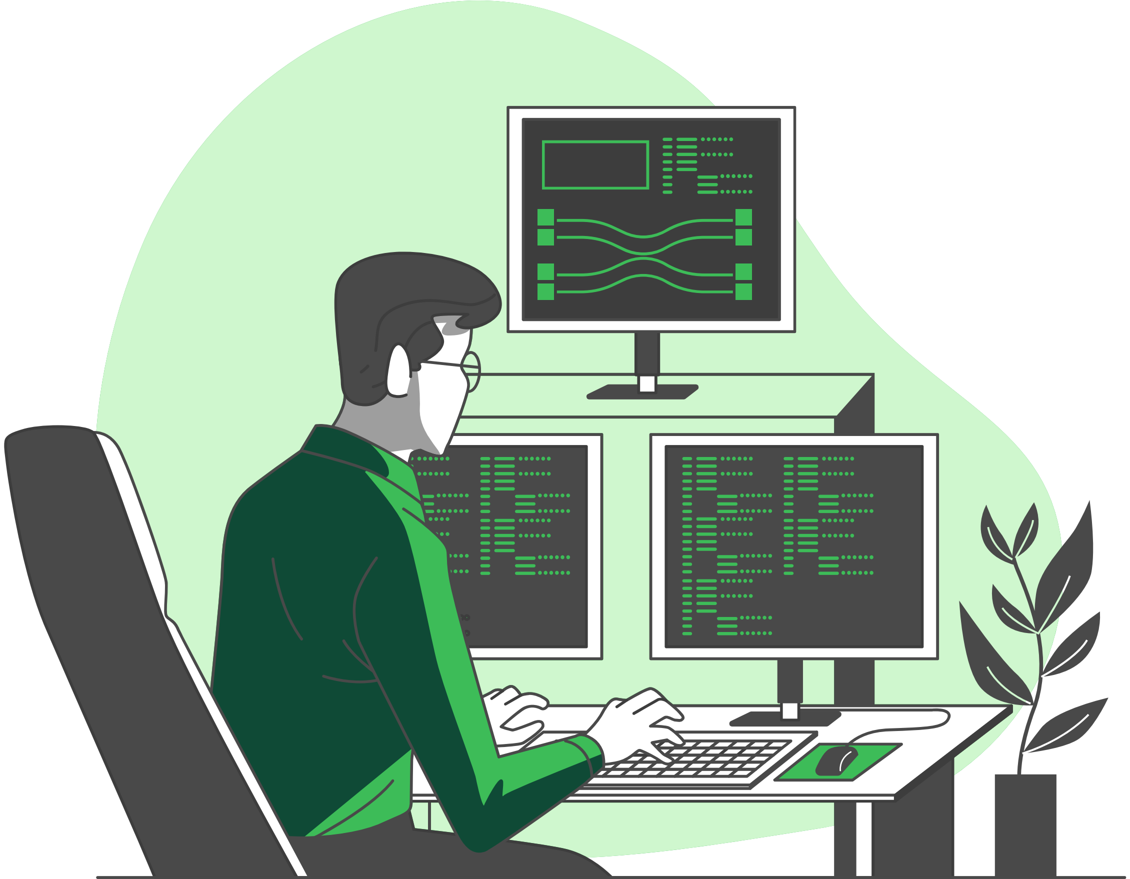
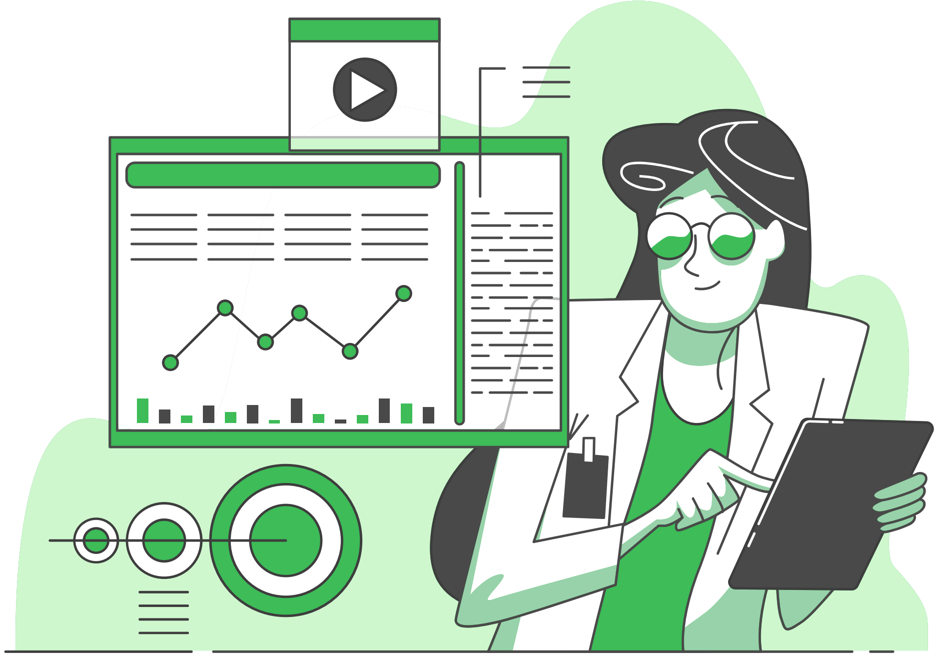


Weighted Graph

CS 251 - Data Structures and Algorithms

Note:
Slides complement the discussion in class

Table of Contents



01

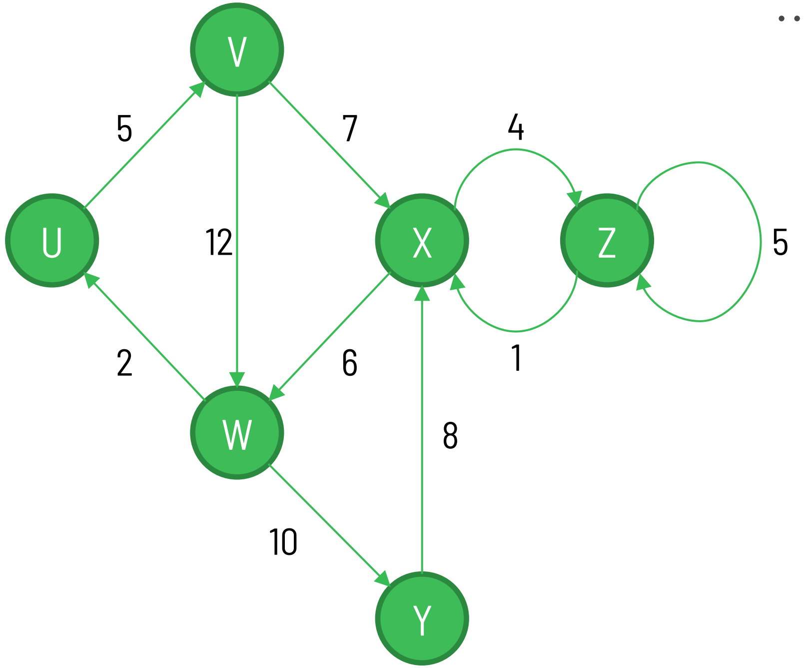
Weighted Graph

When edges have info

Weighted Graph

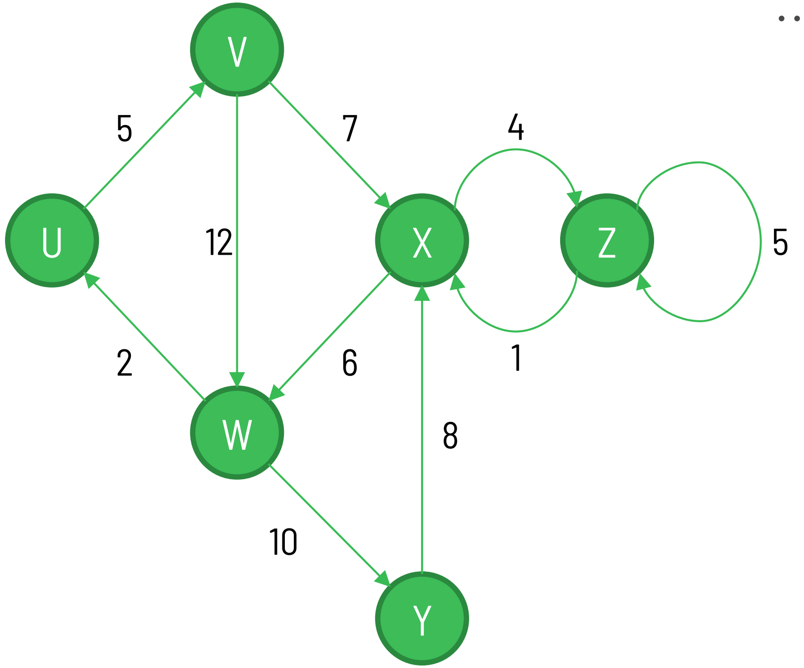
A **weighted graph** is a graph with values associated to its edges (aka. **weights**)

They are useful to represent distances, costs, penalties, loads, capacities, times...



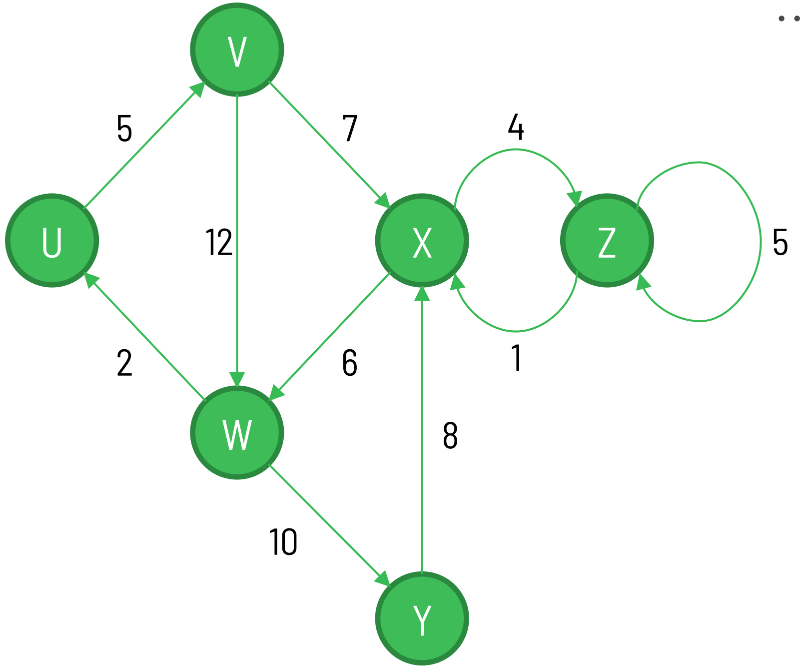
Weighted Graph Representations

	U	V	W	X	Y	Z
U	-	5	-	-	-	-
V	-	-	12	7	-	-
W	2	-	-	-	10	-
X	-	-	6	-	-	4
Y	-	-	-	8	-	-
Z	-	-	-	1	-	5



Weighted Graph Representations

U	(V, 5)
V	(W, 12), (X, 7)
W	(U, 2), (Y, 10)
X	(W, 6), (Z, 4)
Y	(X, 8)
Z	(X, 1), (Z, 5)



02

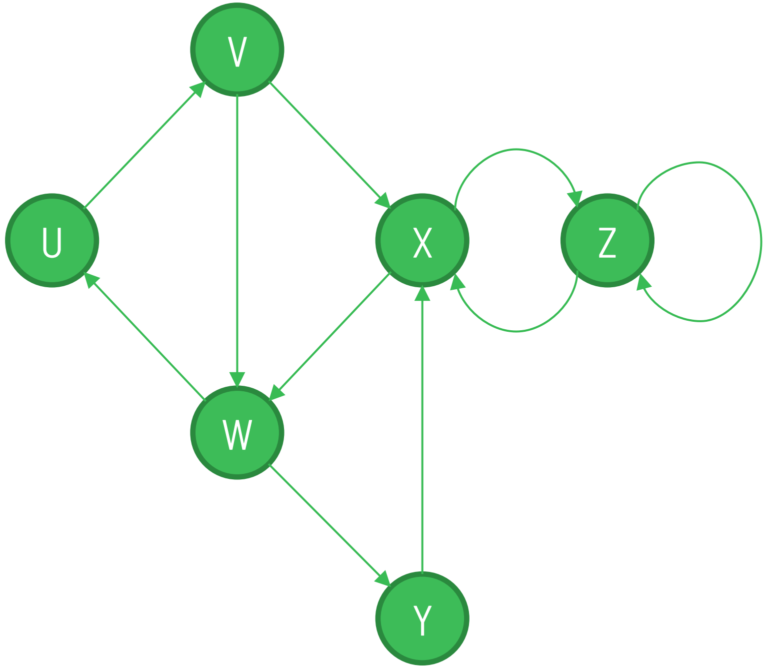
Shortest Path Problem

Finding the shortest path between two vertices

Finding A Shortest Path

Didn't we say that BFS finds the shortest path between a pair of vertices in a graph?

Shortest path between vertices W and X? Call BFS(G, W) and get the path that visits X.

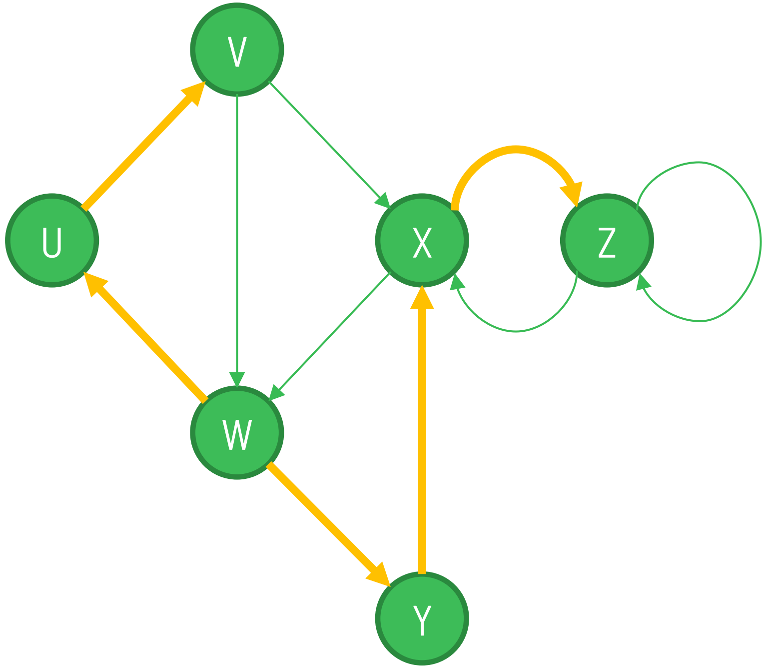


Finding A Shortest Path

Didn't we say that BFS finds the shortest path between a pair of vertices in a graph?

Shortest path between vertices W and X? Call BFS(G, W) and get the path that visits X.

There! Shortest path is {W, Y, X} with length 2.



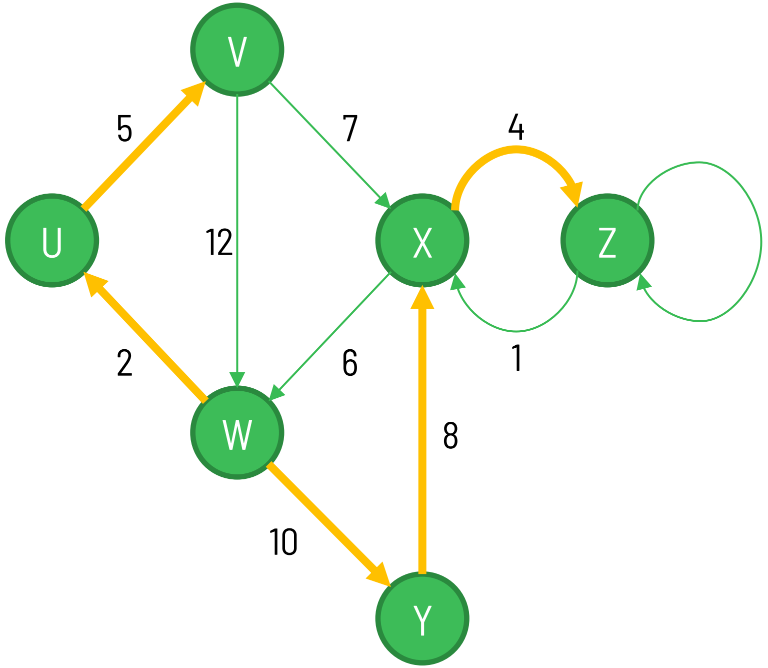
Finding A Shortest Path

Didn't we say that BFS finds the shortest path between a pair of vertices in a graph?

Shortest path between vertices W and X? Call BFS(G, W) and get the path that visits X.

There! Shortest path is {W, Y, X} with length 2.

Is it?

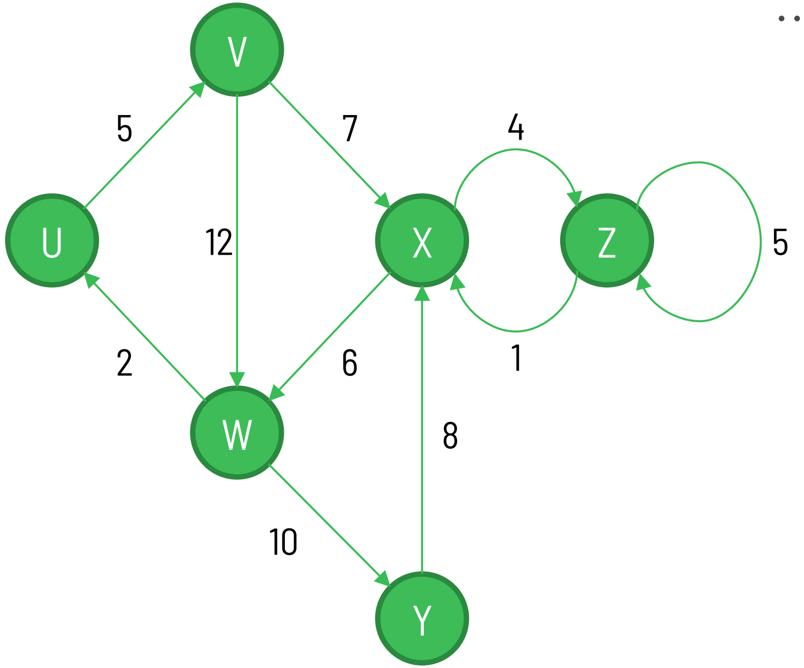


Shortest Path Problem

Given a weighted digraph $G = \{V, E\}$ and two vertices $u, v \in V$, we need to find a path of minimum total weight between u and v.

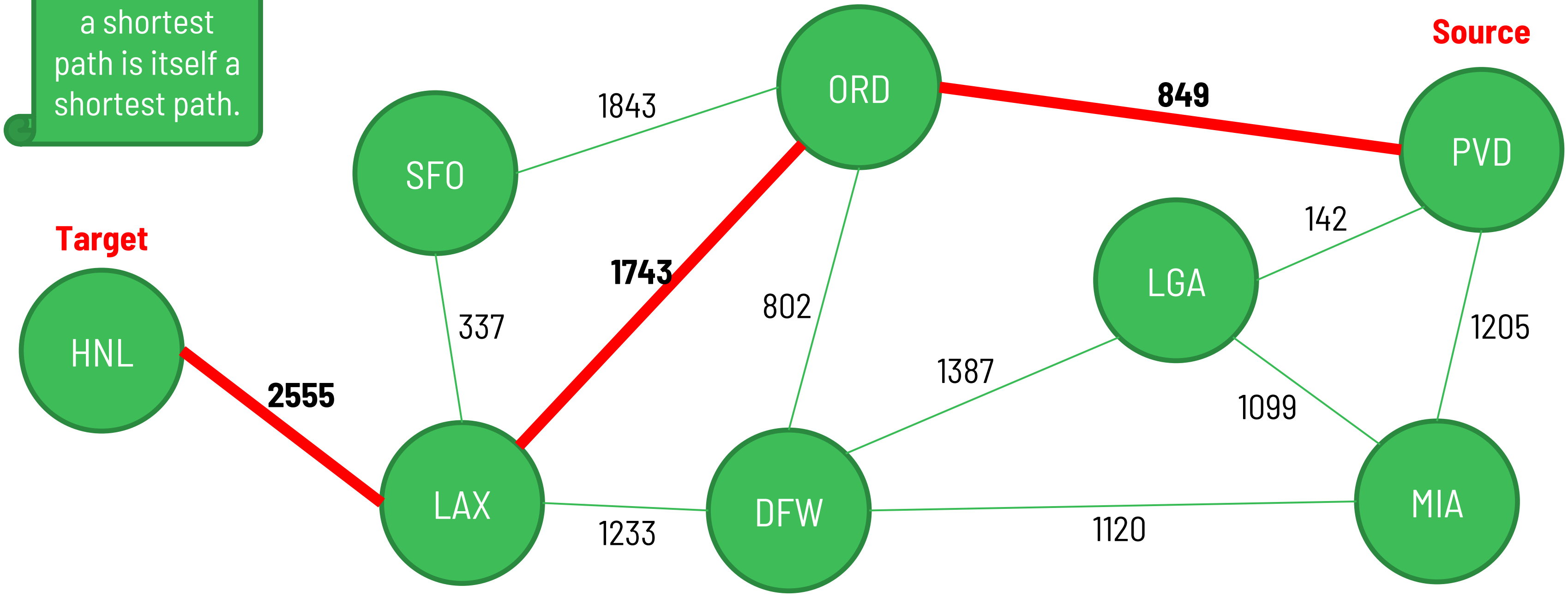
Shortest Path Properties:

1. A sub path of a shortest path is itself a shortest path.
2. There is a tree of shortest paths from a vertex to every other vertex in the graph.



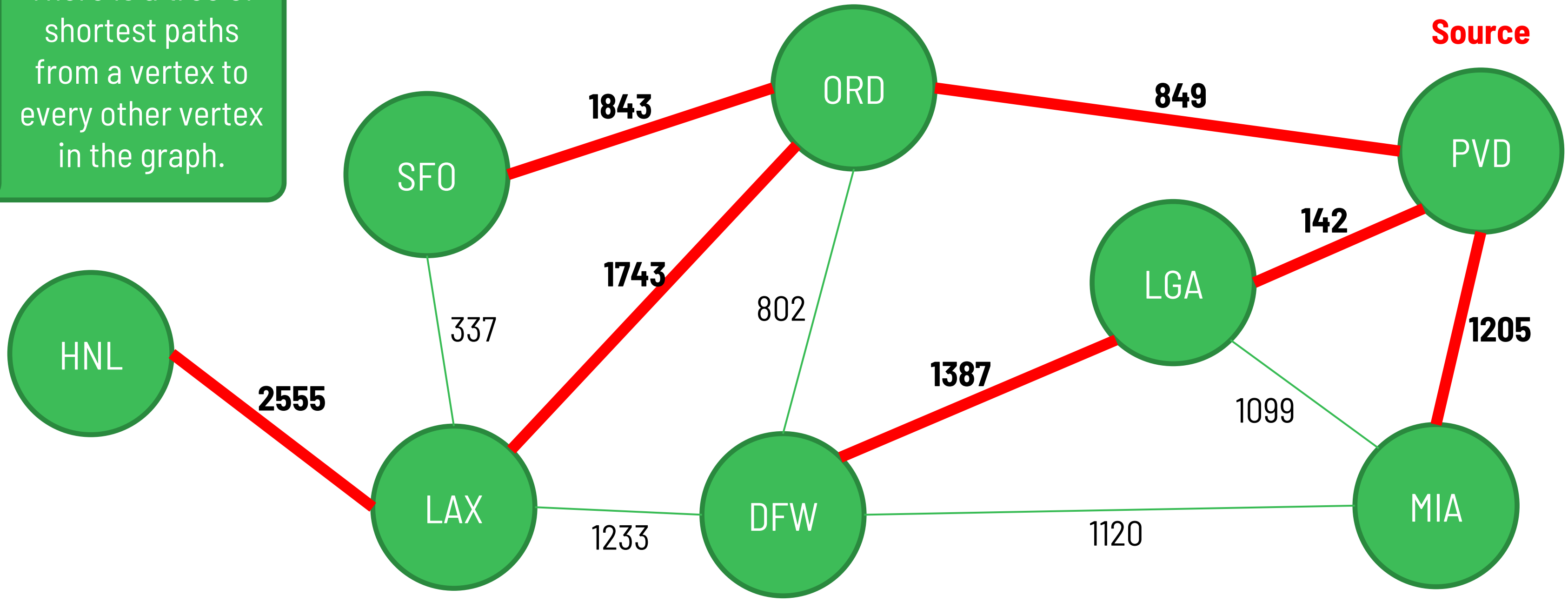
Property 1 Example

A sub path of a shortest path is itself a shortest path.



Property 2 Example

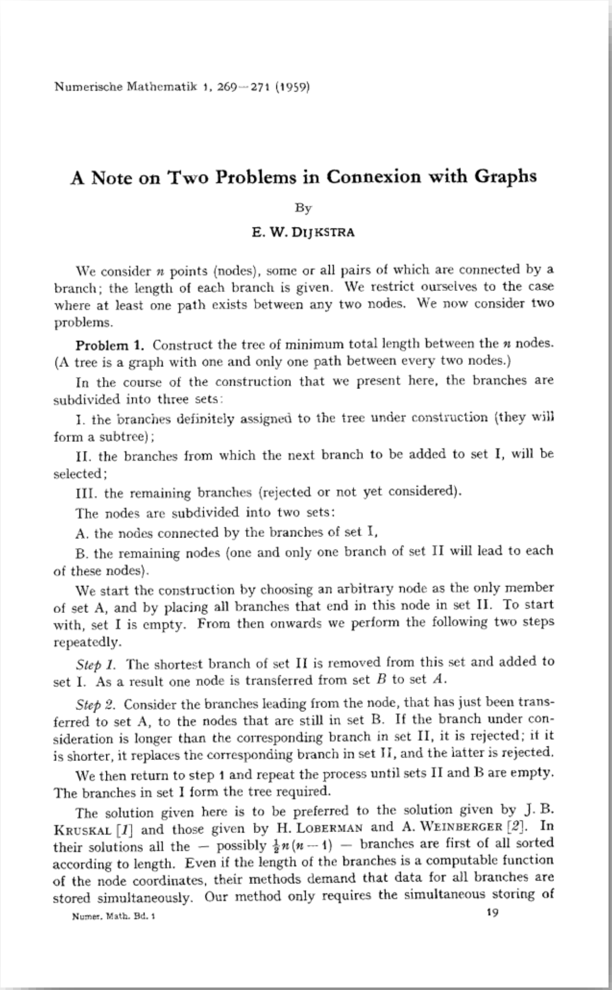
There is a tree of shortest paths from a vertex to every other vertex in the graph.



03
Dijkstra’s Algorithm

Let’s find them paths

Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." Numerische Mathematik, vol. 1, pp. 269–271, 1959.



Numerische Mathematik 1, 269—271 (1959)

A Note on Two Problems in Connexion with Graphs

By

E. W. Dijkstra

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Problem 1. Construct the tree of minimum total length between the n nodes. (A tree is a graph with one and only one path between every two nodes.)

In the course of the construction that we present here, the branches are subdivided into three sets:

I. the branches definitely assigned to the tree under construction (they will form a subtree);

II. the branches from which the next branch to be added to set I, will be selected;

III. the remaining branches (rejected or not yet considered).

The nodes are subdivided into two sets:

A. the nodes connected by the branches of set I,

B. the remaining nodes (one and only one branch of set II will lead to each of these nodes).

We start the construction by choosing an arbitrary node as the only member of set A, and by placing all branches that end in this node in set II. To start with, set I is empty. From then onwards we perform the following two steps repeatedly.

Step 1. The shortest branch of set II is removed from this set and added to set I. As a result one node is transferred from set B to set A.

Step 2. Consider the branches leading from the node, that has just been transferred to set A, to the nodes that are still in set B. If the branch under consideration is longer than the corresponding branch in set II, it is rejected; if it is shorter, it replaces the corresponding branch in set II, and the latter is rejected.

We then return to step 1 and repeat the process until sets II and B are empty. The branches in set I form the tree required.

The solution given here is to be preferred to the solution given by J. B. Kruskal [1] and those given by H. Loberman and A. Weinberger [2]. In their solutions all the — possibly $\frac{1}{2}n(n-1)$ — branches are first of all sorted according to length. Even if the length of the branches is a computable function of the node coordinates, their methods demand that data for all branches are stored simultaneously. Our method only requires the simultaneous storing of

https://link.springer.com/article/10.1007/BF01386390

Dijkstra's Algorithm

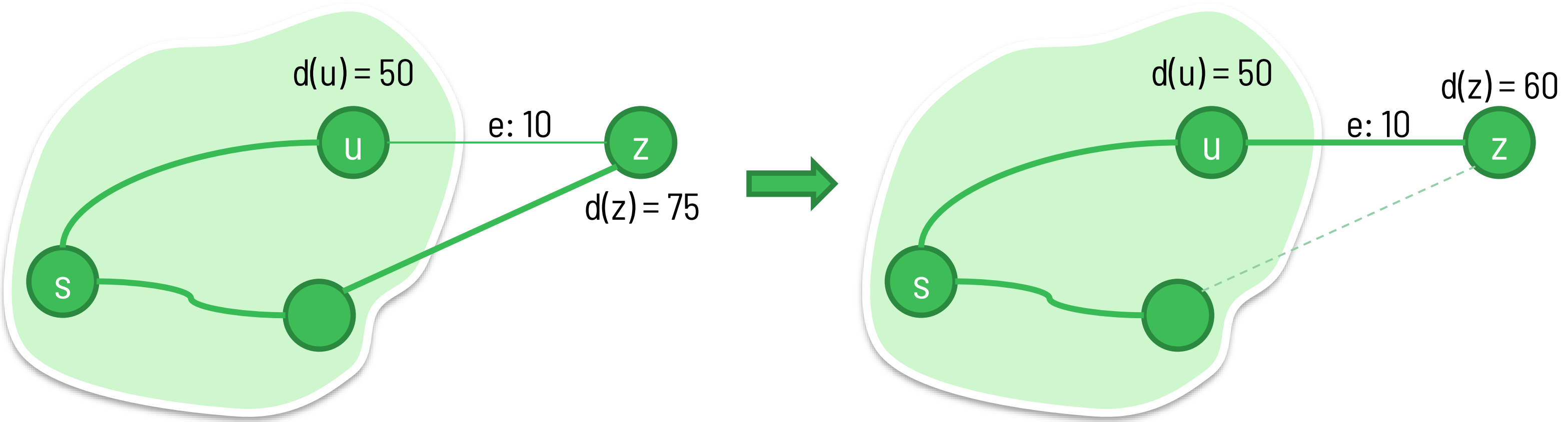
1. Given a digraph $G = \{V, E\}$ and a source vertex $v \in V$, Dijkstra's algorithm finds the shortest path from v to every other vertex in the digraph.

2. Dijkstra's algorithm is a **Greedy Algorithm**.

3. Assumptions:
   - The digraph is connected.
   - Edge weights are nonnegative (**IMPORTANT**)

4. Analogy: Dijkstra's algorithm grows a "**cloud**" of vertices, starting with v, until the cloud covers all vertices in G. In every step of the algorithm, we insert a new vertex into the cloud and keep the current shortest distances from v to the current vertices in the cloud.

Edge Relaxation



d(z) = min{d(z), d(u) + weight(e)}

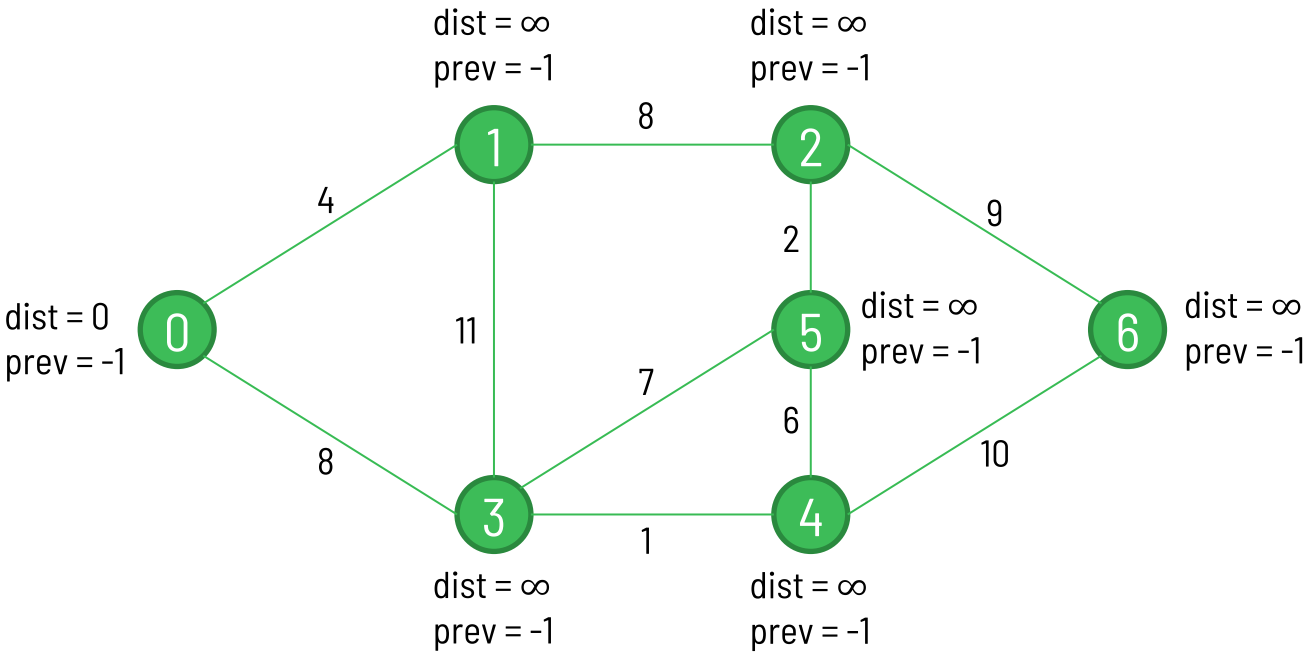
Dijkstra's Shortest Path Algorithm

```
algorithm DijkstraShortestPath(G(V,E), s ∈ V)

   let dist:V → ℤ
   let prev:V → V
   let Q be an empty priority queue

   dist[s] ← 0
   for each v ∈ V do
      if v ≠ s then
         dist[v] ← ∞
      end if
      prev[v] ← -1
      Q.add(dist[v], v)
   end for

   while Q is not empty do
      u ← Q.getMin()
      for each w ∈ V adjacent to u still in Q do
         d ← dist[u] + weight(u, w)
         if d < dist[w] then
            dist[w] ← d
            prev[w] ← u
            Q.set(d, w)
         end if
      end for
   end while

   return dist, prev
end algorithm
```

Remember:

$\text{dist}(v) = \min(dist(v), \text{dist}(u) + \text{weight}(u, v))$

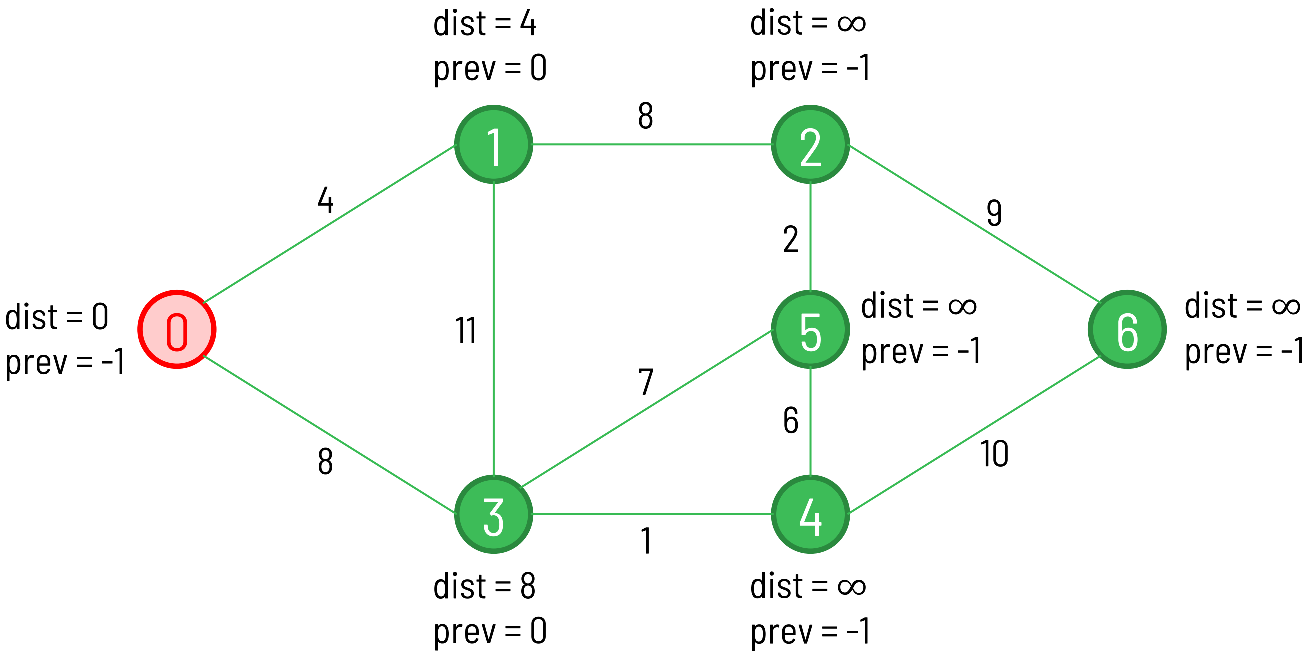
There is a min-heap Q with all the vertices of the graph



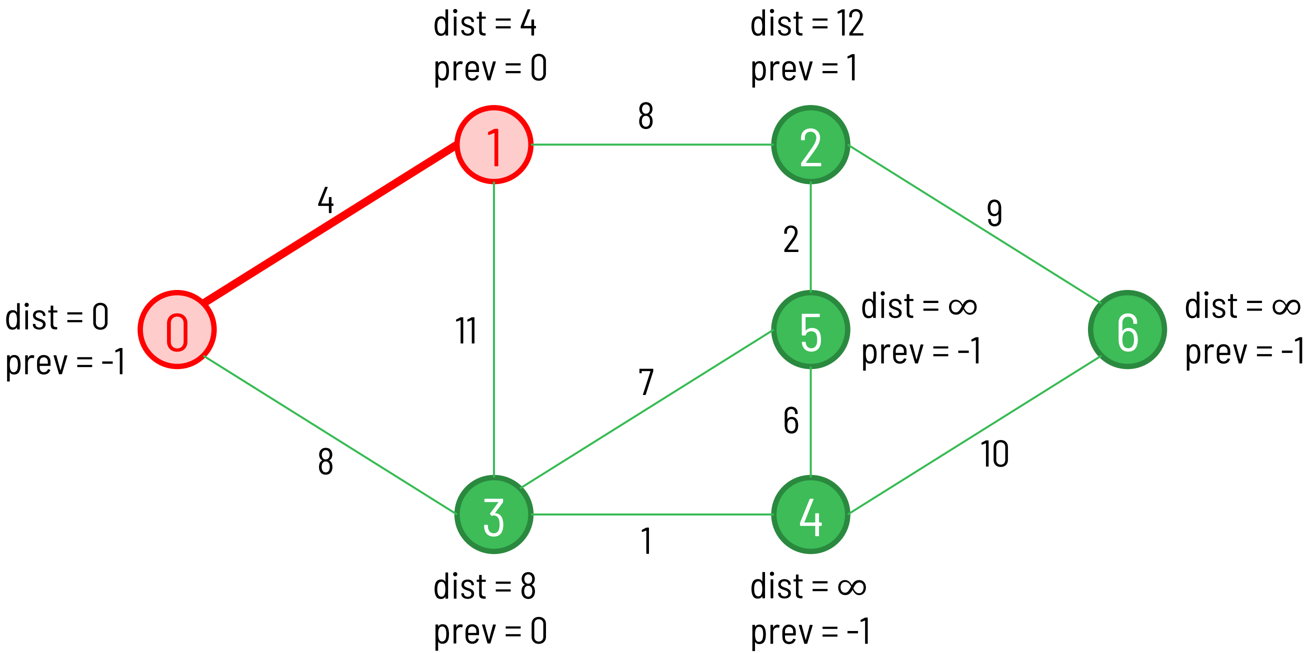
Remember:

$\text{dist}(v) = \min(dist(v), \text{dist}(u) + \text{weight}(u, v))$

There is a min-heap Q with all the vertices of the graph



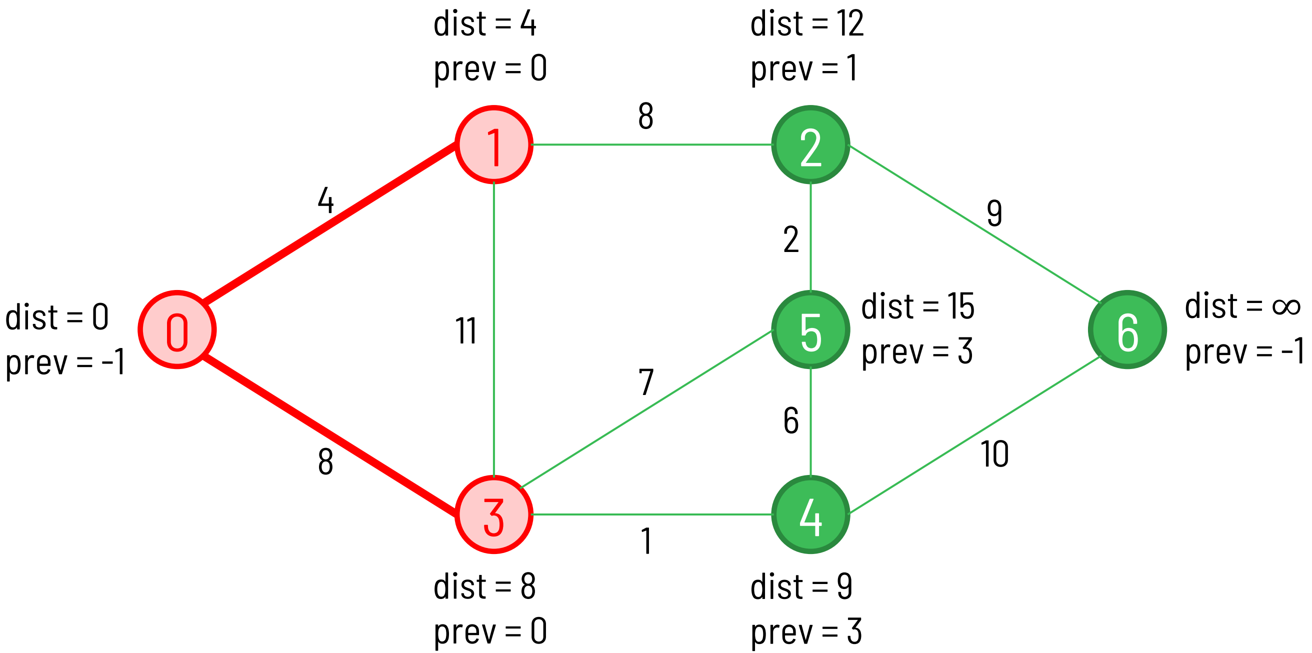
Remember:
$\text{dist}(v) = \min(dist(v), \text{dist}(u) + \text{weight}(u, v))$
There is a min-heap Q with all the vertices of the graph



Remember:

$\text{dist}(v) = \min(dist(v), \text{dist}(u) + \text{weight}(u, v))$

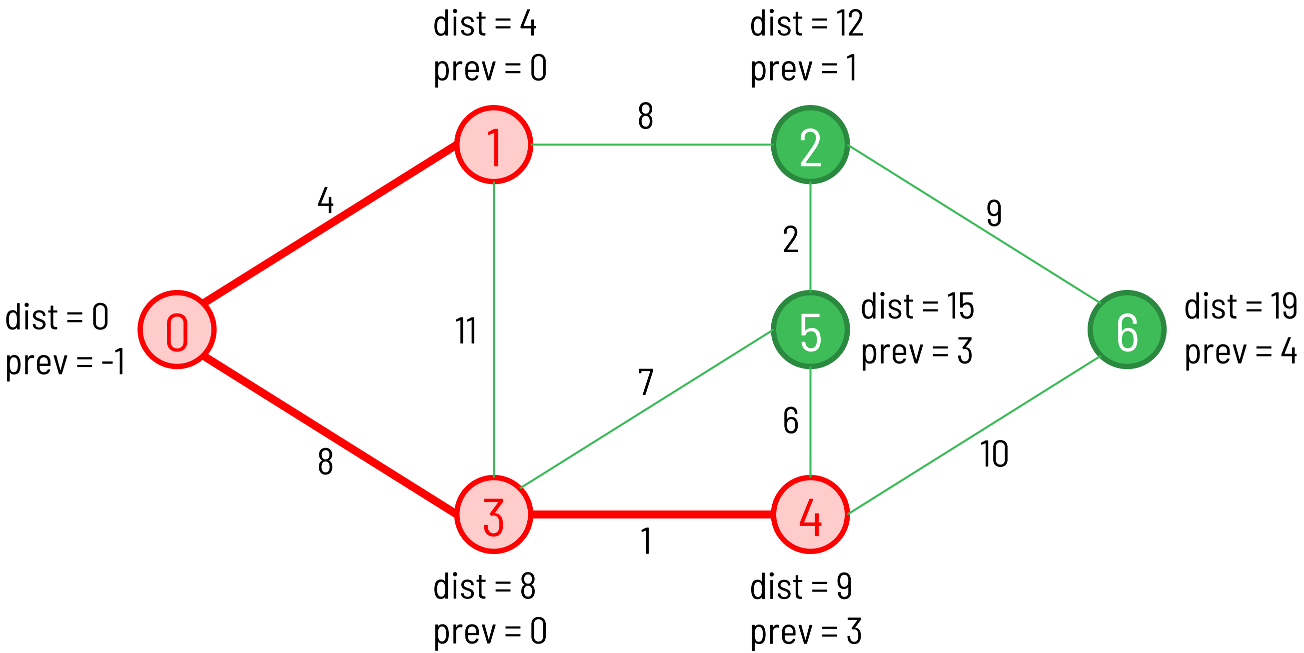
There is a min-heap Q with all the vertices of the graph



Remember:

$\text{dist}(v) = \min(dist(v), \text{dist}(u) + \text{weight}(u, v))$

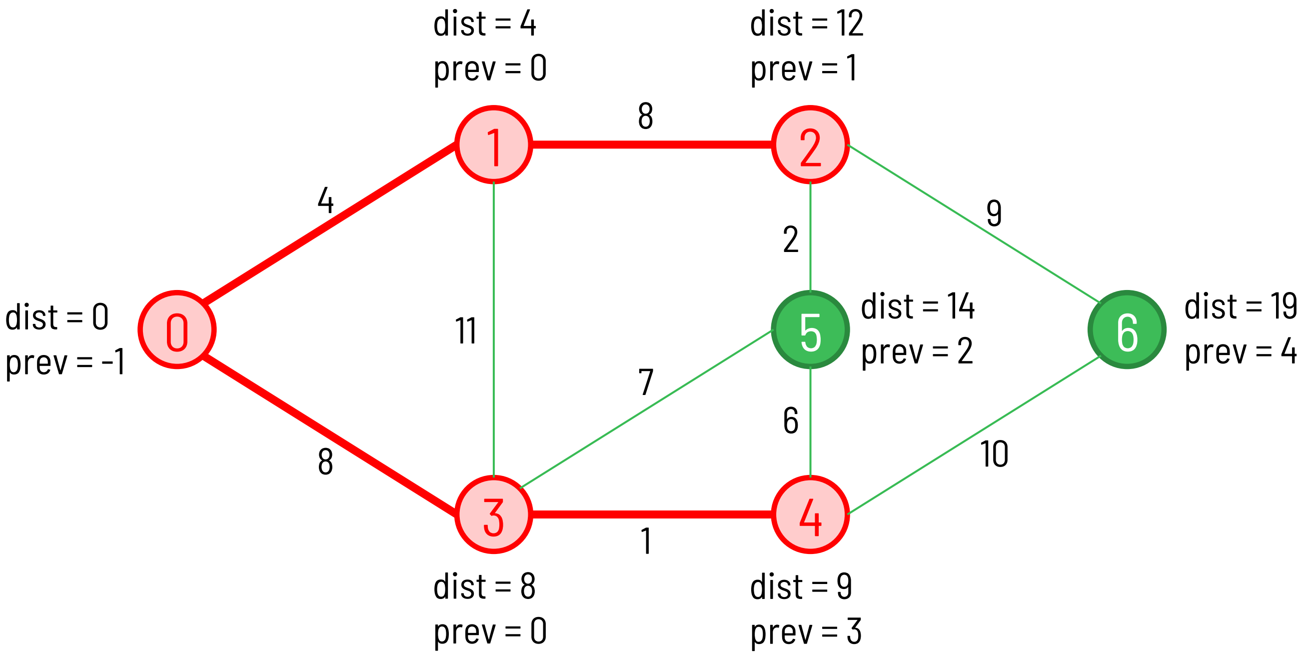
There is a min-heap Q with all the vertices of the graph



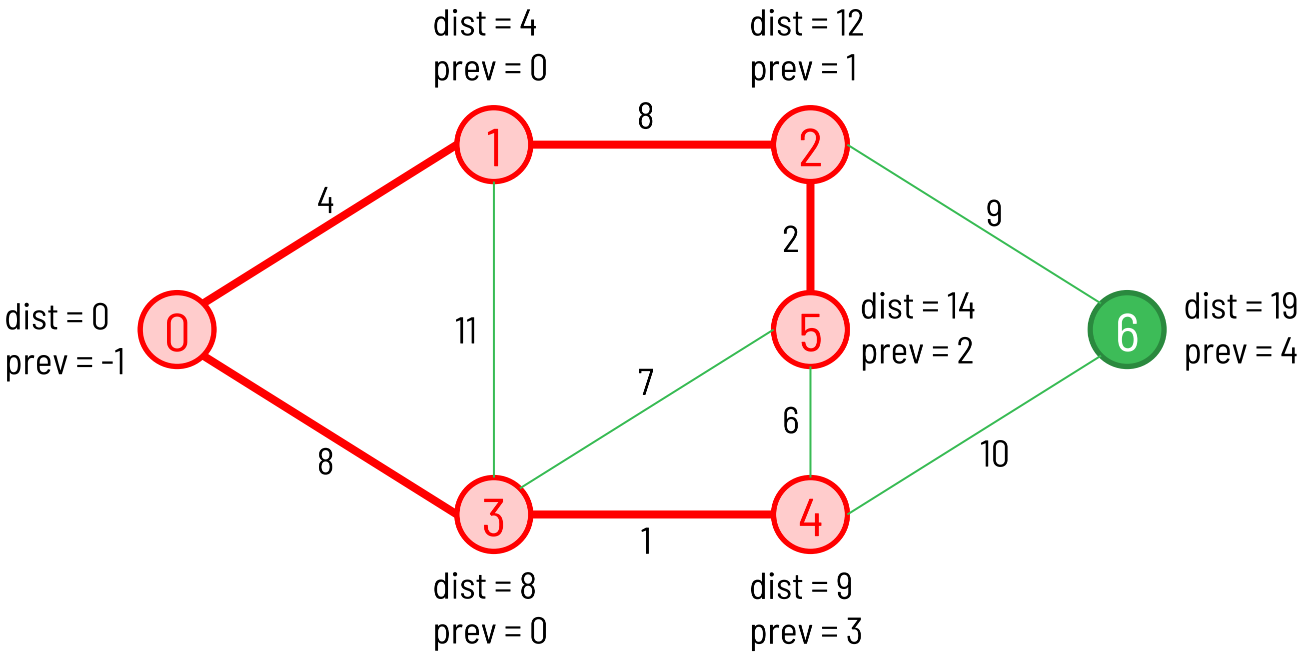
Remember:

$\text{dist}(v) = \min(dist(v), \text{dist}(u) + \text{weight}(u, v))$

There is a min-heap Q with all the vertices of the graph



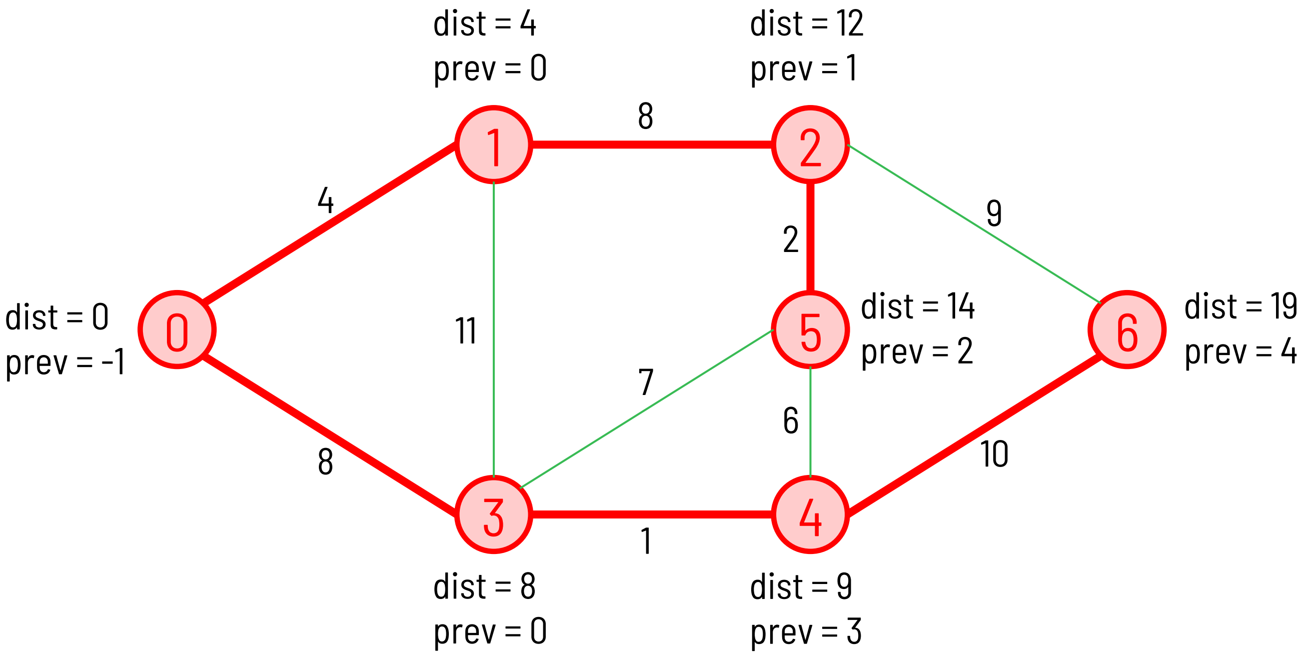
Remember:
$\text{dist}(v) = \min(dist(v), \text{dist}(u) + \text{weight}(u, v))$
There is a min-heap Q with all the vertices of the graph



Remember:

$\text{dist}(v) = \min(dist(v), \text{dist}(u) + \text{weight}(u, v))$

There is a min-heap Q with all the vertices of the graph



```
algorithm DijkstraShortestPath(G(V,E), s ∈ V)

   let dist:V → ℤ
   let prev:V → V
   let Q be an empty priority queue

   dist[s] ← 0
   for each v ∈ V do
      if v ≠ s then
         dist[v] ← ∞
      end if
      prev[v] ← -1
      Q.add(dist[v], v)
   end for

   while Q is not empty do
      u ← Q.getMin()
      for each w ∈ V adjacent to u still in Q do
         d ← dist[u] + weight(u, w)
         if d < dist[w] then
            dist[w] ← d
            prev[w] ← u
            Q.set(d, w)
         end if
      end for
   end while

   return dist, prev
end algorithm
```

Remember: G has **nonnegative** weight values.

Observations:

- We assume Q is a binary heap.
- Q.set(a, b) updates the location of a value b based on a new key a.

Runtime:

- Initializing arrays: $O(|V|)$
- Adding vertices to Q: $O(|V|\log(|V|))$
- Checking if a vertex is in Q: $O(1)$**!**
- Resetting values in the arrays (aka. Edge relaxation): $O(|E|)$
- Resetting values in Q: $O(|E|\log(|V|))$**!**

Dijkstra's Runtime: $O\big((|V| + |E|)\log(|V|)\big)$
We can also say $O(|E|\log(|V|))$ since every vertex is connected to at least one edge.

Runtime Considerations

Checking if a vertex is in Q: $O(1)$!
The algorithm must maintain a link between vertices and their positions in the queue (e.g., an array of size $|V|$ that get updated whenever a vertex changes position in Q). This allows checking is a vertex is in Q in $O(1)$.

Resetting values in Q: $O(|E|\log(|V|))$!
When a key gets updated, at most $\log(|V|)$ vertex positions will have to be updated as the heap is rearranged. So, updating a key can be done in $O(\log(|V|))$.

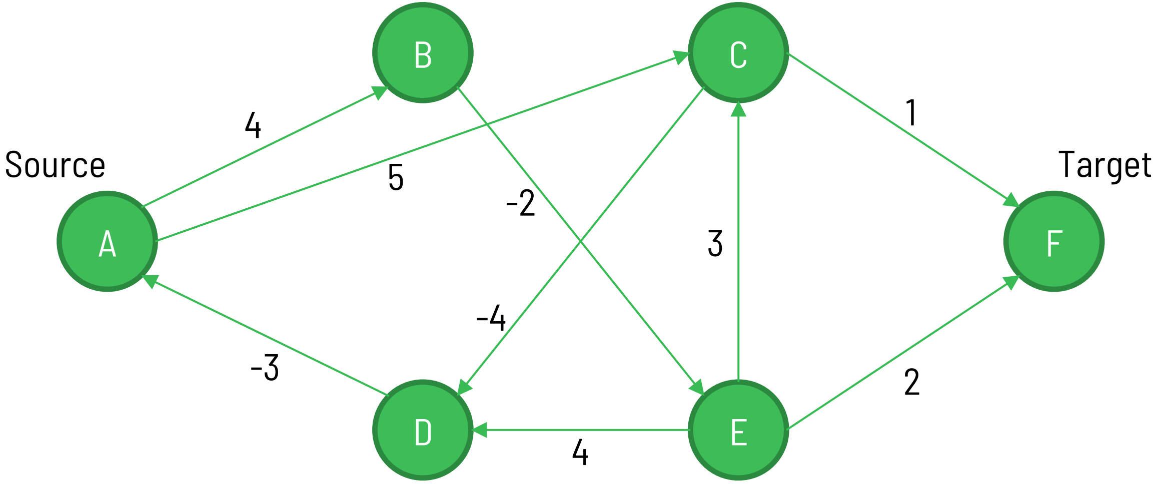
Internet says Dijkstra's runtime is $O(|E| + |V|\log(|V|))$
Yes, it is true if we keep track of vertices using a **Fibonacci Heap** (out of the scope of this course) instead of a Priority Queue implemented with a Binary Heap.

Don’t use Dijkstra’s algorithm with negative weights

Remember that Dijkstra’s algorithm is **Greedy**. Which means the algorithm chooses the best option in every iteration.

A **negative weight** will reduce distances to vertices outside of the cloud. Inserting a vertex incident to a “negative” edge **messes up** with the distances already in the cloud.

Try yo'self



Spoiler alert: Dijkstra should relax an already relaxed vertex.

We’re done

Do you have any questions?

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, infographics & images by Freepik and illustrations by Stories