Graphs

1 Definition

Graphs in algorithms refer to a fundamental data structure used to represent a network of nodes (also known as vertices) and the connections between them, called edges. In math notation, we define graph as:

$$G = (V, E) \tag{1}$$

Here:

1. V: nodes
2. E: edges between pairs of nodes (undirected, directed, weighted)
3. n: number of nodes
4. m: number of edges

2 Adjacency Matrix

The adjacency matrix is a way of representing a graph as a matrix of booleans or integers. In this matrix, the rows and columns represent the vertices (nodes) of the graph, and the matrix elements indicate whether pairs of vertices are adjacent or not in the graph. Some general characteristics for all kinds of n vertices and m edges:

1. Space proportional to n^2.
2. Checking if (u, v) is an edge takes $\Theta(1)$ time.
3. Identifying all edges takes $\Theta(n^2)$ time.

2.1 Undirected Graph

An undirected graph is a type of graph in which the edges have no orientation or direction. One example is shown below:

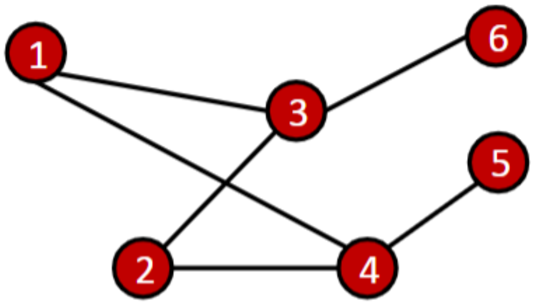
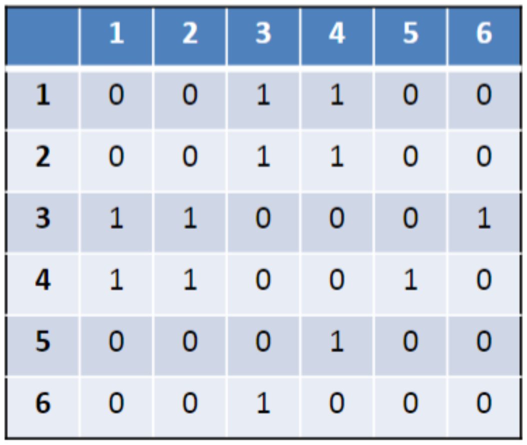


Figure 1: Undirected Graph



	1	2	3	4	5	6
1	0	0	1	1	0	0
2	0	0	1	1	0	0
3	1	1	0	0	0	1
4	1	1	0	0	1	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0

Figure 2: Undirected Graph Matrix

Notice that the undirected graph is **binary and symmetric**.

2.2 Directed Graph

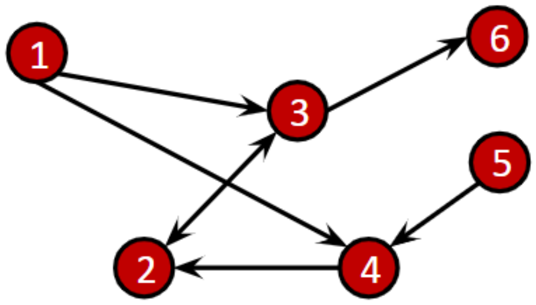


Figure 3: Directed Graph

	1	2	3	4	5	6
1	0	0	1	1	0	0
2	0	0	1	0	0	0
3	0	1	0	0	0	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	0

Figure 4: Directed Graph Matrix

Notice that the directed graph is **binary and not symmetric**.

2.3 Weighted Directed Graph

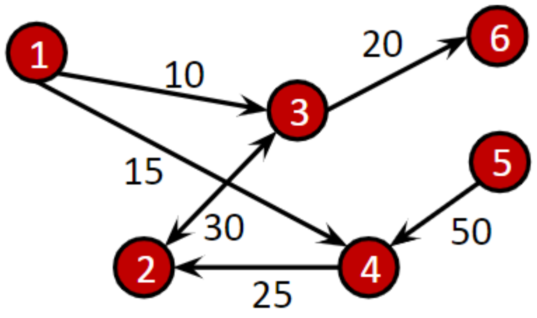


Figure 5: Weighted Directed Graph

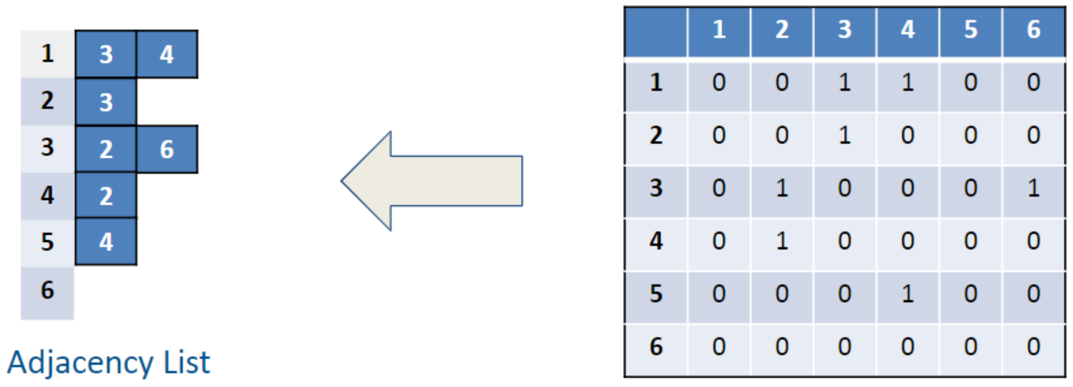
	1	2	3	4	5	6
1	0	0	10	15	0	0
2	0	0	30	0	0	0
3	0	30	0	0	0	20
4	0	25	0	0	0	0
5	0	0	0	50	0	0
6	0	0	0	0	0	0

Figure 6: Weighted Directed Graph Matrix

Notice that the directed graph is **not binary and not symmetric**.

2.4 Adjacency List

The adjacency matrix is actually a sparse matrix. Therefore, we can simplify this matrix into an adjacency list:



1	3	4
2	3	
3	2	6
4	2	
5	4	
6		

	1	2	3	4	5	6
1	0	0	1	1	0	0
2	0	0	1	0	0	0
3	0	1	0	0	0	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	0

Figure 7: Adjacency List

Recall that n is the number of nodes, m is the number of edges, there are some general characteristics of adjacency matrix:

1. Space proportional to $m + n$.
2. Checking if (u, v) is an edge takes $O(\deg(u))$ time. In an adjacency list, each vertex has its own list of adjacent vertices. **The degree of a vertex** u is denoted as $\deg(u)$. Therefore, to find an edge (u, v) we must traverse the list of neighbors of u.
3. Identifying all edges takes $\Theta(m + n)$ time.

3 Shortest Path Problem

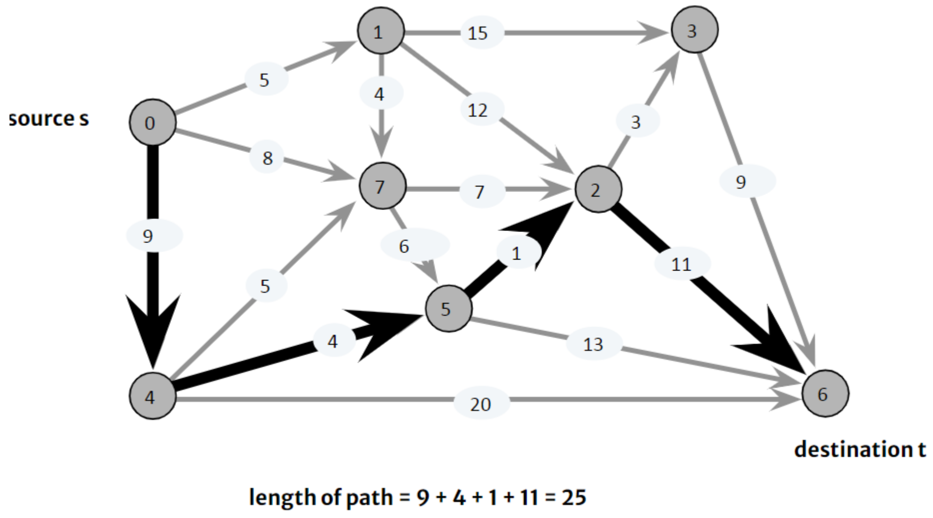


Figure 8: Shortest Path Problem

Given a directed graph $G = (V, E)$, edge lengths $l_e \geq 0$, source $s \in V$, find a shortest directed path from s to every node. The basic assumption is that **there exists a path from s to every node.**

4 Dijkstra's Algorithm

Dijkstra's algorithm is designed for finding the shortest path from a starting node to all other nodes in a graph with **non-negative** edge weights. The algorithm works by **iteratively picking the unvisited node with the lowest distance from the start, calculating the distance through it to each unvisited neighbor, and updating the neighbor's distance if it is smaller. The process is repeated until all nodes have been visited. The result is the shortest path from the start node to every other node in the graph.**

4.1 Math Expressions

Define the following parameters:

1. S: a set of explored nodes
2. u: a node in S, an explored node
3. v: a node not is S, an unexplored node
4. s: starting node
5. $d(u)$: shortest path distance

The algorithm is shown below:

1. Initialize the explored node set by $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$
2. Repeatedly choose unexplored node $v \notin S$ which minimizes:

$$\pi(v) = \min_{e=(u,v):u\in S} (d[u] + l_e) \tag{2}$$

   Here, $d[u] + l_e$ is the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

4.2 Demo

Assume we have this directed weight graph:

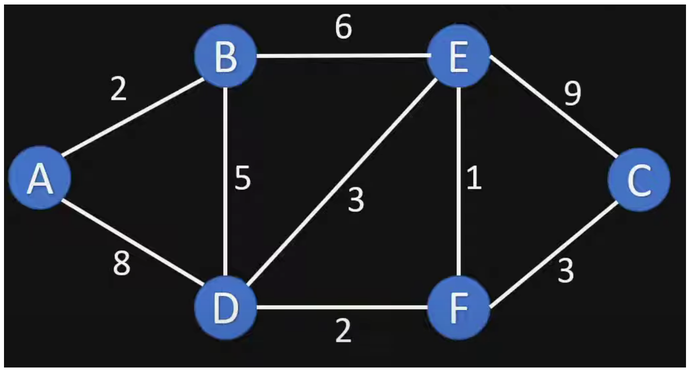


Figure 9: Directed Weight Graph

We want to calculate the shortest distance from A to each other node. Now the visited node set S is empty, and the shortest distances are summarized below:

Node	Shortest Distance	Previous Node
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-
F	∞	-

4.2.1 A

The steps include:

1. Explore all the edges of A point, we have B and D connected.
2. Update the table by their weights.

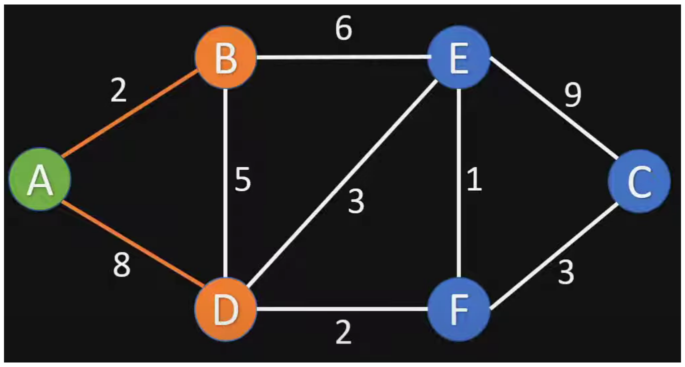


Figure 10: A point

Node	Shortest Distance	Previous Node
A	0	-
B	2	A
C	∞	-
D	8	A
E	∞	-
F	∞	-

And the visited node set $S = \{A\}$.

4.2.2 B

The steps include:

1. Because B has lower weight than D from A, so now we explore the edges (unexplored ones) of B.
2. We can reach E, record it. We can also reach D, and the new path is $2+5 = 7 < 8$, so we update the D row.

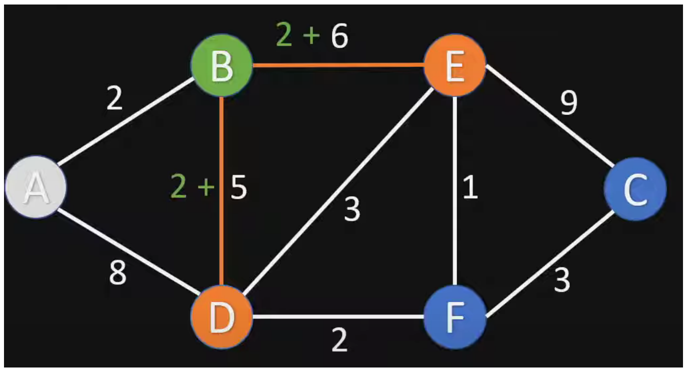


Figure 11: B point

Node	Shortest Distance	Previous Node
A	0	-
B	2	A
C	∞	-
D	7	B
E	8	B
F	∞	-

And the visited node set $S = \{A, B\}$.

4.2.3 D

The steps include:

1. Now because D has lower total path length than E, then now we explore the edges of D.
2. We reach E, but the new path length is $7 + 3 > 8$, so we will not record it.
3. We can also reach F, record it.

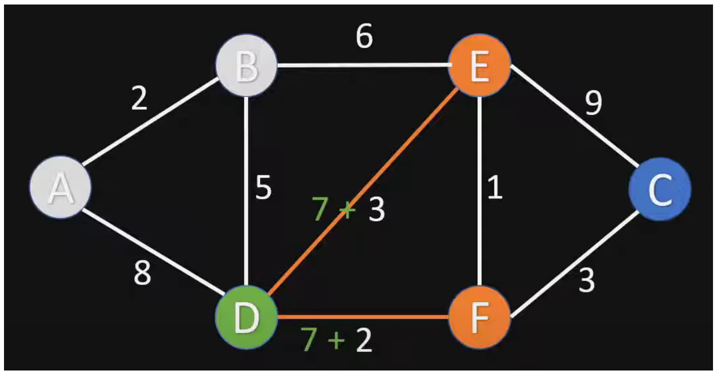


Figure 12: D point

Node	Shortest Distance	Previous Node
A	0	-
B	2	A
C	∞	-
D	7	B
E	8	B
F	9	D

And the visited node set $S = \{A, B, D\}$.

4.2.4 E

The steps include:

1. Now E has lower path length than F, so we choose E.
2. We reach F again, but the path length is the same, so we will not record it.
3. We reach C from E, record it.

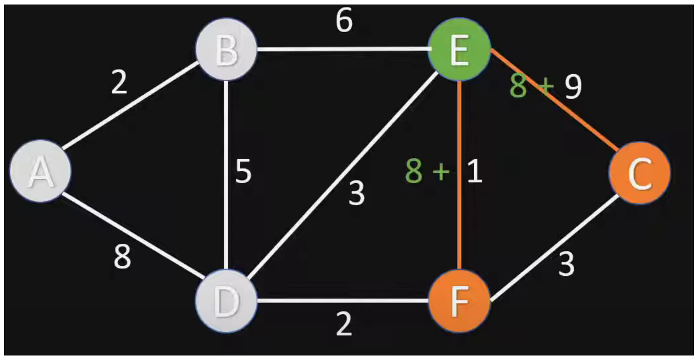


Figure 13: E point

Node	Shortest Distance	Previous Node
A	0	-
B	2	A
C	17	E
D	7	B
E	8	B
F	9	D

And the visited node set $S = \{A, B, D, E\}$.

4.2.5 F

The steps include:

1. Now F has lower path length, start with it.
2. Reach C again from F, and the path length is $9 + 3 = 12 < 17$, so we update it.

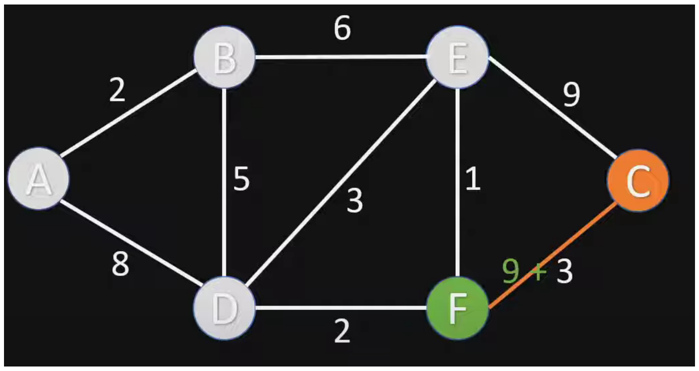


Figure 14: F point

Node	Shortest Distance	Previous Node
A	0	-
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

And the visited node set $S = \{A, B, D, E, F\}$.

4.2.6 C

The steps include:

1. Now we only have C unvisited, but actually we already go through all the edges connected to C, so the table will not updated.

And the visited node set $S = \{A, B, C, D, E, F\}$

4.3 Proof

Now we want to prove the correctness of Dijkstra's algorithm, which has the structure of **greedy stays ahead**. We choose **inductive proof** method.

1. **Invariant**: the invariant is a property or statement that remains true across each step of the induction process. In this case, the invariant is that **for each node $u \in S$, $d[u]$ is the length of a shortest path from s to u**
2. **Proof**: Greedy algorithm is optimal.
3. **Base Case**: When $S = \{s\}$, $d[s] = 0$, true proof.
4. **Inductive Step**:

(a) **Step 1:** Select the next vertex u with the minimum $d[u]$ among all vertices not yet finalized. By our induction hypothesis and the way algorithm works, u is guaranteed to have the shortest distance from s out of all vertices not yet finalized.

(b) **Step 1 Proof:** Suppose there exists a vertex $u' \notin S$ with a shorter path from s that has not been considered yet. This path must pass through at least one vertex not in S, say x, because all paths through vertices in S have already been considered (as those distances are finalized and assumed to be correct). However, this implies that $d[x]$ would be less than $d[u]$, which contradicts the selection of u as the vertex with the minimum value among all non-finalized vertices

(c) **Step 2**: Update the distances to all neighbors v of u by checking if $d[u] + l_e(u,v) < d[v]$. If so, update $d[v]$ to $d[u] + l_e(u,v)$. This step ensures that we are considering the shortest possible distance to v through u, based on the greedy algorithm.

(d) **Conclusion:** By induction, if the algorithm correctly identifies the shortest paths to a set of finalized vertices S at each step, it will also correctly identify the shortest path to the next vertex u to be finalized. This process repeats until all vertices are finalized, ensuring that the algorithm correctly computes the shortest paths from s to all other vertices in S.

5 Dijkstra's Algorithm Optimization

5.1 Introduction to Priority Queue (PQ)

A priority queue is an abstract data type in computer science that operates similarly to a regular queue or stack, but with an important difference: **each element in the priority queue has a priority associated with it**. In a priority queue, an element with higher priority is served before an element with lower priority. If two elements have the same priority, they are served according to their order in the queue. The choice of implementation affects the efficiency of various operations (such as insertion, deletion, and finding the minimum or maximum element), making some implementations more suitable for certain situations than others.

The primary operations of a priority queue include:

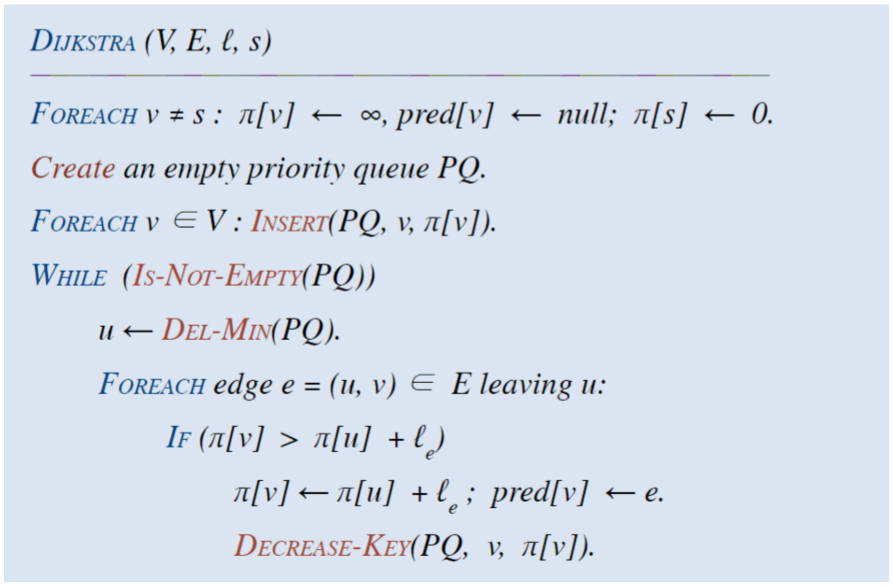
1. **Insertion (Enqueue)**: Adding a new element to the queue with its associated priority.

2. **Deletion (Dequeue)**: Removing the element with the highest priority from the queue and returning it.

3. **Peek**:Returning the element with the highest priority without removing it from the queue.

5.2 Priority Queue Application

We can use **mini-oriented PQ** to choose an unexplored node that minimizes $\pi[v]$. The algorithm is shown below:



```
DIJKSTRA (V, E, ℓ, s)
────────────────────────────
FOREACH v ≠ s : π[v] ← ∞, pred[v] ← null; π[s] ← 0.
Create an empty priority queue PQ.
FOREACH v ∈ V : INSERT(PQ, v, π[v]).
WHILE (IS-NOT-EMPTY(PQ))
    u ← DEL-MIN(PQ).
    FOREACH edge e = (u, v) ∈ E leaving u:
        IF (π[v] > π[u] + ℓ_e)
            π[v] ← π[u] + ℓ_e ; pred[v] ← e.
            DECREASE-KEY(PQ, v, π[v]).
```

Figure 15: PQ Algorithm

The algorithm includes the following part:

5.2.1 Initialization

1. Initialize the distance to all vertices, denoted as $\pi[v]$, to infinity, except for the starting vertex s, which is set to 0.

2. The **predecessor** of all vertices $pred[v]$ is set to null.

5.2.2 Priority Queue

1. A priority queue PQ is created, which will store **all vertices of the graph along with their current shortest distance from source** s.

2. Every vertex v is inserted into PQ with its associated initialized distance $\pi[v]$.

5.2.3 Main Loop

1. The algorithm then enters a loop that continues as long as the priority queue is not empty.

2. Inside the loop, the algorithm **extracts the vertex** u **with the minimum distance value from the priority queue using** $Del-Min(PQ)$. This vertex is the one that is closest to the starting vertex s among all vertices currently in the PQ.

3. Next step is to consider all edges e leaving u. For each edge (u, v), the algorithm checks if the distance to v via u is shorter than the current known distance $\pi[v]$. This is called **relaxation** step.

4. If a shorter path is found, the algorithm updates the distance to v ($\pi[v]$) to reflect this shorter path.

5. It also updates the predecessor of v to u.

6. The $Decrease-key(PQ, v, \pi[v])$ operation updates the distance value of v in PQ, as its priority (which is the same as the shortest distance to s) has now decreased.

5.3 Demo

Suppose we have the following directed weighted graph, we want to find shortest path from s to t using PQ.

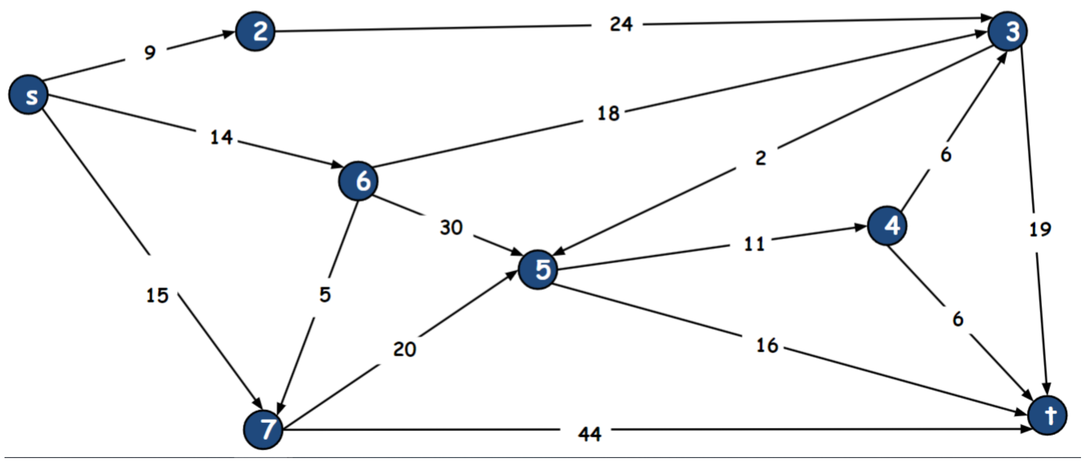


Figure 16: PQ Demo

5.3.1 Initialization

First we have the explored node set as S, PQ, and priority recorder (inside PQ) and the predecessor recorder:

$$S = \{\} \tag{3}$$

$$PQ = \{s, 2, 3, 4, 5, 6, 7, t\} \tag{4}$$

$$priority = \{0, \infty, \infty, \infty, \infty, \infty, \infty, \infty\} \tag{5}$$

Node	Shortest Distance	Previous Node
s	0	-
2	∞	-
3	∞	-
4	∞	-
5	∞	-
6	∞	-
7	∞	-
t	∞	-

5.3.2 Vertex s

Now, s is the vertex with the minimum distance value, so we extract it. It can reach 2, 6, 7 nodes, and their distances will be shorter than ∞ for sure. So we update the priority.

$$S = \{s\} \tag{6}$$

$$PQ = \{2, 3, 4, 5, 6, 7, t\} \tag{7}$$

$$priority = \{9, \infty, \infty, \infty, 14, 15, \infty\} \tag{8}$$

Node	Shortest Distance	Previous Node
s	0	-
2	9	s
3	∞	-
4	∞	-
5	∞	-
6	14	s
7	15	s
t	∞	-

5.3.3 Vertex 2

Now vertex 2 is the minimum priority, extract it. It can reach 3, update the priority of it $(9 + 24 = 33)$.

$$S = \{s, 2\} \tag{9}$$

$$PQ = \{3, 4, 5, 6, 7, t\} \tag{10}$$

$$priority = \{33, \infty, \infty, 14, 15, \infty\} \tag{11}$$

Node	Shortest Distance	Previous Node
s	0	-
2	9	s
3	33	2
4	∞	-
5	∞	-
6	14	s
7	15	s
t	∞	-

5.3.4 Vertex 6

Now vertex 6 is the minimum priority, extract it. It can reach:

1. Vertex 3: $14 + 18 = 32$
2. Vertex 5: $14 + 30 = 44$
3. Vertex 7: $14 + 5 = 19$

Update the priority if we get a lower value:

$$S = \{s, 2, 6\} \tag{12}$$

$$PQ = \{3, 4, 5, 7, t\} \tag{13}$$

$$priority = \{32, \infty, 44, 15, \infty\} \tag{14}$$

Node	Shortest Distance	Previous Node
s	0	-
2	9	s
3	32	6
4	∞	-
5	44	6
6	14	s
7	15	s
t	∞	-

5.3.5 Vertex 7

Now vertex 7 is the minimum priority, extract it. It can reach:

1. Vertex 5: $15 + 20 = 35$
2. Vertex t: $15 + 44 = 59$

Update the priority if we get a lower value:

$$S = \{s, 2, 6, 7\} \tag{15}$$

$$PQ = \{3, 4, 5, t\} \tag{16}$$

$$priority = \{32, \infty, 35, 59\} \tag{17}$$

Node	Shortest Distance	Previous Node
s	0	-
2	9	s
3	32	6
4	∞	-
5	35	7
6	14	s
7	15	s
t	59	7

5.3.6 Vertex 3

Now vertex 3 is the minimum priority, extract it. It can reach:

1. Vertex 5: $32 + 2 = 34$
2. Vertex t: $32 + 19 = 51$

Update the priority if we get a lower value:

$$S = \{s, 2, 6, 7, 3\} \tag{18}$$

$$PQ = \{4, 5, t\} \tag{19}$$

$$priority = \{\infty, 34, 51\} \tag{20}$$

Node	Shortest Distance	Previous Node
s	0	-
2	9	s
3	32	6
4	∞	-
5	34	3
6	14	s
7	15	s
t	51	3

5.3.7 Vertex 5

Now vertex 5 is the minimum priority, extract it. It can reach:

1. Vertex 4: $34 + 11 = 45$
2. Vertex t: $34 + 16 = 50$

Update the priority if we get a lower value:

$$S = \{s, 2, 6, 7, 3, 5\} \tag{21}$$

$$PQ = \{4, t\} \tag{22}$$

$$priority = \{45, 50\} \tag{23}$$

Node	Shortest Distance	Previous Node
s	0	-
2	9	s
3	32	6
4	45	5
5	34	3
6	14	s
7	15	s
t	50	5

5.3.8 Vertex 4

Now vertex 4 is the minimum priority, extract it. It can reach:

1. Vertex 3: $45 + 6 = 51$
2. Vertex t: $45 + 6 = 51$

Update the priority if we get a lower value:

$$S = \{s, 2, 6, 7, 3, 5, 4\} \tag{24}$$

$$PQ = \{t\} \tag{25}$$

$$priority = \{50\} \tag{26}$$

Node	Shortest Distance	Previous Node
s	0	-
2	9	s
3	32	6
4	45	5
5	34	3
6	14	s
7	15	s
t	50	5

5.3.9 Vertex t

Now vertex t is the only one element left in the queue, extract it. Because it could not reach any other nodes, so anything will not be updated.

5.3.10 Conclusion

Finally we have the shortest path as:

$$s \to 6 \to 3 \to 5 \to t \tag{27}$$

And the shortest path length is 50.