.graph[u][j] != 0 and self.visited[j]!=True:

                #update the distance array accordingly
                self.d[j] = self.d[u] + self.graph[u][j]

                #update the predecessor array accordingly
                self.pi[j] = u

                #insert 'j' into the priority queue according to its d[j]
                self.enqueue(j)


    print("The shortest distance to all the vertices from source node is: ")
    print(self.d)
```

Generating Data for List

```python
def makeDenseGraph():
    # stores the number of vertices in the graph
    global denseGraph
    denseGraph = {}


    for x in range(size):
        add_vertex(x,denseGraph)

    for vertex in denseGraph:
        no_of_edges = randint(size//2,size-1)
        check=[]
        for x in range(size):
            check.append(False)
        while no_of_edges != 0:
            connected_vertex = randint(0,size-1)
            if connected_vertex != vertex and check[connected_vertex-1]==False:
                add_edge(vertex, connected_vertex, randint(1,10),denseGraph)
                no_of_edges -=  1
                check[connected_vertex-1]=True
            else:
                connected_vertex = randint(0,size-1)
    print ("Dense Graph: ", denseGraph)
```

```python
def makeSparseGraph():
    # stores the number of vertices in the graph
    global sparseGraph
    sparseGraph = {}

    for x in range(size):
        add_vertex(x,sparseGraph)

    for vertex in sparseGraph:
        no_of_edges = randint(0,size//2)
        check=[]
        for x in range(size):
            check.append(False)
        while no_of_edges != 0:
            connected_vertex = randint(0,size-1)
            if connected_vertex != vertex and check[connected_vertex-1]==False:
                add_edge(vertex, connected_vertex, randint(1,10),sparseGraph)
                no_of_edges -=  1
                check[connected_vertex-1]=True
            else:
                connected_vertex = randint(0,size-1)
    print ("Sparse Graph: ", sparseGraph)
```

Data generated

```
0 -> ( 1 , 4 ) -> ( 3 , 8 ) -> ( 12 , 5 ) -> ( 6 , 4 ) -> ( 10 , 7 ) -> ( 9 , 6 ) -> ( 2 , 7 ) -> ( 4 , 3 ) -> ( 11 , 4 ) -> ( 5 , 6 ) -> ( 13 , 3 ) -> ( 14 , 5 )
1 -> ( 14 , 8 ) -> ( 6 , 3 ) -> ( 12 , 9 ) -> ( 5 , 3 ) -> ( 3 , 2 ) -> ( 10 , 2 ) -> ( 8 , 6 ) -> ( 4 , 7 ) -> ( 7 , 8 ) -> ( 9 , 3 ) ->
2 -> ( 9 , 5 ) -> ( 12 , 5 ) -> ( 14 , 10 ) -> ( 10 , 6 ) -> ( 7 , 5 ) -> ( 6 , 4 ) -> ( 8 , 10 ) ->
3 -> ( 2 , 5 ) -> ( 9 , 7 ) -> ( 7 , 2 ) -> ( 6 , 4 ) -> ( 12 , 7 ) -> ( 4 , 4 ) -> ( 13 , 6 ) -> ( 5 , 2 ) -> ( 0 , 1 ) -> ( 10 , 2 ) -> ( 8 , 2 ) -> ( 14 , 4 ) -
4 -> ( 7 , 4 ) -> ( 11 , 5 ) -> ( 3 , 5 ) -> ( 0 , 8 ) -> ( 1 , 10 ) -> ( 8 , 8 ) -> ( 10 , 2 ) -> ( 13 , 4 ) -> ( 2 , 8 ) -> ( 14 , 5 ) -> ( 9 , 6 ) -> ( 6 , 8 )
5 -> ( 13 , 4 ) -> ( 12 , 1 ) -> ( 3 , 5 ) -> ( 2 , 3 ) -> ( 14 , 3 ) -> ( 4 , 8 ) -> ( 8 , 4 ) -> ( 6 , 2 ) -> ( 0 , 9 ) -> ( 7 , 9 ) ->
6 -> ( 4 , 8 ) -> ( 2 , 1 ) -> ( 13 , 3 ) -> ( 11 , 1 ) -> ( 3 , 2 ) -> ( 9 , 5 ) -> ( 1 , 9 ) -> ( 12 , 8 ) -> ( 14 , 1 ) -> ( 0 , 4 ) -> ( 10 , 5 ) -> ( 5 , 5 )
7 -> ( 8 , 7 ) -> ( 10 , 7 ) -> ( 2 , 2 ) -> ( 9 , 6 ) -> ( 14 , 9 ) -> ( 11 , 6 ) -> ( 3 , 7 ) -> ( 13 , 3 ) -> ( 6 , 7 ) -> ( 1 , 2 ) -> ( 5 , 8 ) -> ( 4 , 3 ) -
8 -> ( 1 , 1 ) -> ( 12 , 2 ) -> ( 4 , 3 ) -> ( 9 , 5 ) -> ( 13 , 2 ) -> ( 6 , 7 ) -> ( 11 , 7 ) -> ( 0 , 5 ) ->
9 -> ( 2 , 10 ) -> ( 14 , 8 ) -> ( 7 , 3 ) -> ( 0 , 1 ) -> ( 12 , 8 ) -> ( 4 , 5 ) -> ( 3 , 4 ) -> ( 13 , 4 ) -> ( 8 , 10 ) -> ( 6 , 1 ) -> ( 5 , 1 ) -> ( 1 , 6 )
```

```
0 -> ( 14 , 6 ) -> ( 5 , 4 ) -> ( 3 , 3 ) ->
1 -> ( 9 , 2 ) -> ( 10 , 5 ) -> ( 13 , 1 ) -> ( 11 , 2 ) -> ( 0 , 5 ) -> ( 8 , 4 ) -> ( 5 , 7 ) ->
2 -> ( 10 , 4 ) -> ( 4 , 7 ) -> ( 5 , 5 ) -> ( 11 , 10 ) -> ( 6 , 9 ) -> ( 14 , 7 ) -> ( 12 , 1 ) ->
3 -> ( 14 , 10 ) -> ( 11 , 5 ) ->
4 -> ( 2 , 2 ) ->
5 -> ( 7 , 9 ) -> ( 6 , 10 ) -> ( 3 , 3 ) -> ( 8 , 8 ) ->
6 -> ( 13 , 3 ) -> ( 12 , 8 ) -> ( 5 , 2 ) -> ( 2 , 7 ) -> ( 3 , 2 ) -> ( 9 , 4 ) ->
7 -> ( 13 , 7 ) -> ( 9 , 7 ) ->
8 -> ( 7 , 3 ) -> ( 11 , 2 ) -> ( 13 , 8 ) ->
9 -> ( 12 , 10 ) ->
```

Minimizing Heap Algorithm

```
class Heap():
    def __init__(self):
        self.array = []
        self.size = 0
        self.pos = []

    def newNode(self,v, dist):
        node = [v, dist]
        return node

    def swapMinHeapNode(self,a, b):
        t = self.array[a]
        self.array[a] = self.array[b]
        self.array[b] = t

    #will be heapifying the heap when parent node is removed
    def minHeap(self,parent):
        smallest = parent
        left = 2*parent+1
        right = 2*parent+2
        #To check whether the left node is smaller compared to right
        if left < self.size and self.array[left][1] < self.array[smallest][1]:
            smallest = left

        #To check whether the right node is smaller compared to left
        if right < self.size and self.array[right][1] < self.array[smallest][1]:
            smallest = right

        #it will swap the smaller child node with parent node IFF child node is smaller than parent node
        if smallest != parent:
            # Swap positions
            self.pos[self.array[smallest][0]] = parent
            self.pos[self.array[parent][0]] = smallest

            self.swapMinHeapNode(smallest,parent)

            self.minHeap(smallest)
```

```
def extractMin(self):
    # Return NULL wif heap is empty
    if self.isEmpty() == True:
        return

    # Store the root node
    root = self.array[0]

    # Replace root node with last node
    lastNode = self.array[self.size - 1]
    self.array[0] = lastNode

    # Update position of last node
    self.pos[lastNode[0]] = 0
    self.pos[root[0]] = self.size - 1

    # Reduce heap size and heapify root
    self.size -= 1
    self.minHeap(0)

    return root

def isEmpty(self):
    return True if self.size == 0 else False

def decreaseKey(self, v, dist):

    # Get the index of v in  heap array

    i = self.pos[v]

    # Get the node and update its dist value
    self.array[i][1] = dist

    # Travel up while the complete tree is
    # not hepified. This is a O(Logn) loop
    while i > 0 and self.array[i][1] < self.array[(i - 1) // 2][1]:

        # Swap this node with its parent
        self.pos[ self.array[i][0] ] = (i-1)//2
        self.pos[ self.array[(i-1)//2][0] ] = i
        self.swapMinHeapNode(i, (i - 1)//2 )

        # move to parent index
        i = (i - 1) // 2;

def isInMinHeap(self, v):

    if self.pos[v] < self.size:
        return True
    return False
```

Time Complexity

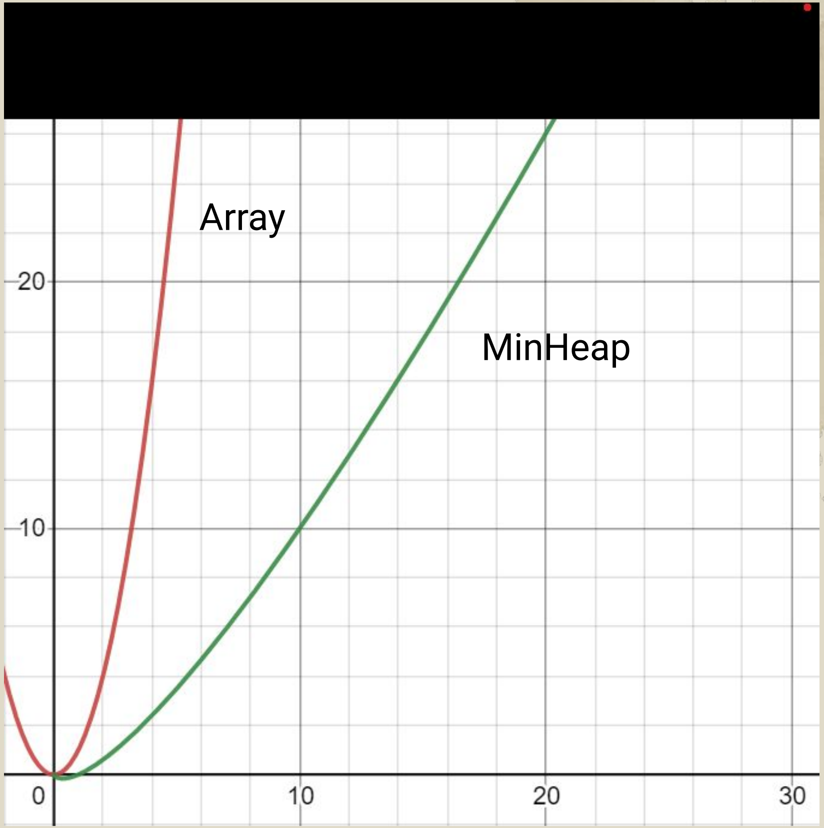
Dijkstra's Algorithm:

Insert into Priority Queue	**O(|V|)**
Extract min	**O(|V|)**
Relaxation/Decrease Key Comparison	**O(|E|)**

Array	
Extract min	**O(|V|)**
Relaxation	**O(1)**
O(|V|*|V| + |E|) = O(|V|² + |E|)	

Minimising Heap	
Extract min	**O(log |V|)**
Relaxation	**O(log |V|)**
O(|V|*log |V| + |E|*log |V|) = O(|V| log |V| + |E| log |V|)	

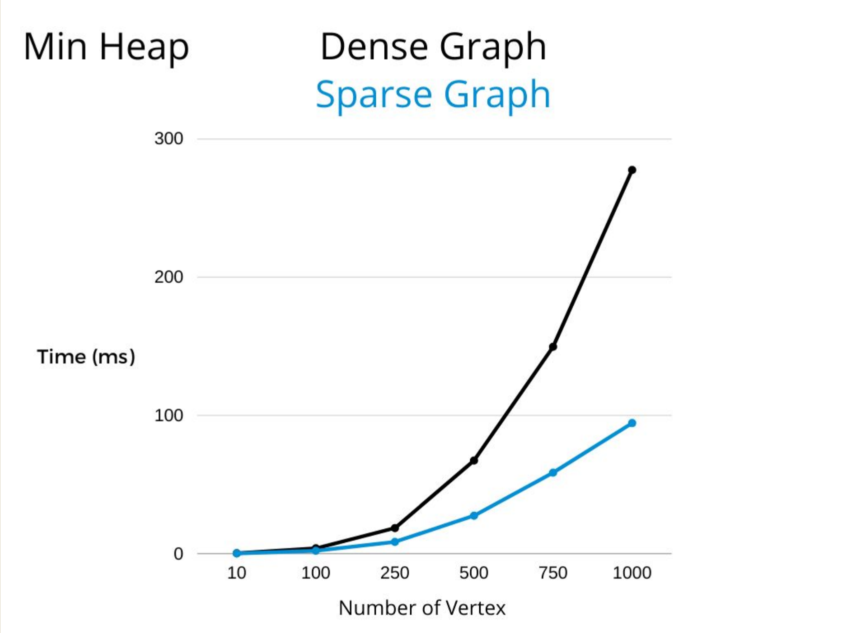
Mathematical Modeling of Time Complexity

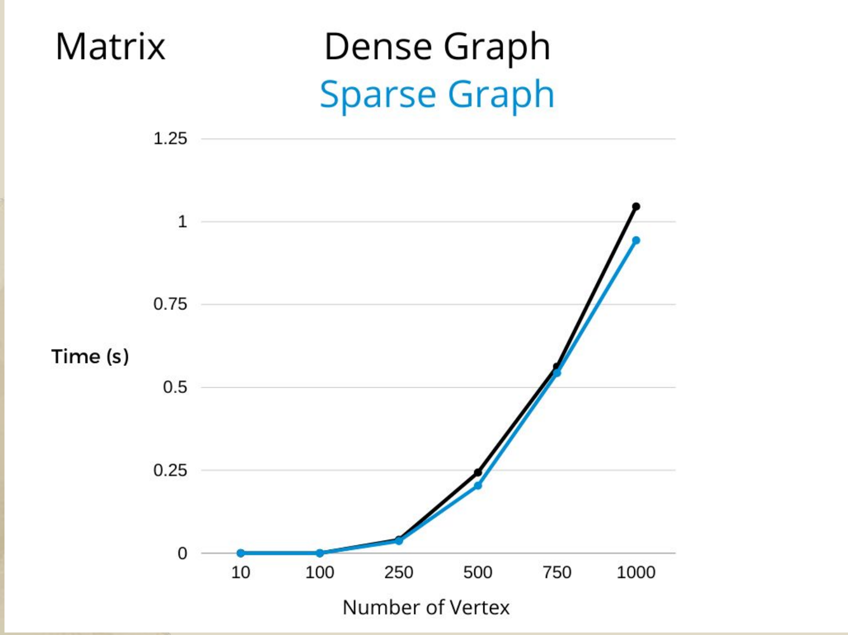


Empirical Time Complexity

Graph	Number of Vertex	Array	Minimizing Heap
Dense	10	3.80039×10^{-7}	2.17826×10^{-4}
	100	4.45259×10^{-5}	3.75468×10^{-3}
	250	0.039949	0.018481
	500	0.24348	0.067302
	750	0.56292	0.14963
	1000	1.04577	0.27761
Sparse	10	2.73466×10^{-7}	1.13496×10^{-5}
	100	4.22599×10^{-5}	2.03583×10^{-3}
	250	0.036412	0.0084645
	500	0.20357	0.027383
	750	0.54340	0.058521
	1000	0.94379	0.094407

Empirical Time Complexity





Conclusion

There is a positive relation between the following and CPU runtime:

1) Number of Vertices (|V|)
2) Type of Graph (|E|, number of edges)
3) Algorithm used

Both Minimizing Heap and Array priority queue are greatly affected by the number of vertex, but Minimizing Heap is also greatly affected by number of edges compared to Array.

Array Priority Queue has a slower run time as compared to Minimizing Heap when the number of vertices goes beyond 100.

Thank You!