Report on Networking Routing

Sabal Subedi
Master's in Computer Science
Idaho State University
Pocatello, ID, 83209 USA
sabalsubedi@isu.edu

February 27, 2024

Abstract

This project report summarizes the use of the Dijkstra's algorithm to find paths through a graph representing a network routing problem.

1 Introduction

Dijkstra's algorithm is widely used algorithm in graph theory for finding the shortest path between nodes in a graph with non-negative edge weights.
Dijkstra's algorithm relies heavily on the use of a priority queue to efficiently select the next node to visit during the exploration of the graph. The priority queue is used to keep track of the nodes that are candidates for visiting, with the priority being determined by the tentative distance from the start node.
I will use two of such priority queue: array and binary heap, to analyze the Asymptotic notation of Dijkstra's algorithm.

2 Pseudocode and Asymptotic Analysis

2.1 Pseudocode for unsorted array priority queue

```
1     def dijkstra_implementataion(self, srcIndex,
         use_heap=False):
2         pq = object of array queue if not use_heap
             else object of binary heap queue
3         # initializing the prev array to nil
4         self.prev_array = [None] *
             len(self.network.nodes)
5
6         # Start with the source node and updating the
             dist value to 0
7         pq.insert(srcIndex)
8         pq.decrease_key(0, srcIndex)
9
10        # iterate until there is node in queue
11        while pq.queue:
12            curr_min_node_index = pq.delete_min()
13            curr_node =
                 self.network.nodes[curr_min_node_index]
14
15            # Check each edge from the lowest
                 distance node
16            for edge in curr_node.neighbors:
17                node_index = edge.dest.node_id
18                new_distance =
                     pq.dist[curr_min_node_index] +
                     edge.length
19                if new_distance < pq.dist[node_index]:
20                    pq.insert(node_index)
21                    self.prev_array[node_index] =
                         curr_min_node_index
22                    pq.decrease_key(new_distance,
                         node_index)
```

Time complexity and space complexity
Using unsorted Array

1. insert method takes $O(1)$ time and $O(1)$ space; this method appends new node to the queue
2. decrease_key method takes $O(1)$ time and $O(1)$ space; this method updates the distance of node
3. delete_min method takes $O(n)$ times and $O(1)$ space; this method scans the queue and return the the node with minimum distance value

$$Time\_complexity = O(1) + O(1) + O(n) * (O(1) + O(n)) \approx O(n^2) \approx O(|V|^2)$$
$$Space\_complexity = O(n) \approx O(|V|)$$
$$where,\ V\ is\ verticies\ in\ graph$$

(1)

Using binary heap

1. insert method takes $O(\log n)$ time and $O(1)$ space; this method appends new node to the queue
2. decrease_key method has two function calls
   bubble_up takes $O(\log n)$ times; this method switch the parent and child node in a tree of depth $O(\log n)$
   swap_nodes takes $O(1)$ time
   and space complexity is $O(1)$ space
3. delete_min method has two function calls
   bubble_down takes $O(\log n)$ times; this method switch the root and child node in a tree of depth $O(\log n)$
   swap_nodes takes $O(1)$ time
   space complexity is $O(1)$ space

$$Time\_complexity = O(1) + O(\log n) + O(n) * (O(\log n) + O(\log n)) \approx O(n \log n) \approx O(|V| \log |V|)$$
$$Space\_complexity = O(n) \approx O(|V|)$$
$$where,\ V\ is\ verticies\ in\ graph$$

(2)

2.2 Pseudocode for unsorted array priority queue

```
1     class ArrayPriorityQueue:
2         def __init__(self, nodes):
3             # initializing the dist array to infinite
4             self.dist = [math.inf] * len(nodes)
5             self.queue = []
6
7         # adds new item in queue
8         def insert(self, node_index):
9             self.queue.append(node_index)
10
11        # updates the distance value of a node
12        def decrease_key(self, dist, node_index):
13            self.dist[node_index] = dist
14
15        # finds a node with minimum distance and
              return it after removing it from the queue
16        def delete_min(self):
17            if not self.queue:
18                return None
19
20            min_node = None
21            min_distance = math.inf
22
23            for node in self.queue:
24                if self.dist[node] < min_distance:
25                    min_node = node
26                    min_distance = self.dist[node]
27
28            self.queue.remove(min_node)
29            return min_node
```

Time complexity and space complexity

1. initialization takes $O(1)$ times to initialize the dist, and queue, and $O(n)$ space to store the dist, and queue
2. insert method takes $O(1)$ time and $O(1)$ space
3. decrease_key method takes $O(1)$ time and $O(1)$ space
4. delete_min method takes $O(n)$ times and $O(1)$ space

$$
\begin{aligned}
&Time\_complexity = O(1) + O(1) + O(1) + O(n) \approx O(n) \approx O(|V|) \\
&Space\_complexity = O(n) + O(1) + O(1) + O(1) \approx O(n) \approx O(|V|) \\
&where,\ V\ is\ verticies\ in\ graph
\end{aligned} \tag{3}
$$

2.3 Pseudocode for binary heap priority queue

```
1     class HeapPriorityQueue:
2     def __init__(self, nodes):
3         # initializing the distance to infinity
4         self.dist = [math.inf] * len(nodes)
5         self.queue = []
6         # initializing the pointer of the tree to
              infinity
7         self.q_pointer = [math.inf] * len(nodes)
8
9     # add a node into the heap
10    def insert(self, node_index):
11        self.queue.append(node_index)
12        self.q_pointer[node_index] = len(self.queue)
              - 1
13        self.bubble_up(len(self.queue) - 1)
14
15    # adjust the distance update of nodes
16    def decrease_key(self, dist, node_index):
17        self.dist[node_index] = dist
18        self.bubble_up(self.q_pointer[node_index])
19
```

```
20      # bubble_up maintains the property of heap after
           insertion or decrease key operation
21      def bubble_up(self, child_index):
22          while child_index > 0:
23              parent_index = (child_index - 1) // 2
24              if (
25                  self.dist[self.queue[child_index]]
26                  >= self.dist[self.queue[parent_index]]
27              ):
28                  break
29              self.swap_nodes(child_index, parent_index)
30              child_index = parent_index
31
32      # return the minimum element from the heap and
           removes it
33      def delete_min(self):
34          if not self.queue:
35              return None
36
37          min_index = self.queue[0]
38          if len(self.queue) == 1:
39              self.queue.pop()
40              return min_index
41
42          last_index = len(self.queue) - 1
43          self.swap_nodes(0, last_index)
44          self.queue.pop() # swaps the item at the end
               with root and return the root node
45          self.bubble_down(0)
46
47          return min_index
48
49      # restores the heap property starting from the
           given parent_index
50      def bubble_down(self, parent_index):
51      while True:
52          left_child_index = 2 * parent_index + 1
53          if left_child_index >= len(self.queue):
```

```
54              break
55
56          child_index_left = left_child_index
57          child_index_right = left_child_index + 1
58          if (
59              child_index_right < len(self.queue)
60              and
                   self.dist[self.queue[child_index_right]]
61              < self.dist[self.queue[left_child_index]]
62          ):
63              child_index_left = child_index_right
64
65          # checking the distance of parent node and
               the smallest child node
66          if (
67              self.dist[self.queue[parent_index]]
68              <= self.dist[self.queue[child_index_left]]
69          ):
70              break
71
72          self.swap_nodes(parent_index,
               child_index_left)
73          parent_index = child_index_left
74
75      # perform the swap between nodes
76      def swap_nodes(self, i, j):
77          self.queue[i], self.queue[j] = self.queue[j],
               self.queue[i]
78          self.q_pointer[self.queue[i]] = i
79          self.q_pointer[self.queue[j]] = j
```

Time complexity and space complexity

1. initialization takes $O(1)$ times to create the dist, queue, and pointer,
   and $O(n)$ space to store the dist, queue, and pointer
2. insert method takes $O(\log n)$ time and $O(1)$ space
3. decrease_key method has two function calls
   bubble_up takes $O(\log n)$ times
   swap_nodes takes $O(1)$ time
   and space complexity is $O(1)$ space
4. delete_min method has two function calls
   bubble_down takes $O(\log n)$ times
   swap_nodes takes $O(1)$ time
   space complexity is $O(1)$ space

$$Time\_complexity = O(1) + O(\log n) + O(\log n) + O(\log n) \approx O(\log n) \approx O(\log |V|)$$
$$Space\_complexity = O(n) + O(1) + O(1) + O(1) + O(1) \approx O(n) \approx O(|V|)$$
$$where,\ V\ is\ verticies\ in\ graph$$

(4)

3 Observations and Results

3.1 Observation Table

Queue Type	Size	1	2	3	4	5	Mean time
Array	100	0.0000	0.0000	0.0000	0.0000	0.0010	0.0002
Array	1000	0.0109	0.0096	0.0119	0.0095	0.0101	0.0104
Array	10000	0.9208	1.1623	1.2870	1.2569	1.1836	1.1621
Array	100000	135.5281	122.2875	141.2688	165.9068	130.2860	139.0554
Binary Heap	100	0.0000	0.0010	0.0000	0.0010	0.0000	0.0002
Binary Heap	1000	0.0050	0.0050	0.0045	0.0055	0.0051	0.0050
Binary Heap	10000	0.0738	0.0951	1.1093	0.1054	1.1090	0.4985
Binary Heap	100000	1.1646	1.0881	1.3380	1.1206	1.1153	1.1652
Binary Heap	1000000	21.2837	20.0075	20.0632	20.4115	20.2848	20.4101

Table 1: Time taken to run Dijkstra's Algorithm in different priority queue

The table (in Table 1) shows different time taken to run the different sample size with random seed, source and target. Observing the obtained data, I can estimate the time needed to compute the tree using array with size 1000000. The time taken by the array is growing at a rate of 10^2.
To estimate the run time of Array for size 1000000, I can use the average time taken by array queue to get the tree for size 100000 using following formula,

$$\begin{aligned} Array\ queue\ of\ size\ 1000000 &= ((1000000/100000)^2) * avg\ time\ for\ array\ of\ size\ 100000 \\ &= 100 * 139.0554 \\ &= 13905.544\ sec \end{aligned}$$

.

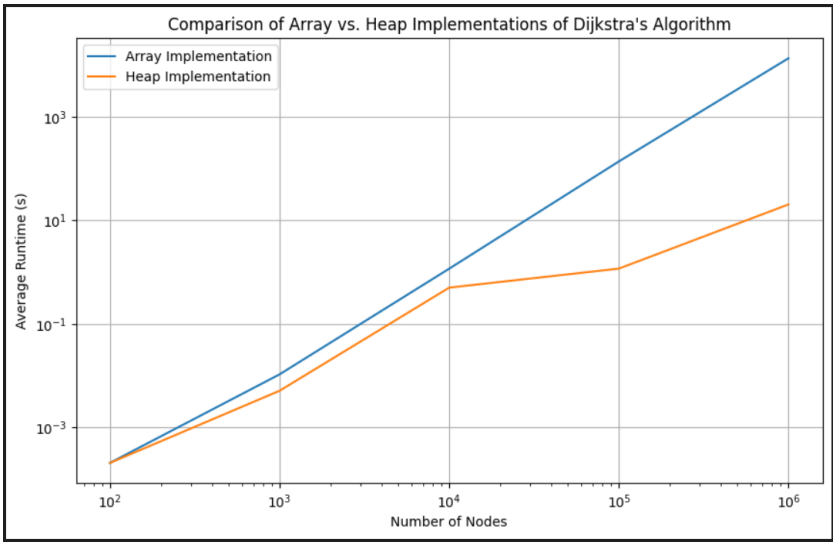

Figure 1: Plot between the different sample sizes and the time taken by the project

From the figure (in figure 1), I can see the time taken by Dijkstra's algorithm using the array priority queue vs the Dijkstra's algorithm using the binary heap. We can observe that the binary heap can compute the Dijkstra algorithm faster then the array queue. As the size of sample increases, the binary heap computes the tree comparatively faster i.e. takes less time.

3.2 Results

.

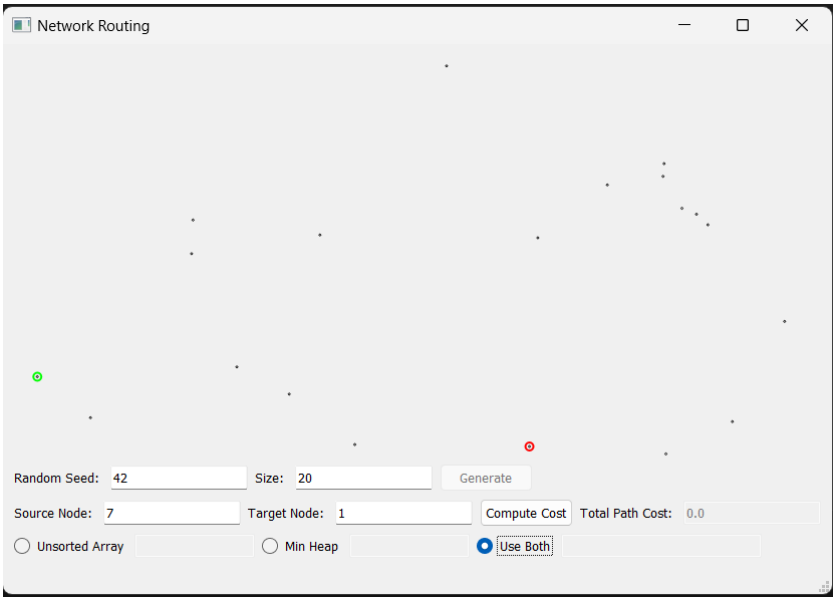

Figure 2: For Random seed 42 - Size 20, use node 7 (the left-most node) as the source and node 1 (on the bottom toward the right) as the destination

The figure (in figure 2) shows the 20 sample generated using seed 42. The source node is 7 and the target node is 1. But the target was unreachable from the given source. Thus, I could not include the time taken by the program and conclude that the target is unreachable.

.

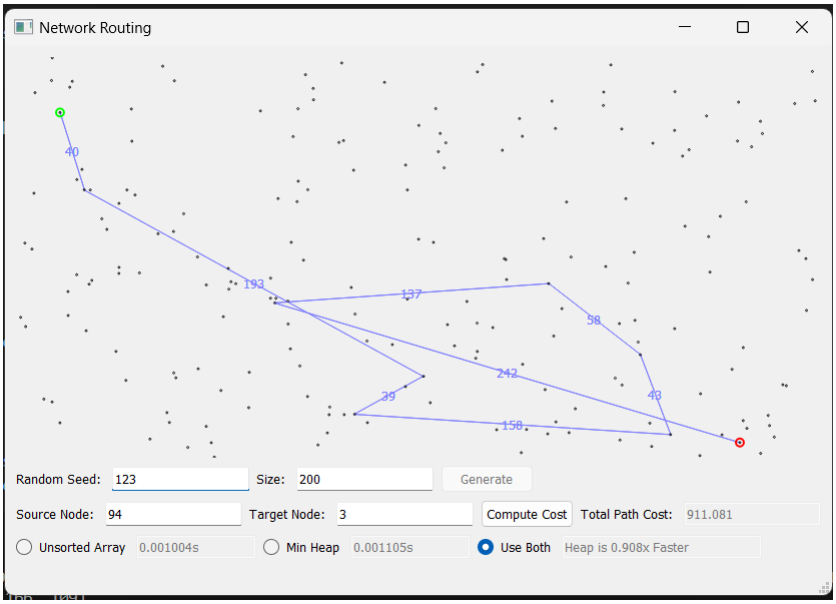

Figure 3: For Random seed 123 - Size 200, use node 94 (near the upper left) as the source and node 3 (near the lower right) as the destination

The figure (in figure 3) shows the 200 sample generated using seed 123. The source node is 94 and the target node is 3. The total time taken to compute the Dijkstra's algorithm using array is 0.00100 sec and for binary heap it took 0.001105 sec i.e. the heap is 0.908 times faster. And the total cost of tree is 911.081.

.

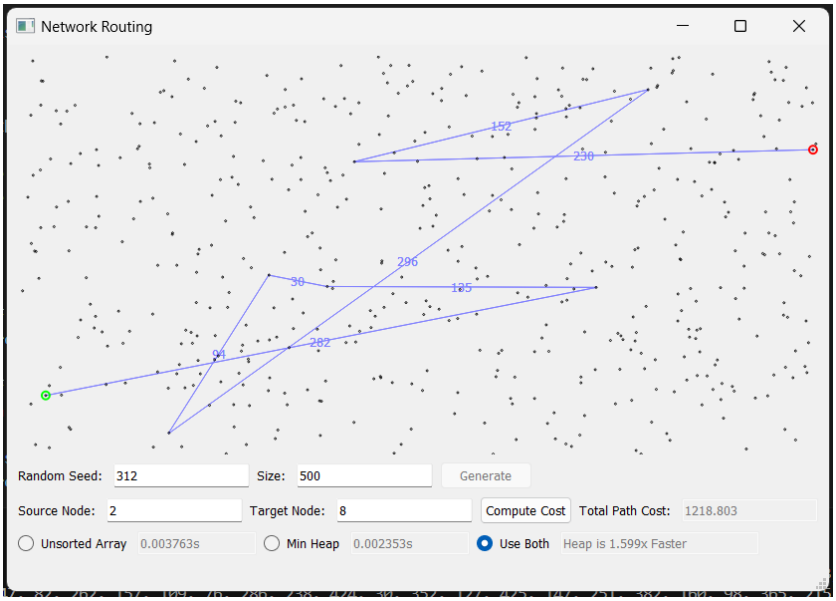

Figure 4: For Random seed 312 - Size 500, use node 2 (near the lower left) as the source and node 8 (near the upper right) as the destination

The figure (in figure 4) shows the 500 sample generated using seed 312. The source node is 2 and the target node is 8. The total time taken to compute the Dijkstra's algorithm using array is 0.0.00376 sec and for binary heap it took 0.0.00235 sec i.e. the heap is 1.5999 times faster. And the total cost of tree is 1218.803.

4 Conclusion

I was able to generate the code to compute the Dijkstra's algorithm using the array and binary priority queue. I was able to estimate the time taken to compute the sample size of 1000000 for array queue. Also, I have analyzed the time complexity of both the priority queue.