**Dijkstra’s Algorithm**

**Iheb Gafsi**\*

INSAT Student

**Iheb.engineer@gmail.com**

**Definition:**

Dijkstra's algorithm is a popular graph traversal and shortest path finding algorithm. Given a weighted graph with non-negative edge weights and a source node, it efficiently computes the shortest path from the source node to all other nodes in the graph. The algorithm maintains a priority queue of nodes, starting with the source node and iteratively expands to find the next closest node. It keeps track of the minimum distance from the source to each node and updates the distances whenever a shorter path is found. Dijkstra's algorithm is known for its optimality when dealing with non-negative edge weights and can be implemented using various data structures like priority queues or min-heaps to achieve time complexity of , where V is the number of vertices and E is the number of edges in the graph.

**Lazy Dijkstra’s Algorithm:**

We first have to create a distances array where the distance to every node is positive infinity and the starting node has a distance of 0.

Then we should create a priority queue of key-value pairs of (node\_index, distance) tuple which will help us figure out which node to visit based on sorted minimum value.

Then insert (s, 0) into the PQ and loop while PG is not empty pulling out the next most promising tuple.

Then Iterate over all edges outwards from the current node and relax them and appending the tuple to the PQ for every relaxation.

1. #Global Variables

2. graph = adjacency list

3. s = starting node index

4.

5. #Dijkstra’s Algorithm

6. def dijkstra(graph, start):

7. n = len(graph)

8. distances = [float('inf')] \* n

9. pq = PriorityQueue()

10. visited = set()

11. distances[start] = 0

12. pq.put((start, 0))

13. while not pq.empty():

14. i, d = pq.get()

15. visited.add(i)

16. for el in graph[i]:

17. if el[0] in visited:

18. continue

19. dist = d + el[1]

20. if dist < distances[ el[0] ]:

21. distances[ el[0] ] = dist

22. pq.put( (el[0], dist) )

23. return distances

24.

25. print(dijkstra(graph, 0))

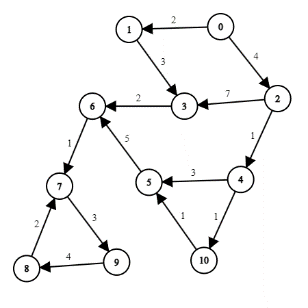
The reason we use an (index, value) tuple is because it’s faster at a time complexity of rather than using a key which is a lot slower at a linear time complexity.

**Eager Dijkstra’s Algorithm:**

Lazy Dijkstra is called lazy because we lazily remove the edges that are out of date, therefore to optimize our task, we can use an indexed priority queue to update these elements. You can try that out on your own but the lazy Dijkstra is the one that is mostly used.

**Example:**

Here’s a small example illustrating an example of input outputs for the Dijkstra:



We will use the Python code down below to outline the output of the algorithm on this graph:

1. from queue import PriorityQueue

2.

3. # Variables

4. graph = [

5.     [(1, 2), (2, 4)],

6.     [(3, 3)],

7.     [(3, 7), (4, 1)],

8.     [(6, 2)],

9.     [(5, 3), (10, 1)],

10.     [(6, 5)],

11.     [(7, 1)],

12.     [(9, 3)],

13.     [(7, 2)],

14.     [(8, 4)],

15.     [(5, 1)]

16. ]

17.

18. def dijkstra(graph, start):

19.     n = len(graph)

20.     distances = [float('inf')] \* n

21.     pq = PriorityQueue()

22.     visited = set()

23.

24.     distances[start] = 0

25.     pq.put((start, 0))

26.

27.     while not pq.empty():

28.         i, d = pq.get()

29.         visited.add(i)

30.

31.         for el in graph[i]:

32.             if el[0] in visited:

33.                 continue

34.             dist = d + el[1]

35.             if dist < distances[ el[0] ]:

36.                 distances[ el[0] ] = dist

37.                 pq.put( (el[0], dist) )

38.     return distances

39.

40. print(dijkstra(graph, 0))

The corresponding output is:

1. Python>> [0, 2, 4, 5, 5, 7, 7, 8, 15, 11, 6]

