So far, we have seen how to represent and traverse graphs through DFS and BFS.

Lecture 23: Shortest Paths

- Introduction to Shortest Paths
- API
- Properties
- Dijkstra's Algorithm

Some slides adopted from Algorithms 4th Edition or COS226

Today we will see an algorithm that will allow us to calculate the shortest path from one vertex to every other vertex in the graph.

Edge-weighted digraph

- Edge-weighted digraph: a digraph where we associate weights or costs with each edge.

edge-weighted digraph

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93

We will work with edge-weighted digraphs, that is digraphs where we associate weights or costs with each edge. For example, the graph on the right only shows vertices and edges that connect them, but we also maintain a list of the weight for each edge.

Shortest Paths

- Shortest path from vertex s to vertex t: a directed path from s to t with the property that no other such path has a lower weight (total weight sum of edges it consists).

edge-weighted digraph

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93

shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

An edge-weighted digraph and a shortest path

A shortest path from vertex s to vertex t is a directed path from s to t with the property that no other such path has a lower weight, i.e., the total weight sum of edges it consists. (it's ok to have another path with the same total weight). For example, a shortest path from vertex 0 to vertex 6 in the graph above could take us from 0 to 2 to 7 to 3 to 6 with a total cost of 1.51 and no other path will have a lower weight than that.

Shortest Path variants

- Single source: from one vertex `s` to every other vertex.
- Single sink: from every vertex to one vertex `t`.
- Source-sink: from one vertex `s` to another vertex `t`.
- All pairs: from every vertex to every other vertex.

- What version is there in your navigation app?

The problem of finding shortest paths has a lot of variants. For example:
Single source: from one vertex s to every other vertex.
Single sink: from every vertex to one vertex t.
Source-sink: from one vertex s to another vertex t.
All pairs: from every vertex to every other vertex.

What version do you think your navigation app follows? That's right, source-sink, for the most part.

Shortest Paths Assumptions

- Not all vertices need to be reachable.
  - We will assume so in this lecture.
- Weights are non-negative.
  - There are algorithms that can handle negative weights.
- Shortest paths are not necessarily unique but they are simple.

Let's state some assumptions for our shortest paths problem. Not all vertices need to be reachable (remember reachability refers to the ability to get from one vertex to another within a graph). Also we will assume that weights are non-negative (although there are algorithms that can handle negative weights). And finally, although shortest paths are not necessarily unique they have to be simple (a path in a graph which does not have repeating vertices).

Lecture 23: Shortest Paths

- Introduction to Shortest Paths
- API
- Properties
- Dijkstra's Algorithm

How would we go about modeling this problem?

Weighted directed edge API

- `public class DirectedEdge`
  - `DirectedEdge(int v, int w, double weight)`
    - Constructs a weighted edge from v to `w` `(v->w)` with the provided `weight`.
  - `int from()`
    - Returns vertex source of this edge.
  - `int to()`
    - Returns vertex destination of this edge.
  - `double weight()`
    - Returns weight of this edge.
  - `String toString()`
    - Returns the string representation of this edge.

We will need to introduce the concept of a directed edge, and we will do so with a DirectedEdge class. Its constructor will create a weighted edge from v to w (v->w) with the provided weight. We should have a weight of getting the source and destination of the edge as well as its weight. And it would be convenient to have a string representation of the edge.

Weighted directed edge in Java

```
public class DirectedEdge {
    private final int v;
    private final int w;
    private final double weight;

   public DirectedEdge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

   public int from() {
        return v;
    }

    public int to() {
        return w;
    }

    public double weight() {
        return weight;
    }
```

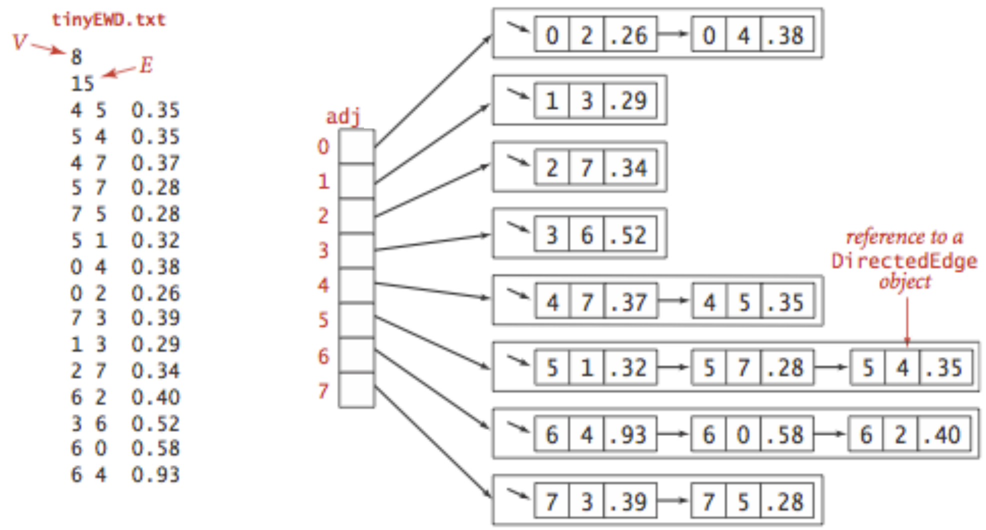
This is truly simple to code!

Edge-weighted digraph API

- `public class EdgeWeightedDigraph`
  - `EdgeWeightedDigraph(int v)`
    - Constructs an edge-weighted digraph with v vertices.
  - `void addEdge(DirectedEdge e)`
    - Add weighted directed edge e.
  - `Iterable<DirectedEdge> adj(int v)`
    - Returns edges adjacent from v.
  - `int V()`
    - Returns number of vertices.
  - `int E()`
    - Returns number of edges.
  - `Iterable<DirectedEdge> edges()`
    - Returns all edges.

Now we can use the concept of a weighted directed edge to specify the API for an edge-weight digraph. We would need to provide a way of constructing such a digraph with v vertices. A way to add weighted directed edges as well as get back edges adjacent to v or all edges.

Edge-weighted digraph adjacency list representation



Edge-weighted digraph representation

We will again follow the adjacency list representation for edge-weighted digraphs, where now we will hold references to a DirectedEdge object.

Edge-weighted digraph in Java

```
public class EdgeWeightedDigraph {
    private final int V;                // number of vertices in this digraph
    private int E;                      // number of edges in this digraph
    private ArrayList<ArrayList<DirectedEdge>> adj; // adj.get(v) = adjacency list for v

    public EdgeWeightedDigraph(int V) {
       this.V = V;
       this.E = 0;
       adj = new ArrayList<ArrayList<DirectedEdge>>(V);
        for (int v = 0; v < V; v++)
            adj.add(new ArrayList<DirectedEdge>());
    }
    public void addEdge(DirectedEdge e) {
        int v = e.from();
        int w = e.to();
        adj.get(v).add(e);
        E++;
    }

   public Iterable<DirectedEdge> adj(int v) {
       return adj.get(v);
    }
```

And that's how we would code such a class in Java. You will notice that the only difference from undirected and directed graphs is that we are working with DirectedEdges instead of Integers.

Single-source shortest path API

- Goal: find shortest path from s to every other vertex in the digraph.
- `public class SP`
  - `SP(EdgeWeightedDigraph G, int s)`
    - Shortest paths from s in digraph G.
  - `double distTo(int v)`
    - Length of shortest path from s to v.
  - `Iterable<DirectedEdge> pathTo(int v)`
    - Returns edges along the shortest path from s to v.
  - `boolean hasPathTo(int v)`
    - Returns whether there is a path from s to v.

That brings us to the API for the single-source shortest path problem which states that our goal is to find the shortest path from s to EVERY other vertex in the digraph. We can imagine a class SP (for shortest paths) whose constructor takes an edge weighted digraph and the index of the starting vertex and calculates the shortest paths from s to every other vertex in the digraph. Some convenient methods to provide would be given a vertex, what is the length of the shortest path from s to that vertex, as well as a method that returns edges along such a path and a method that confirms whether such a path exists at all.

Lecture 23: Shortest Paths

- Introduction to Shortest Paths
- API
- Properties
- Dijkstra's Algorithm

Now let's talk about some of the properties of the shortest paths problem.

Data structures for single-source shortest paths

- Goal: find shortest path from `s` to every other vertex in the digraph.
- Shortest-paths tree (SPT): a subgraph which will be a directed tree rooted at `s` which will contain all the vertices reachable from `s` and every tree path in the SPT is a shortest path in the digraph.
- Representation of shortest paths with two vertex-indexed arrays.
  - Edges on the shortest-paths tree: `edgeTo[v]` is the last edge on a shortest path from `s` to `v`.
  - Distance to the source: `distTo[v]` is the length of the shortest path from `s` to `v`.

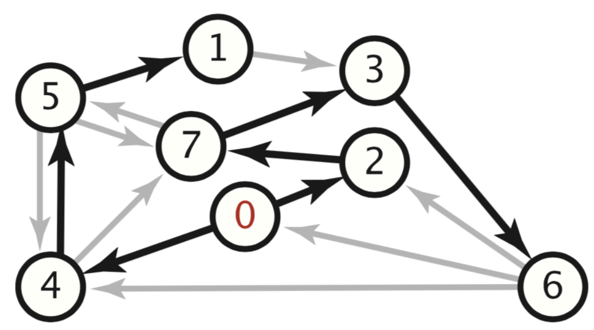
In our effort to find the shortest path from s to every other vertex in the digraph, we will calculate the shortest-paths tree (or SPT for short), which will be a directed tree rooted at s which will contain all the vertices reachable from s and every tree path in the SPT is a shortest path in the digraph.
We will keep two vertex-indexed arrays. edgeTo will keep the edges on the shortest-paths tree, with edgeTo[v] being the LAST edge on the shortest path from s to v, and disco which will keep the distance to the source, that is distTo[v] will be the length of the shortest path from s to v.

PROPERTIES


```
public Iterable<DirectedEdge> pathTo(int v) {
       Stack<DirectedEdge> path = new Stack<DirectedEdge>();
       for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
           path.push(e);
       }
       return path;
   }
```

edge-weighted digra

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.39	0.99
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.51
7	2->7 0.34	0.60

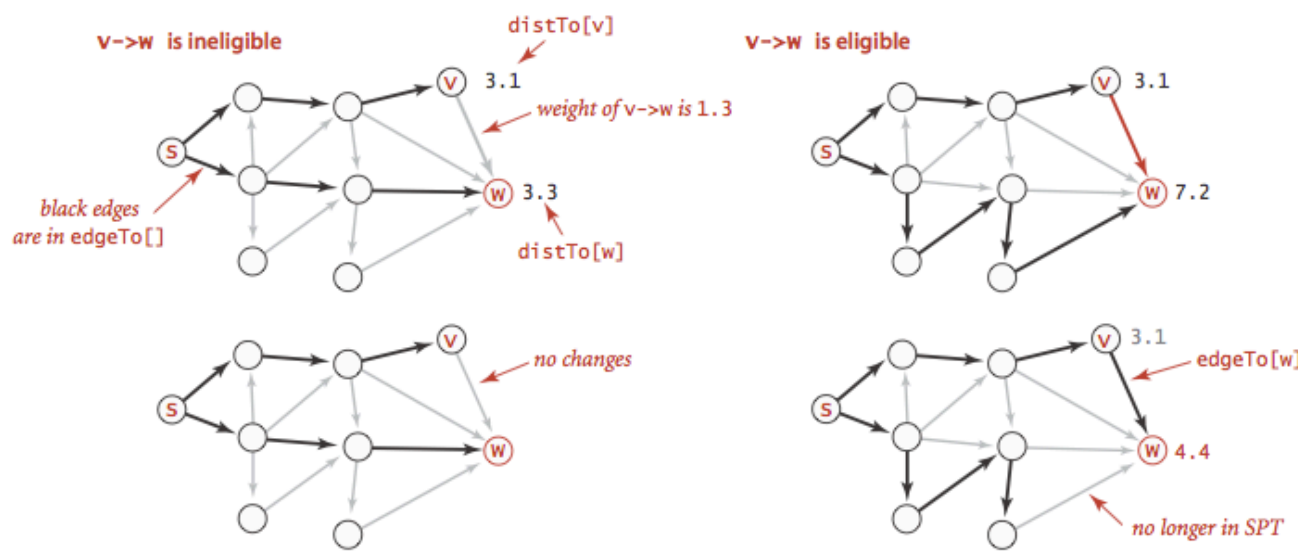
For example, here, we have an edge weighted digraph and you can see the edgeTo and distTo contents for all shortest paths from 0. To help us out return back a full shorts path from 0 to any vertex, we will use a pathTo method that uses a stack to give us back the edges from a destination vertex to the source vertex s in an iterable structure.

Edge relaxation

- Relax edge `e = v->w`
  - `distTo[v]` is the length of the shortest **known** path from `s` to `v`.
  - `distTo[w]` is the length of the shortest **known** path from `s` to `w`.
  - `edgeTo[w]` is the last edge on shortest **known** path from `s` to `w`.
  - If `e = v->w` yields shorter path to `w`, update `distTo[w]` and `edgeTo[w]`.

To calculate the contents of distTo and edgeTo, we will need the concept of edge relaxation. Remember, distTo[v] will store the length of the shortest path from s to v SO FAR. And similarly, edgeTo[w] will be the last edge of the shortest path from s to w SO FAR. Our ultimate goal, will be that when we are done with our algorithm, we will have found the overall shortest path. To get there, every time we encounter an edge e from v to w, we will ask whether it yields a shorter path from s to w. If yes, we will need to update distTo[w] and edgeTo[w].

Edge relaxation



There are two possible outcomes of an edge-relaxation operation. Either the edge is ineligible (as in the example at left, where the distTo[w] so far is 3.3 and going from s to w through v would actually increase the weight to 3.1+1.3=4.4) and no changes are made, or the edge v->w leads to a shorter path to w (as in the example at the right, where the best known path from s to w so far costs us 7.2, but if we were to go through v we would instead pay 4.4 which is cheaper) and we updated edgeTo[e] and distTo[e]

Edge relaxation implementation

```
private void relax(DirectedEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
    }
}
```

Implementing edge relaxation is extremely simple. Given an edge, we get the source and destination vertices and we update the distTo and edgeTo arrays for the destination only if the path through the source is shorter.

Framework for shortest-paths algorithm

- Generic algorithm to compute a SPT from s
  - distTo[v]=∞ for each vertex v.
  - edgeTo[v]=null for each vertex v.
  - distTo[s]=0.
  - Repeat until done:
    - Relax any edge.
- distTo[v] is the length of a simple path from s to v.
- distTo[v] does not increase.

Putting it all together, it's not hard to imagine a generic algorithm to compute a SPT from s. We will initiate distTo for every vertex as a very large number, which here I will represent as infinity, and edgeTo for every vertex as null. The distTo[s] will be obviously 0 :)
We will repeatedly relax any edge until no change is registered. Remember, distTo[v] is the length of a simple path from s to v and distTo[v] does not increase.

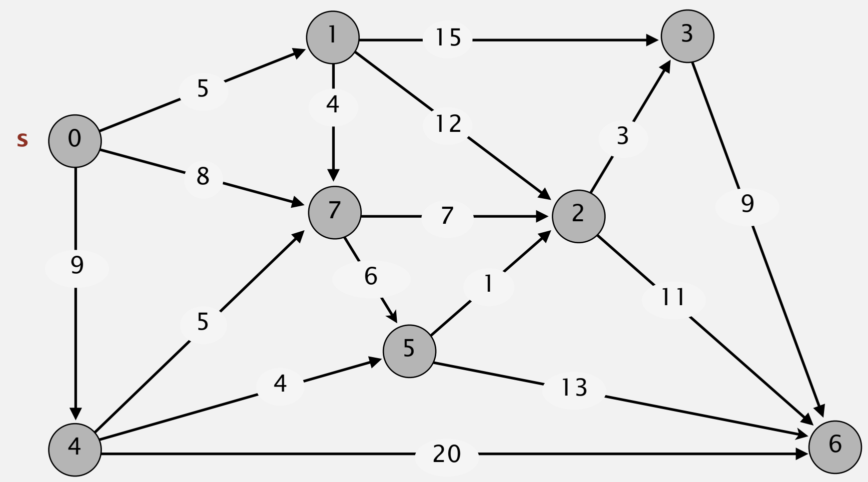
Lecture 23: Shortest Paths

- Introduction to Shortest Paths
- API
- Properties
- Dijkstra's Algorithm

All that can be done systematically using Dijkstra's algorithm.

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

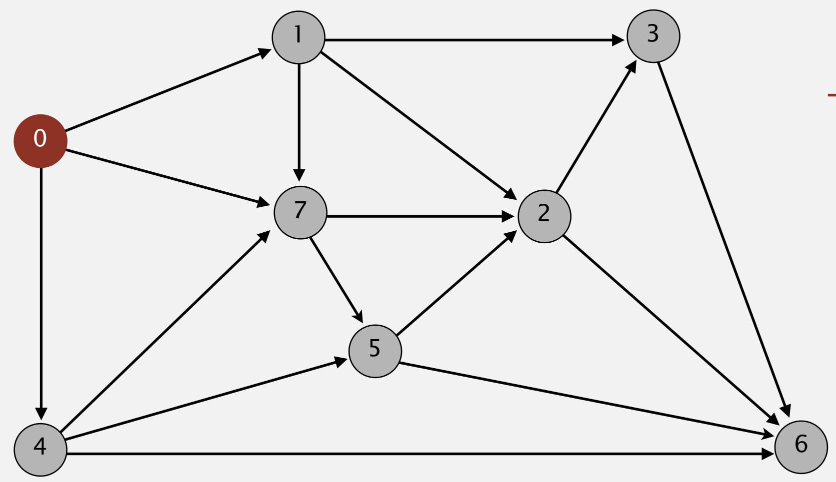
The algorithm considers vertices in increasing order of distance from s (let's say for this graph, vertex 0) by looking into the lowest disco values for vertices that are not part of the SPT. It then adds the vertex to the SPT and relaxes all edges adjacent from that vertex.

Let's take this graph as an example. On the right, you can see the edges and weights which are also indicated on the graph itself.

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



	v	distTo[]	edgeTo[]
→	0	0.0	-
	1		
	2		
	3		
	4		
	5		
	6		
	7		

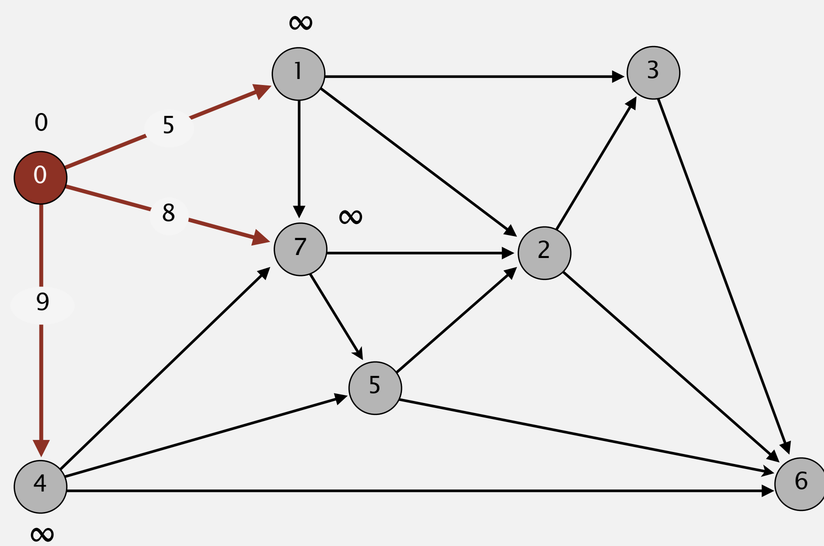
choose source vertex 0

We start at the source vertex 0, add the vertex to the SPT (marked in red), and set the distTo[0] as 0.0 and the edgeTo[0] as null.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
→ 0	0.0	-
1		
2		
3		
4		
5		
6		
7		

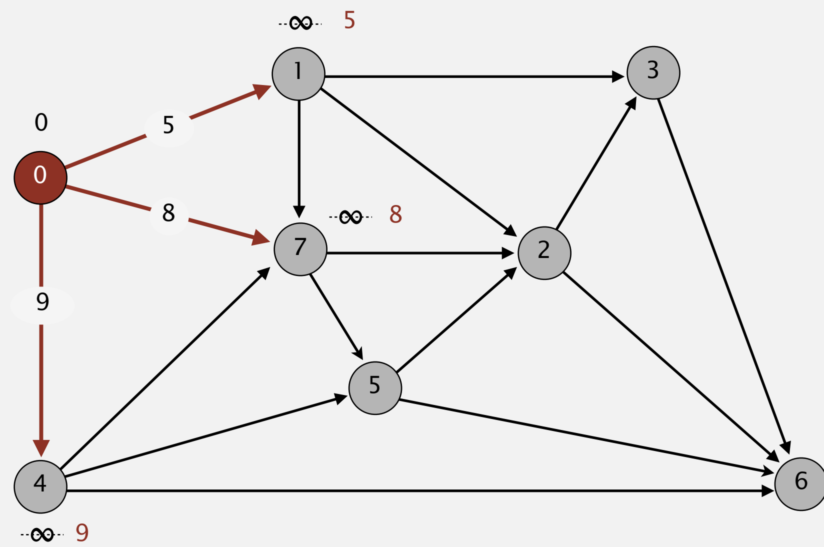
relax all edges adjacent from 0

We then consider all adjacent vertices to 0 and relax them. Initially, all have been initialized to infinity. So the distance to them from 0 will be updated since they can all be relaxed.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

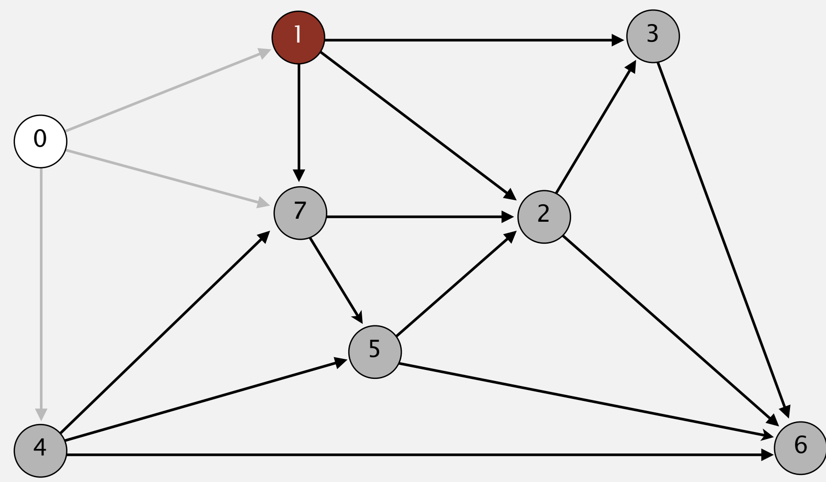
relax all edges adjacent from 0

We will be updating the table as we go. Now we will consider all vertices that have not been added to the SPT yet (everything but 0 so far) and pick the one with the smallest distance to s.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



	v	distTo[]	edgeTo[]
	0	0.0	-
→	1	5.0	0→1
	2		
	3		
	4	9.0	0→4
	5		
	6		
	7	8.0	0→7

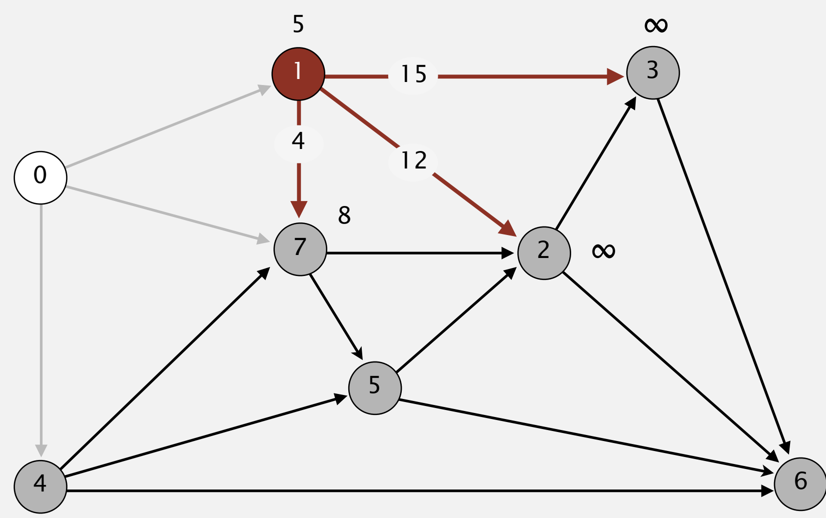
choose vertex 1

That would be vertex 1 with a total distance of 5.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



	v	distTo[]	edgeTo[]
	0	0.0	-
→	1	5.0	0→1
	2		
	3		
	4	9.0	0→4
	5		
	6		
	7	8.0	0→7

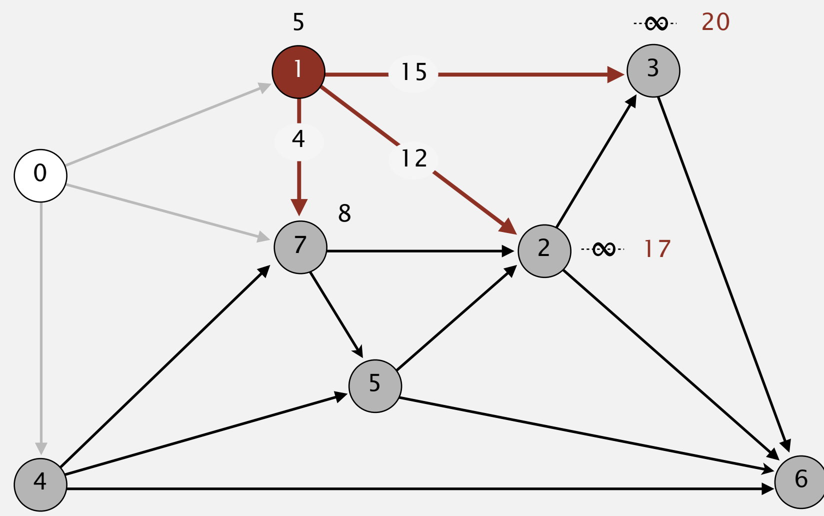
relax all edges adjacent from 1

We can add 1 to the SPT. 1's adjacent vertices are 2, 3, and 7. We will need to relax them by comparing their current distance (infinity, infinity, and 8, respectively) with the cost we would have to pay if we were to go from 0 to them through 1 (i.e. 5+15=20, 5+12 = 17, and 5+4 = 9).

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
→ 1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0 ✔	0→7

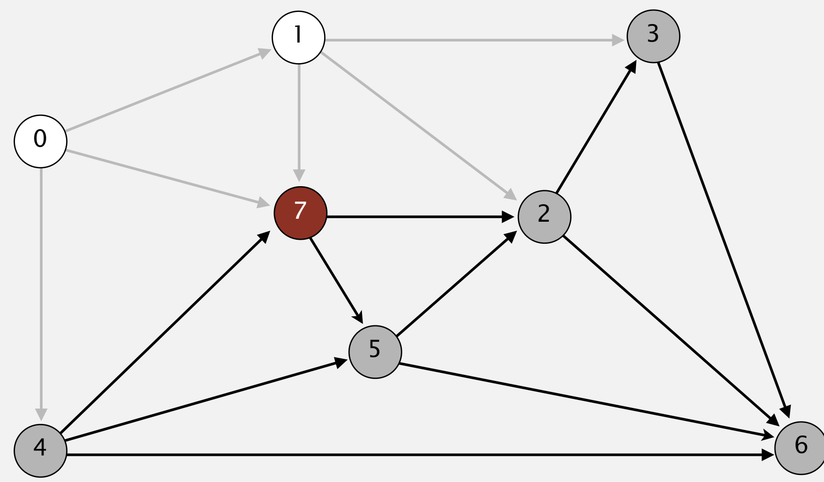
relax all edges adjacent from 1

We will update the table for vertices 2 and 3 but not for 7 since paying 9 would be higher than the current cost of going directly to it (8). We will consider all vertices that are not on SPT yet (all except for 0 and 1) and pick the one with the shortest distance.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
→ 7	8.0	0→7

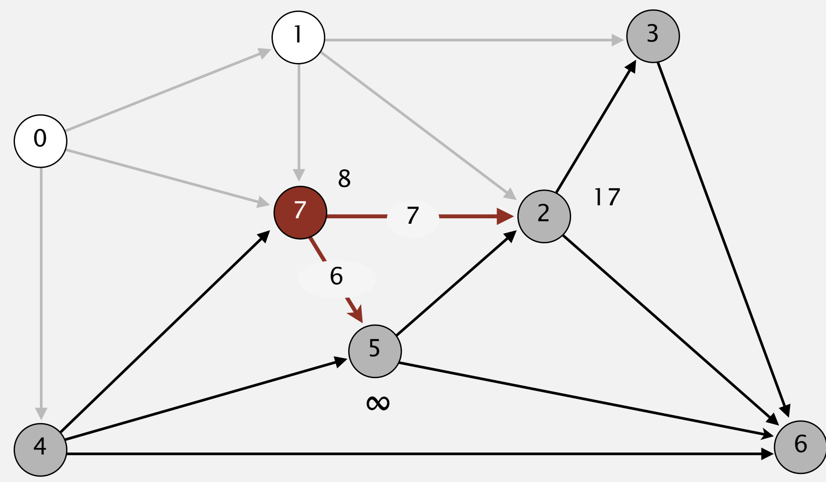
choose vertex 7

That would be vertex 7 with a cost of 8. Don't forget to update edgeTo as the last edge to the best known shortest path so far from 0 to that edge.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
→ 7	8.0	0→7

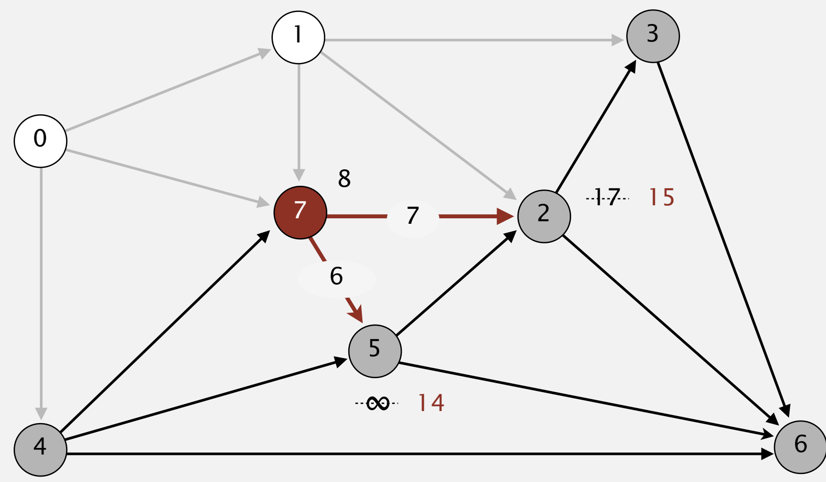
relax all edges adjacent from 7

Consider the adjacent vertices of 7. These would be 2 and 5 with a cost of 8+7=15 and 8+6 = 14.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

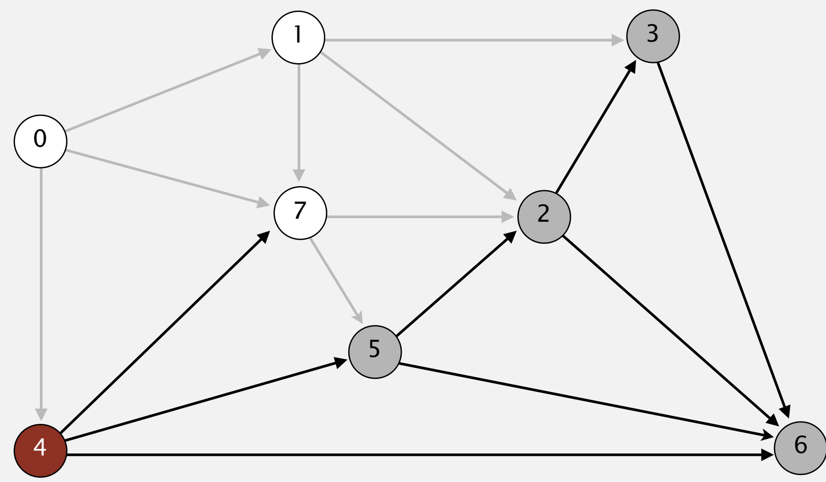
relax all edges adjacent from 7

We will update both vertices with the shorter new total weights. We are ready to move on to the next vertex that is not part of the SPT and has the smallest distance.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
→ 4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

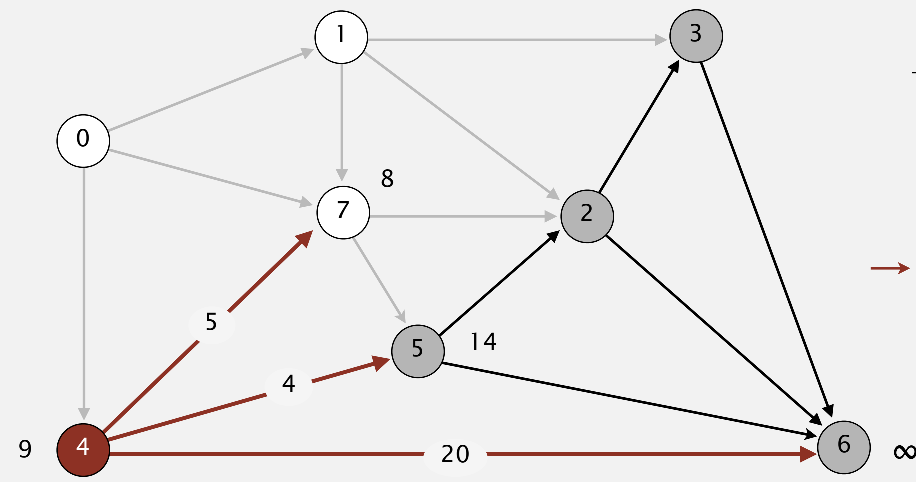
select vertex 4

That would be vertex 4 for a cost of 9.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



	v	distTo[]	edgeTo[]
	0	0.0	-
	1	5.0	0→1
	2	15.0	7→2
	3	20.0	1→3
→	4	9.0	0→4
	5	14.0	7→5
	6		
	7	8.0	0→7

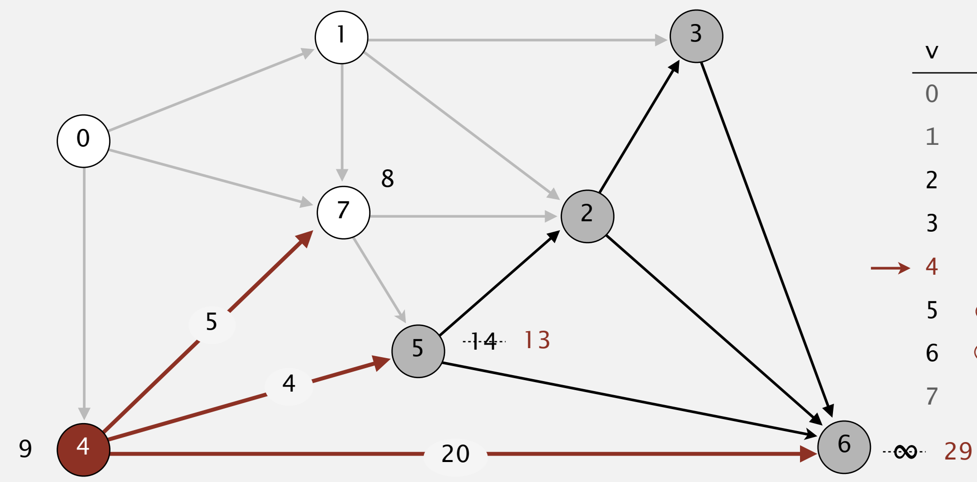
relax all edges adjacent from 4

4's adjacent vertices are 5, 6, and 7. If we were to go from 0 to them through 4, we would pay 9+4=13, 9+20=29, and 9+5=14, respectively.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
→ 4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0 ✔	0→7

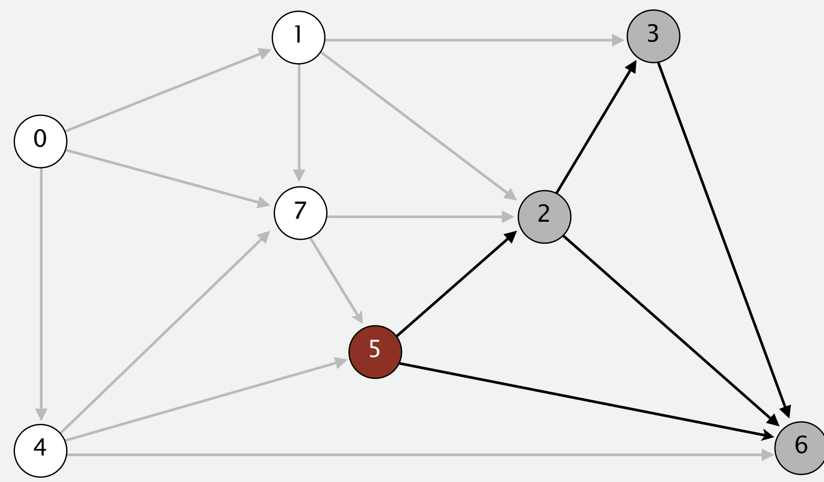
relax all edges adjacent from 4

Which means we will update only vertices 5 and 6 (since the cost to 7 is higher than the existing 8). Vertex 4 is now part of the SPT.

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

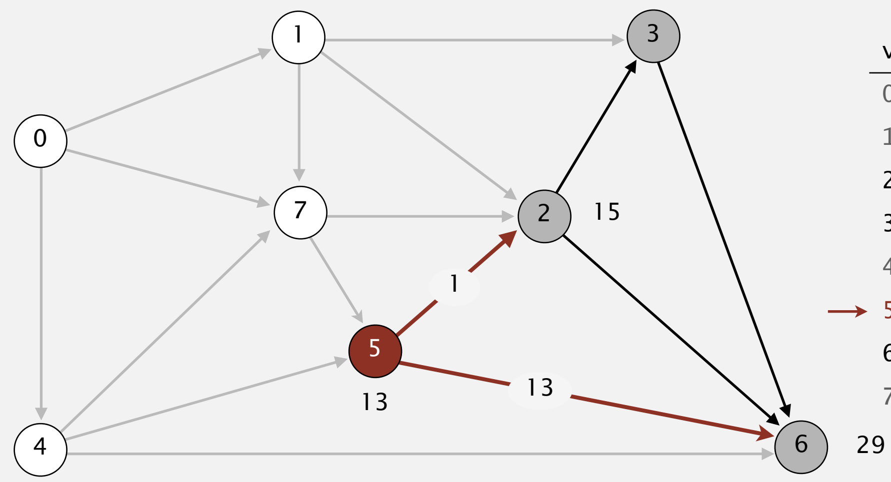
select vertex 5

The next vertex would be 5.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



	v	distTo[]	edgeTo[]
	0	0.0	-
	1	5.0	0→1
	2	15.0	7→2
	3	20.0	1→3
	4	9.0	0→4
→	5	13.0	4→5
	6	29.0	4→6
	7	8.0	0→7

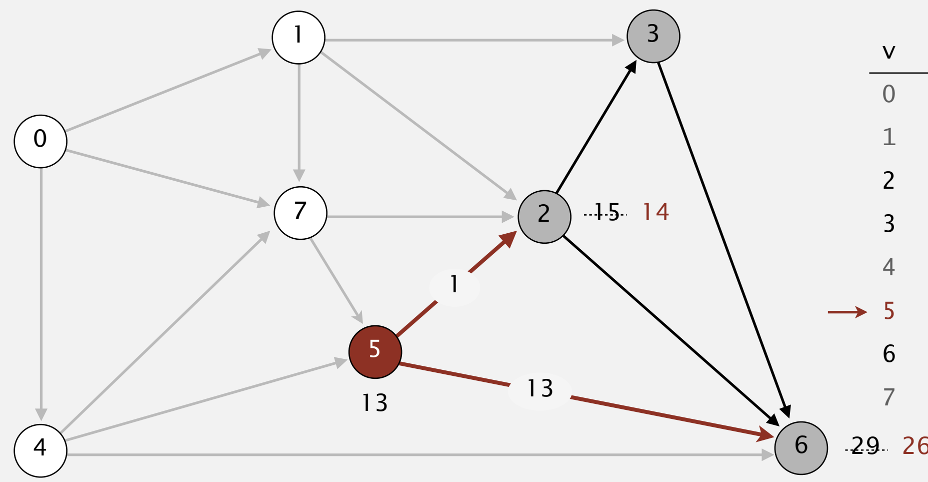
relax all edges adjacent from 5

Its adjacent vertices are 2 and 6

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

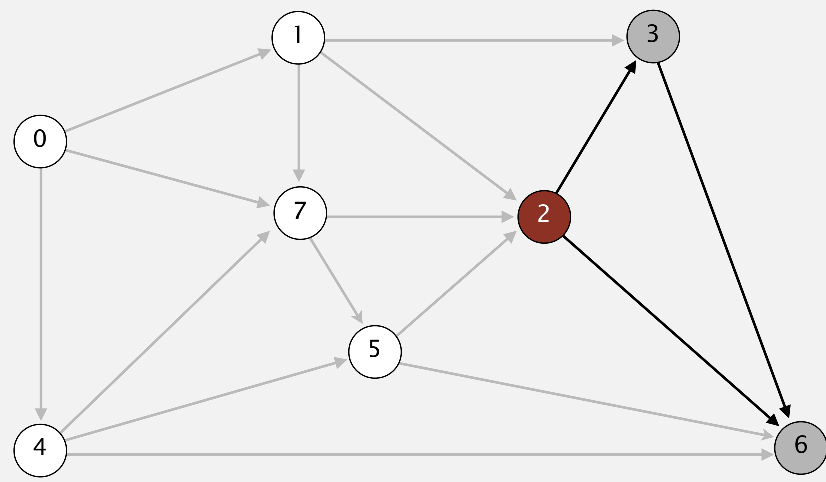
relax all edges adjacent from 5

and both are relaxed to new total distances.

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



	v	distTo[]	edgeTo[]
	0	0.0	-
	1	5.0	0→1
→	2	14.0	5→2
	3	20.0	1→3
	4	9.0	0→4
	5	13.0	4→5
	6	26.0	5→6
	7	8.0	0→7

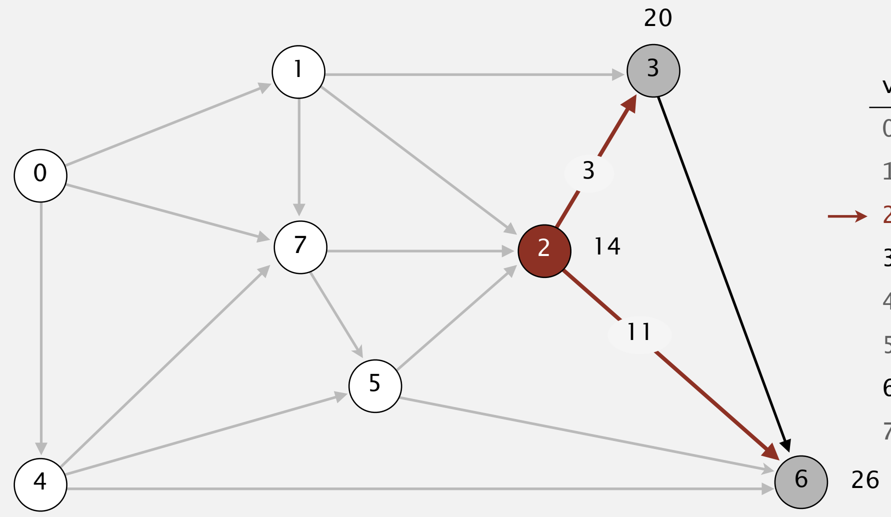
select vertex 2

We then proceed with vertex 2.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
→ 2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

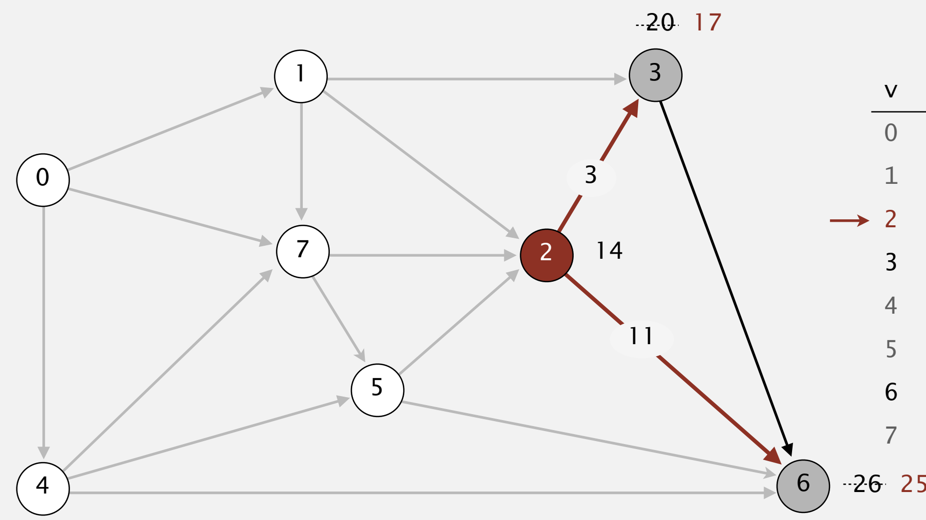
relax all edges adjacent from 2

its adjacent vertices are 3 and 6

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



	v	distTo[]	edgeTo[]
	0	0.0	-
	1	5.0	0→1
→	2	14.0	5→2
	3	17.0	2→3
	4	9.0	0→4
	5	13.0	4→5
	6	25.0	2→6
	7	8.0	0→7

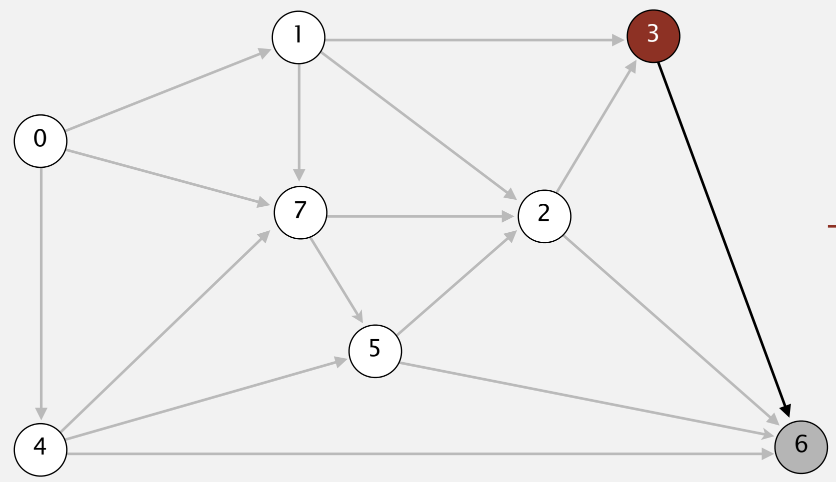
relax all edges adjacent from 2

and are both relaxed.

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

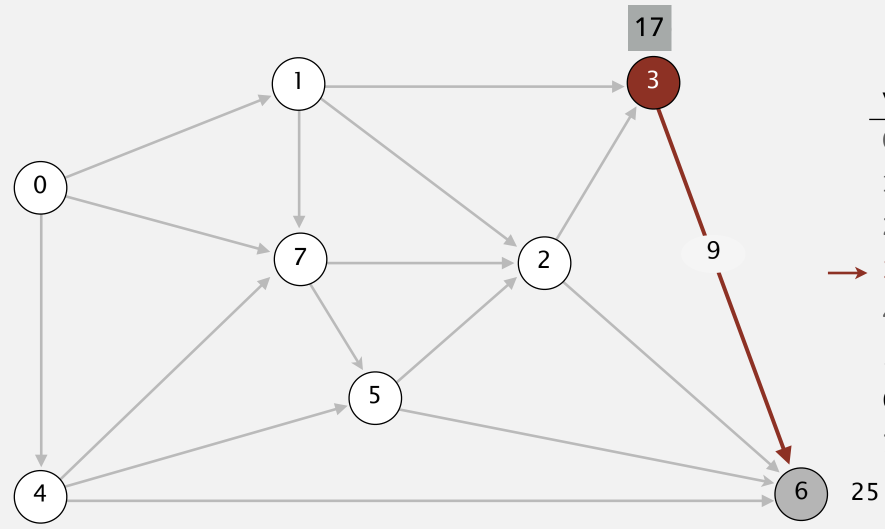
select vertex 3

next vertex is 3

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
→ 3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

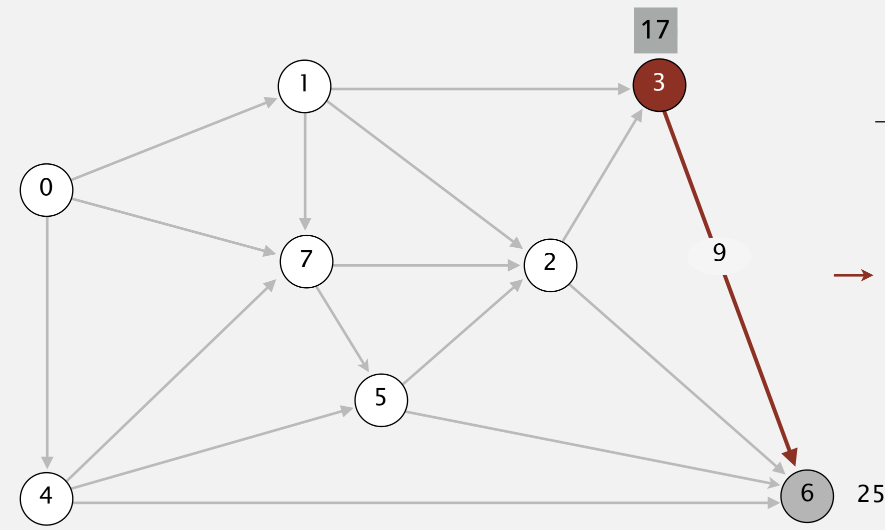
relax all edges adjacent from 3

it has only one adjacent vertex, 6,

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]		edgeTo[]
0	0.0		-
1	5.0		0→1
2	14.0		5→2
→ 3	17.0		2→3
4	9.0		0→4
5	13.0		4→5
6	25.0	✔	2→6
7	8.0		0→7

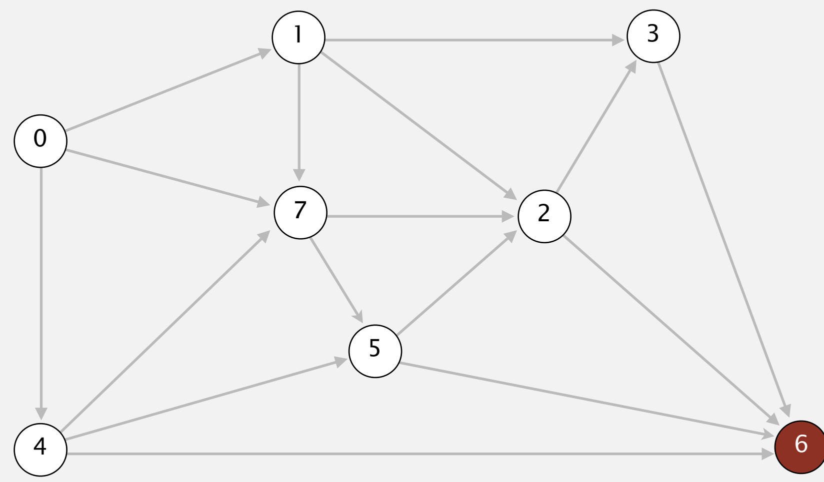
relax all edges adjacent from 3

which won't be relaxed because the cost to go to it from 0 through 3 would be higher than go through 2.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
→ 6	25.0	2→6
7	8.0	0→7

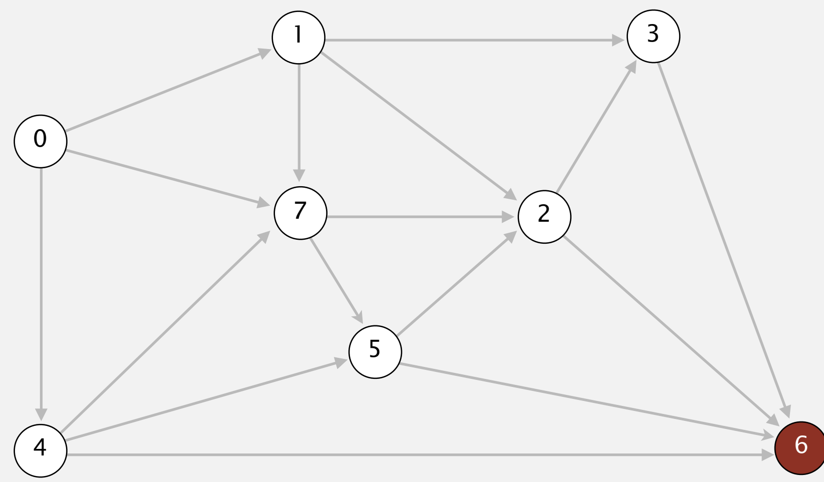
select vertex 6

Last vertex is 6

Dijkstra's algorithm demo

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
→ 6	25.0	2→6
7	8.0	0→7

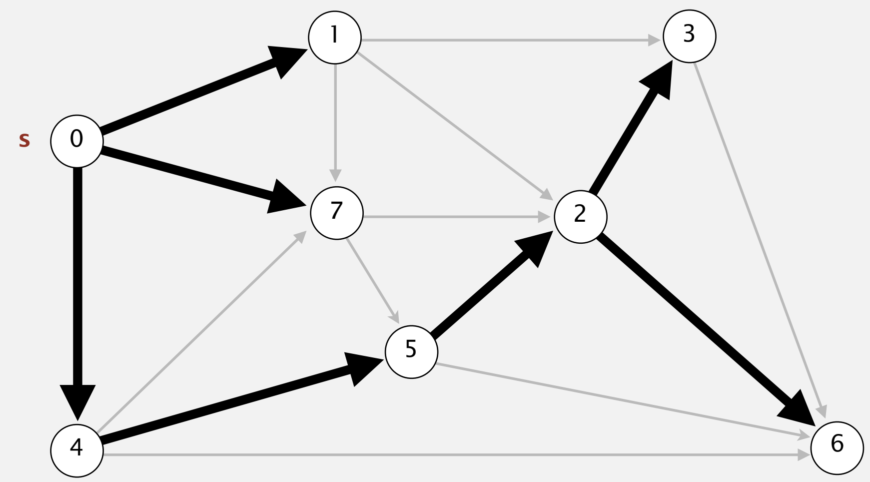
relax all edges adjacent from 6

which happens to not have any adjacent vertices.

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges adjacent from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Here's our SPT from vertex s and the final arrays distTo and edgeTo.

Indexed min-priority queue (Section 2.4 in recommended textbook)

- Associate an index between 0 and n-1 with each key in a priority queue.
  - Insert a key associated with a given index.
  - Delete a minimum key and return associated index.
  - Decrease the key associated with a given index.
- `public class IndexMinPQ<Key extends Comparable<Key>>`
  - `IndexMinPQ(int n)`
    - Create indexed PQ with indices 0,1,…n-1
  - `void insert(int i, Key key)`
    - Associate key with index i.
  - `int delMin()`
    - Remove a minimal key and return its associated index.
  - `void decreaseKey(int i, Key key)`
    - Decrease the key with index i to the specified value.

How do we go about implementing Dijkstra's algorithm? We will use a min priority queue (essentially a min-heap whose root is the minimum value) and work with inserting deleting and decreasing keys.

```java
public class DijkstraSP {
    private double[] distTo;          // distTo[v] = distance  of shortest s->v path
    private DirectedEdge[] edgeTo;    // edgeTo[v] = last edge on shortest s->v path
    private IndexMinPQ<Double> pq;    // priority queue of vertices


    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        // relax vertices in order of distance from s
        pq = new IndexMinPQ<Double>(G.V());
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }

    // relax edge e and update pq if changed
    private void relax(DirectedEdge e) {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
        }
    }
```

We won't really focus on the implementation (we used to have an assignment on this), but you can see that the implementation of Dijkstra's algorithm is not hard.

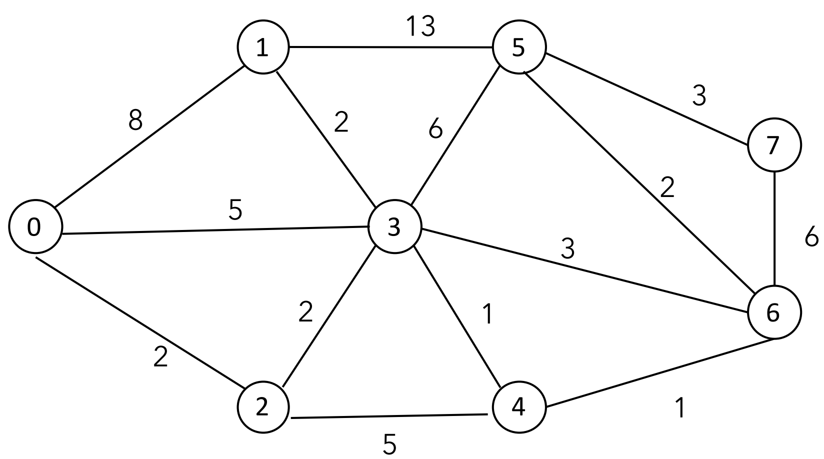
Running time depends on PQ implementation

- Many variations. Assuming binary heap, running time is proportional to $|E|\log|V|$ and $|V|$ extra space.
  - Cost of insert, delete-min, decrease-key are all $\log|V|$.
- More complicated version with a Fibonacci heap takes $O(|E| + |V|\log|V|)$ time but in practice it's not worth implementing.

We won't do the analysis (stay tuned for CS140) but assuming you use a binary heap, the running time will be proportional to E log V and you will need V extra space (the cost for insertion, deletion of min and decreasing a key are all logarithmic). There is a fancier version that uses a Fibonacci heap and makes the algorithm faster but it's not worth implementing.

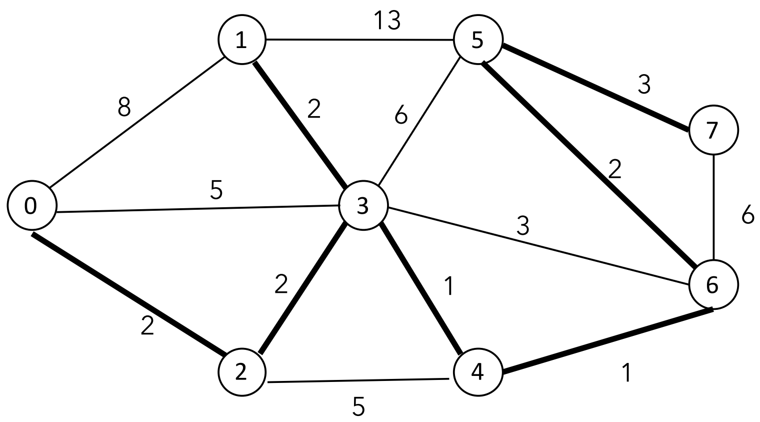
Practice Time

- Run Dijkstra's algorithm on the following graph with 0 being the starting vertex.



Let's see whether we understand how the algorithm works. Run Dijkstra's algorithm on the following graph with 0 being the starting vertex.

Answer



v	distTo[]	edgeTo[]
0	0	-
1	6	3->1
2	2	0->2
3	4	2->3
4	5	3->4
5	8	6->5
6	6	4->6
7	11	5->7

You should have ended up with this SPT and table.

Lecture 23: Shortest Paths

- Introduction to Shortest Paths
- API
- Properties
- Dijkstra's Algorithm

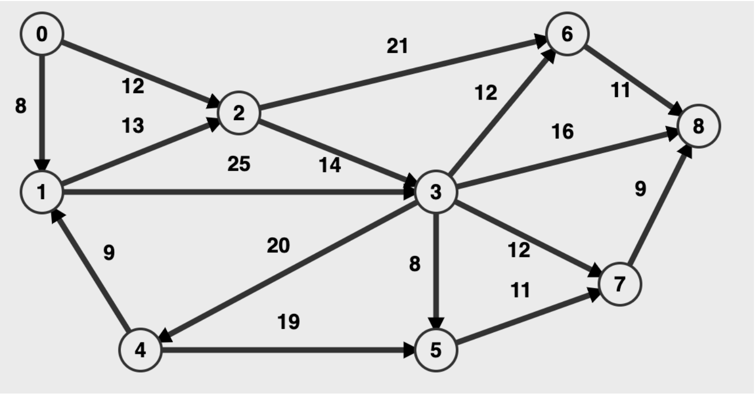
And that's all for today where we saw how to solve the shortest paths problem using Dijkstra's algorithm.

Readings:

- Recommended Textbook: Chapter 4.4 (Pages 638-676)
- Website:
  - https://algs4.cs.princeton.edu/44sp/
- Visualization
  - https://visualgo.net/en/sssp

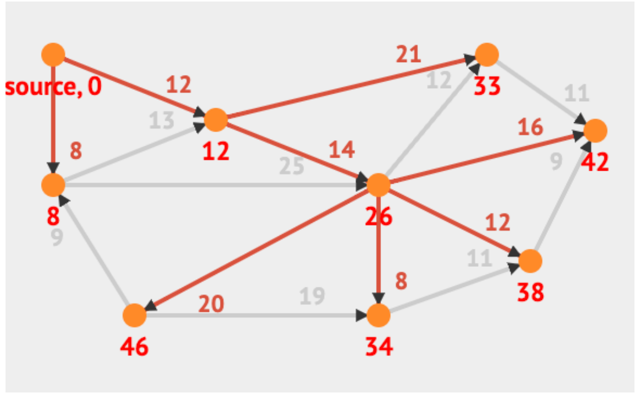
Problem

- Run Dijkstra's algorithm on the following graph with 0 being the starting vertex.



Answer

- Run Dijkstra's algorithm on the following graph with 0 being the starting vertex.



v	distTo[]	edgeTo[]
0	0	-
1	8	0->1
2	12	0->2
3	26	2->3
4	46	3->4
5	34	3->5
6	33	3->6
7	38	3->7
8	42	3->8