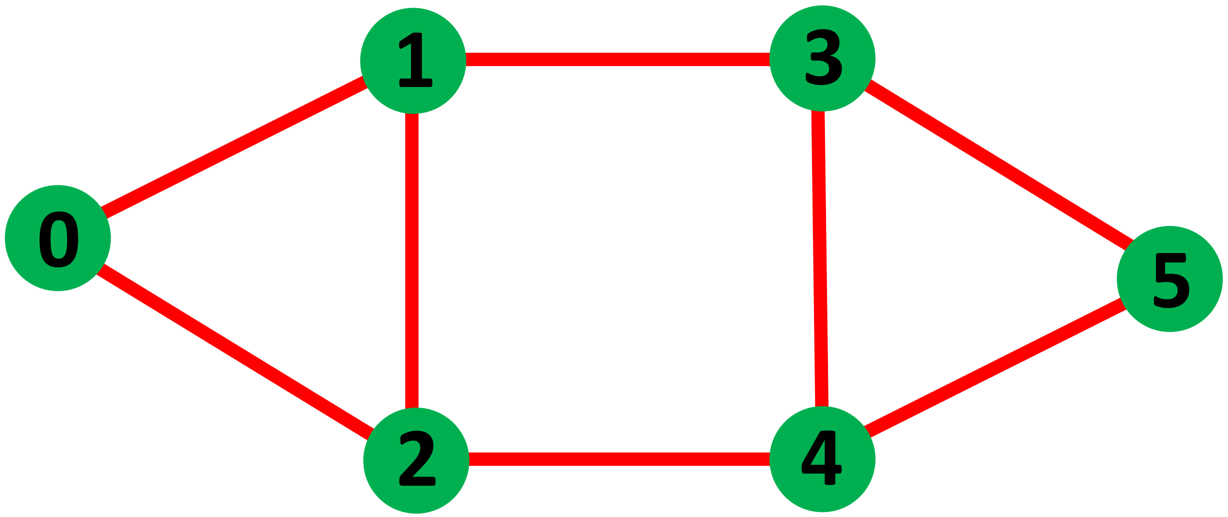
CSCI 232:
Data Structures and Algorithms

Shortest Path

Reese Pearsall
Summer 2025

Graphs

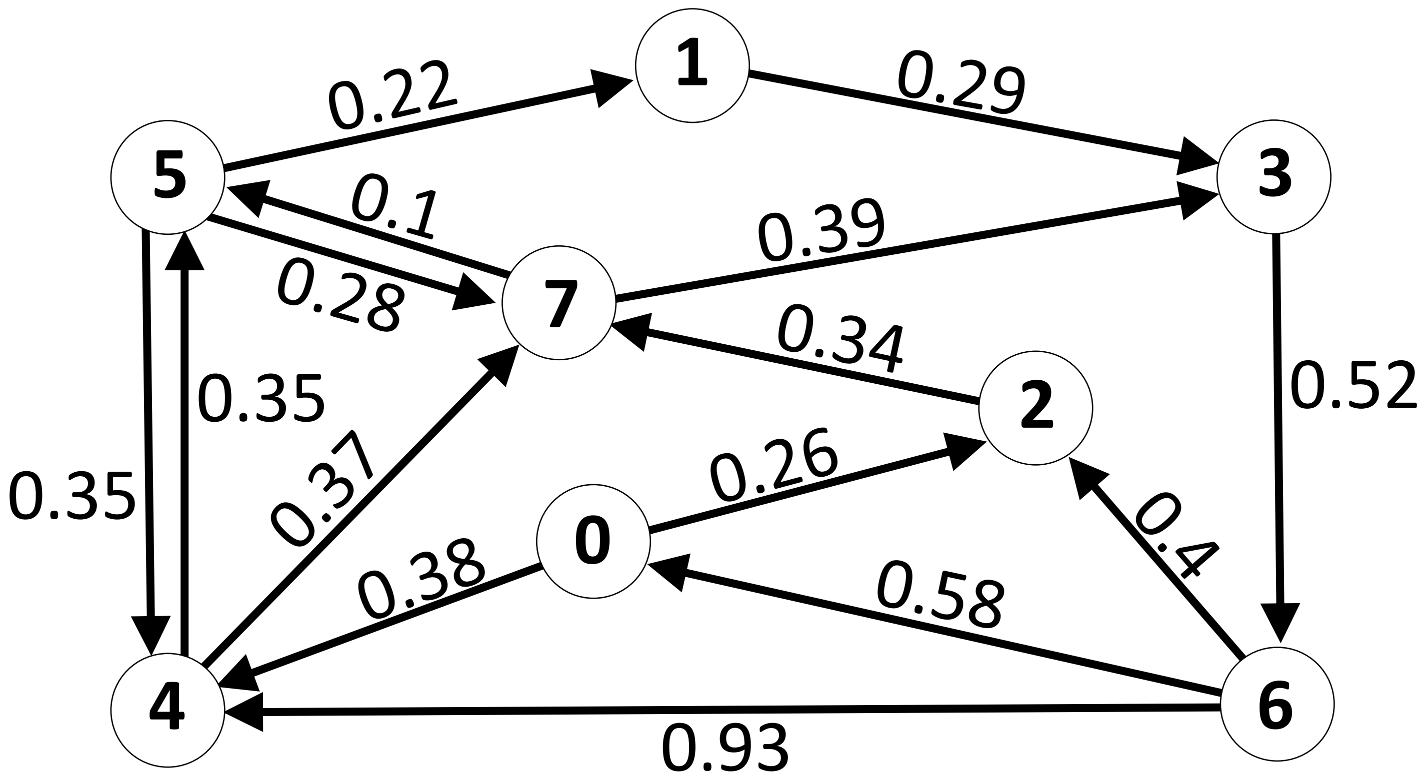
G = (V, E)



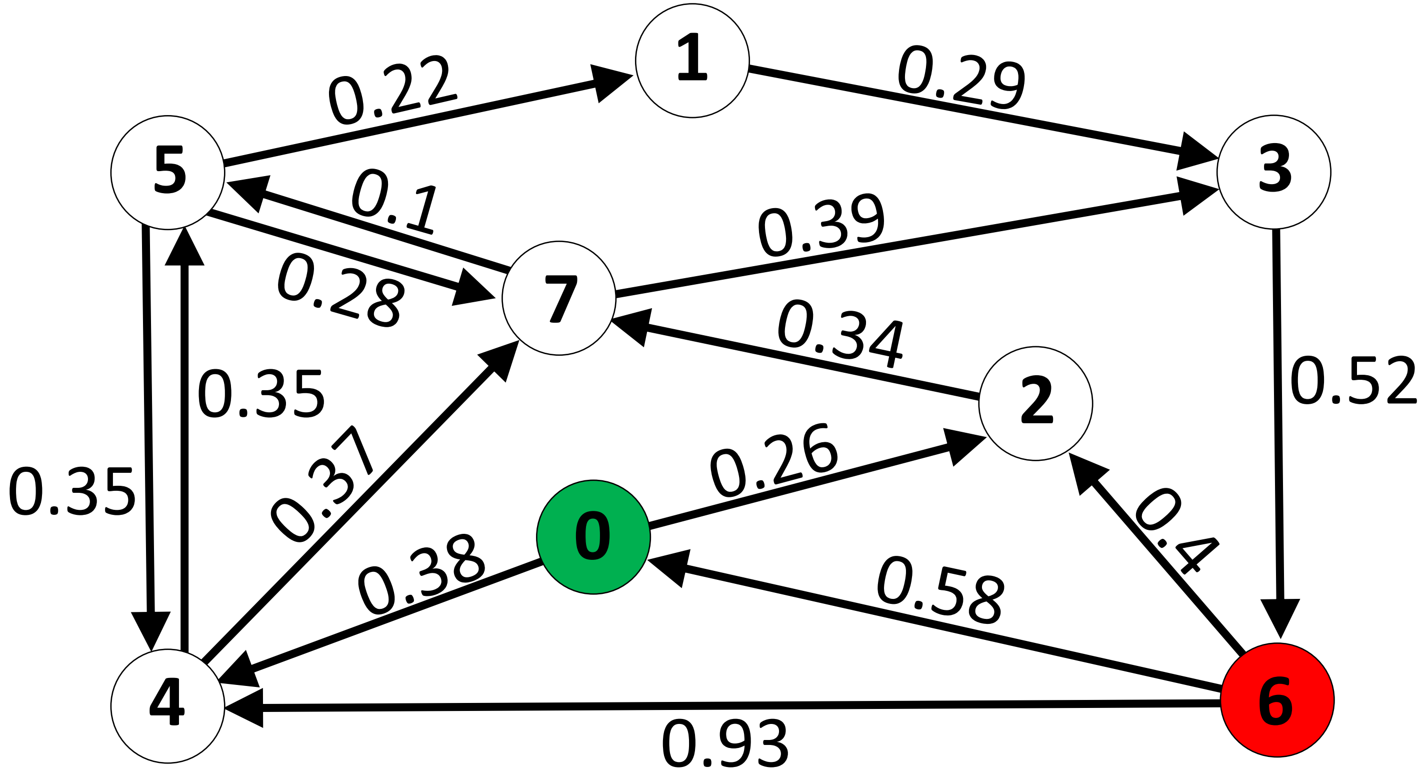
Adjacency List

0	→	{1,2}
1	→	{0,2,3}
2	→	{0,1,4}
3	→	{1,4,5}
4	→	{2,3,5}
5	→	{3,4}

Shortest Path



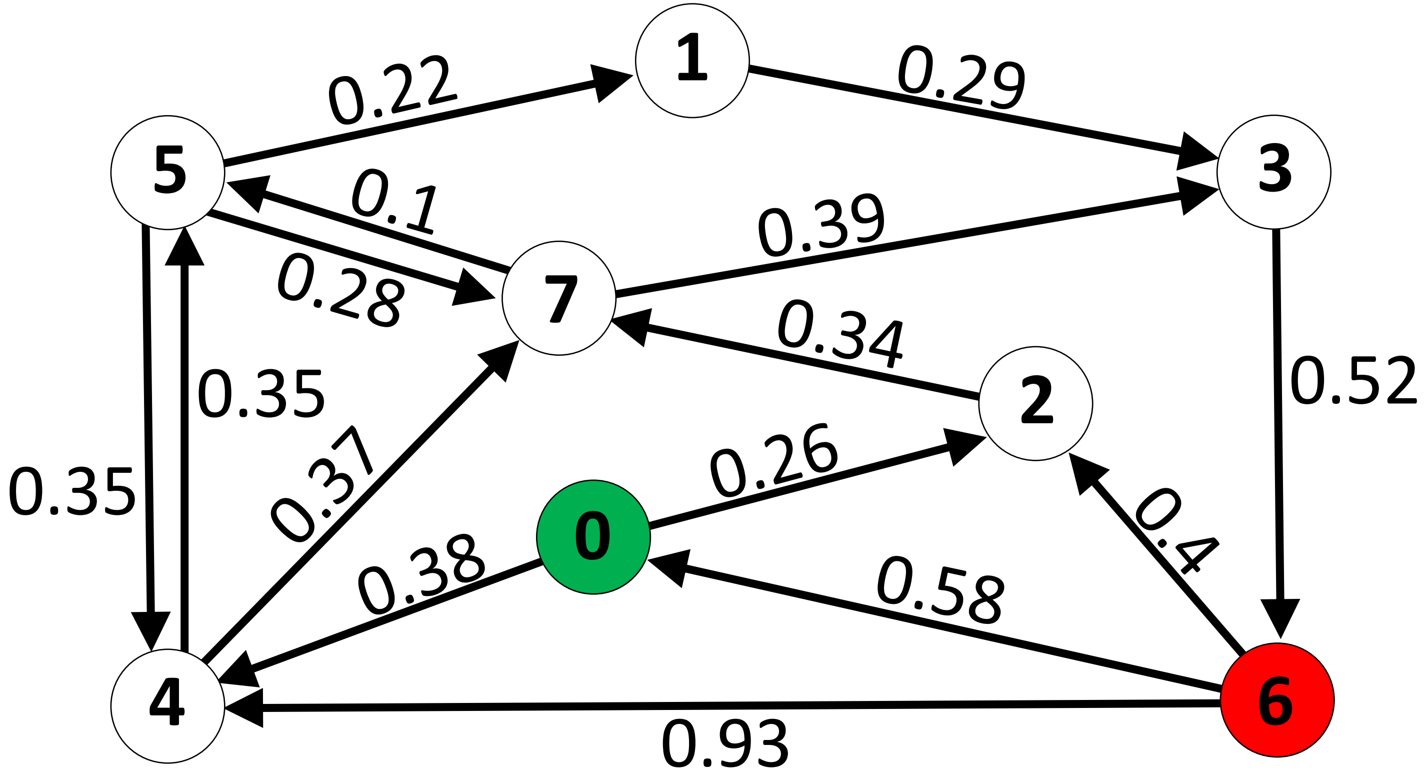
Shortest Path



Path with the smallest sum of edge weights.

What is the shortest path between **vertex 0** and **vertex 6**?

Shortest Path

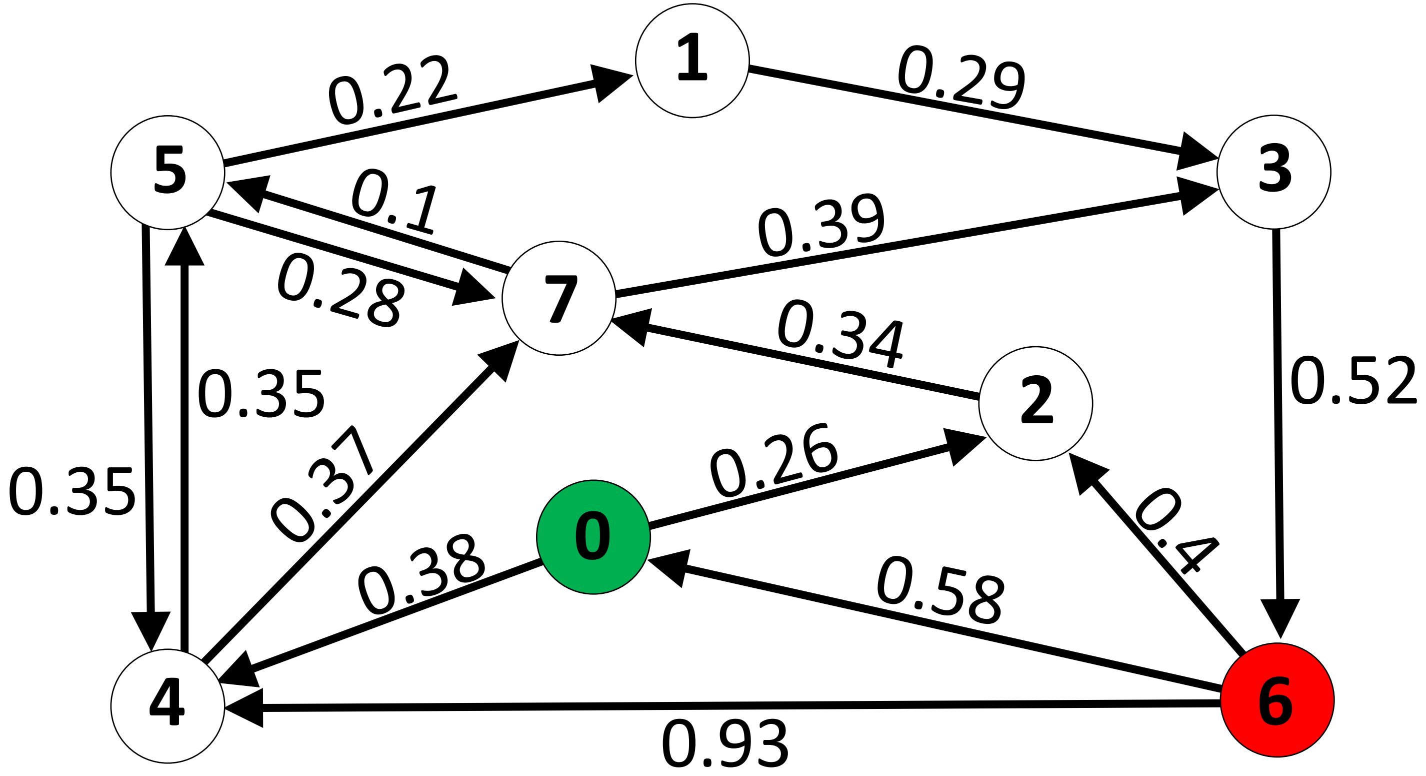


Assumptions:

- Graph is directed.
- Graph is edge-weighted.
- Edge weights are non-negative.
- Graph need not be simple (though our example will be).

What is the shortest path between vertex 0 and vertex 6?

Shortest Path



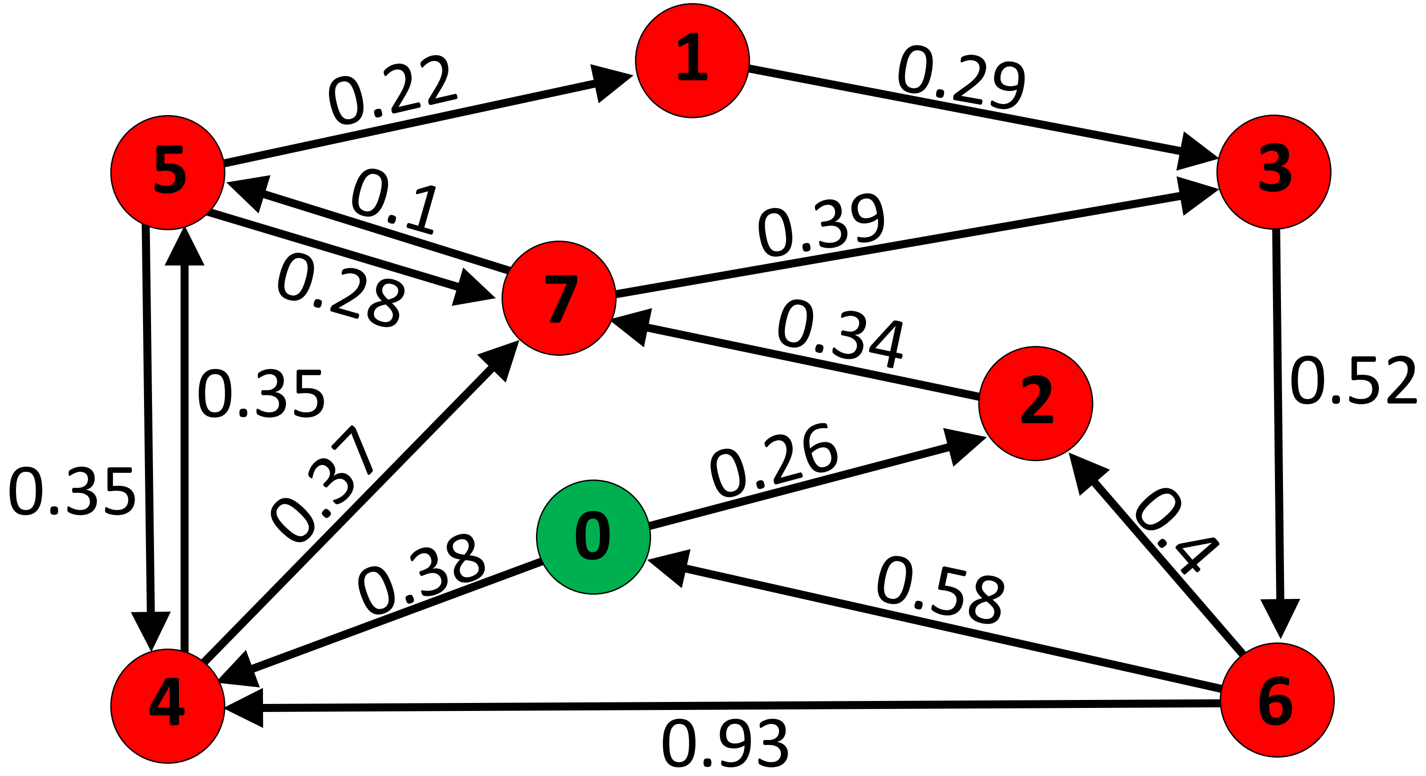
Assumptions:

- Graph is directed.
- Graph is edge-weighted.
- Edge weights are non-negative.
- Graph need not be simple (though our example will be).

Ideas?

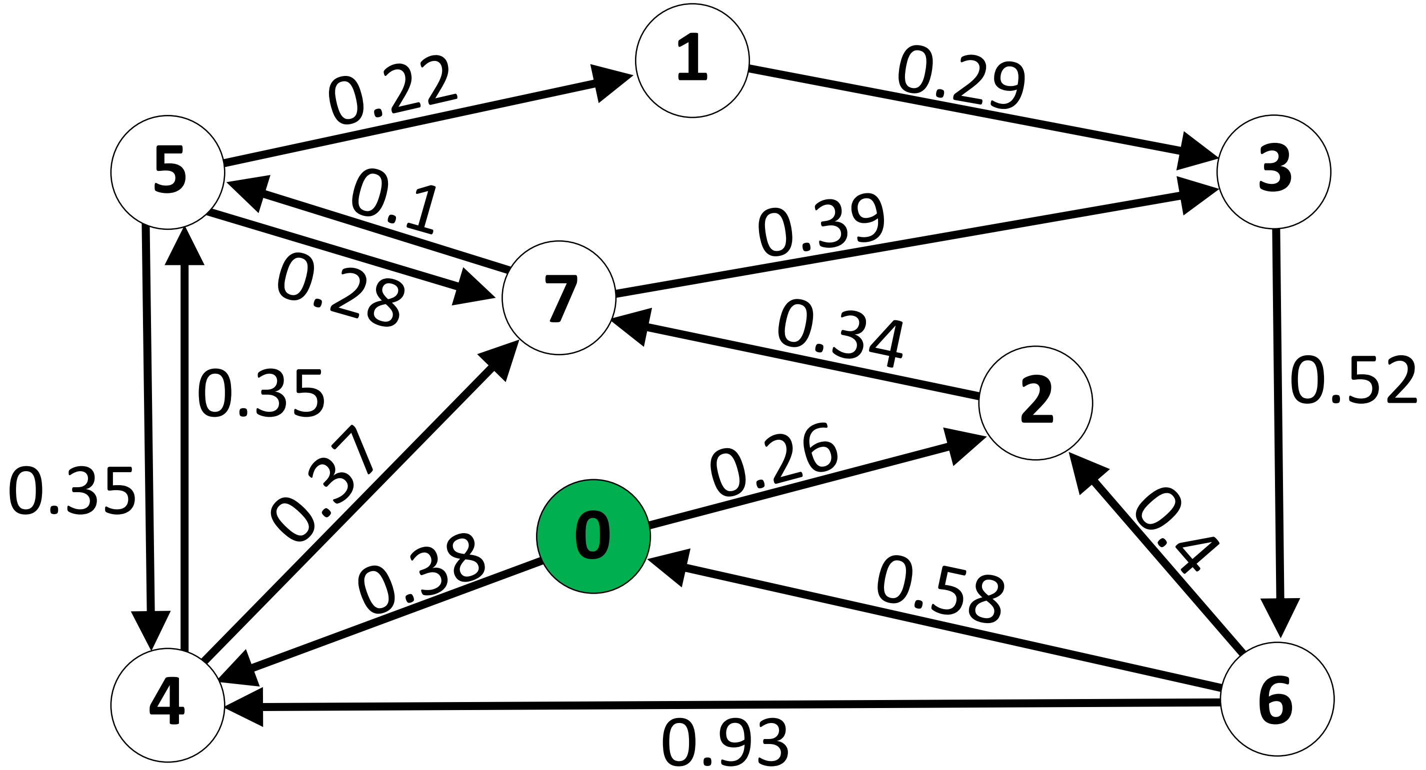
What is the shortest path between vertex 0 and vertex 6?

Shortest Path



We are going to find the shortest path between vertex 0 and every other vertex, flooding out from 0.

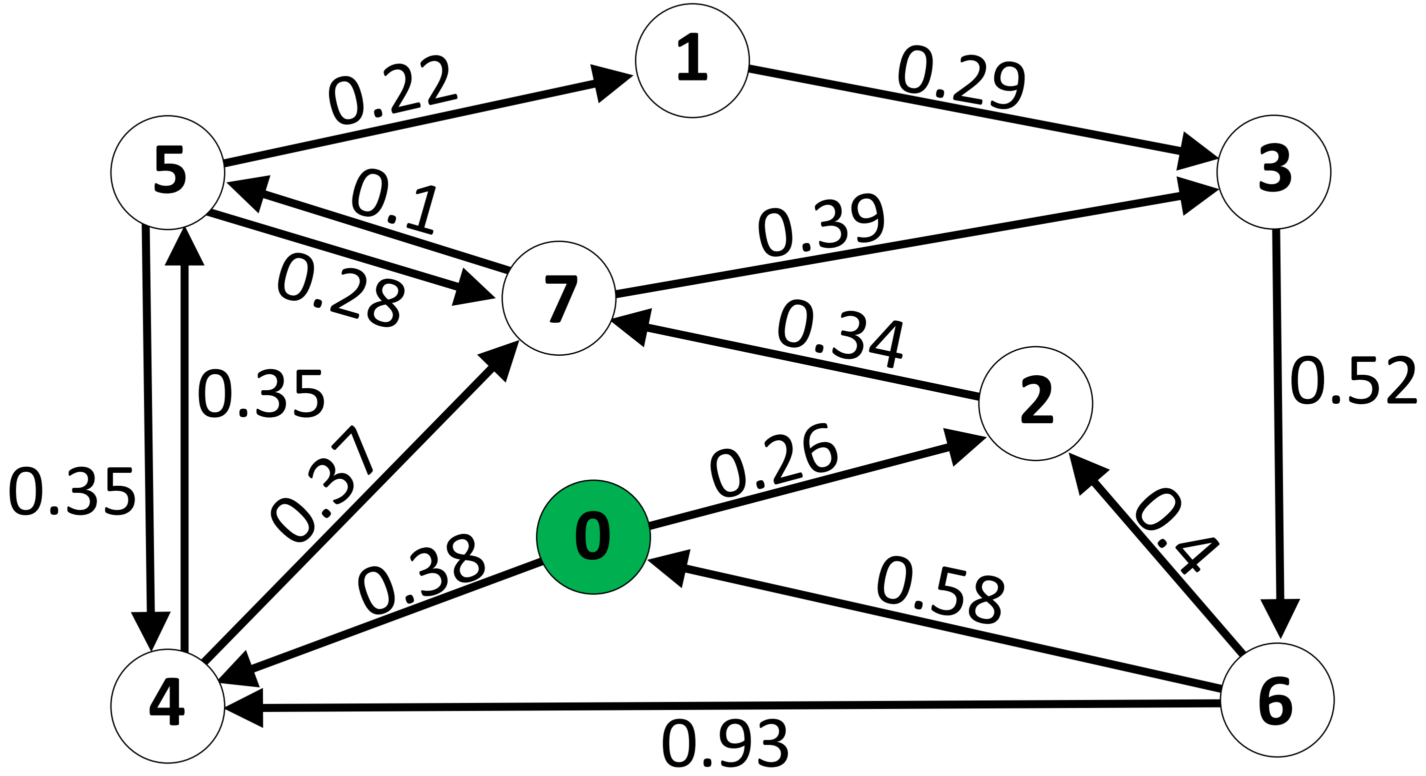
Shortest Path



Distance from 0

0	?
1	?
2	?
3	?
4	?
5	?
6	?
7	?

Shortest Path



Distance from 0

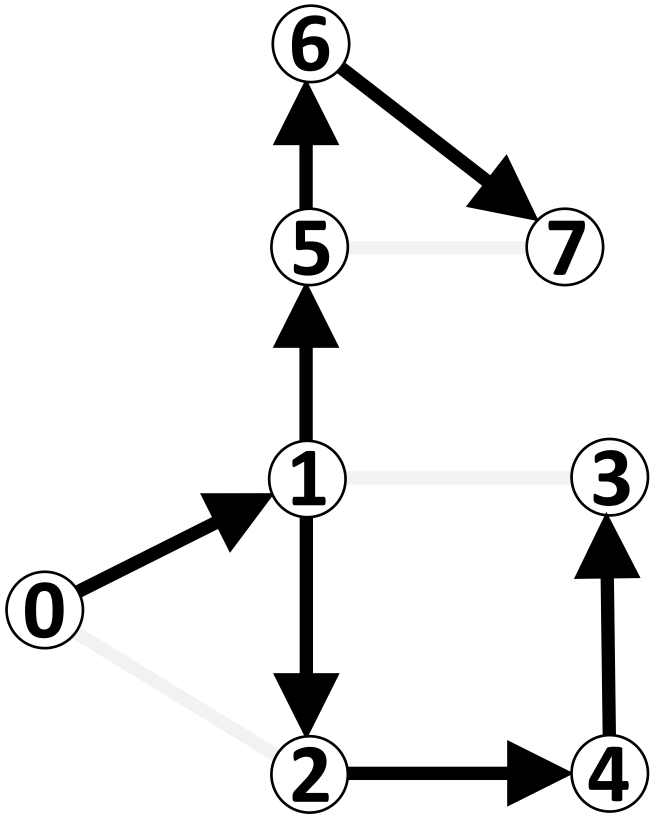
0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

How can we keep track of routes?

Graphs - Paths

`int[] previousVertex`

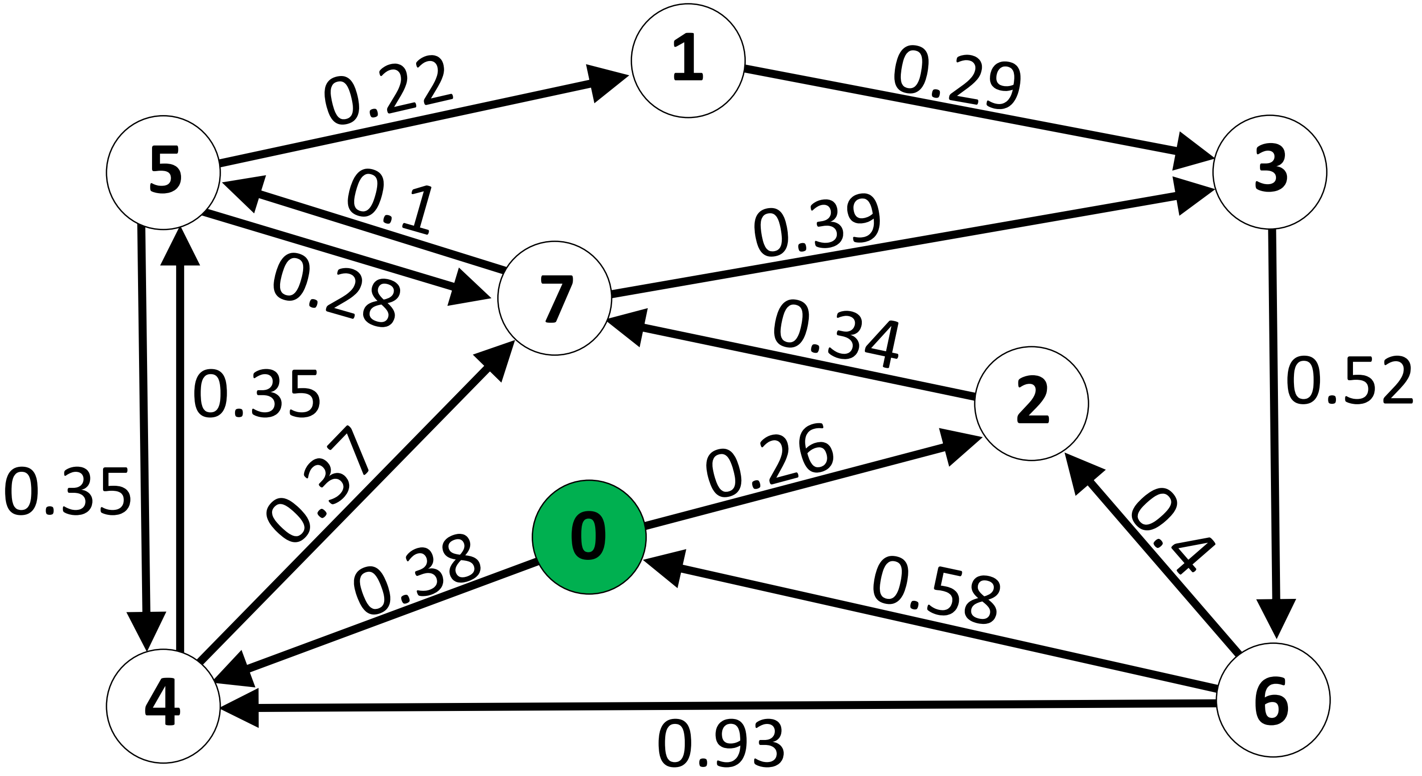
0	-
1	0
2	1
3	4
4	2
5	1
6	5
7	6



How do we determine the path from 0 to 6?

Start at vertex 6. Find its previous vertex. Find its previous vertex… until we get back to the start (0).

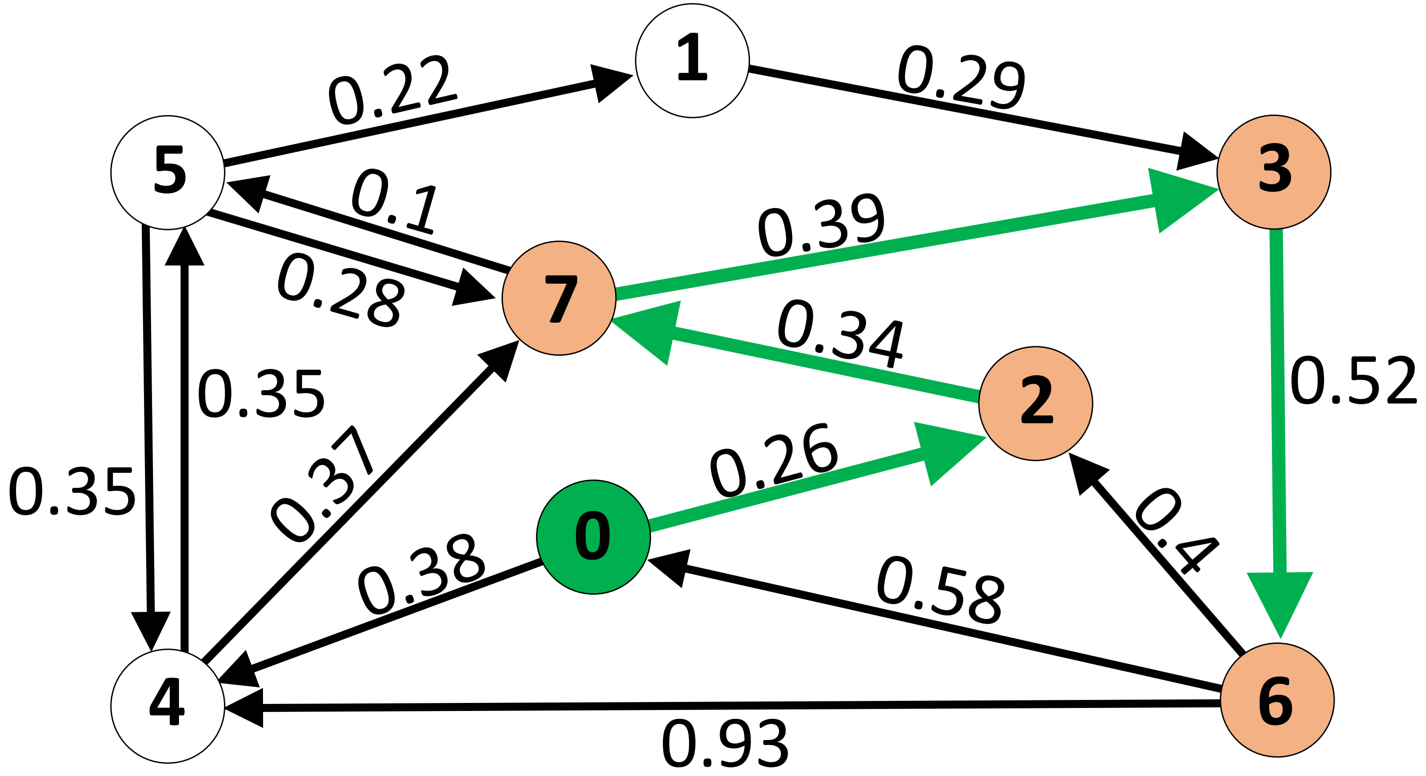
Shortest Path



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

How can we keep track of routes?

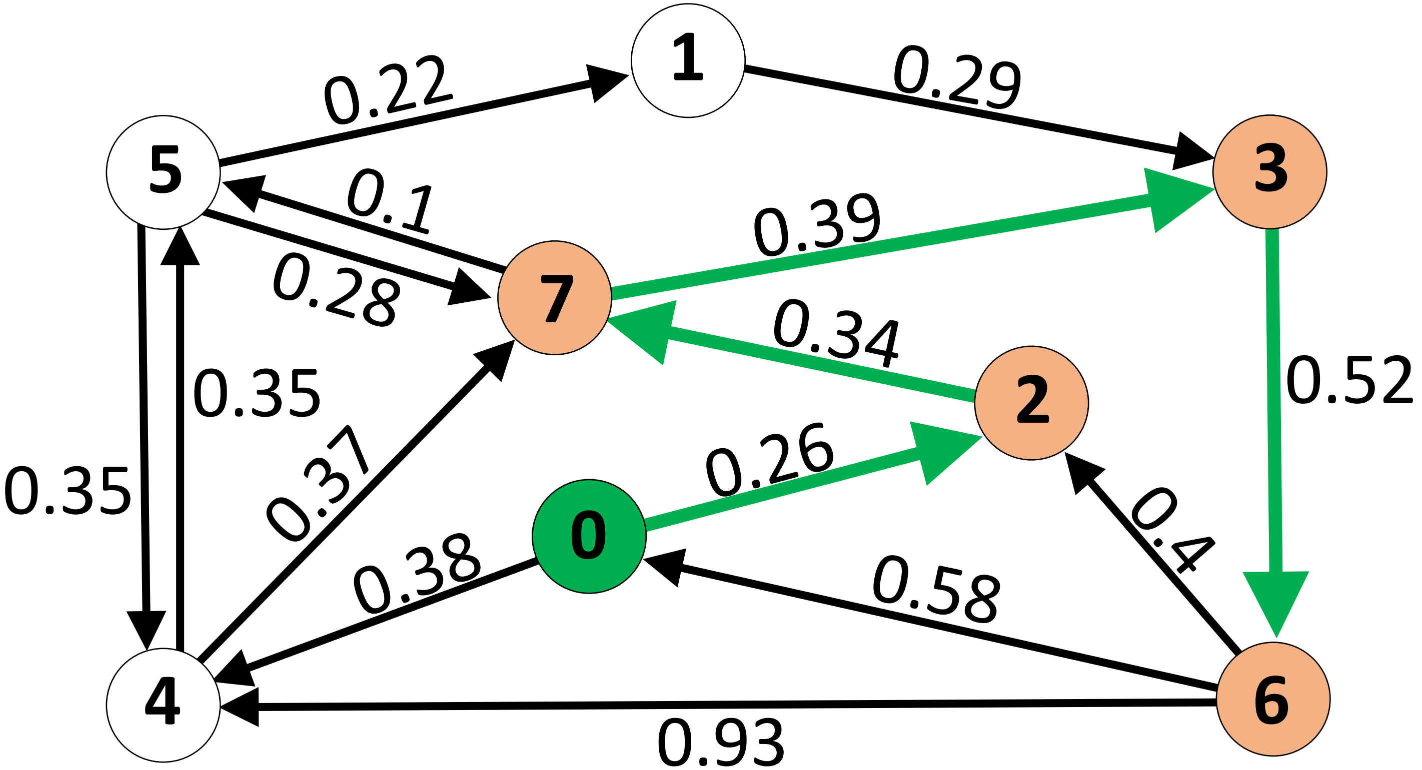
Shortest Path



Vertex	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	0.99	7
4	∞	
5	∞	
6	1.51	3
7	0.60	2

How can we keep track of routes?

Shortest Path

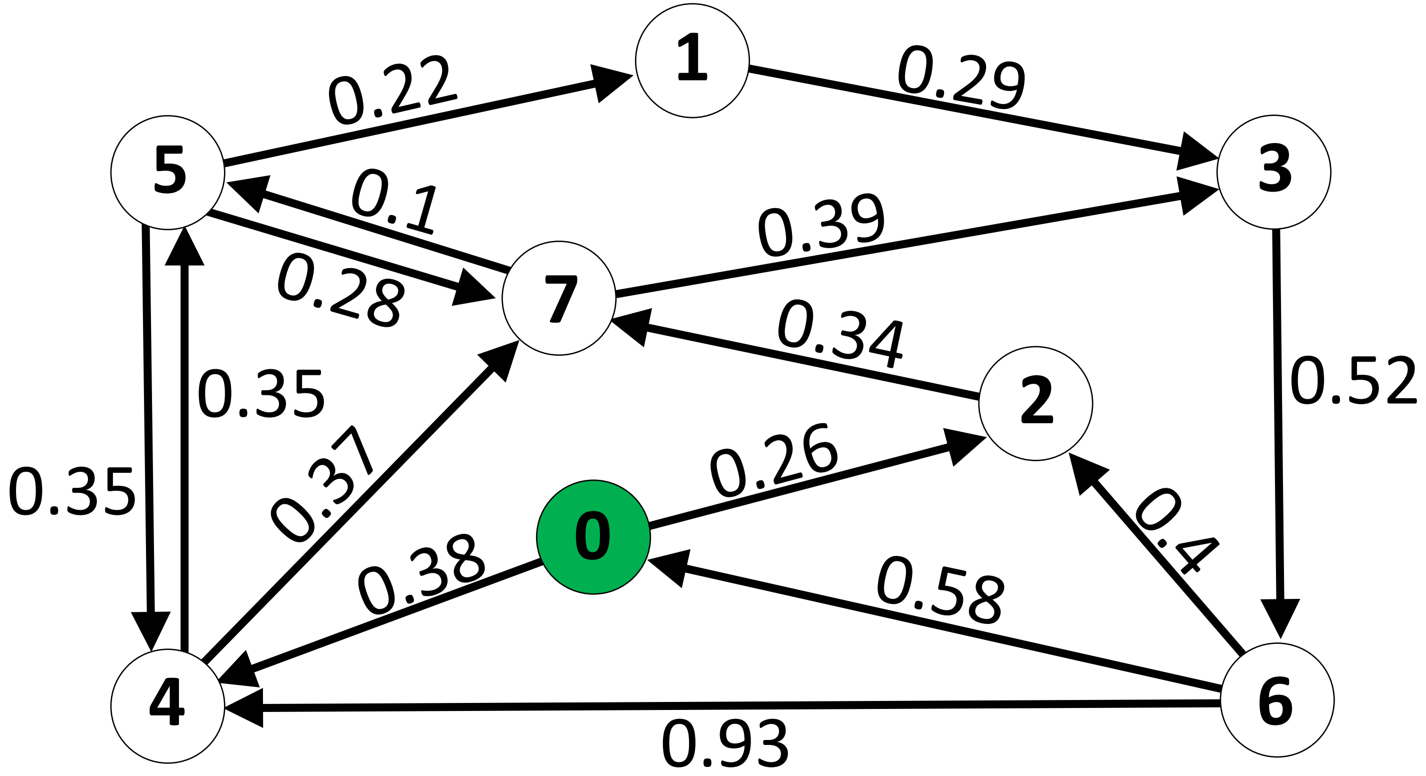


	Distance from 0
0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

	Previous vertex
0	-
1	
2	0
3	7
4	
5	
6	3
7	2

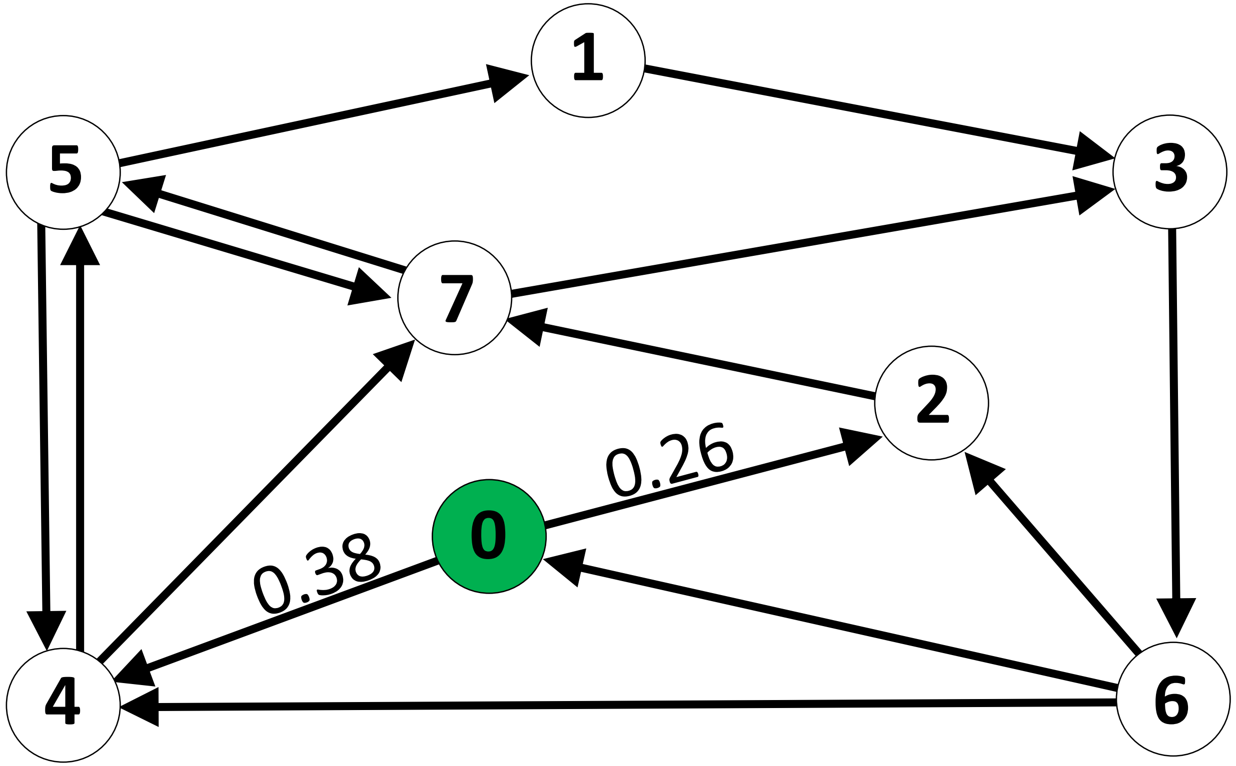
If this is the shortest path from 0 to 6, what can we say about the shortest path from 0 to 3?

Shortest Path



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

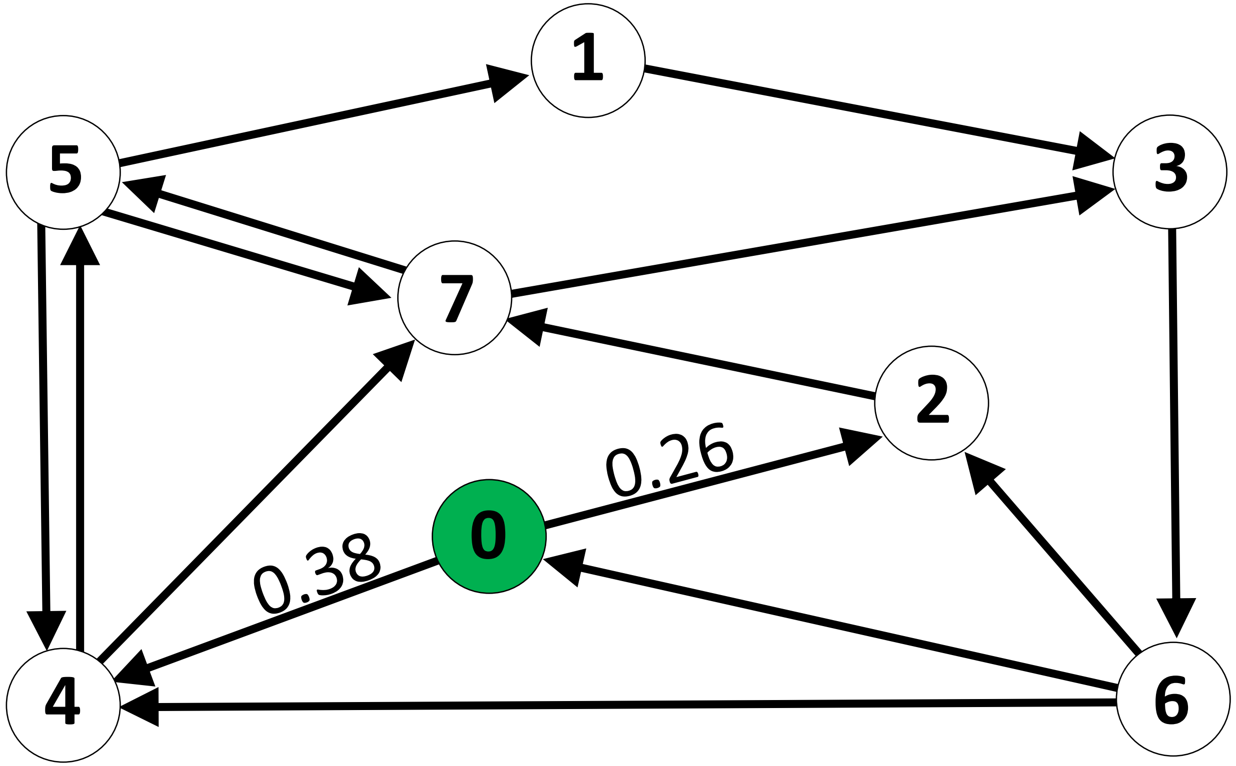
Shortest Path



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

Claim: There cannot possibly be a shorter path from 0 to 2 than the edge from 0 to 2 because…?

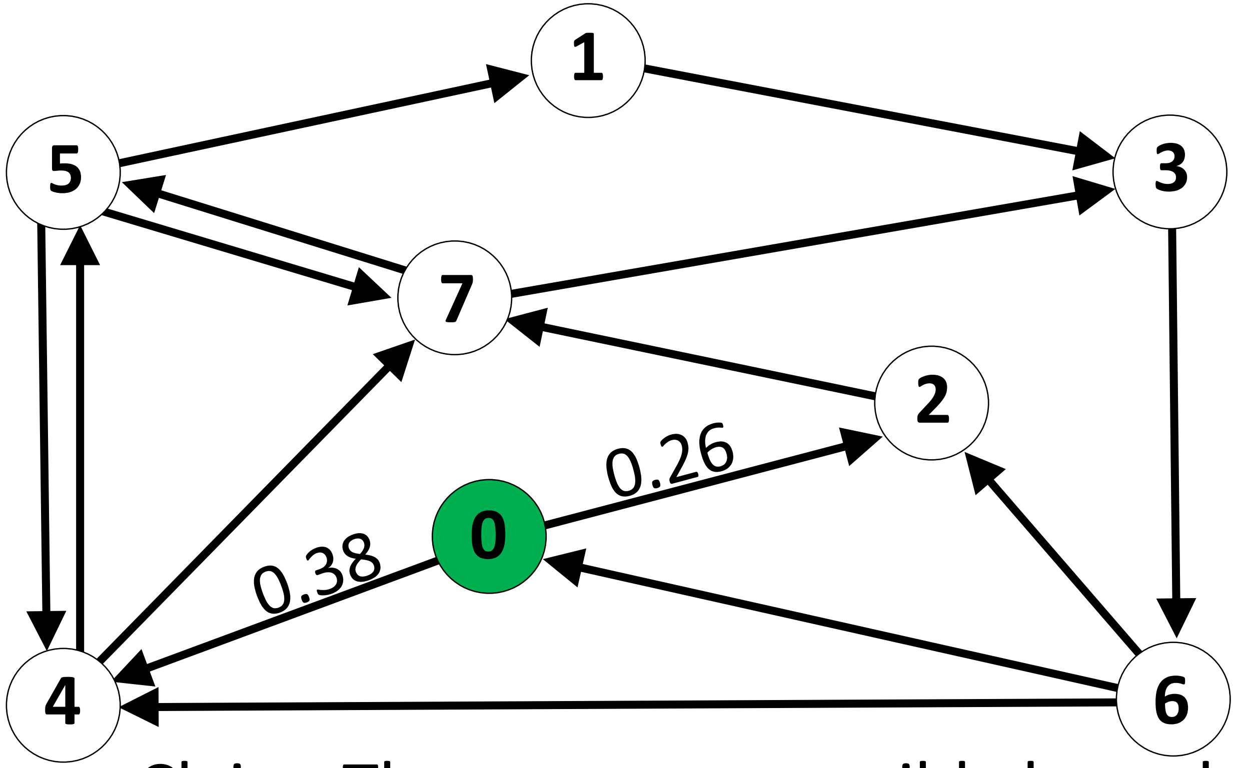
Shortest Path



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

Claim: There cannot possibly be a shorter path from 0 to 2 than the edge from 0 to 2 because non-negative edge weights mean every other path is at least 0.38 or 0.26.

Shortest Path



	Distance from 0
0	0
1	∞
7	∞

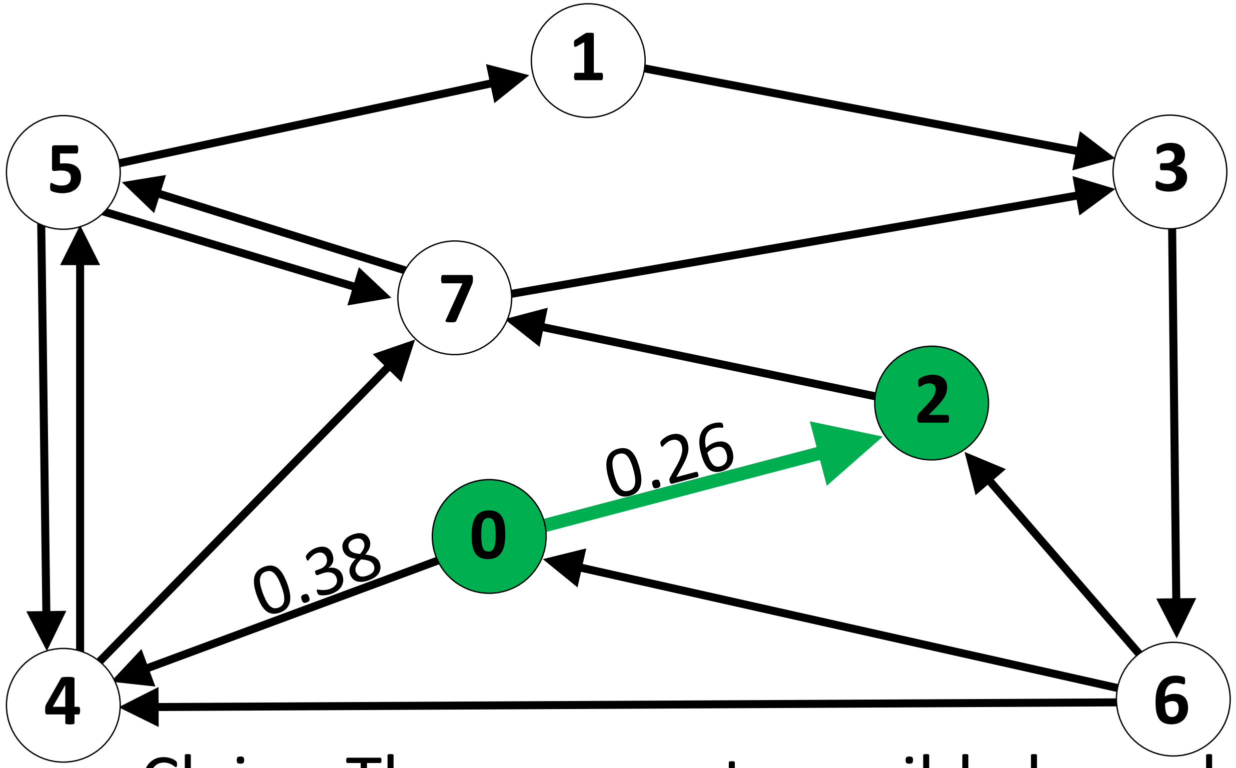
	Previous vertex
0	-
1	
7	

Can we say the same thing about the edge from 0 to 4?

I.e., Could there be a shorter path from 0 to 4 other than the edge from 0 to 4?

Claim: There cannot possibly be a shorter path from 0 to 2 than the edge from 0 to 2 because non-negative edge weights mean every other path is at least 0.38 or 0.26.

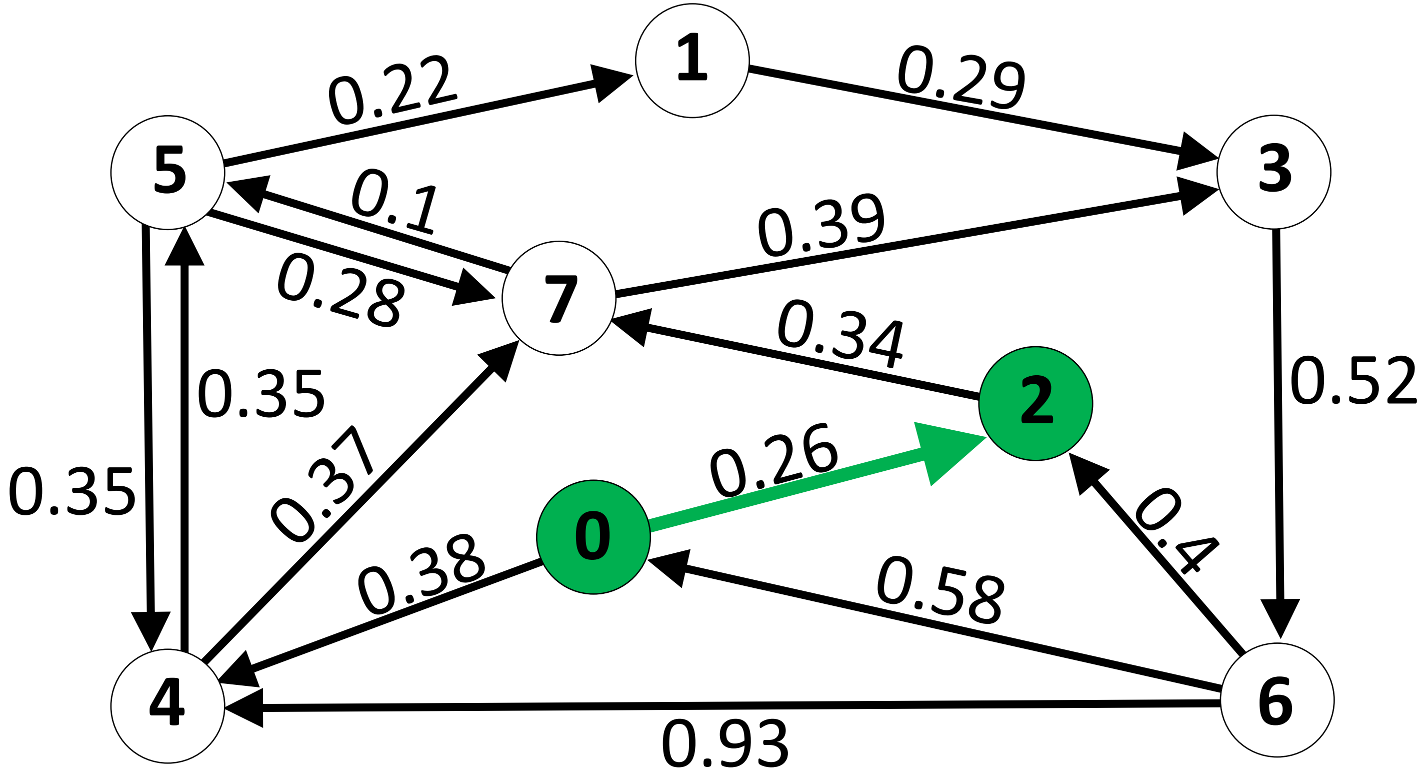
Shortest Path



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

Claim: There cannot possibly be a shorter path from 0 to 2 than the edge from 0 to 2 because non-negative edge weights mean every other path is at least 0.38 or 0.26.

Shortest Path

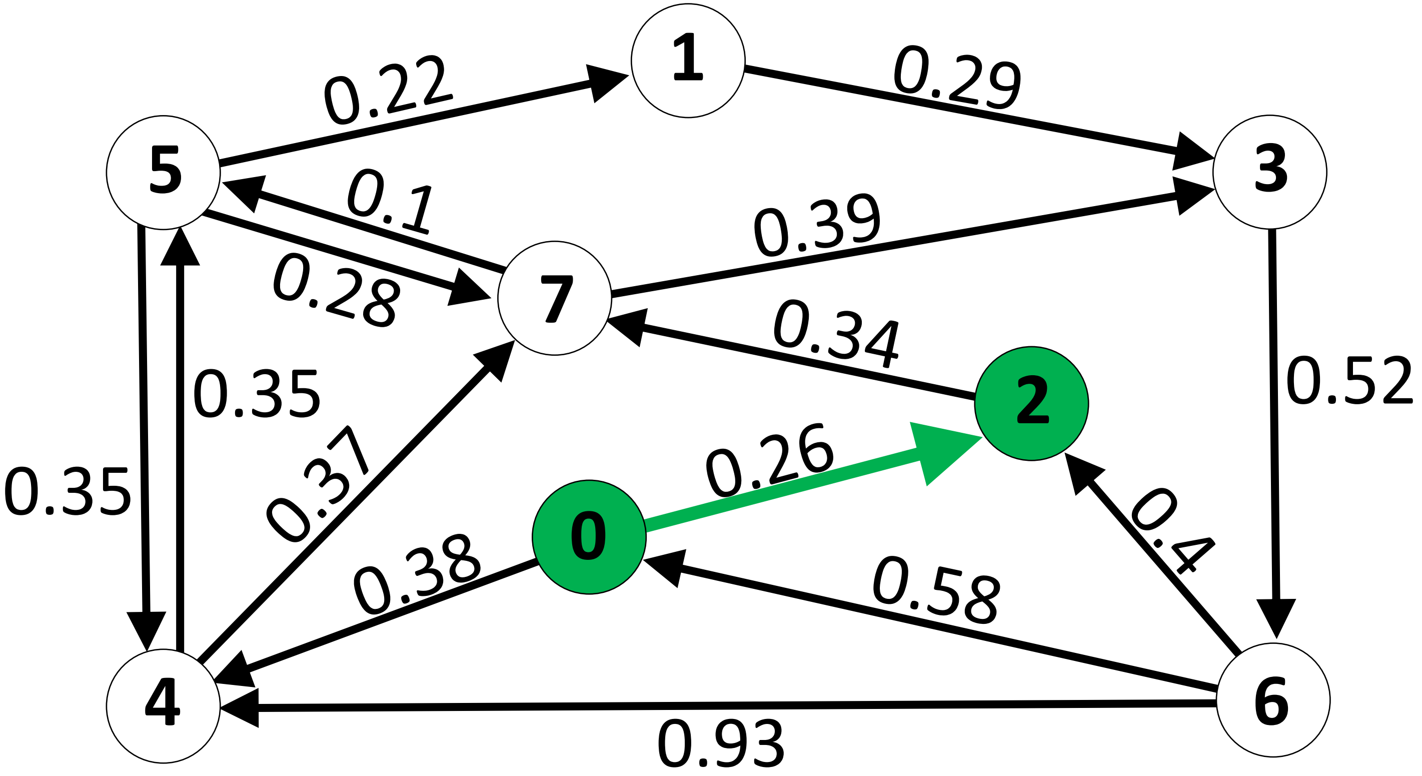


	Distance from 0
0	0
1	∞
2	0.26
3	∞
4	∞
5	∞
6	∞
7	∞

	Previous vertex
0	-
1	
2	0
3	
4	
5	
6	
7	

We need some process for progressing through the graph.

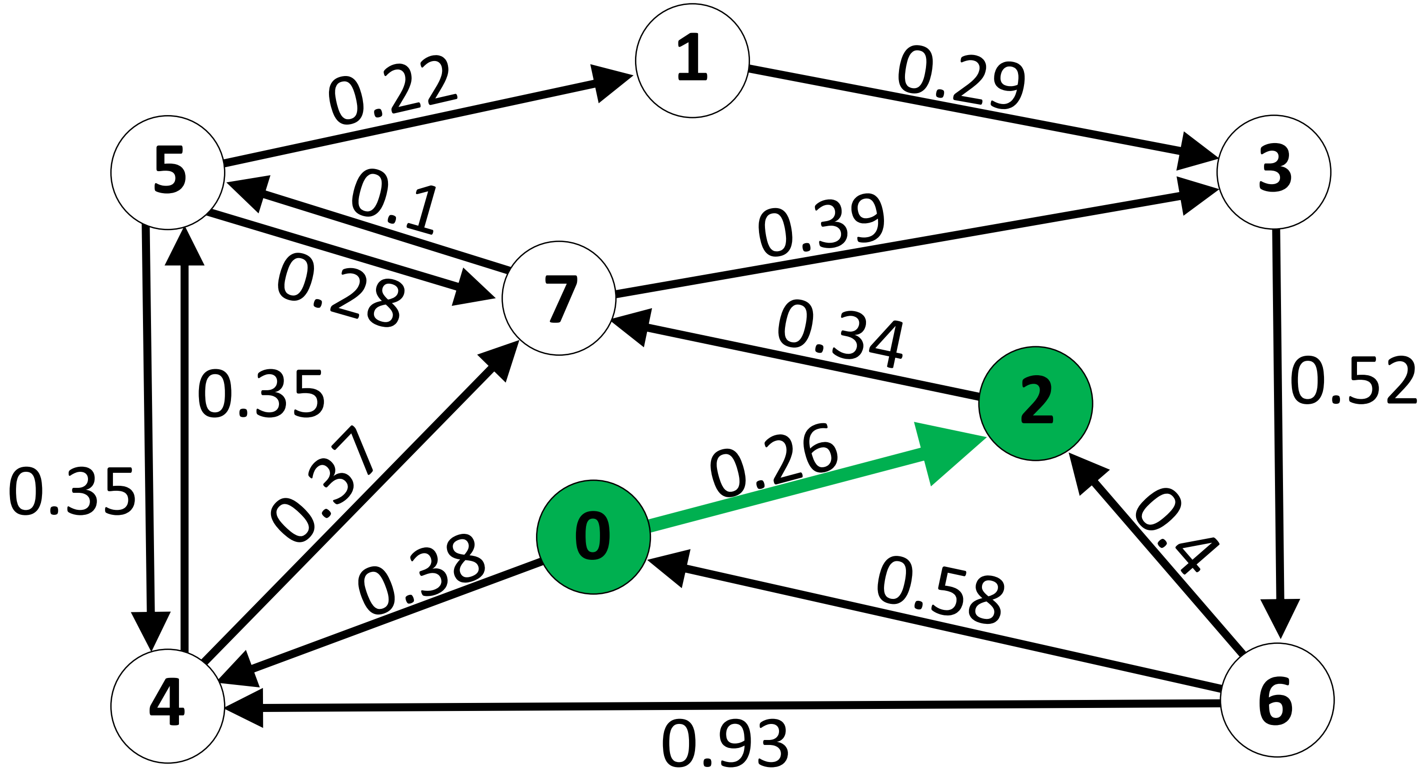
Shortest Path



Vertex	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

We need some process for progressing through the graph.
What if we prioritized neighbors based on path (not edge) distance?

Shortest Path



Distance from 0

0	0
1	∞
2	0.26
3	∞
4	∞
5	∞
6	∞
7	∞

Previous vertex

0	-
1	
2	0
3	
4	
5	
6	
7	

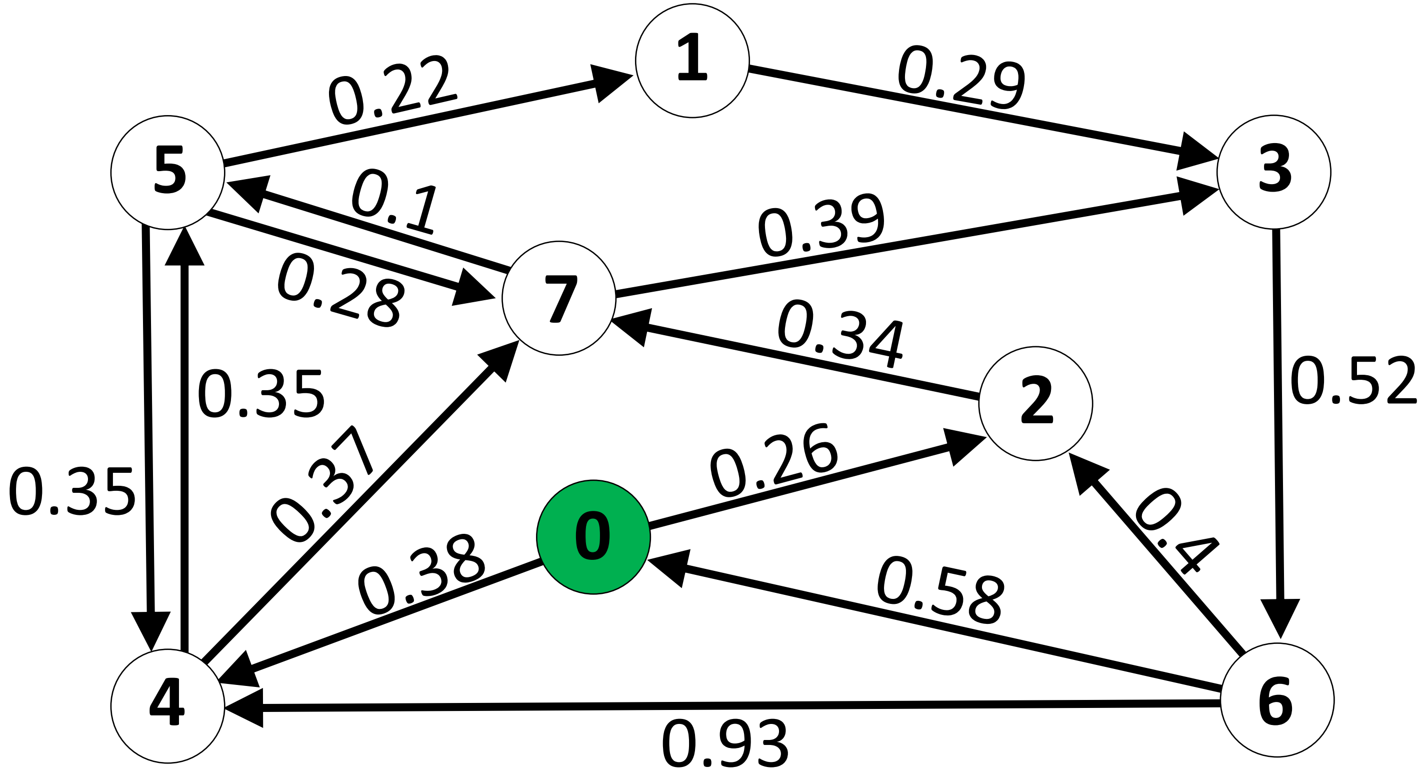
Priority queue

We need some process for progressing through the graph.

What if we prioritized neighbors based on path (not edge) distance?

vertex (distance)

Shortest Path



Distance from 0

0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

Previous vertex

0	-
1	
2	
3	
4	
5	
6	
7	

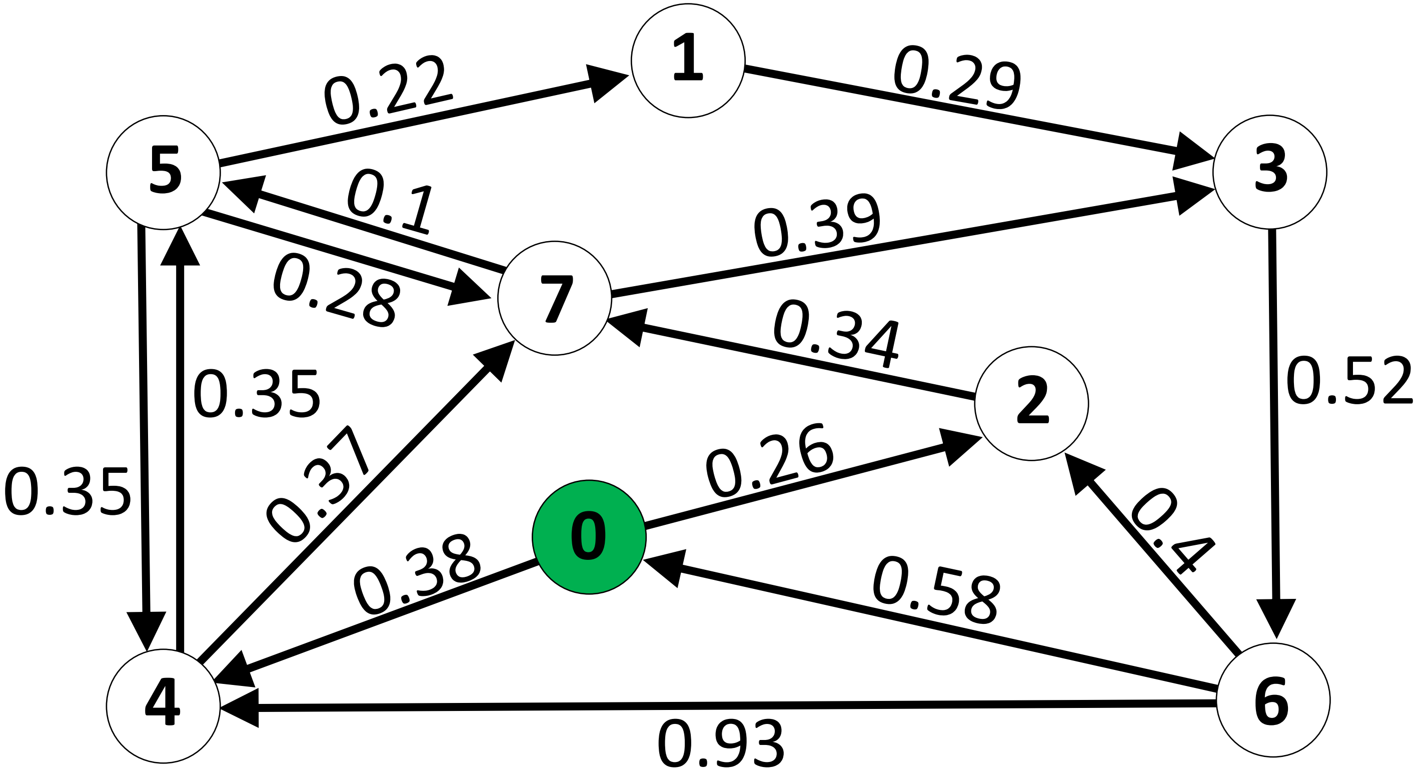
Priority queue

We need some process for progressing through the graph.

What if we prioritized neighbors based on path (not edge) distance?

vertex (distance)

Shortest Path



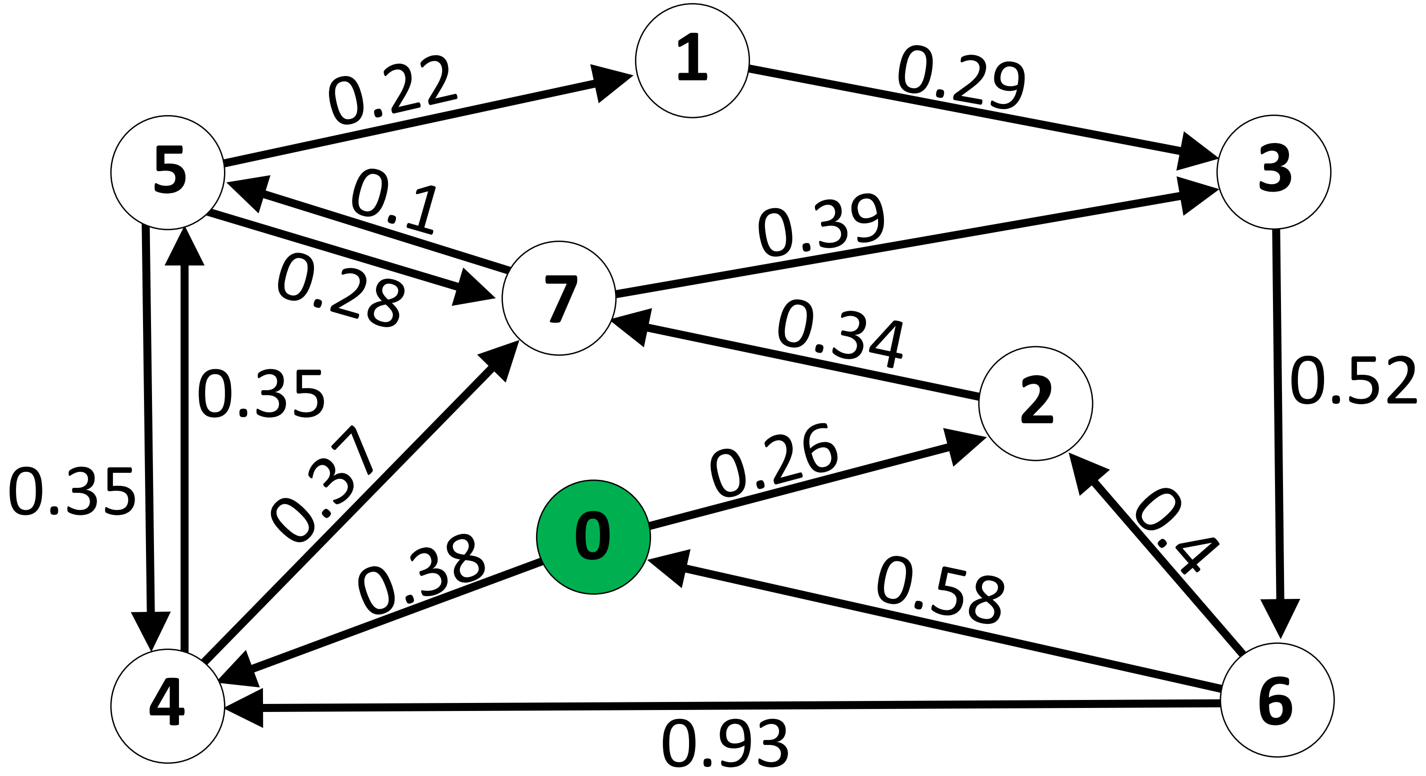
Vertex	Distance from 0	Previous vertex
0	0	-
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

Priority queue

vertex (distance)

What can we reach from connected vertices and at what distance (from 0)?

Shortest Path



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

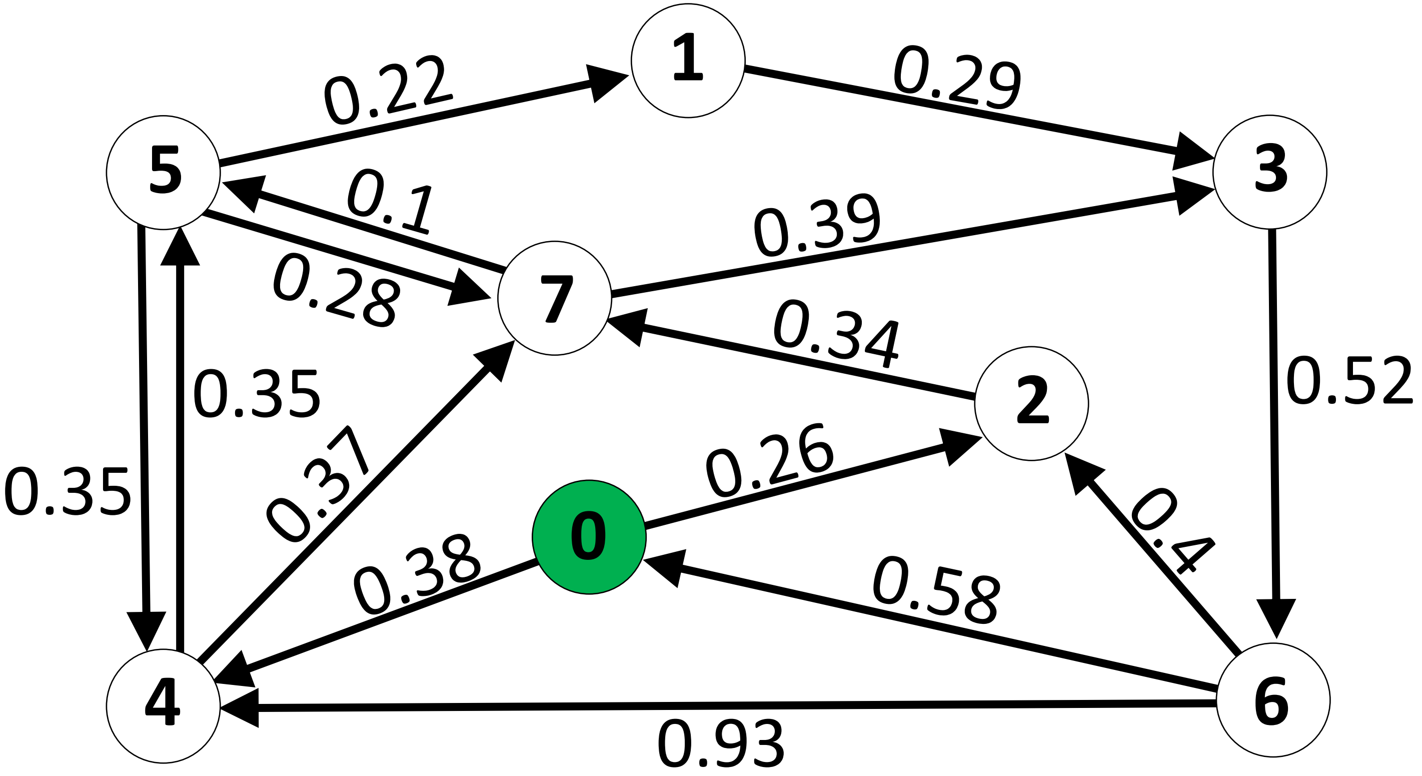
Priority queue

2 (0.26)
4 (0.38)

vertex (distance)

What can we reach from connected vertices and at what distance (from 0)?

Shortest Path



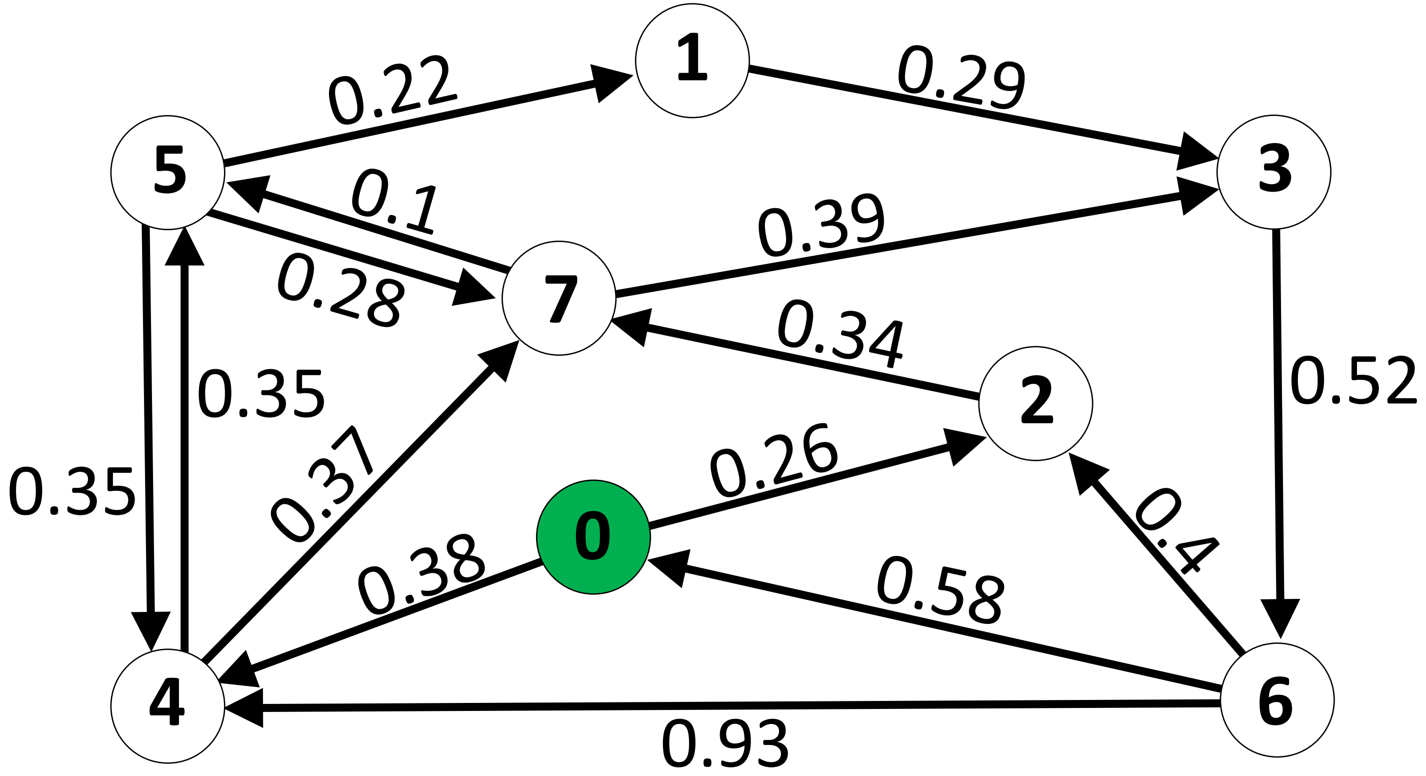
	Distance from 0	Previous vertex
0	0	-
1	∞	
2	∞	0
3	∞	
4	∞	0
5	∞	
6	∞	
7	∞	

Priority queue
2 (0.26)
4 (0.38)

vertex (distance)

What can we reach from connected vertices and at what distance (from 0)?

Shortest Path



Distance from 0	
0	0
1	∞
2	0.26
3	∞
4	0.38
5	∞
6	∞
7	∞

Previous vertex	
0	-
1	
2	0
3	
4	0
5	
6	
7	

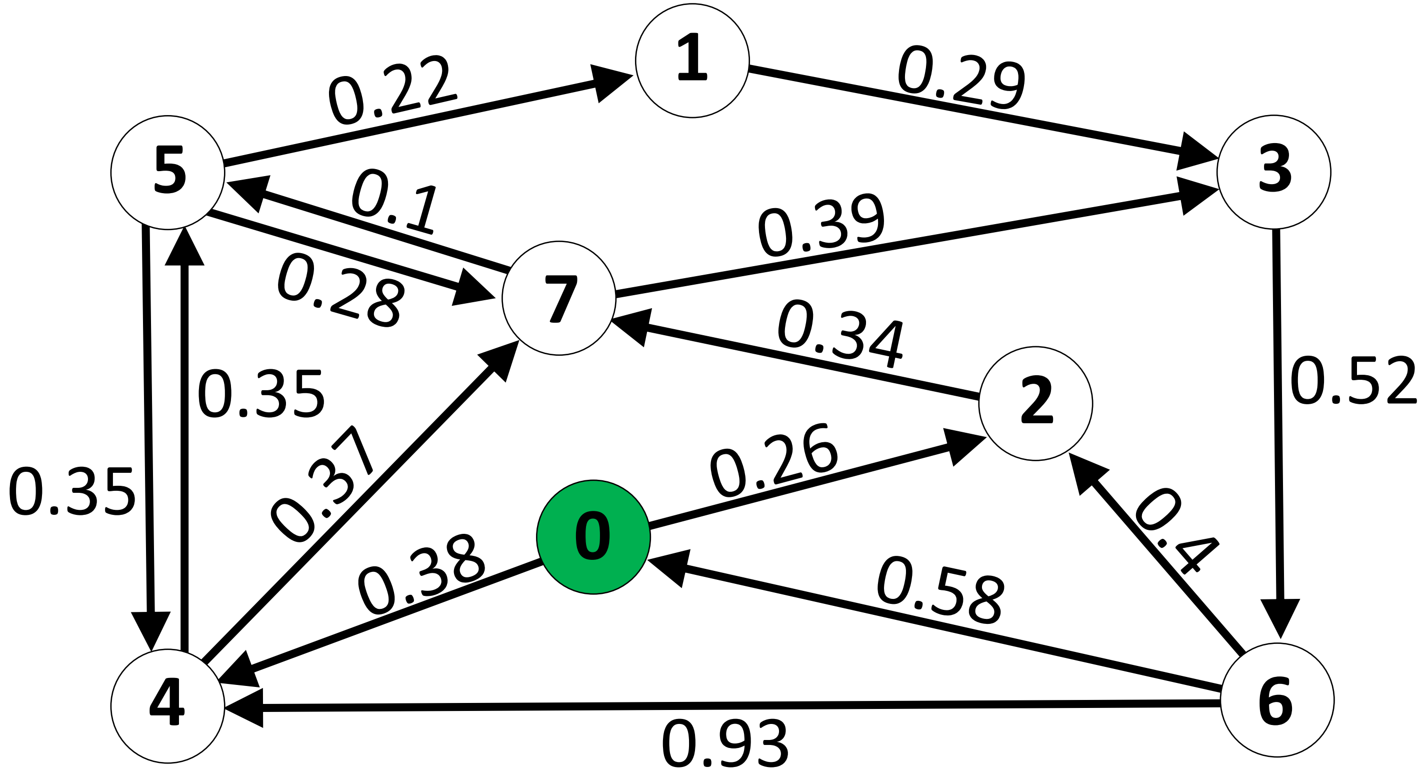
Priority queue
2 (0.26)
4 (0.38)

vertex (distance)

What can we reach from connected vertices and at what distance (from 0)?

Shortest Path

queue top = 2 (0.26)



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	0.38	0
5	∞	
6	∞	
7	∞	

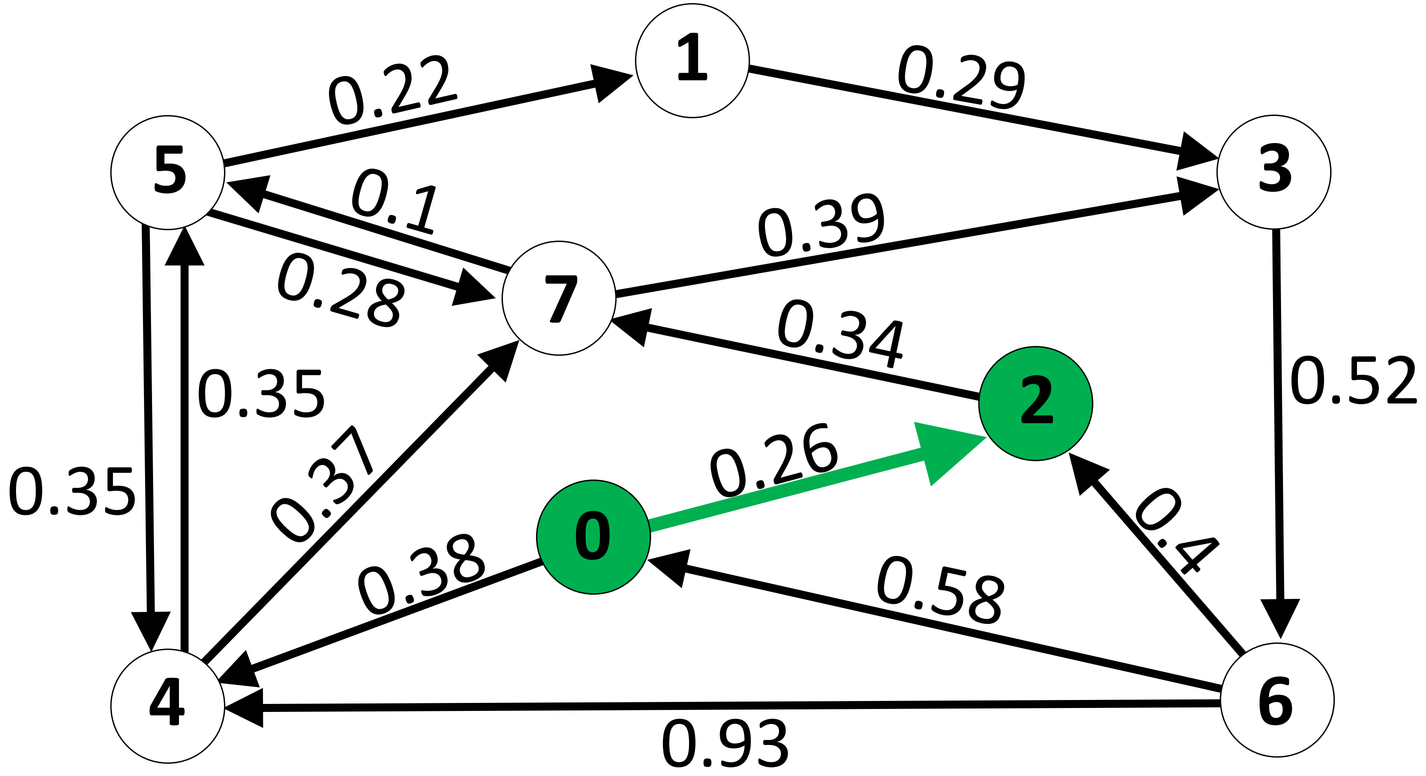
Priority queue
4 (0.38)

vertex (distance)

What can we reach from connected vertices and at what distance (from 0)?

Shortest Path

queue top = 2 (0.26)



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	0.38	0
5	∞	
6	∞	
7	∞	

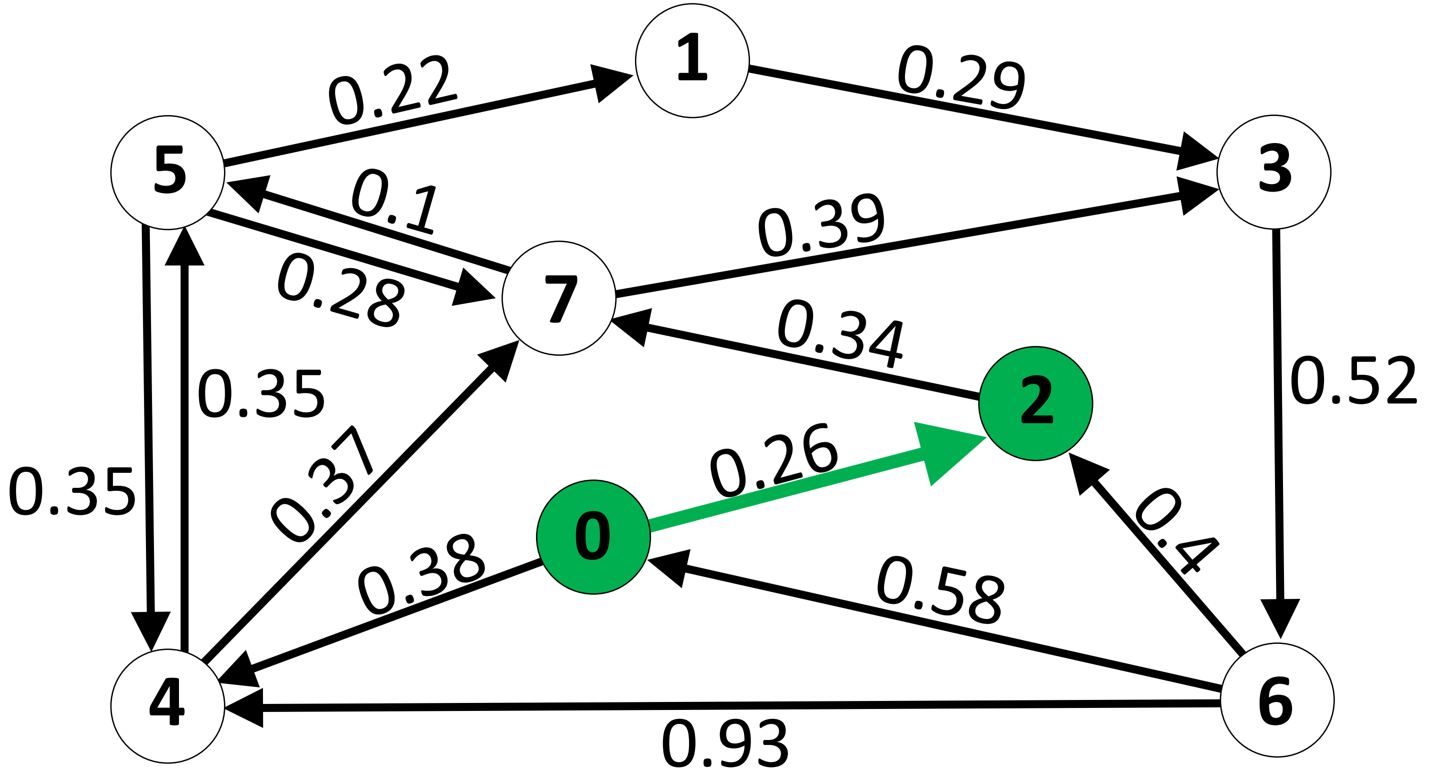
Priority queue
4 (0.38)

vertex (distance)

What can we reach from connected vertices and at what distance (from 0)?

Shortest Path

queue top = 2 (0.26)



Distance from 0	
0	0
1	∞
2	0.26
3	∞
4	0.38
5	∞
6	∞
7	0.60

Previous vertex	
0	-
1	
2	0
3	
4	0
5	
6	
7	2

Priority queue

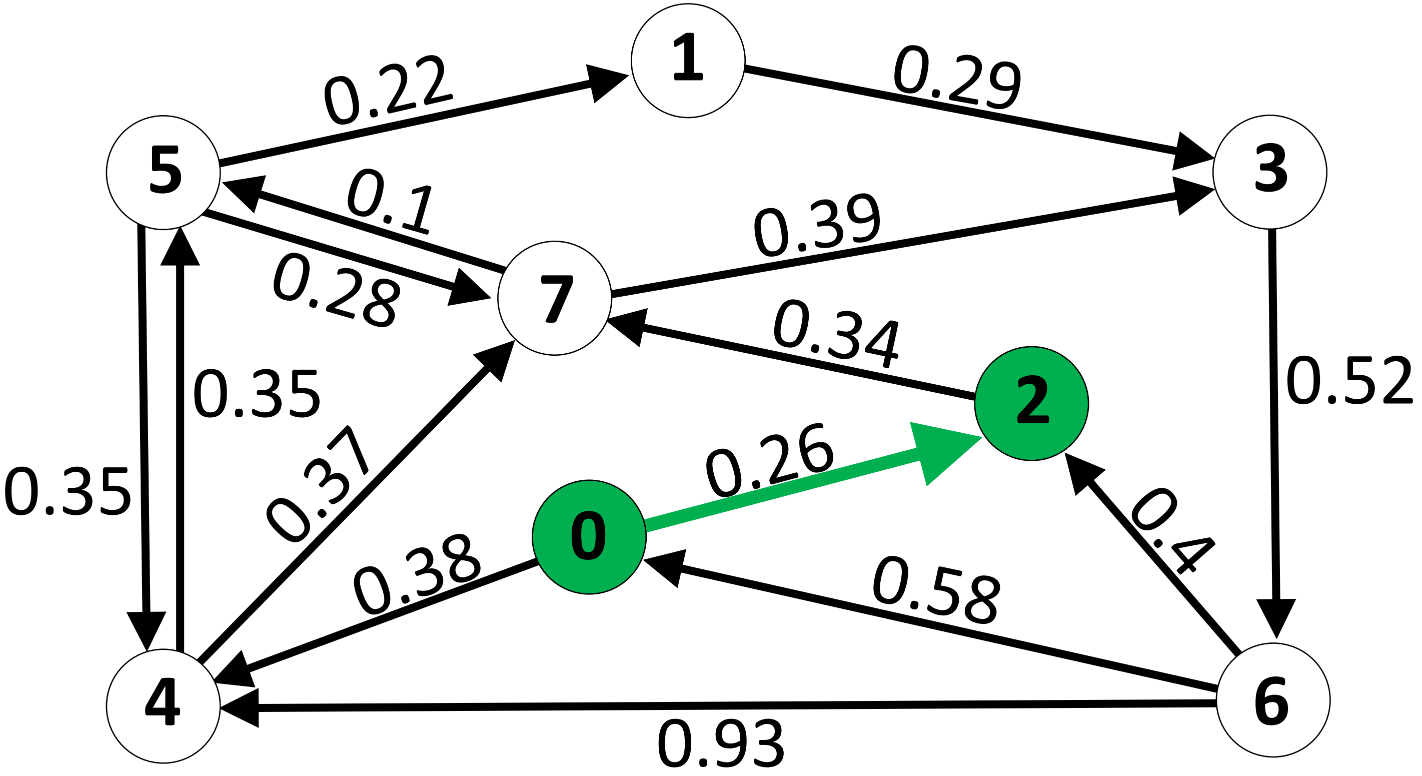
4 (0.38)
7 (0.60)

vertex (distance)

What can we reach from connected vertices and at what distance (from 0)?

Shortest Path

queue top = 4 (0.38)



Repeat.

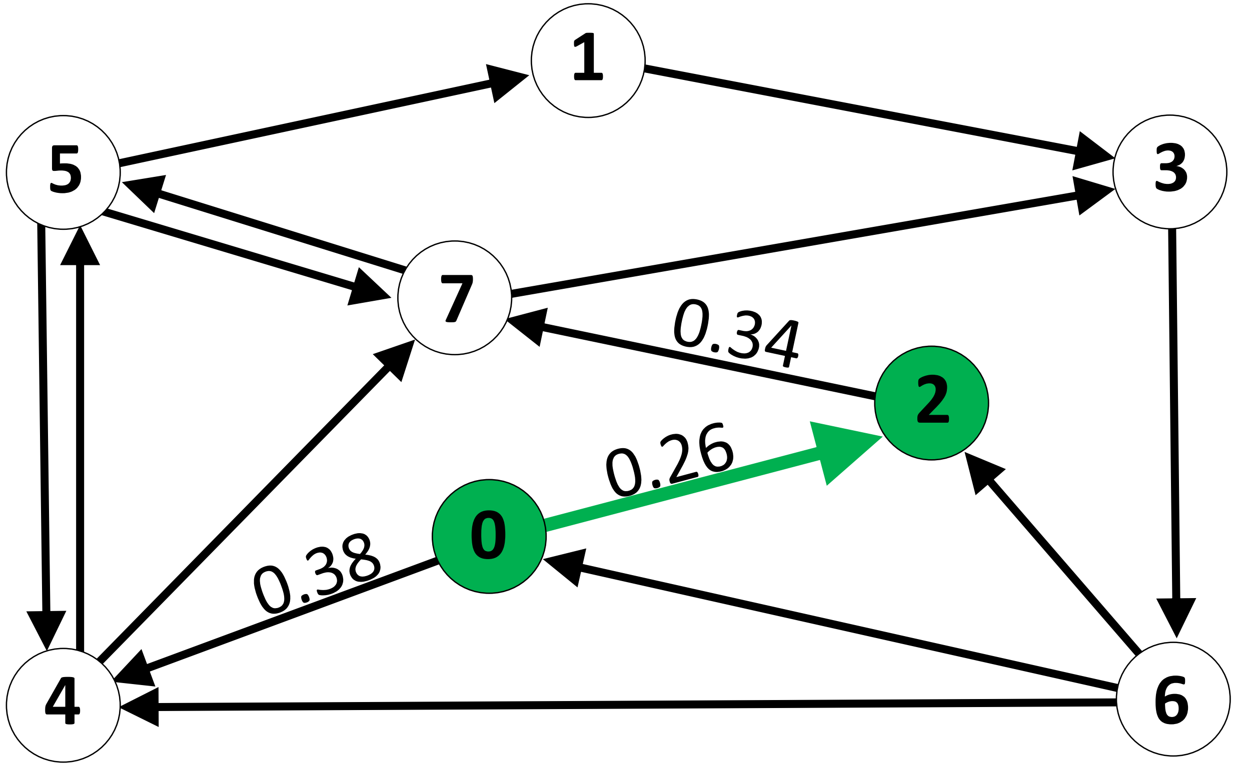
	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	0.38	0
5	∞	
6	∞	
7	0.60	2

Priority queue
7 (0.60)

vertex (distance)

Shortest Path

queue top	= 4 (0.38)



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	0.38	0
5	∞	
6	∞	
7	0.60	2

Priority queue

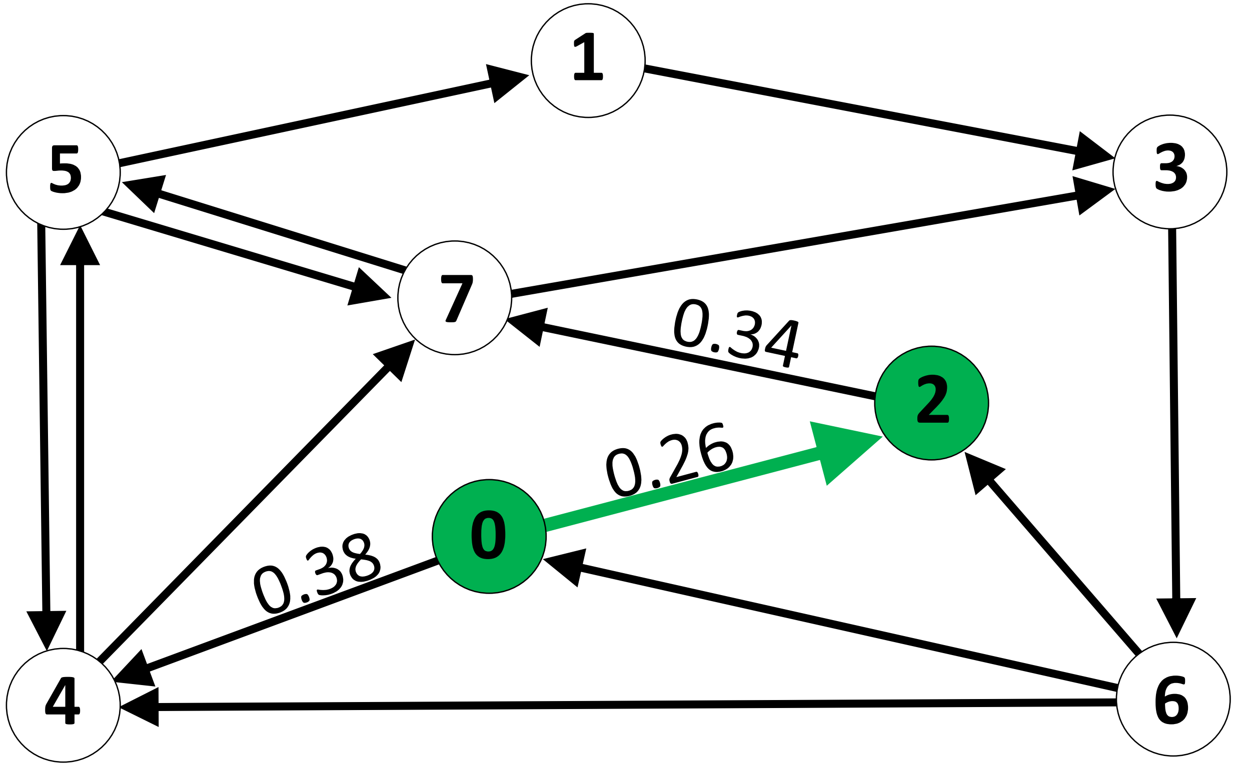
7 (0.60)

vertex (distance)

What can we say about the shortest path from 0 to 4?

Shortest Path

queue top = 4 (0.38)



	Distance from 0
0	0
1	∞
2	0.26
3	∞
4	0.38
5	∞
6	∞
7	0.60

	Previous vertex
0	-
1	
2	0
3	
4	0
5	
6	
7	2

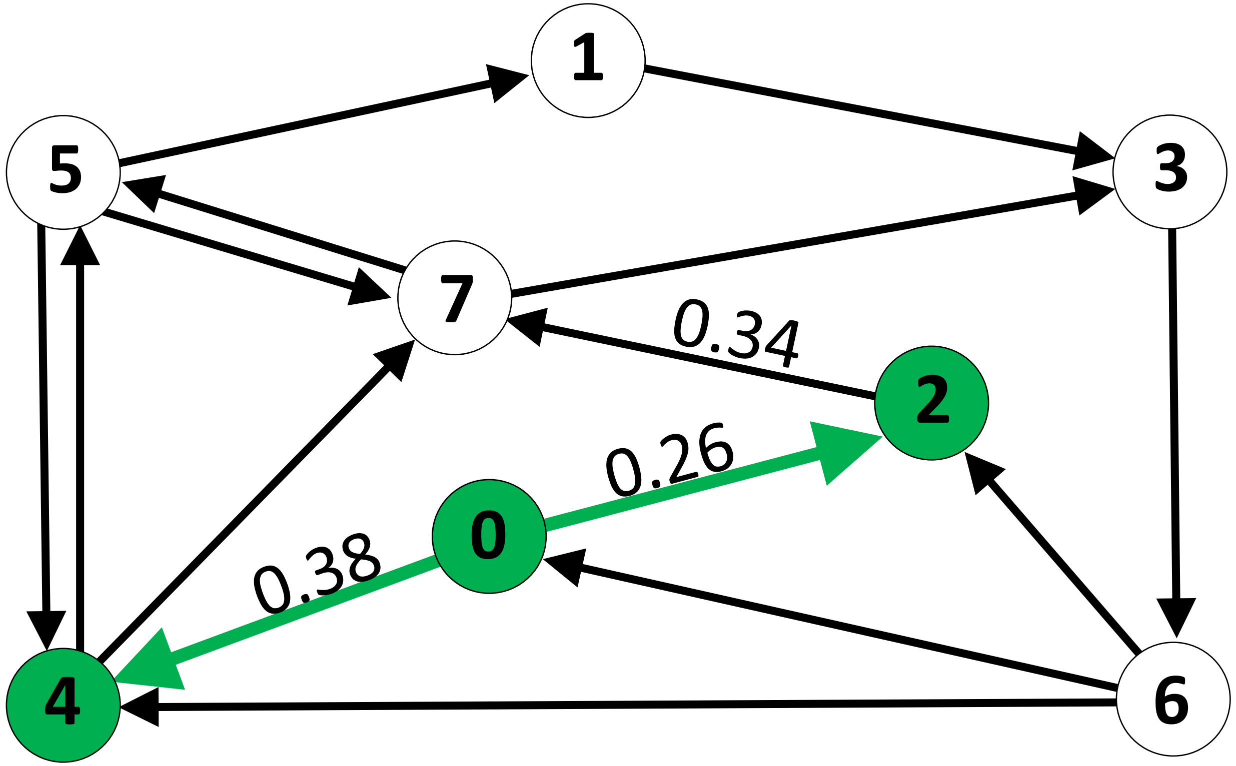
Priority queue
7 (0.60)

vertex (distance)

The 0 to 4 edge has to be the shortest path between 0 and 4, since any other path would go from 0 -> 2 -> 7 -> ? at cost at least 0.26 + 0.34 = 0.6 > 0.38

Shortest Path

queue top = 4 (0.38)



Vertex	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	0.38	0
5	∞	
6	∞	
7	0.60	2

Priority queue

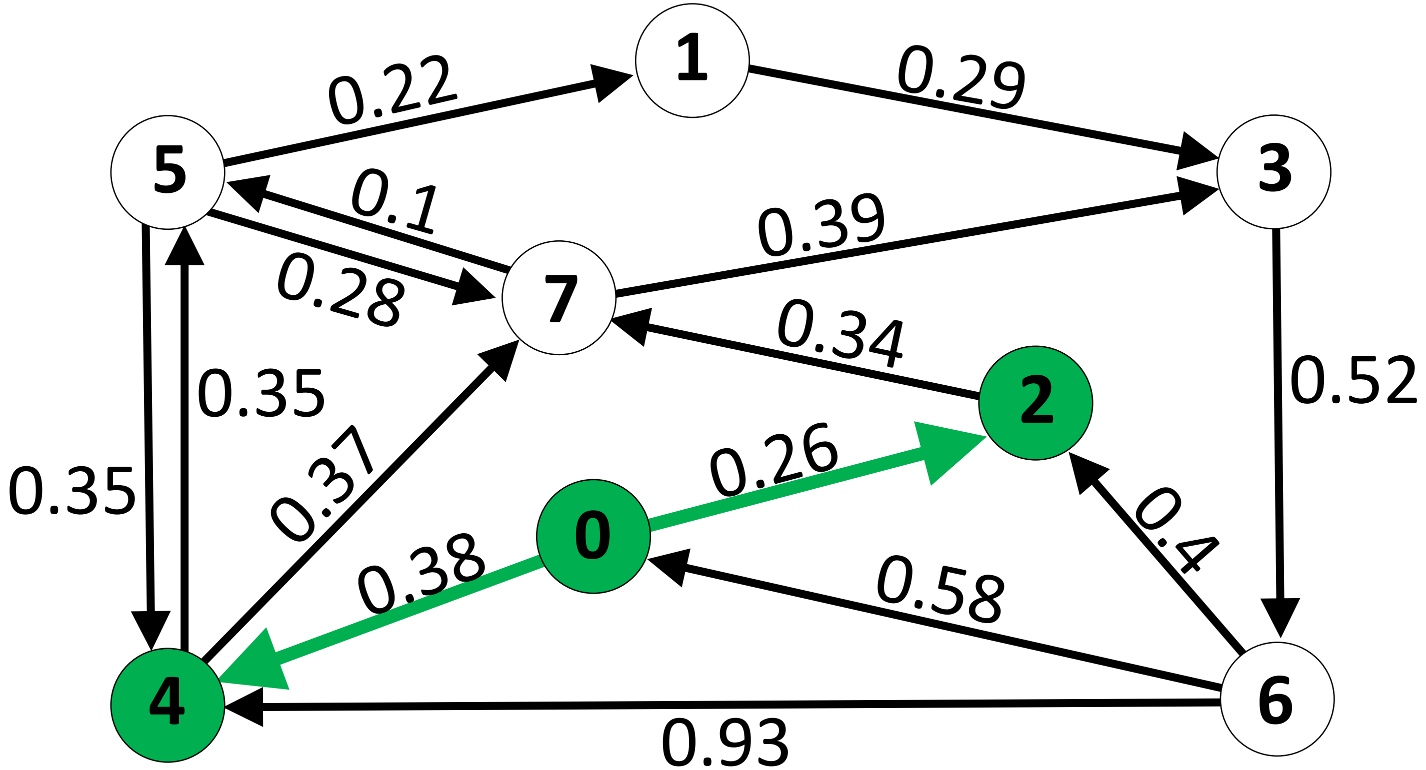
7 (0.60)

vertex (distance)

The 0 to 4 edge has to be the shortest path between 0 and 4, since any other path would go from 0 -> 2 -> 7 -> ? at cost at least 0.26 + 0.34 = 0.6 > 0.38

Shortest Path

queue top = 4 (0.38)



Add neighbors to queue/previous.

	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	0.38	0
5	∞	
6	∞	
7	0.60	2

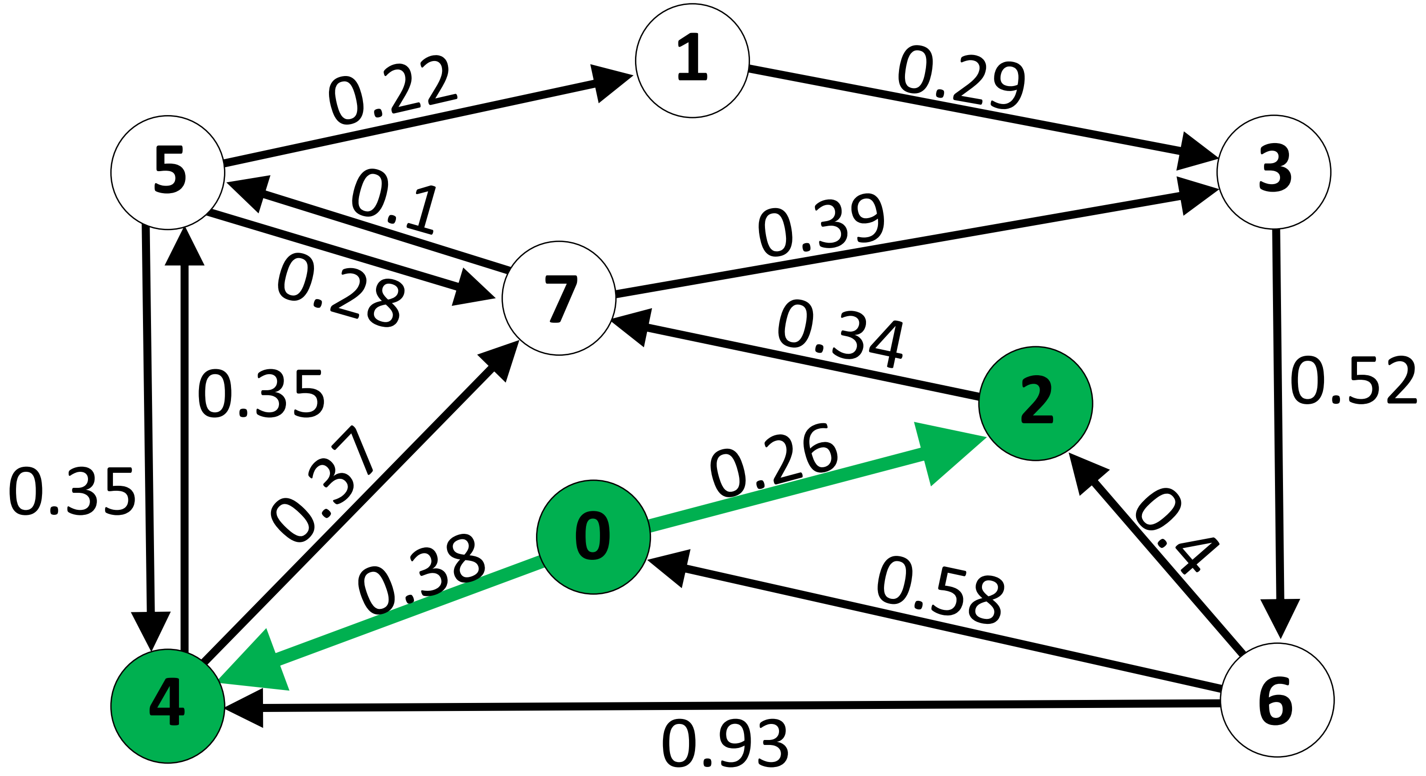
Priority queue

7 (0.60)

vertex (distance)

Shortest Path

queue top = 4 (0.38)



Distance from 0	
0	0
1	∞
2	0.26
3	∞
4	0.38
5	0.73
6	∞
7	0.60

Previous vertex	
0	-
1	
2	0
3	
4	0
5	4
6	
7	2

Priority queue

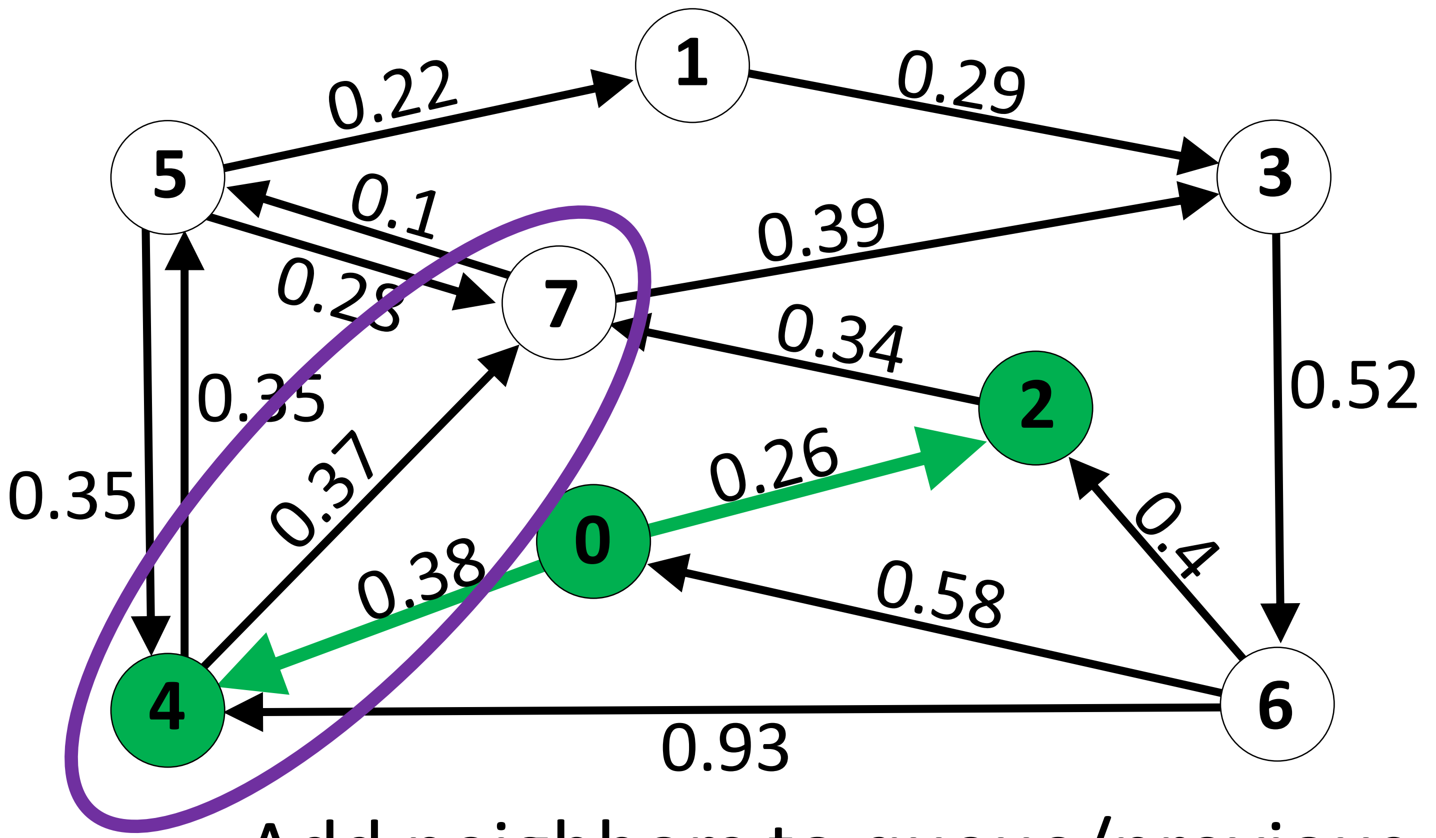
7 (0.60)
5 (0.73)

vertex (distance)

Add neighbors to queue/previous.

Shortest Path

queue top	= 4 (0.38)



Add neighbors to queue/previous.

We have another route to 7!

Vertex	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	0.38	0
5	0.73	4
6	∞	
7	0.60	2

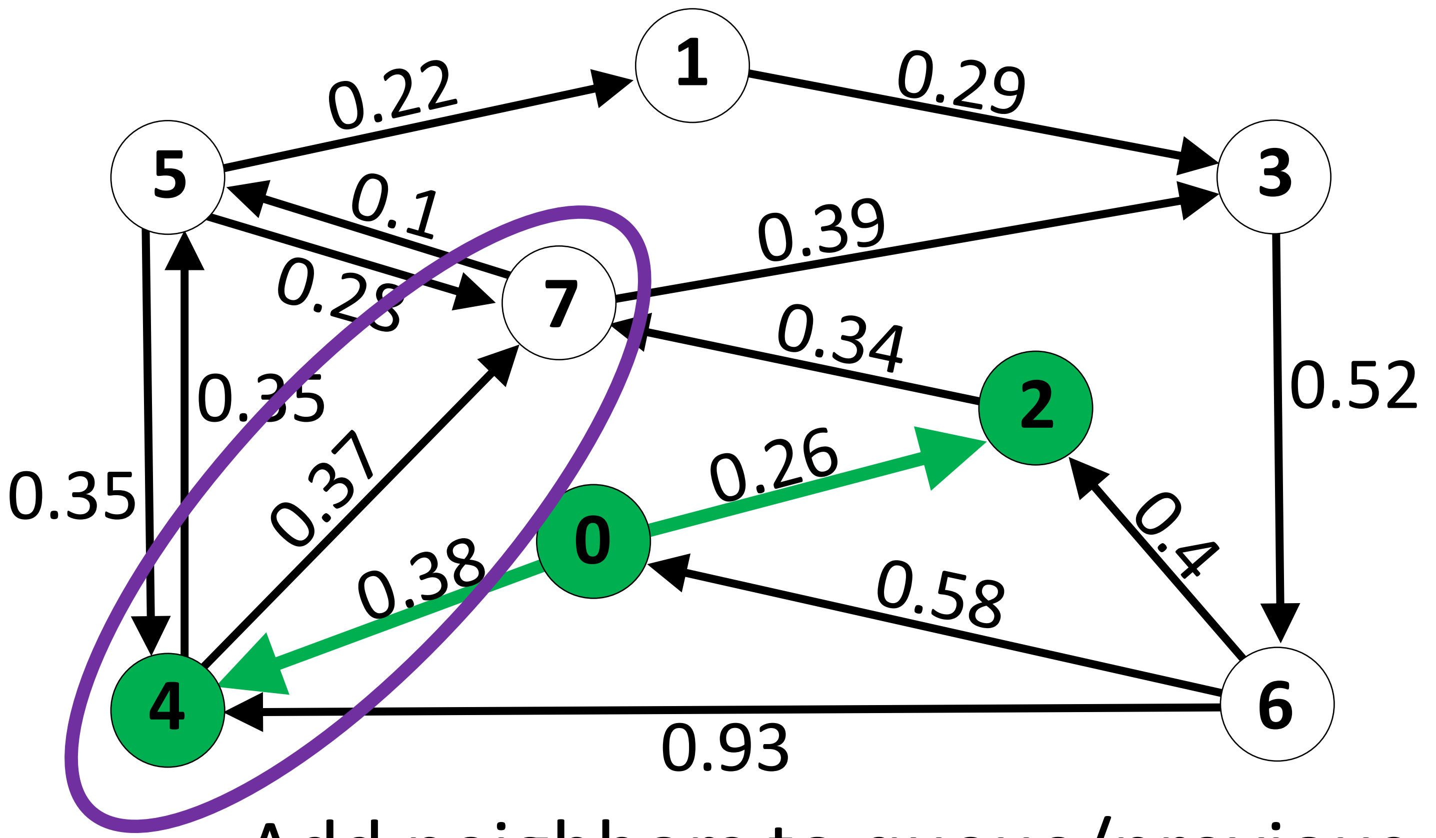
Priority queue

7 (0.60)
5 (0.73)

vertex (distance)

Shortest Path

queue top	= 4 (0.38)



Add neighbors to queue/previous.

	Distance from 0
0	0
1	∞
2	0.26
3	∞
4	0.38
5	0.73
6	∞
7	0.60

	Previous vertex
0	-
1	
2	0
3	
4	0
5	4
6	
7	2

Priority queue

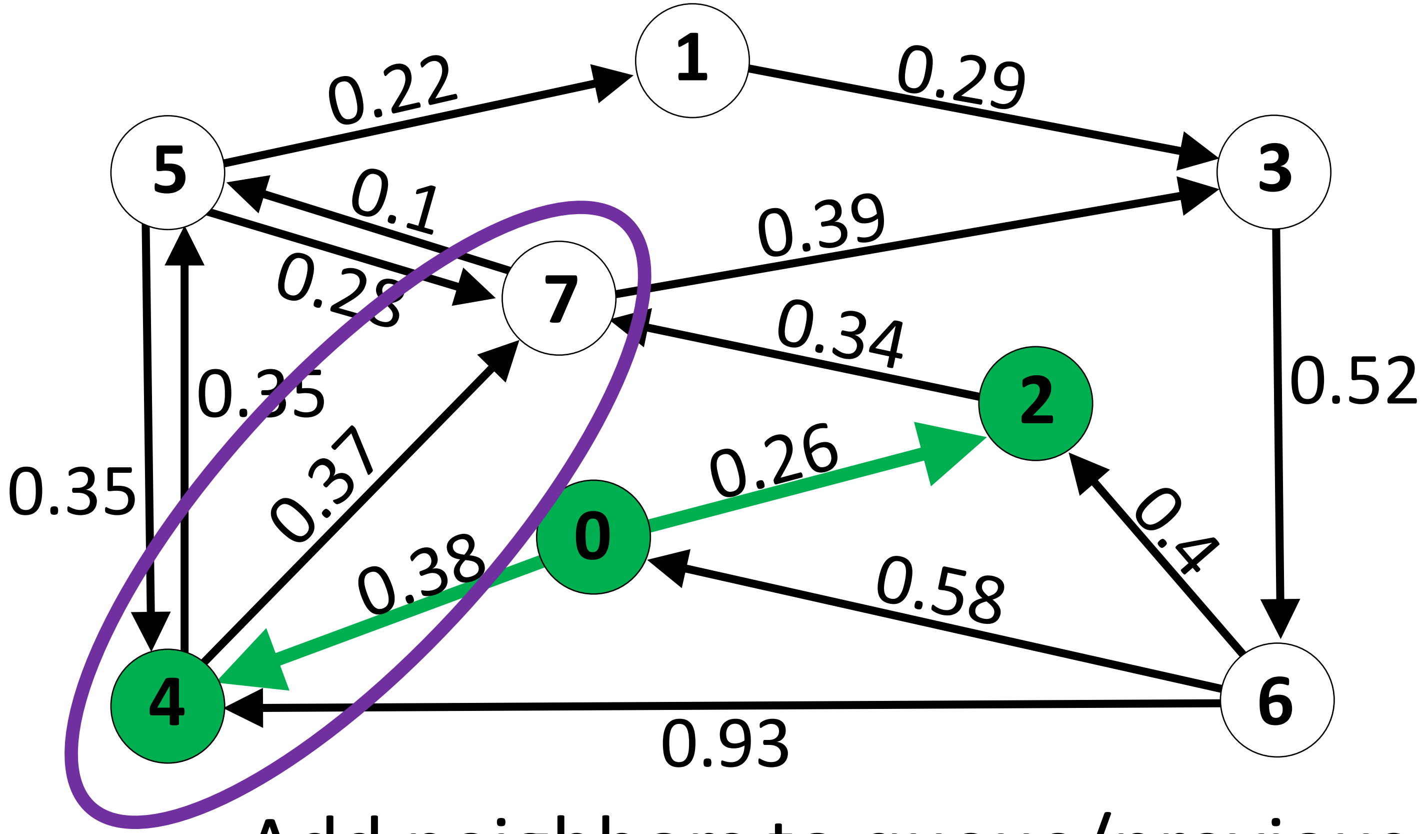
7 (0.60)
5 (0.73)

vertex (distance)

We have another route to 7! Check to see if it is shorter!

Shortest Path

queue top = 4 (0.38)



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	∞	
4	0.38	0
5	0.73	4
6	∞	
7	0.60	2

Priority queue

7 (0.60)
5 (0.73)

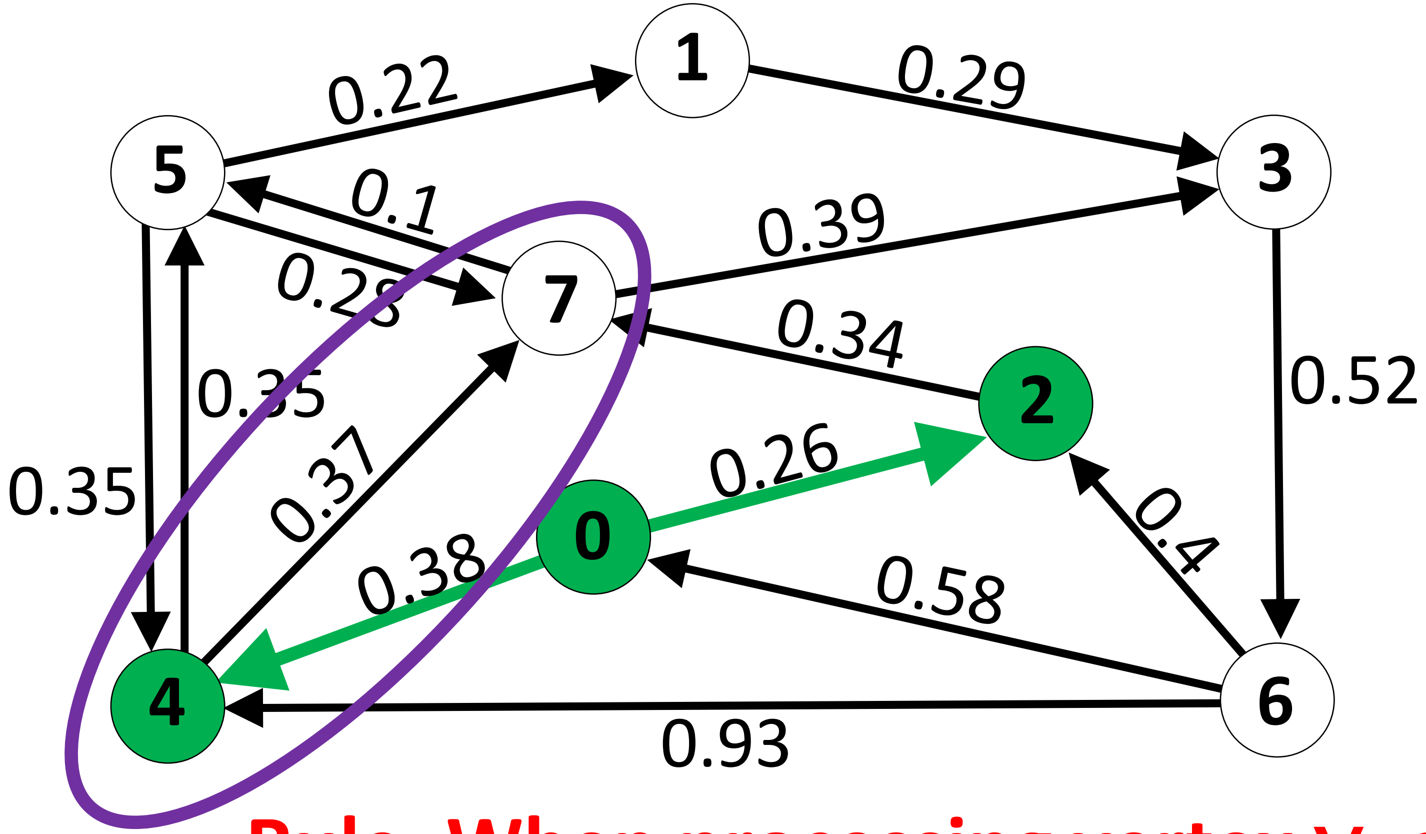
vertex (distance)

Add neighbors to queue/previous.

We have another route to 7! **Check to see if it is shorter! It's not (0.38 + 0.37 = 0.75 > 0.60).**

Shortest Path

queue top = 4 (0.38)



	Distance from 0
0	0
1	∞
2	0.26
3	∞
4	0.38
5	0.73
6	∞
7	0.60

	Previous vertex
0	-
1	
2	0
3	
4	0
5	4
6	
7	2

Priority queue
7 (0.60)
5 (0.73)

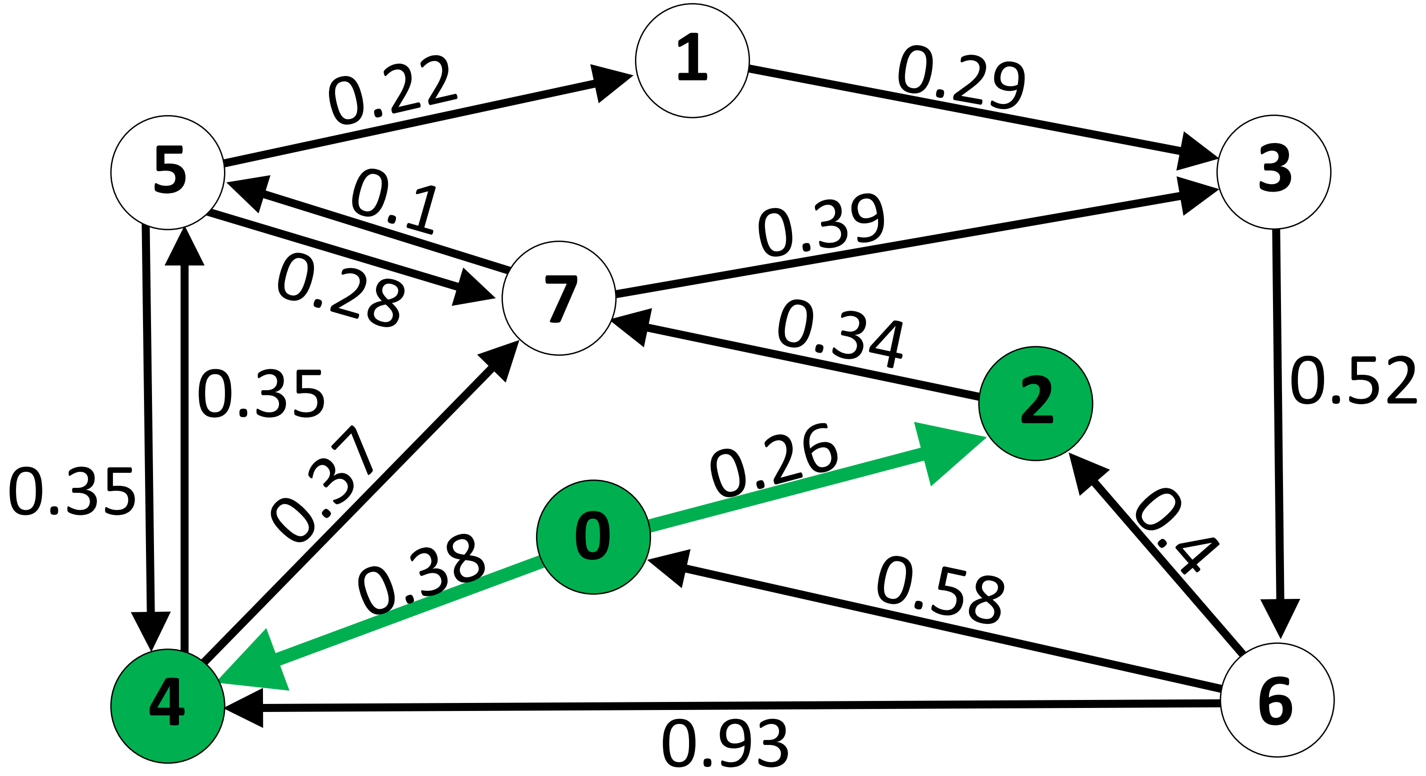
vertex (distance)

Rule: **When processing vertex v, only add/modify queue for neighbor u if and only if:**

`distance[v] + weight(v, u) < distance[u]`

Shortest Path

queue top =



Repeat.

	Distance from 0
0	0
1	∞
2	0.26
3	∞
4	0.38
5	0.73
6	∞
7	0.60

	Previous vertex
0	-
1	
2	0
3	
4	0
5	4
6	
7	2

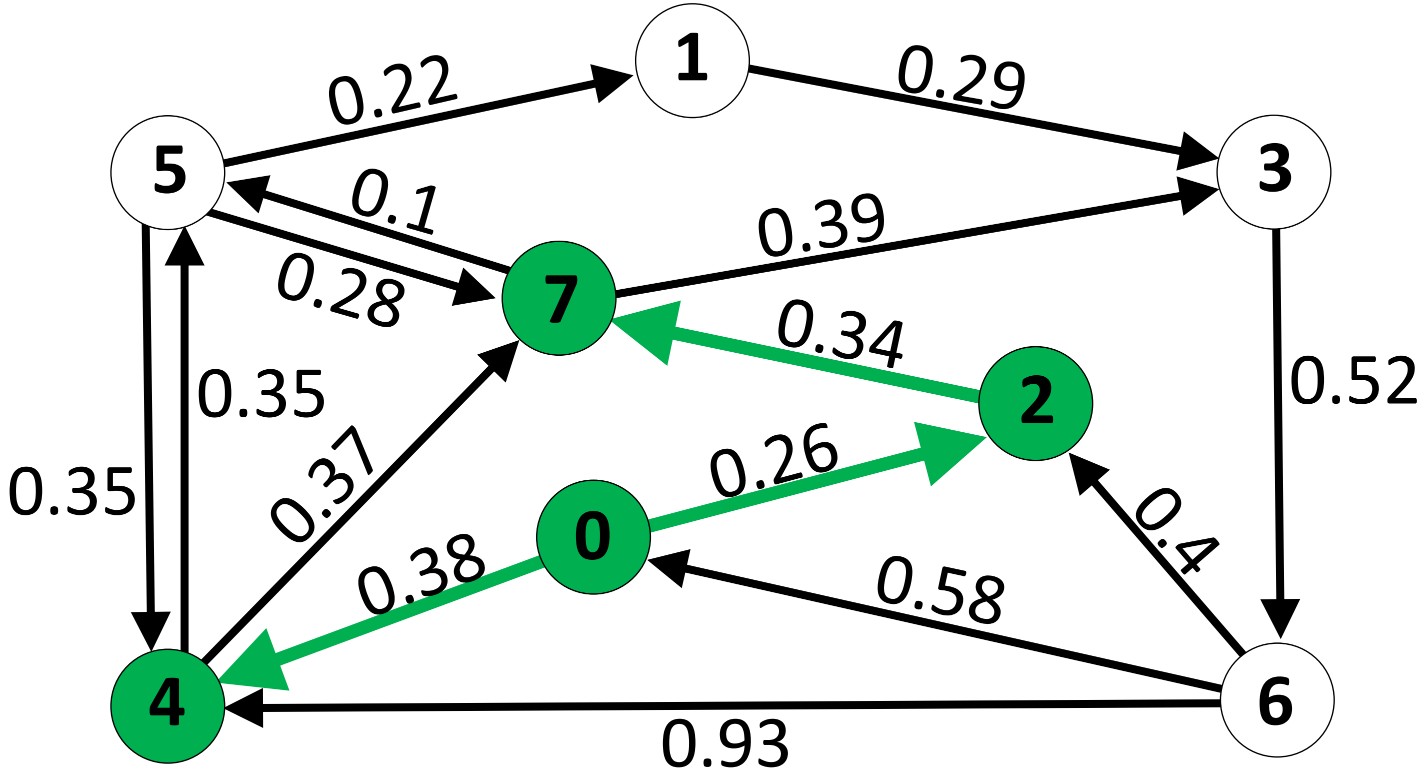
Priority queue

7 (0.60)
5 (0.73)

vertex (distance)

Shortest Path

queue top = 7 (0.60)



Repeat.

	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	0.99	7
4	0.38	0
5	0.73	4
6	∞	
7	0.60	2

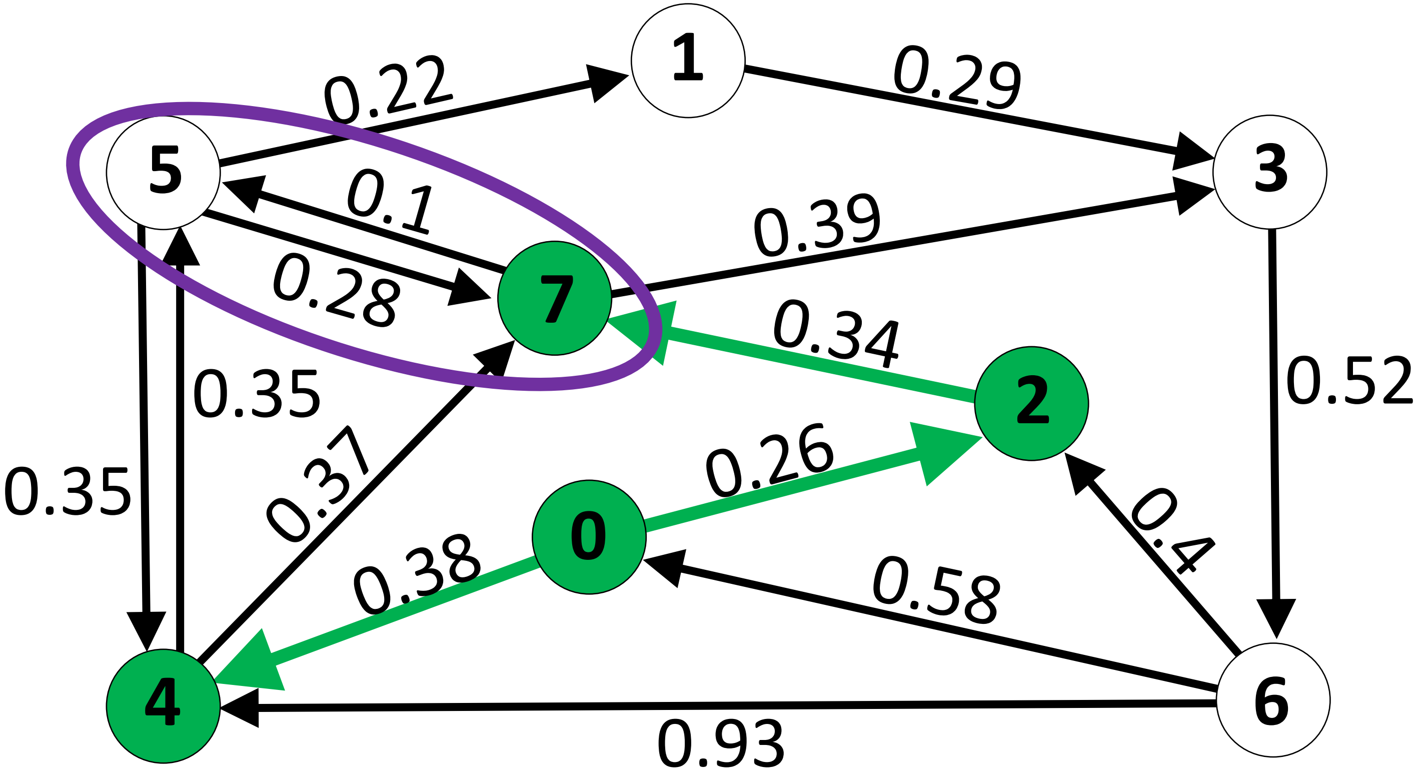
Priority queue

5 (0.73)
3 (0.99)

vertex (distance)

Shortest Path

queue top	= 7 (0.60)



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	0.99	7
4	0.38	0
5	0.73	4
6	∞	
7	0.60	2

Priority queue
5 (0.73)
3 (0.99)

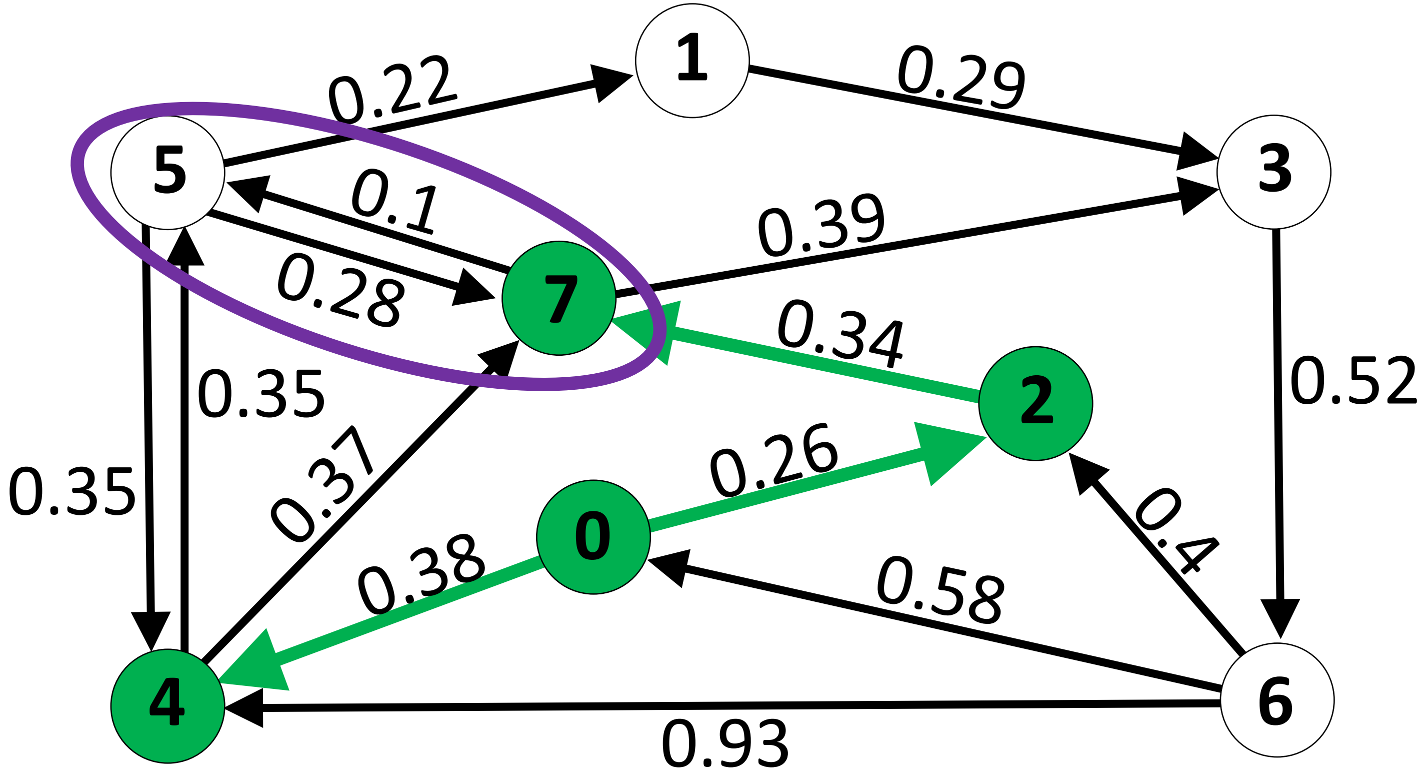
vertex (distance)

Repeat.

We have another route to 5, and at cost 0.7 < 0.73.

Shortest Path

queue top = 7 (0.60)



	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	0.99	7
4	0.38	0
5	0.73	4
6	∞	
7	0.60	2

Priority queue

5 (0.73)
3 (0.99)

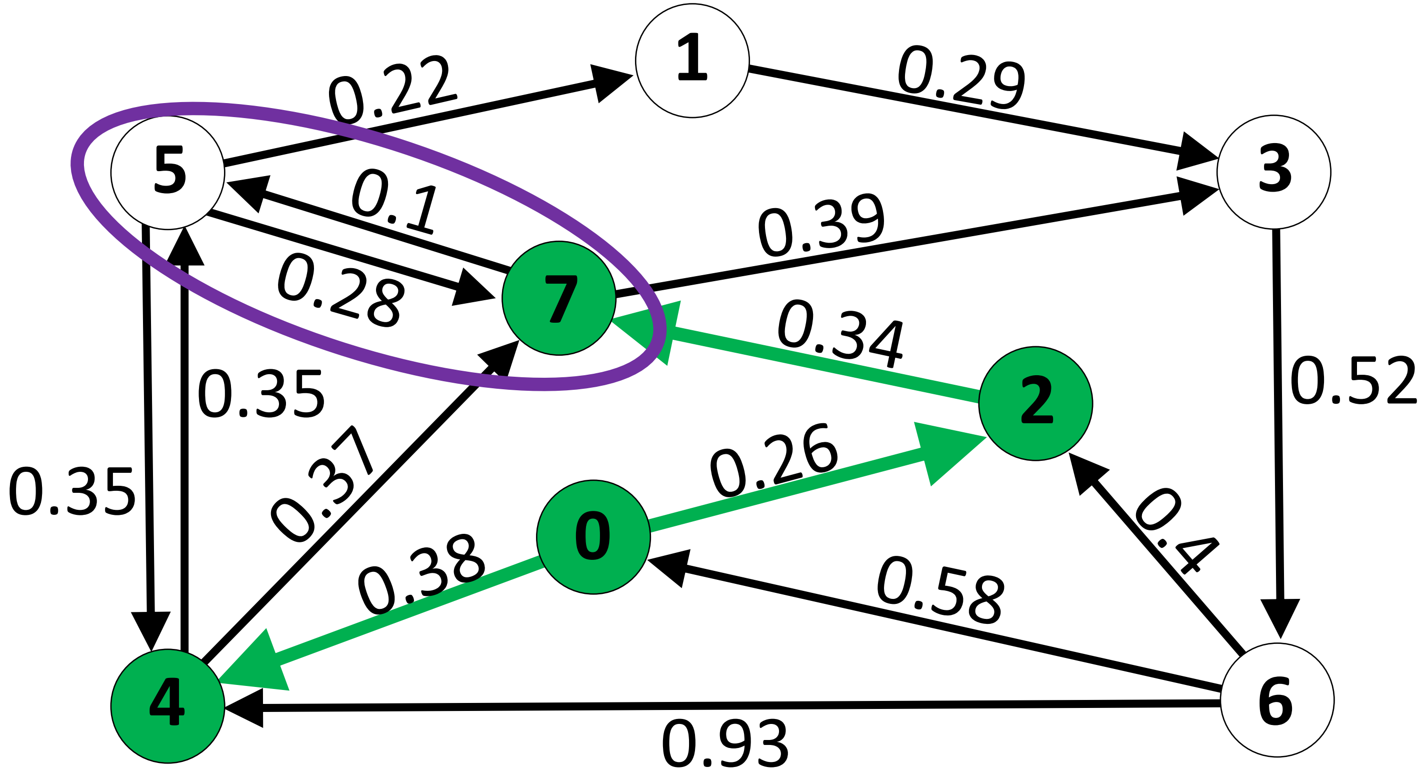
vertex (distance)

Repeat. **We have another route to 5, and at cost 0.7 < 0.73.**

i.e., `distance[v] + weight(v, u) < distance[u]`

Shortest Path

queue top = 7 (0.60)



Distance from 0

0	0
1	∞
2	0.26
3	0.99
4	0.38
5	~~0.73~~ 0.70
6	∞
7	0.60

Previous vertex

0	-
1	
2	0
3	7
4	0
5	4 7
6	
7	2

Priority queue

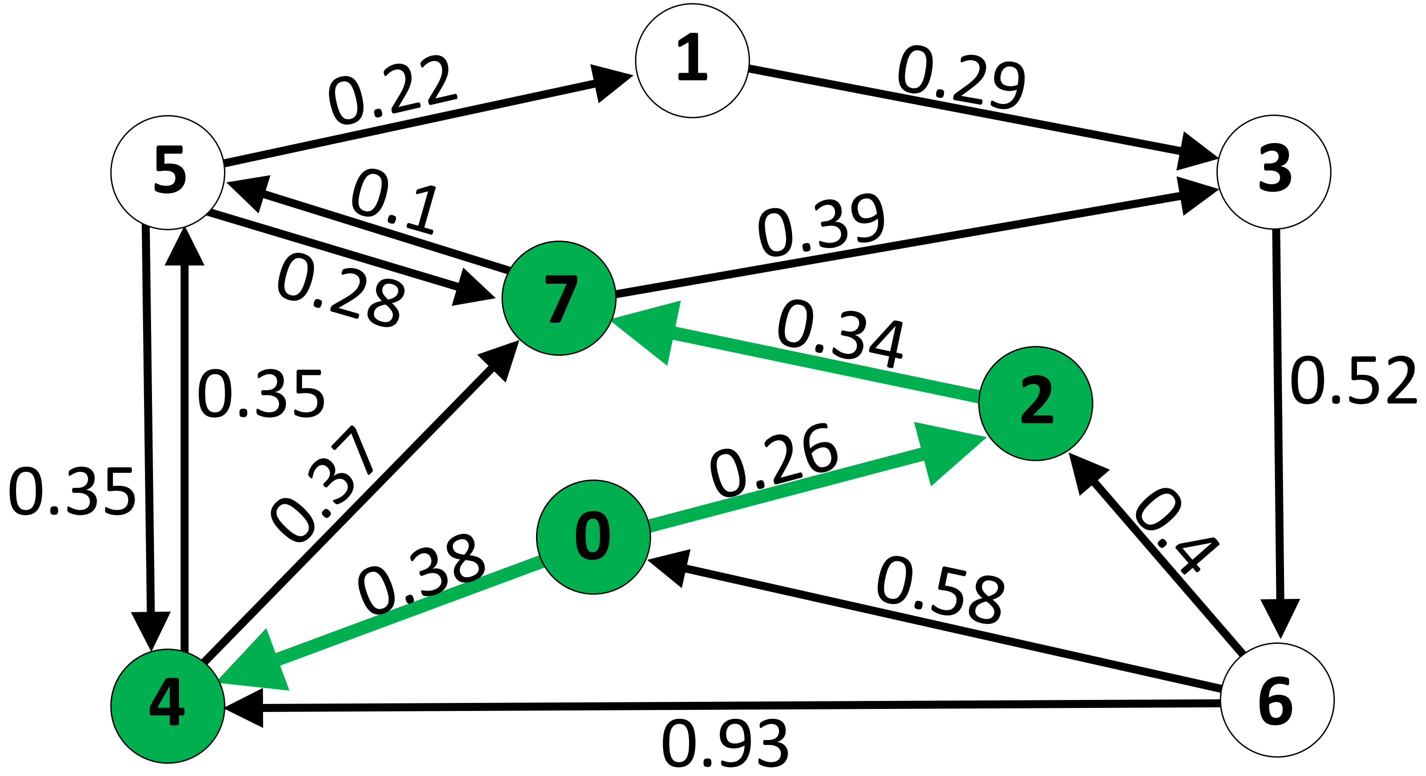
5 (~~0.73~~ 0.70)
3 (0.99)

vertex (distance)

Repeat. **We have another route to 5, and at cost 0.7 < 0.73.**
So updated queue/previous/distance.

Shortest Path

queue top = 7 (0.60)



	Distance from 0
0	0
1	∞
2	0.26
3	0.99
4	0.38
5	0.70
6	∞
7	0.60

	Previous vertex
0	-
1	
2	0
3	7
4	0
5	7
6	
7	2

Priority queue

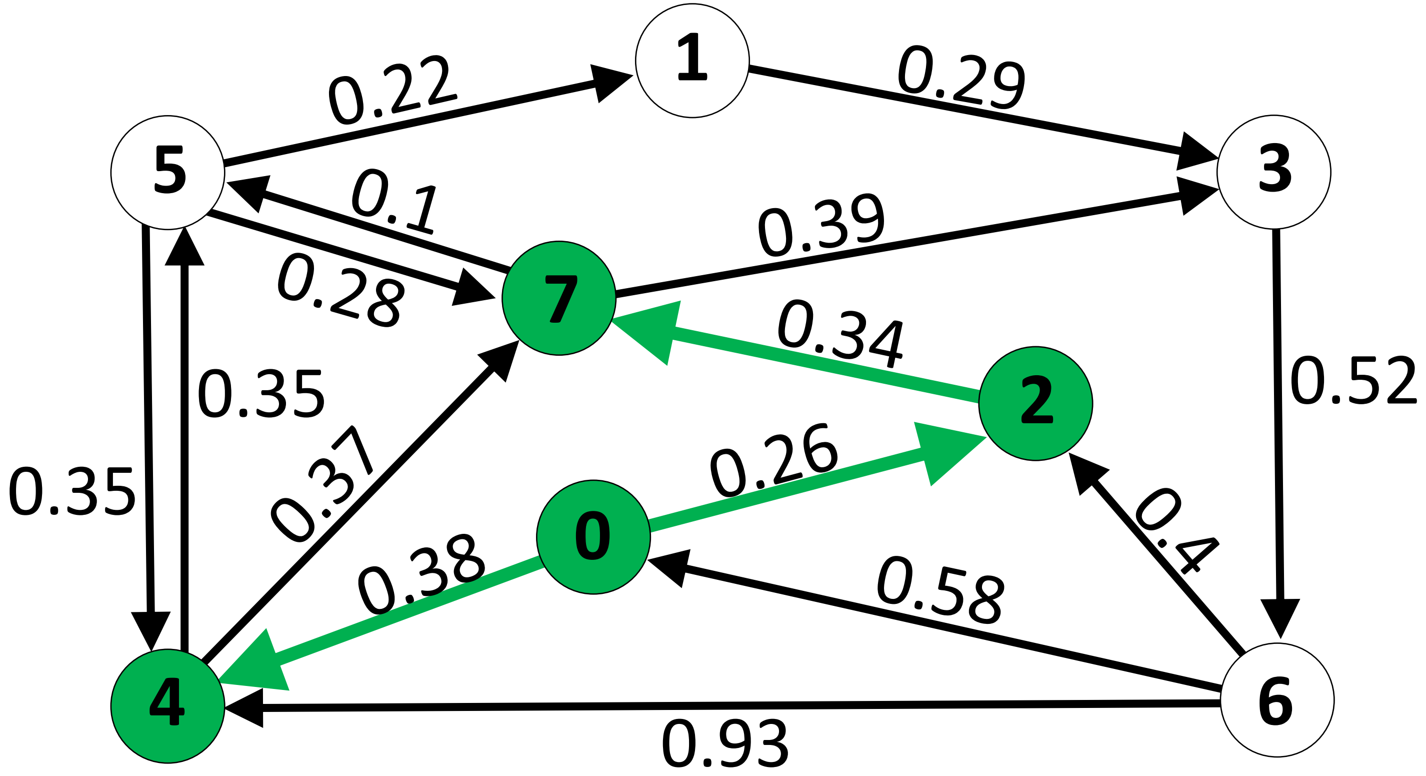
5 (0.70)
3 (0.99)

vertex (distance)

Repeat. **We have another route to 5, and at cost 0.7 < 0.73. So updated queue/previous/distance.**

Shortest Path

queue top = 7 (0.60)



Repeat.

	Distance from 0
0	0
1	∞
2	0.26
3	0.99
4	0.38
5	0.70
6	∞
7	0.60

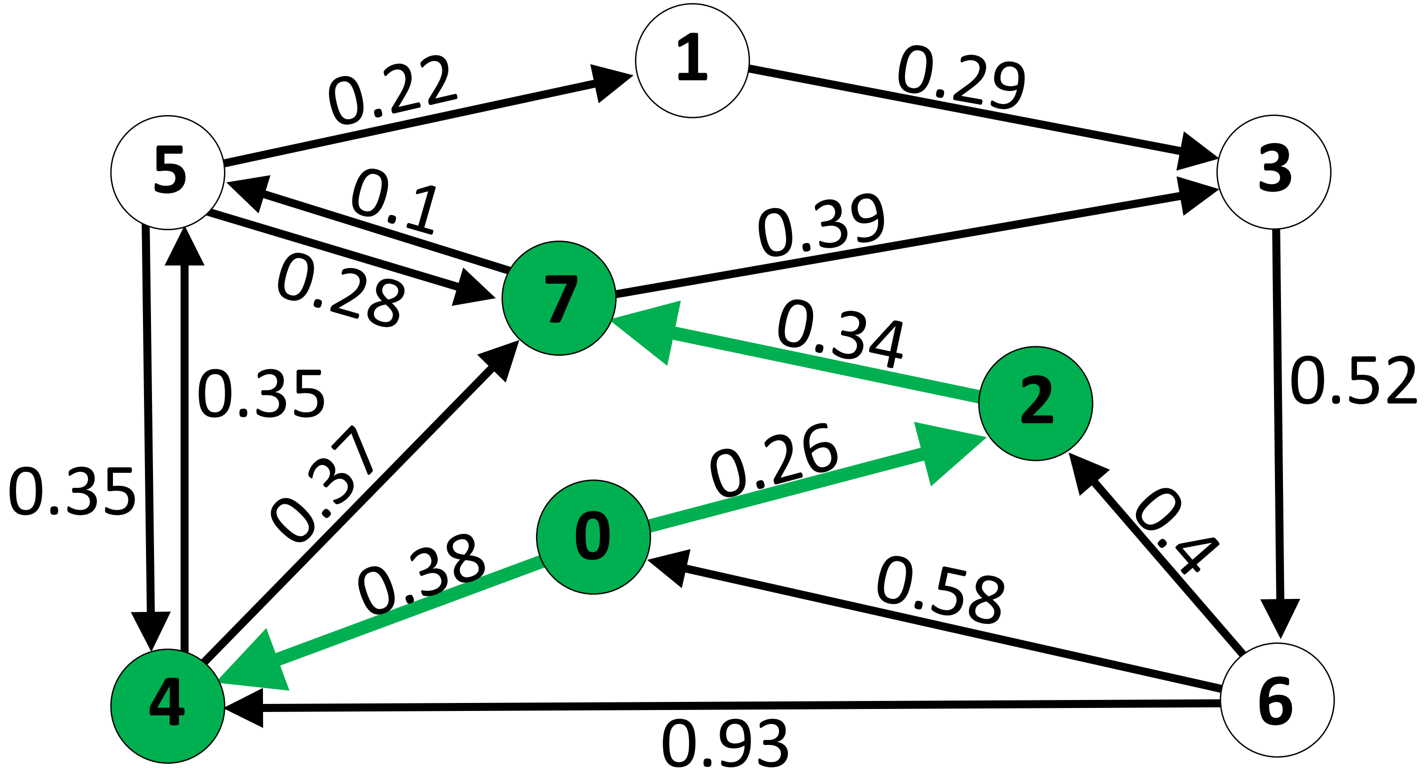
	Previous vertex
0	-
1	
2	0
3	7
4	0
5	7
6	
7	2

Priority queue
5 (0.70)
3 (0.99)

vertex (distance)

Shortest Path

queue top = 5 (0.70)



Repeat.

Vertex	Distance from 0	Previous vertex
0	0	-
1	∞	
2	0.26	0
3	0.99	7
4	0.38	0
5	0.70	7
6	∞	
7	0.60	2

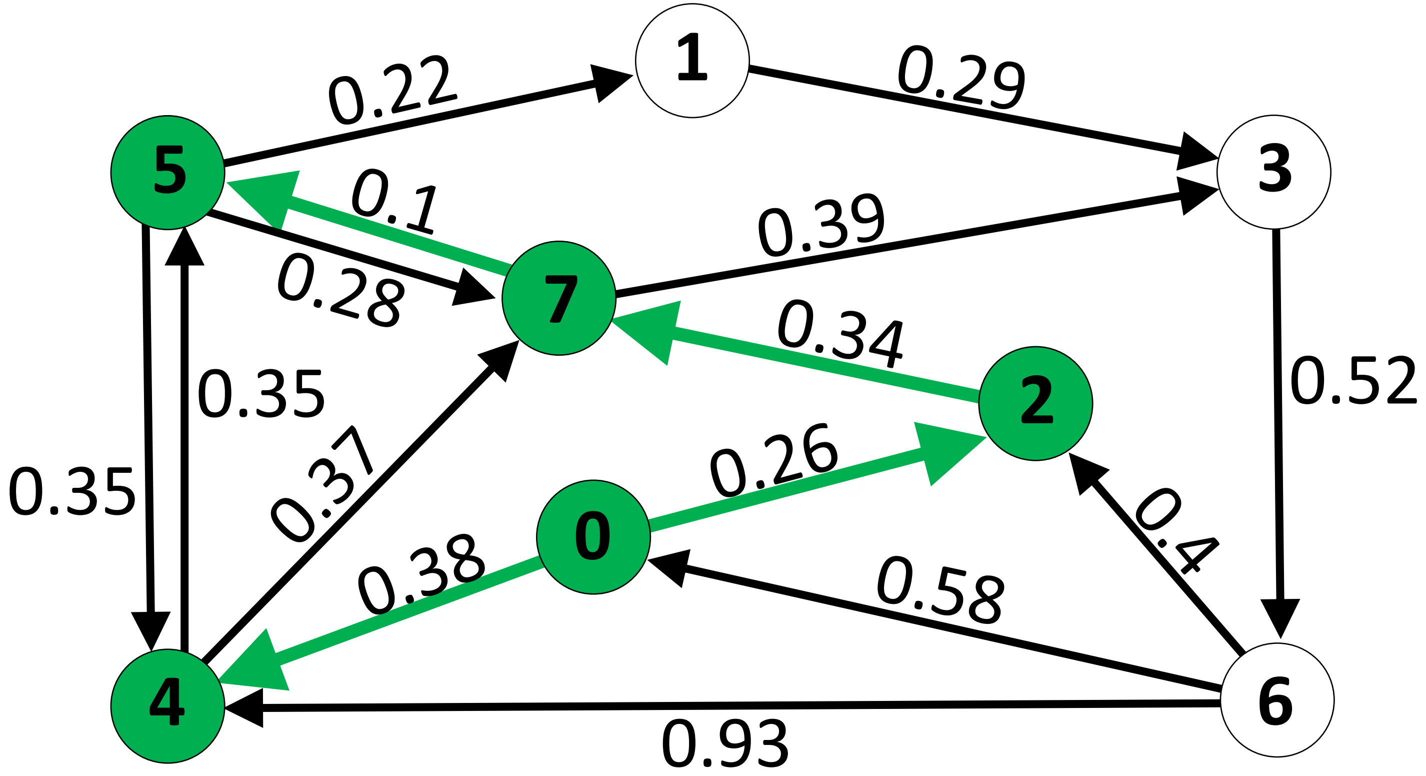
Priority queue

3 (0.99)

vertex (distance)

Shortest Path

queue top = 5 (0.70)



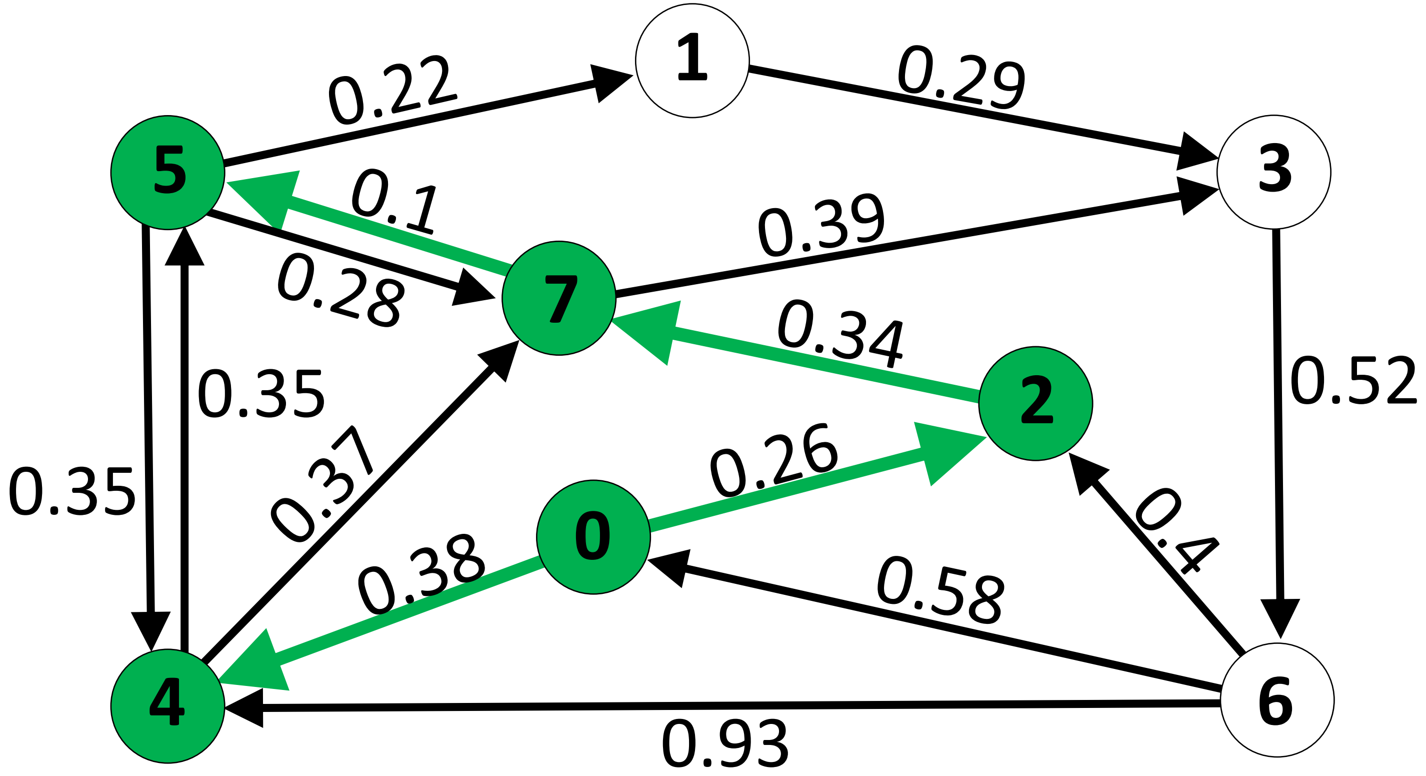
	Distance from 0	Previous vertex	Priority queue
0	0	-	3 (0.99)
1	∞		
2	0.26	0	
3	0.99	7	
4	0.38	0	
5	0.70	7	
6	∞		
7	0.60	2	

vertex (distance)

Repeat.

Shortest Path

queue top = 5 (0.70)



Repeat.

	Distance from 0
0	0
1	0.92
2	0.26
3	0.99
4	0.38
5	0.70
6	∞
7	0.60

	Previous vertex
0	-
1	5
2	0
3	7
4	0
5	7
6	
7	2

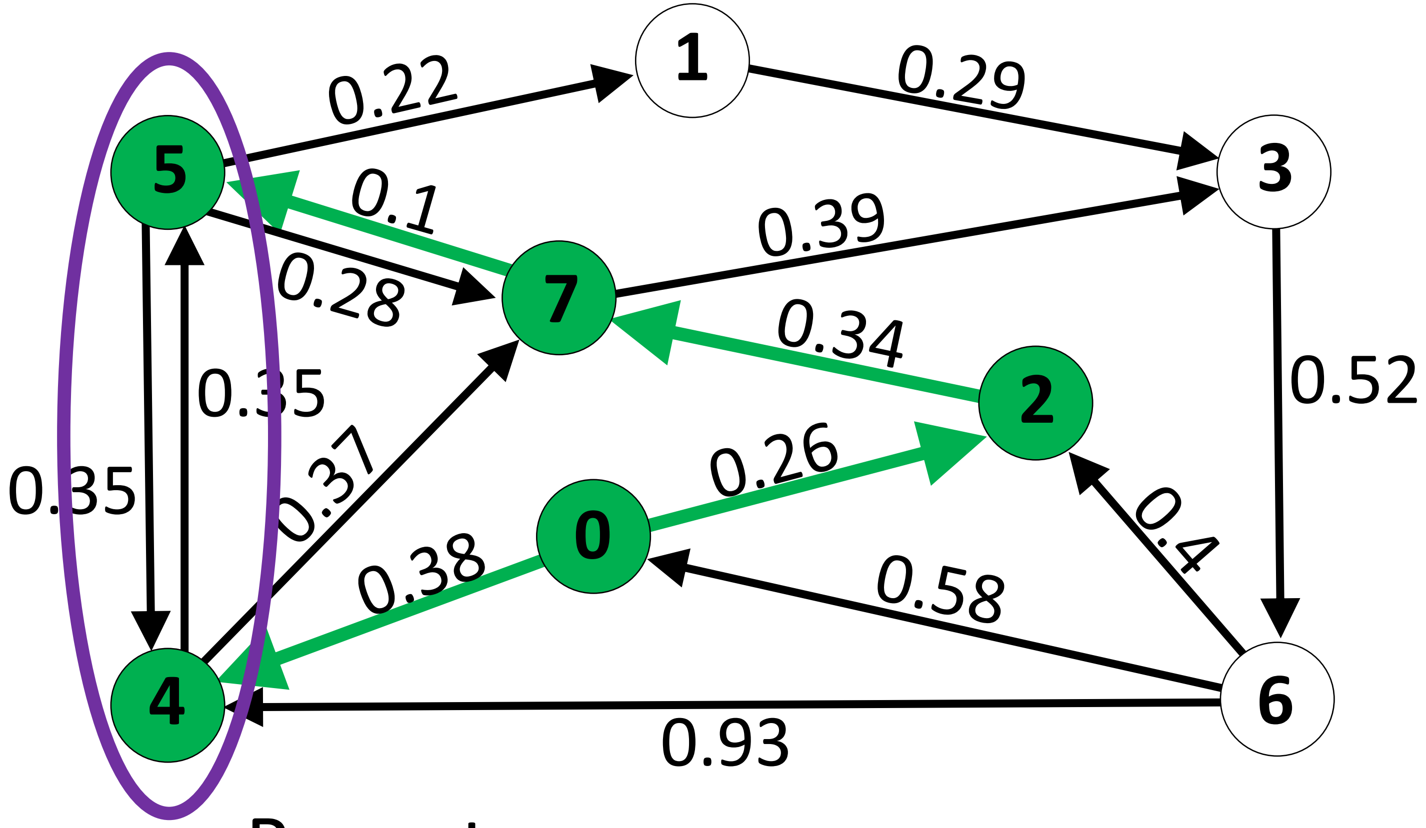
Priority queue

1 (0.92)
3 (0.99)

vertex (distance)

Shortest Path

queue top = 5 (0.70)



Repeat.

What about neighbor 4?

Vertex	Distance from 0	Previous vertex
0	0	-
1	0.92	5
2	0.26	0
3	0.99	7
4	0.38	0
5	0.70	7
6	∞	
7	0.60	2

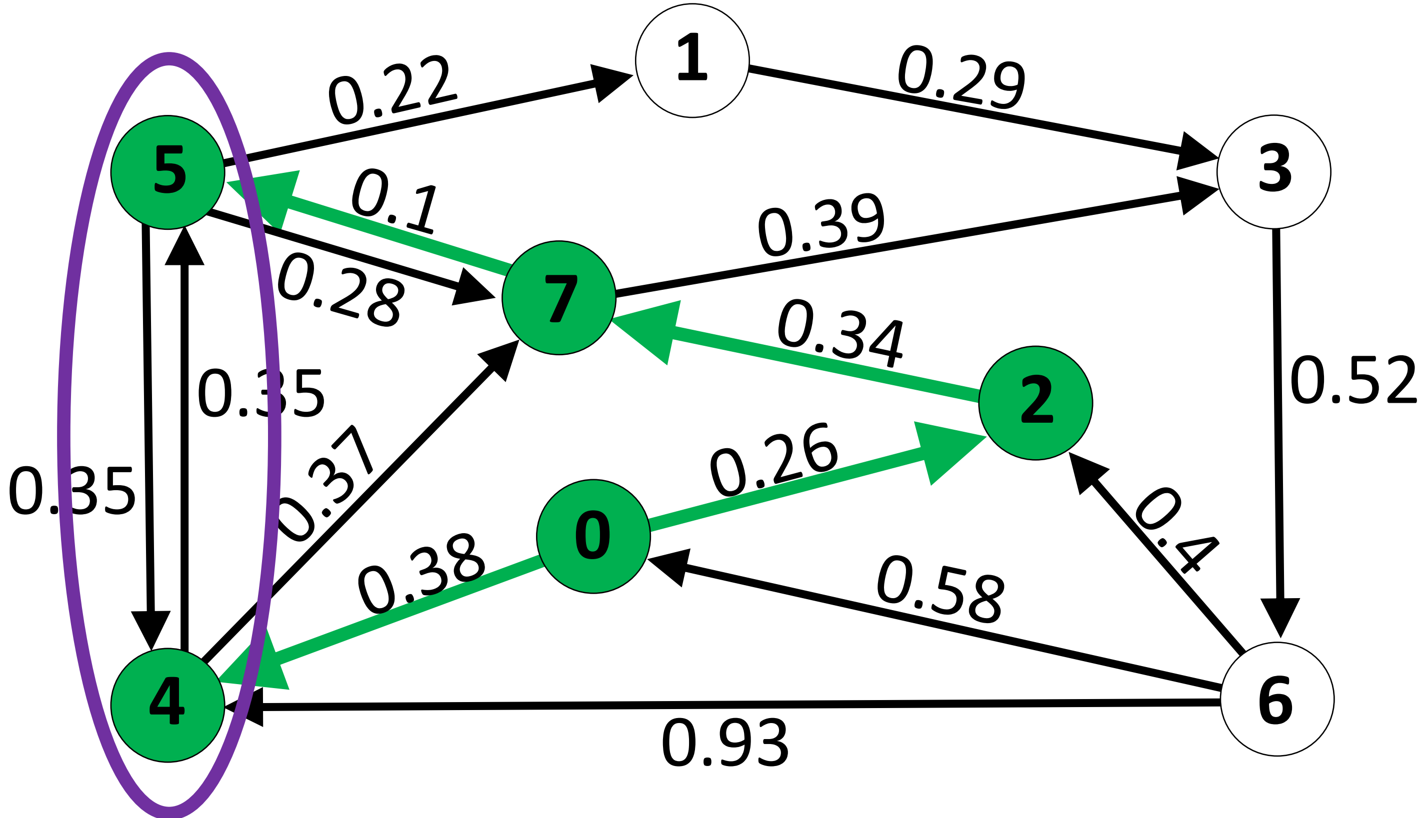
Priority queue

- 1 (0.92)
- 3 (0.99)

vertex (distance)

Shortest Path

queue top = 5 (0.70)



Distance from 0	
0	0
1	0.92
2	0.26
3	0.99
4	0.38
5	0.70
6	∞
7	0.60

Previous vertex	
0	-
1	5
2	0
3	7
4	0
5	7
6	
7	2

Priority queue

1 (0.92)
3 (0.99)

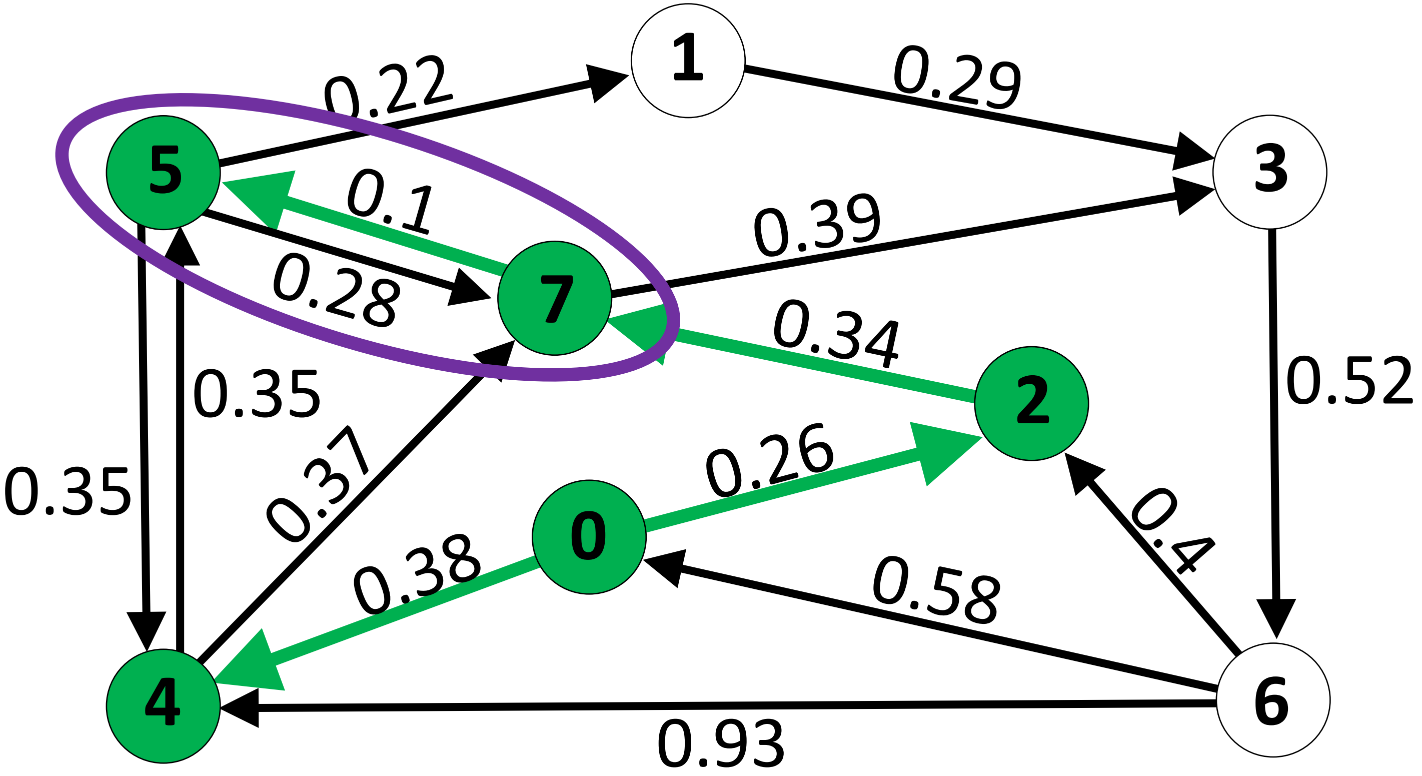
vertex (distance)

Repeat.

What about neighbor 4? distance[5] + weight(5, 4) = 0.70 + 0.35 = 1.05 ≮ 0.38 = distance[4]

Shortest Path

queue top	= 5 (0.70)



Vertex	Distance from 0	Previous vertex
0	0	-
1	0.92	5
2	0.26	0
3	0.99	7
4	0.38	0
5	0.70	7
6	∞	
7	0.60	2

Priority queue

1 (0.92)
3 (0.99)

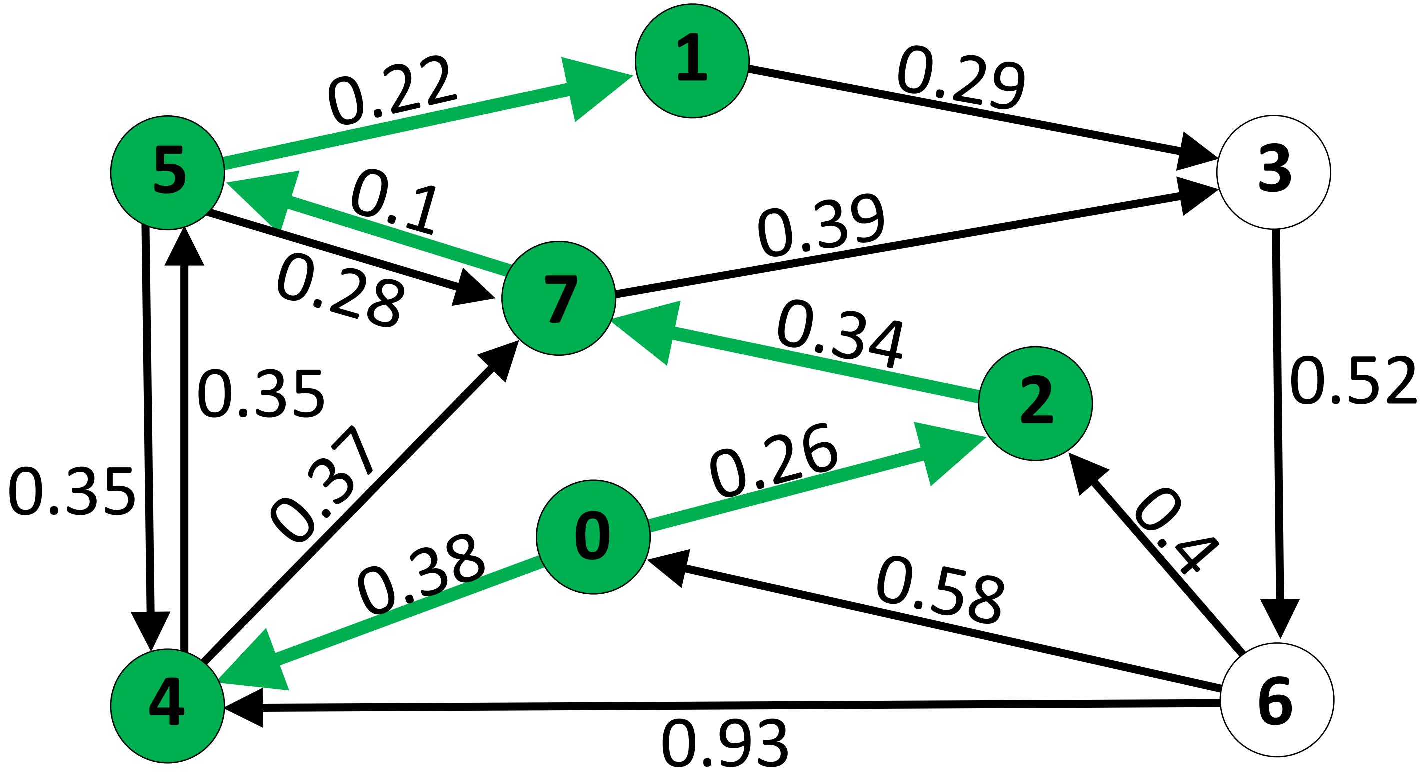
vertex (distance)

Repeat. **What about neighbor 7?**

distance[5] + weight(5, 7) = 0.70 + 0.28 = 0.98 $\not<$ 0.60 = distance[7]

Shortest Path

queue top = 1 (0.92)



Repeat.

Vertex	Distance from 0	Previous vertex
0	0	-
1	0.92	5
2	0.26	0
3	0.99	7
4	0.38	0
5	0.70	7
6	∞	
7	0.60	2

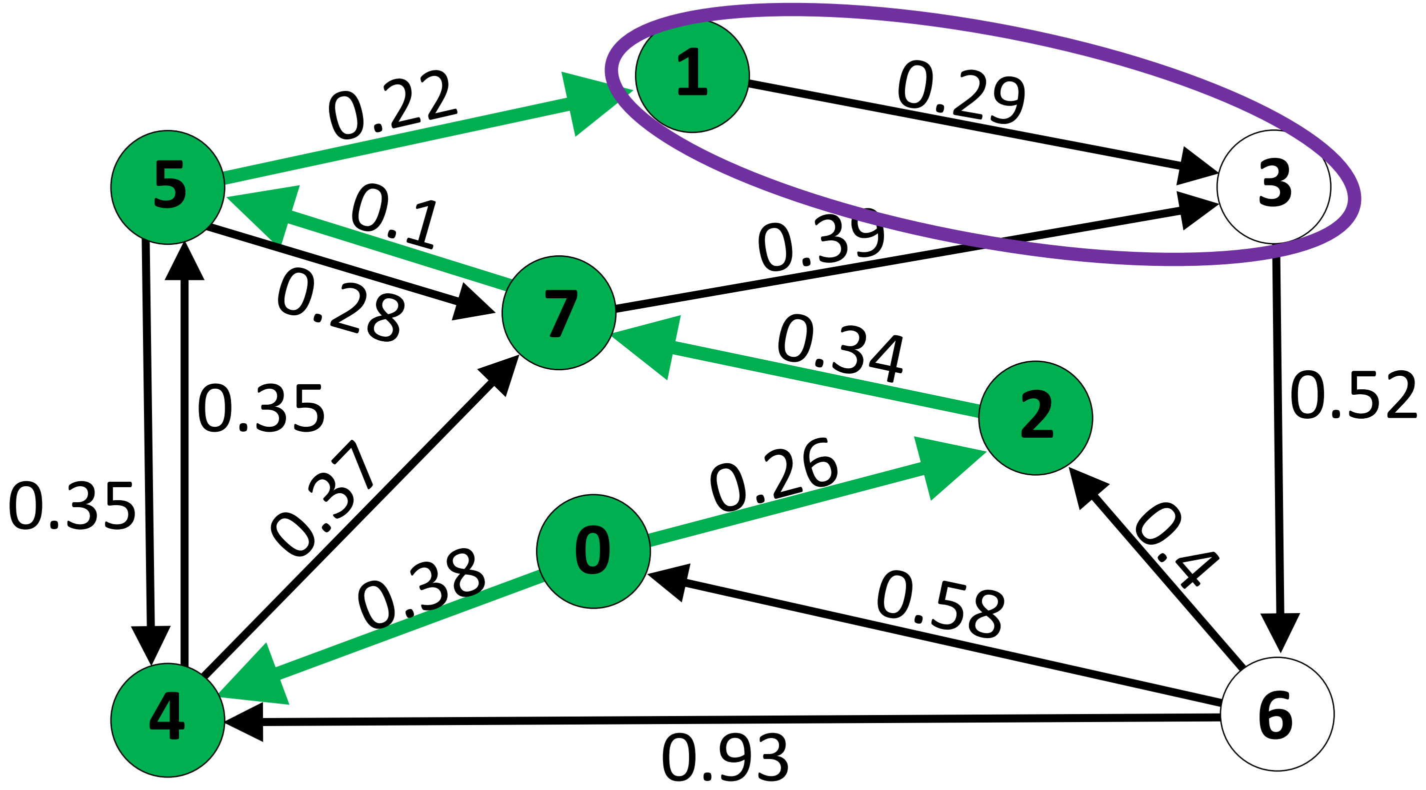
Priority queue

3 (0.99)

vertex (distance)

Shortest Path

queue top = 1 (0.92)



	Distance from 0	Previous vertex
0	0	-
1	0.92	5
2	0.26	0
3	0.99	7
4	0.38	0
5	0.70	7
6	∞	
7	0.60	2

Priority queue

3 (0.99)

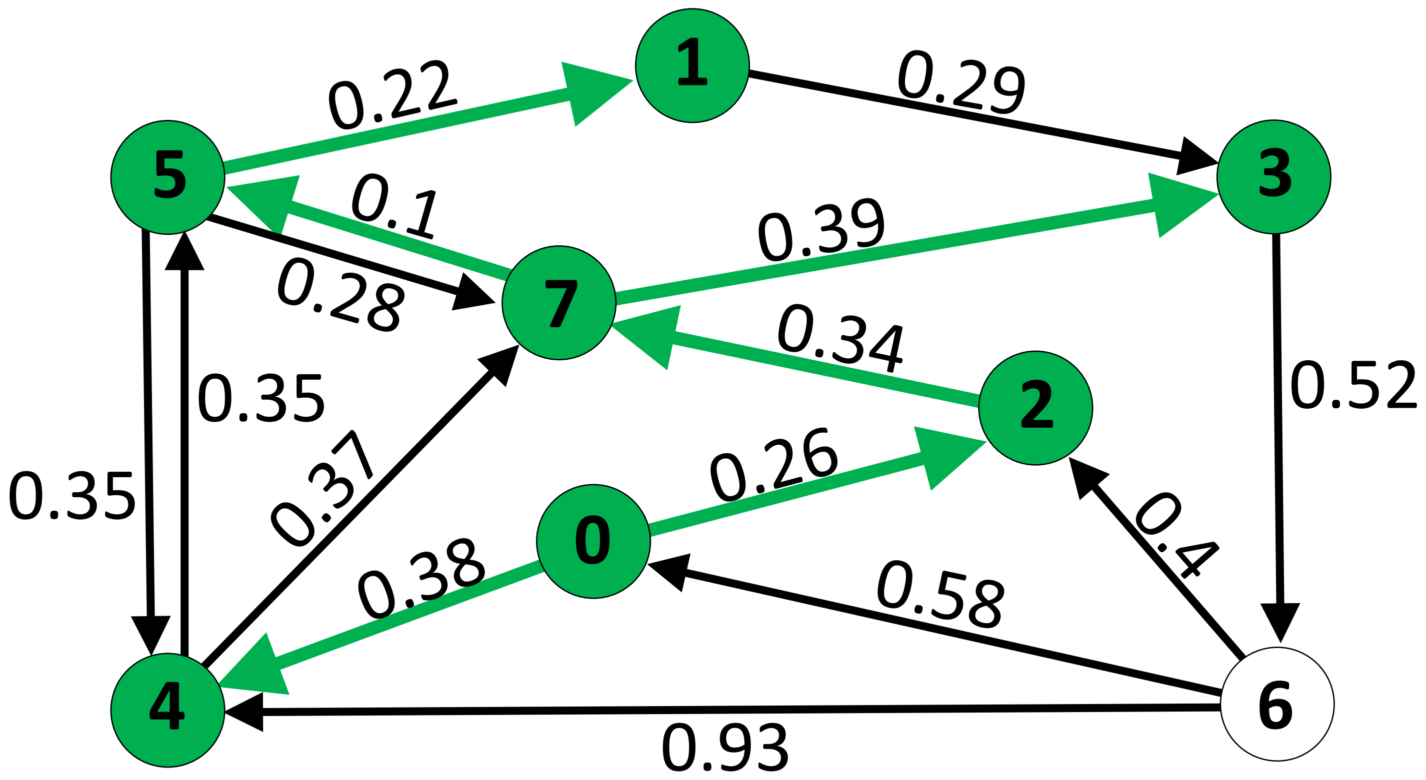
vertex (distance)

Repeat.

What about neighbor 3? 0.92 + 0.29 = 1.21 > 0.99

Shortest Path

queue top = 3 (0.99)



	Distance from 0	Previous vertex
0	0	-
1	0.92	5
2	0.26	0
3	0.99	7
4	0.38	0
5	0.70	7
6	∞	
7	0.60	2

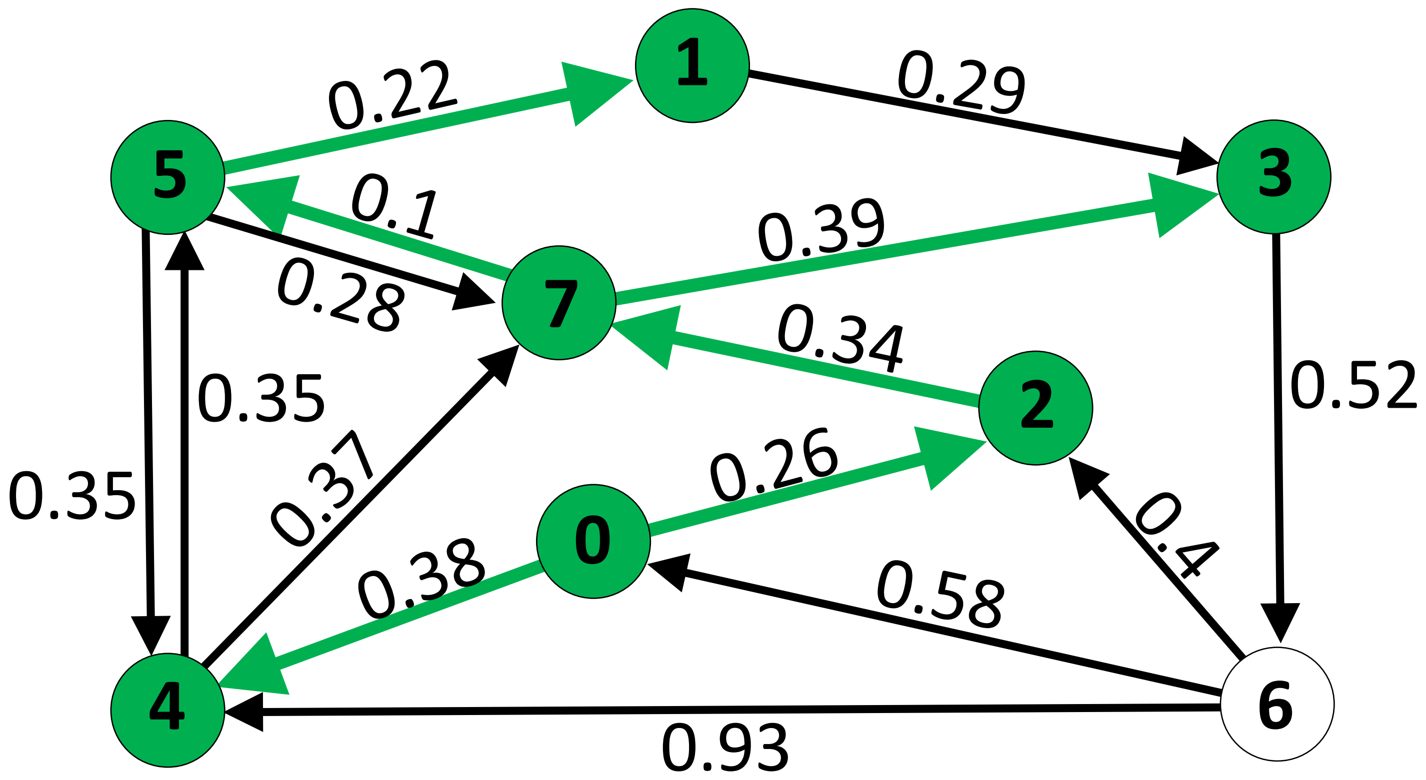
Priority queue

vertex (distance)

Repeat.

Shortest Path

queue top = 3 (0.99)



	Distance from 0
0	0
1	0.92
2	0.26
3	0.99
4	0.38
5	0.70
6	1.51
7	0.60

	Previous vertex
0	-
1	5
2	0
3	7
4	0
5	7
6	3
7	2

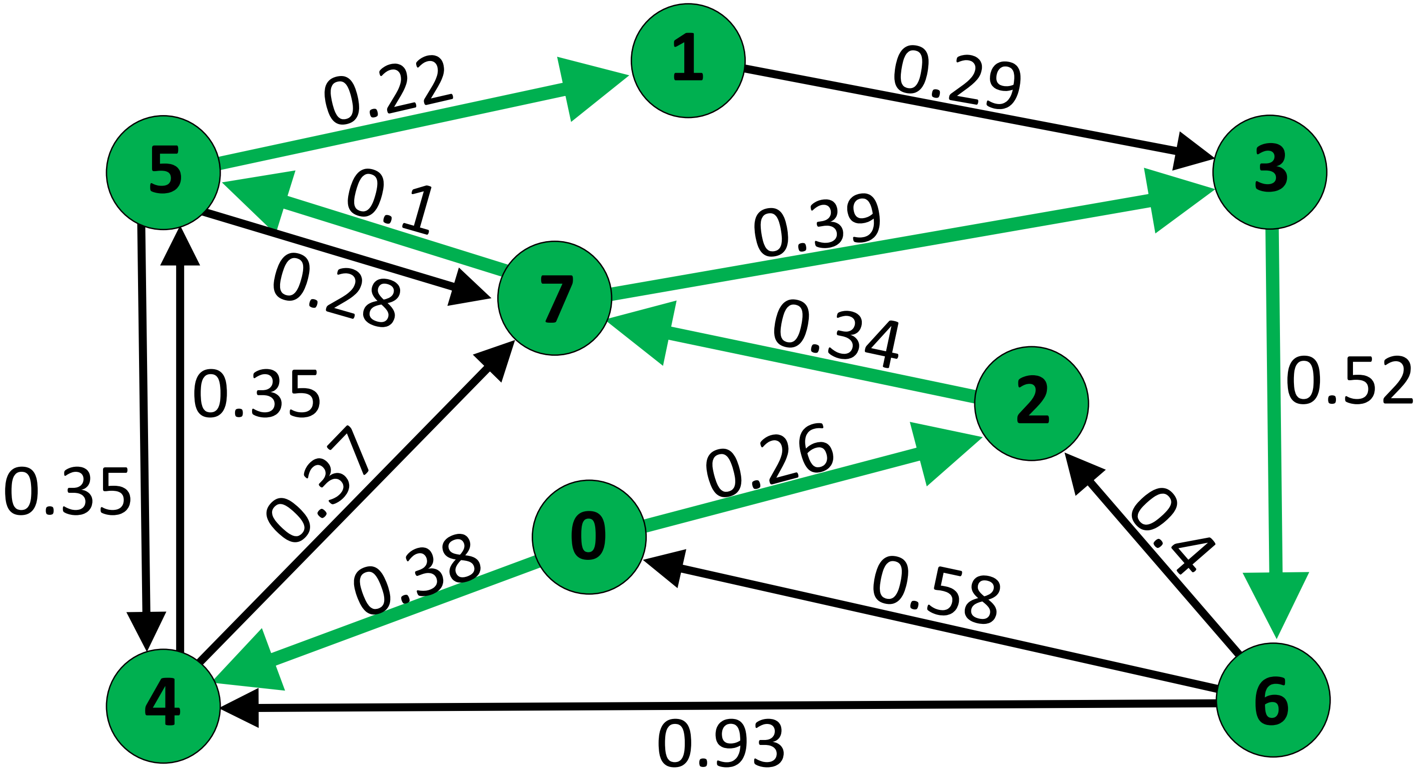
Priority queue
6 (1.51)

vertex (distance)

Repeat.

Shortest Path

queue top	= 6 (1.51)



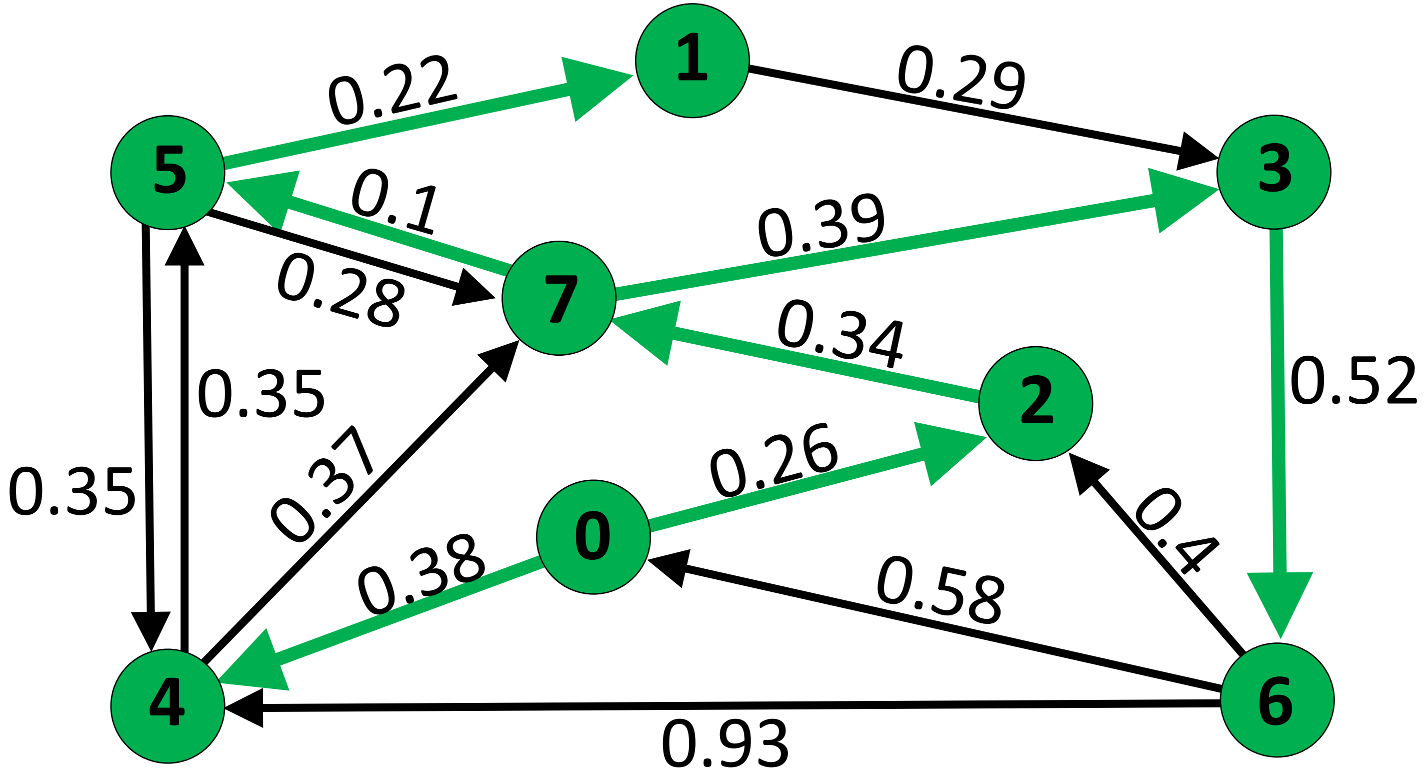
Repeat.

	Distance from 0	Previous vertex	Priority queue
0	0	-	
1	0.92	5	
2	0.26	0	
3	0.99	7	
4	0.38	0	
5	0.70	7	
6	1.51	3	
7	0.60	2	

vertex (distance)

Shortest Path

queue top = 6 (1.51)



Repeat?

	Distance from 0
0	0
1	0.92
2	0.26
3	0.99
4	0.38
5	0.70
6	1.51
7	0.60

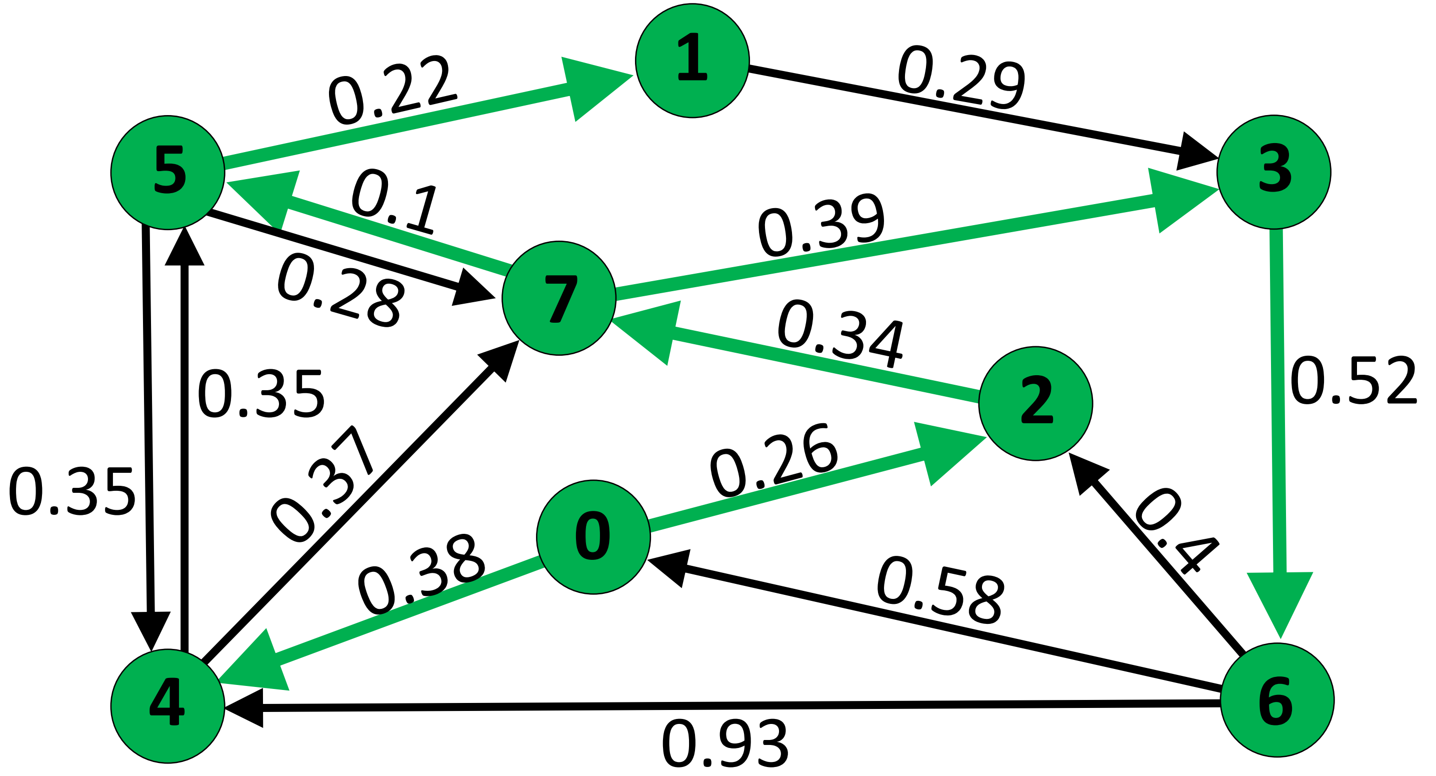
	Previous vertex
0	-
1	5
2	0
3	7
4	0
5	7
6	3
7	2

Priority queue

vertex (distance)

Shortest Path

queue top = 6 (1.51)



Distance from 0	
0	0
1	0.92
2	0.26
3	0.99
4	0.38
5	0.70
6	1.51
7	0.60

Previous vertex	
0	-
1	5
2	0
3	7
4	0
5	7
6	3
7	2

Priority queue

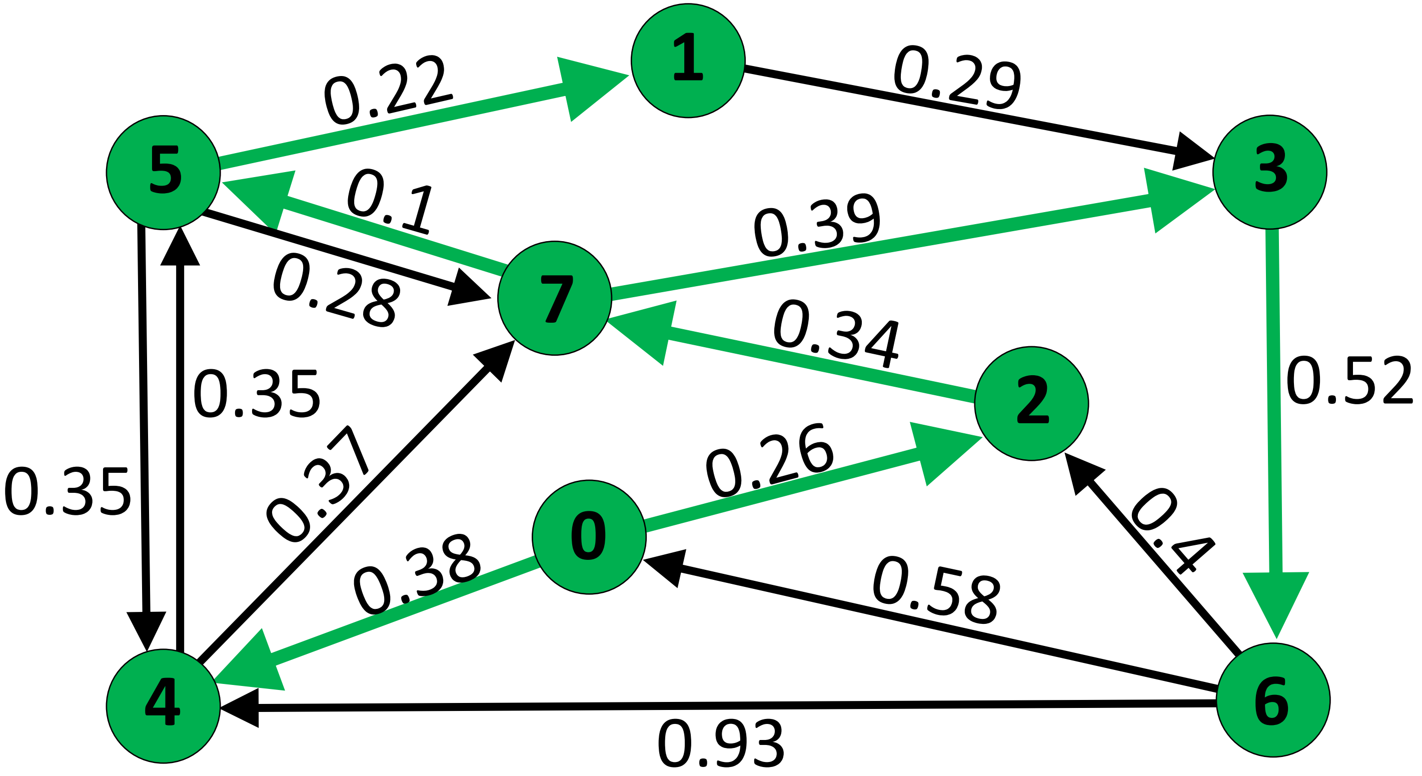
vertex (distance)

Repeat?

Neighbor 4? 1.51 + 0.93 > 0.38

Shortest Path

queue top = 6 (1.51)



Repeat?

Neighbor 0? 1.51 + 0.58 > 0

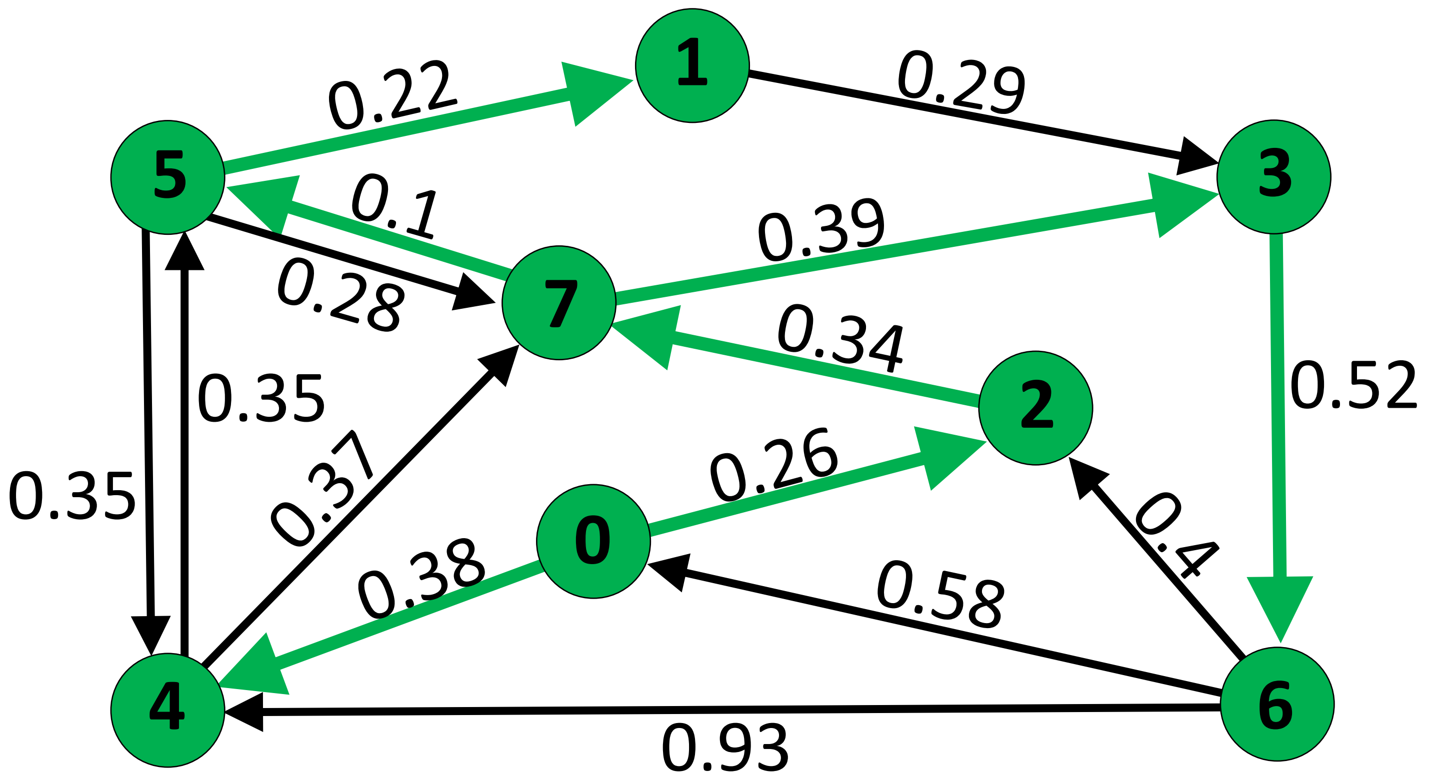
	Distance from 0	Previous vertex
0	0	-
1	0.92	5
2	0.26	0
3	0.99	7
4	0.38	0
5	0.70	7
6	1.51	3
7	0.60	2

Priority queue

vertex (distance)

Shortest Path

queue top = 6 (1.51)



	Distance from 0	Previous vertex
0	0	-
1	0.92	5
2	0.26	0
3	0.99	7
4	0.38	0
5	0.70	7
6	1.51	3
7	0.60	2

Priority queue

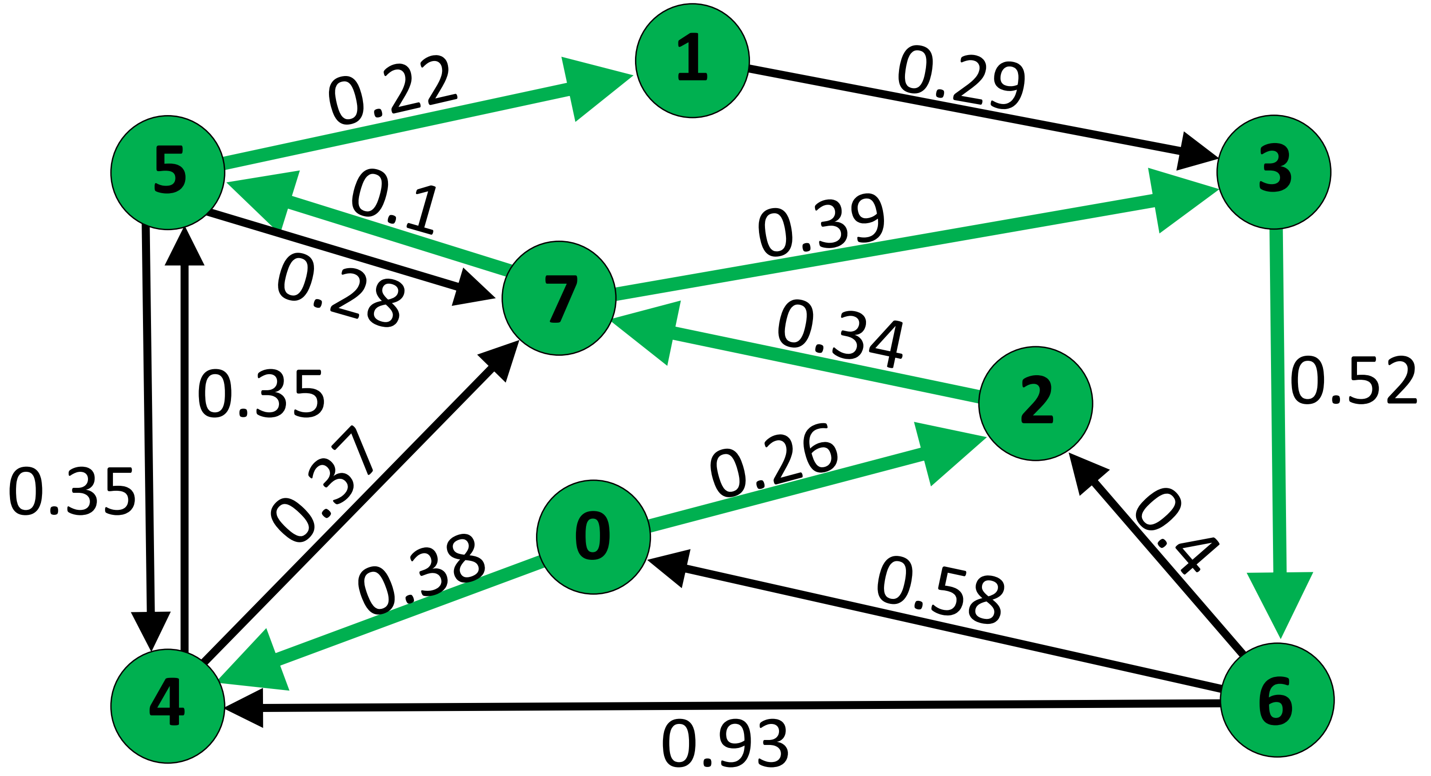
vertex (distance)

Repeat?

Neighbor 2? 1.51 + 0.4 > 0.26

Shortest Path

queue top = 6 (1.51)



When are we done?

	Distance from 0
0	0
1	0.92
2	0.26
3	0.99
4	0.38
5	0.70
6	1.51
7	0.60

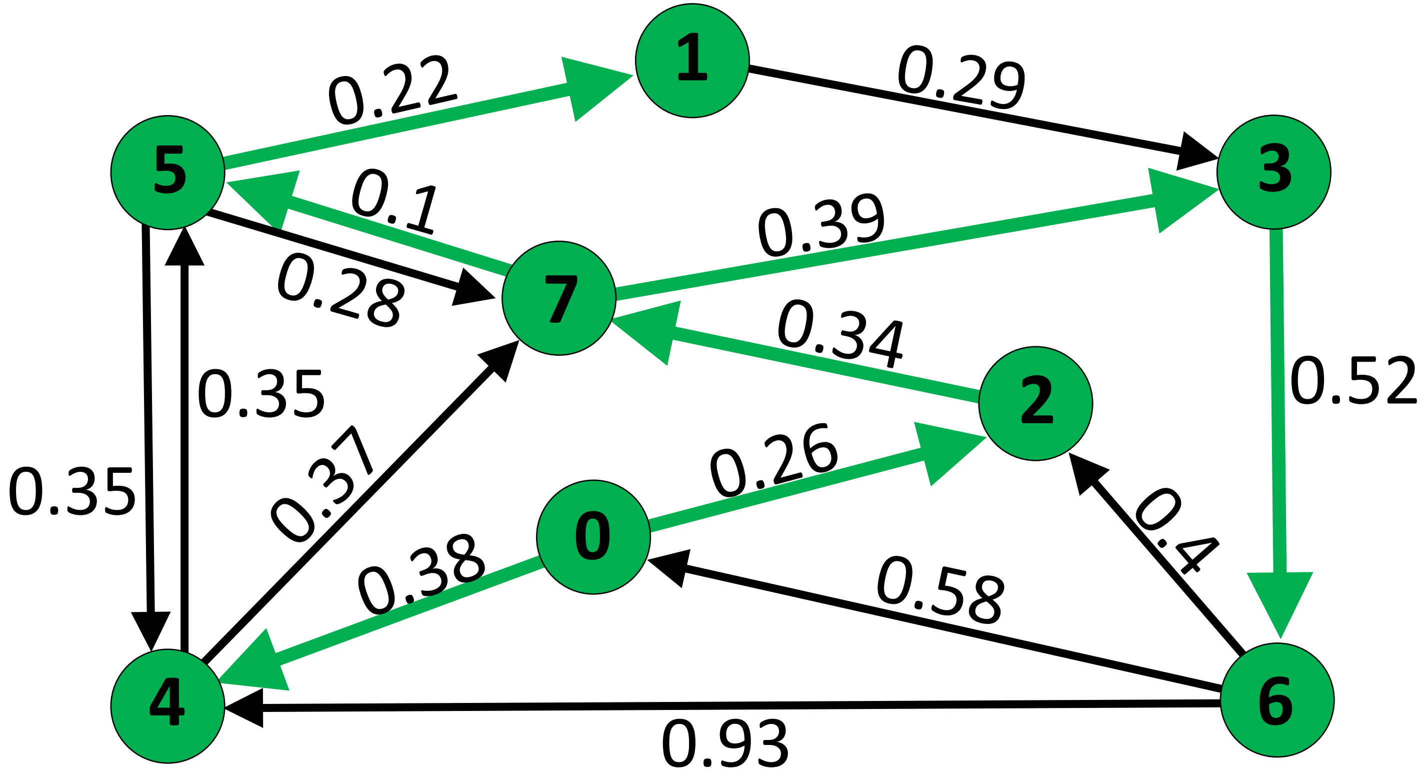
	Previous vertex
0	-
1	5
2	0
3	7
4	0
5	7
6	3
7	2

Priority queue

vertex (distance)

Shortest Path

queue top = 6 (1.51)



When are we done?

When the queue is empty!

Vertex	Distance from 0
0	0
1	0.92
2	0.26
3	0.99
4	0.38
5	0.70
6	1.51
7	0.60

Vertex	Previous vertex
0	-
1	5
2	0
3	7
4	0
5	7
6	3
7	2

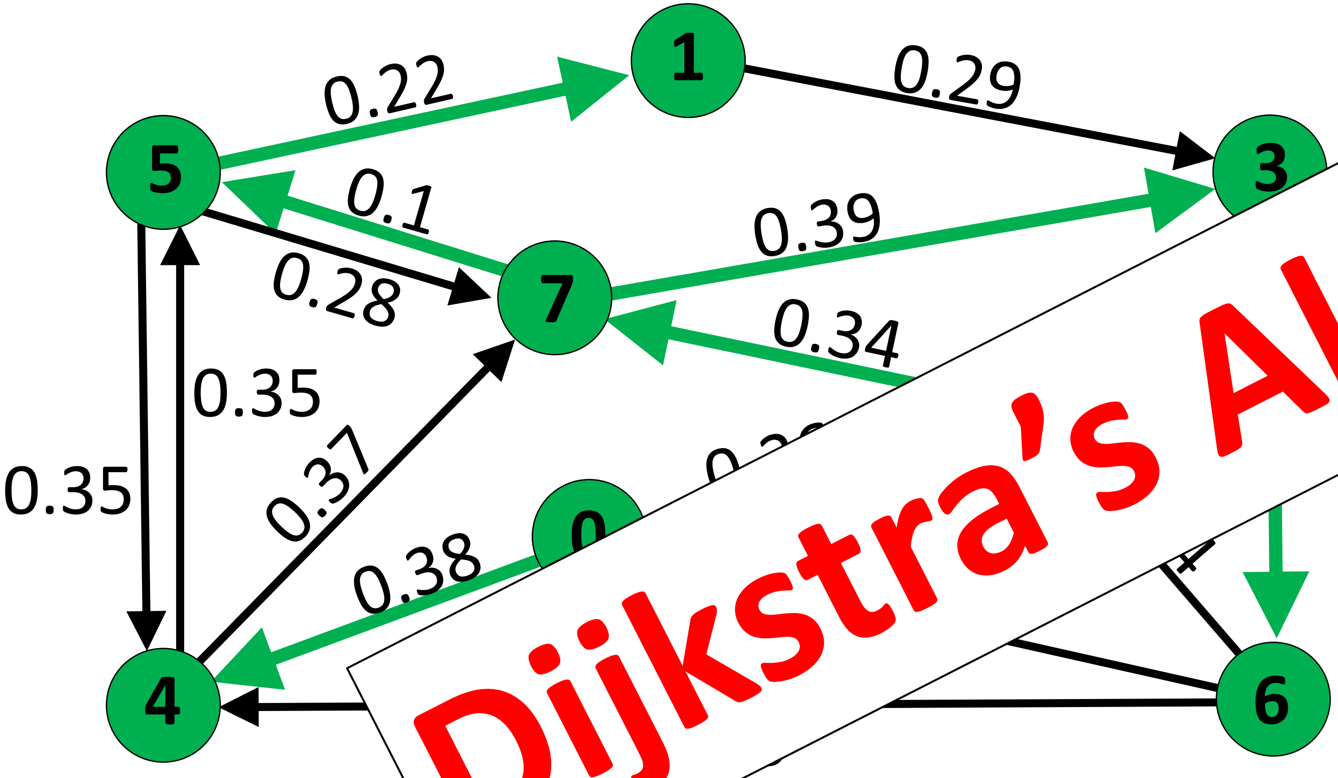
Priority queue

vertex (distance)

Shortest Path

queue top = 6 (1.51)

Dijkstra's Algorithm



	Distance from 0
0	0
1	[illegible]
2	[illegible]
3	[illegible]
4	0.38
5	0.70
6	1.51
7	0.60

	Previous vertex
0	-
1	[illegible]
2	0
3	7
4	0
5	7
6	3
7	2

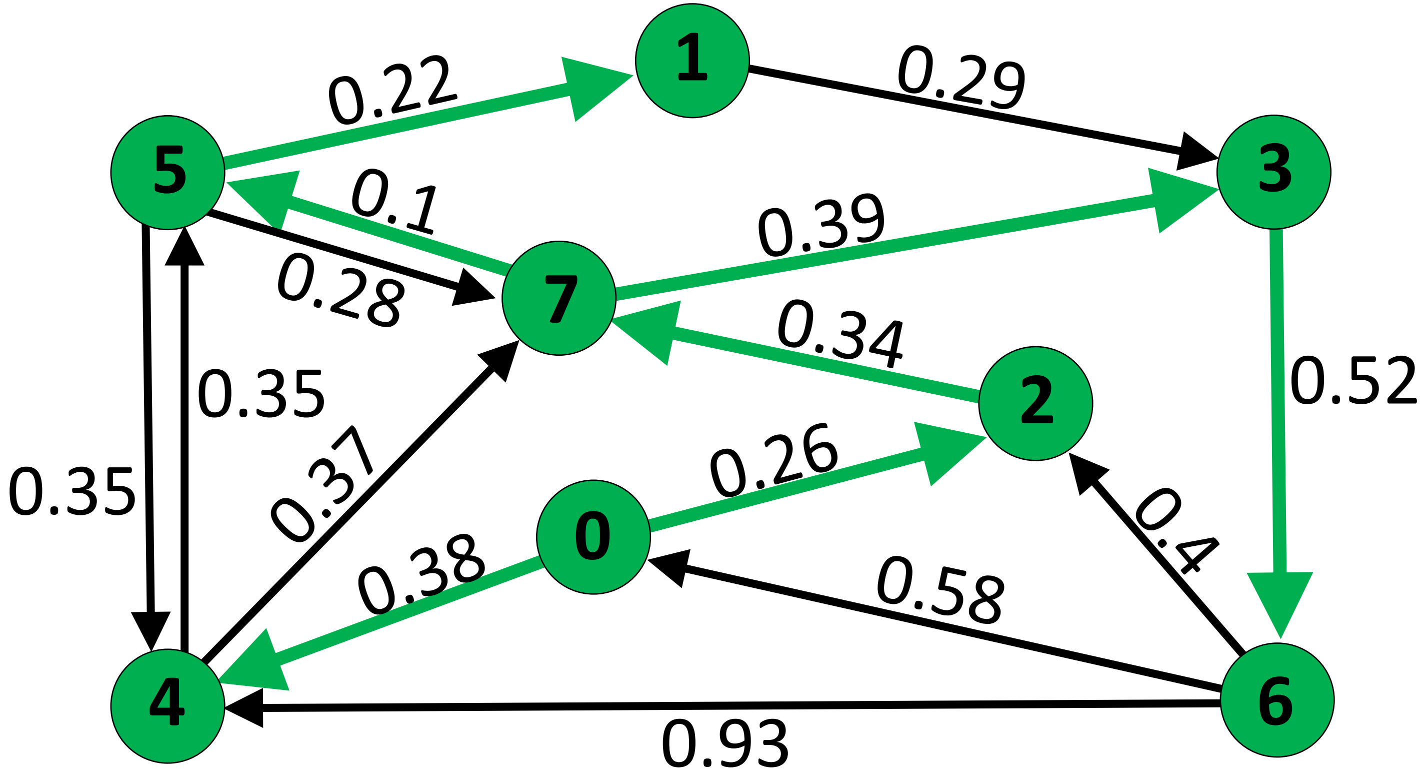
Priority queue

vertex (distance)

When … we done?

When the queue is empty!

Shortest Path

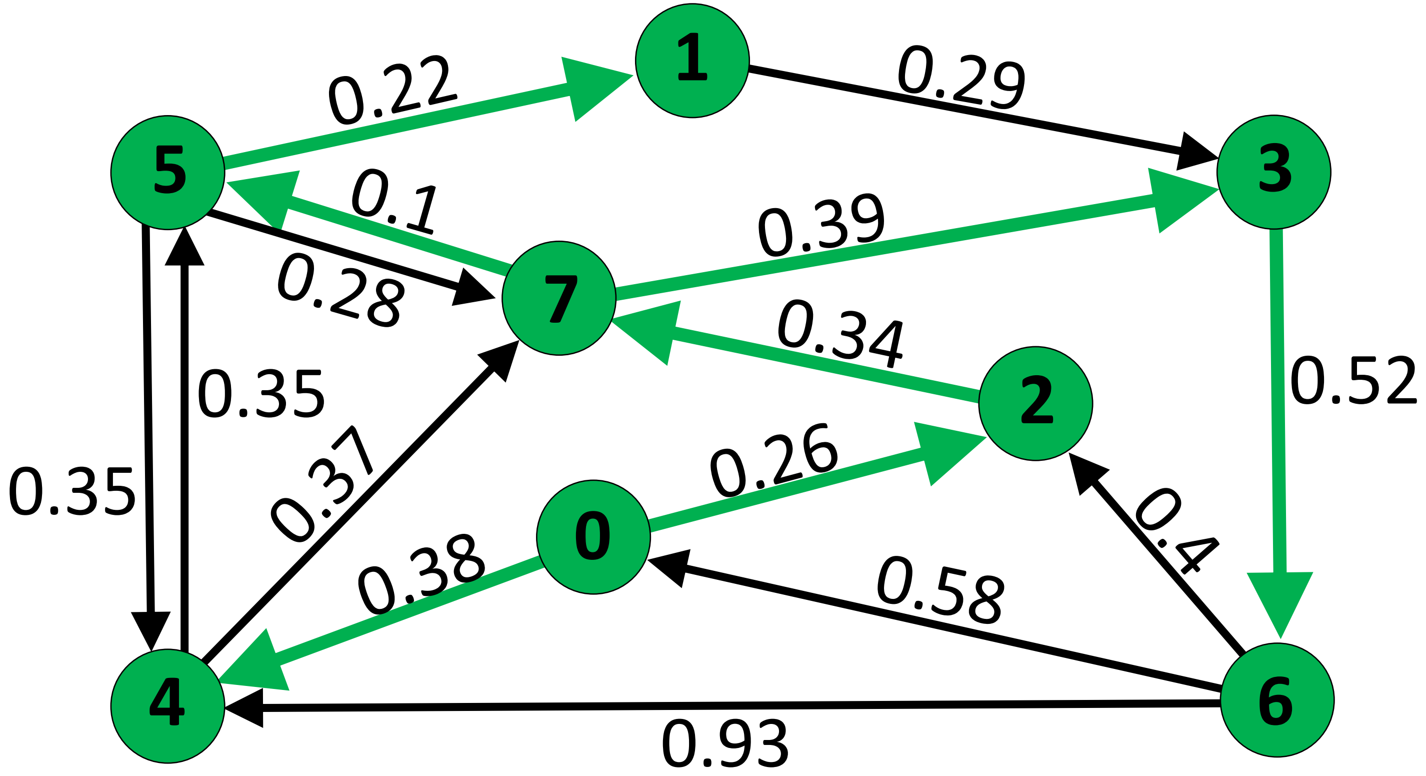


Assumptions:

- Graph is directed.
- Graph is edge-weighted.
- Edge weights are non-negative.
- Graph need not be simple (though our example will be).

What happens if there are self-loops?

Shortest Path



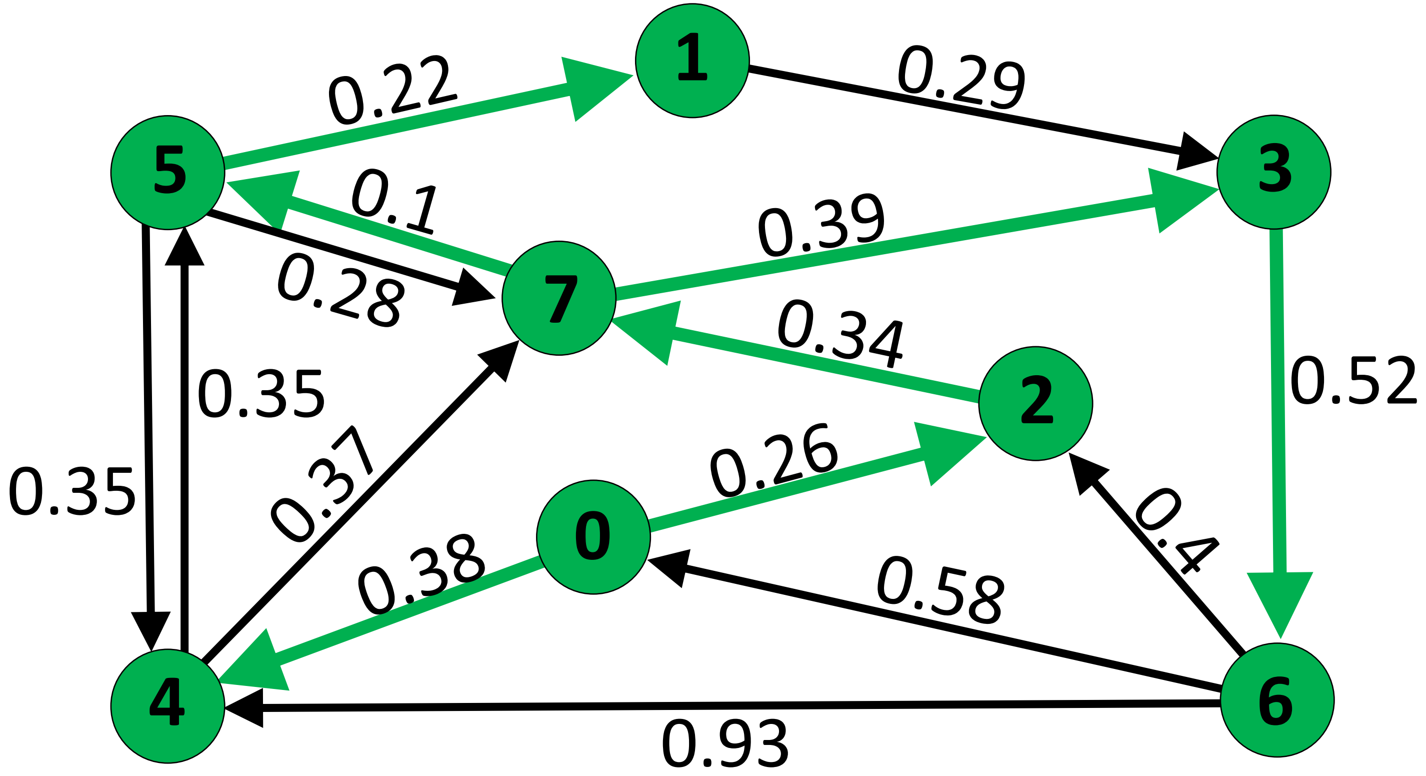
Assumptions:

- Graph is directed.
- Graph is edge-weighted.
- Edge weights are non-negative.
- Graph need not be simple (though our example will be).

What happens if there are self-loops?

They are never taken, since they will never lower the cost of a path.

Shortest Path

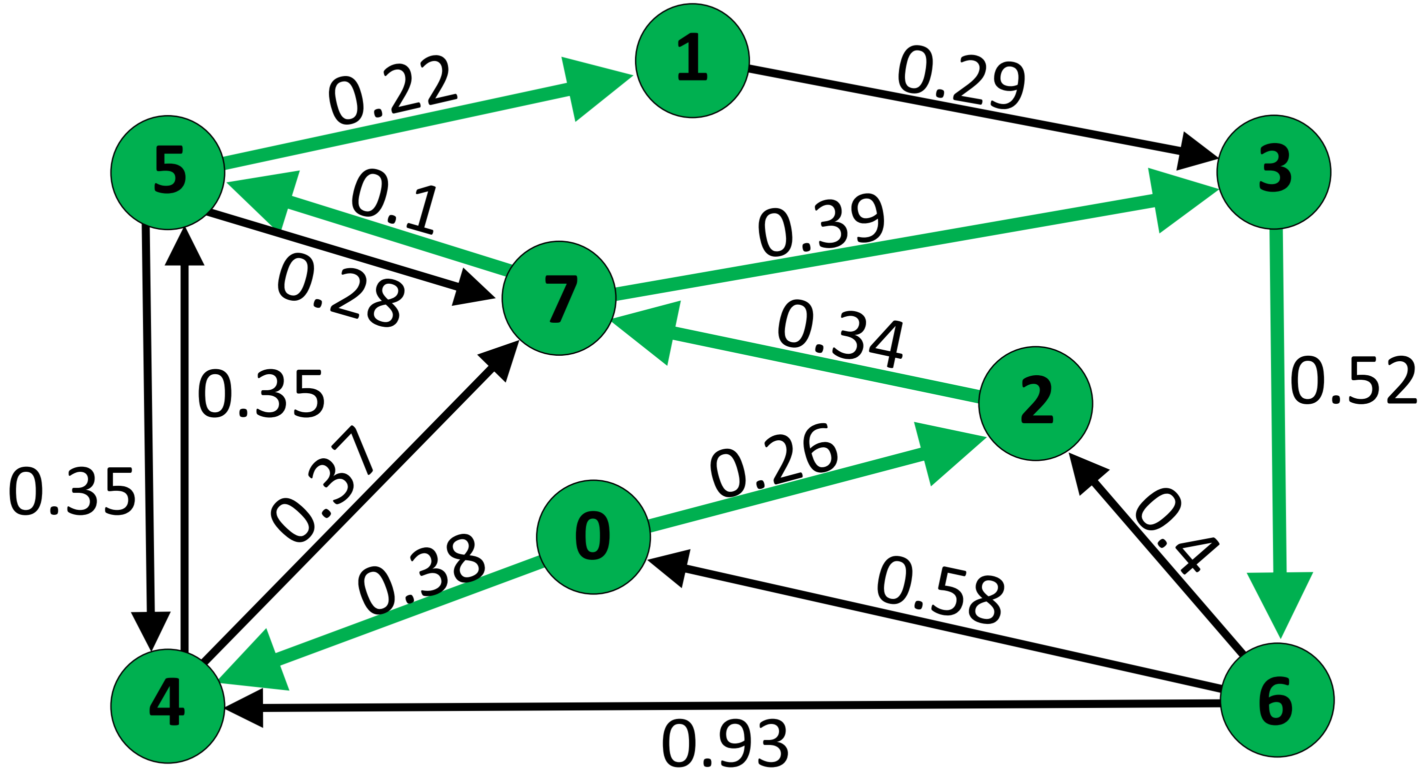


Assumptions:

- Graph is directed.
- Graph is edge-weighted.
- Edge weights are non-negative.
- Graph need not be simple (though our example will be).

What happens if there are parallel edges?

Shortest Path



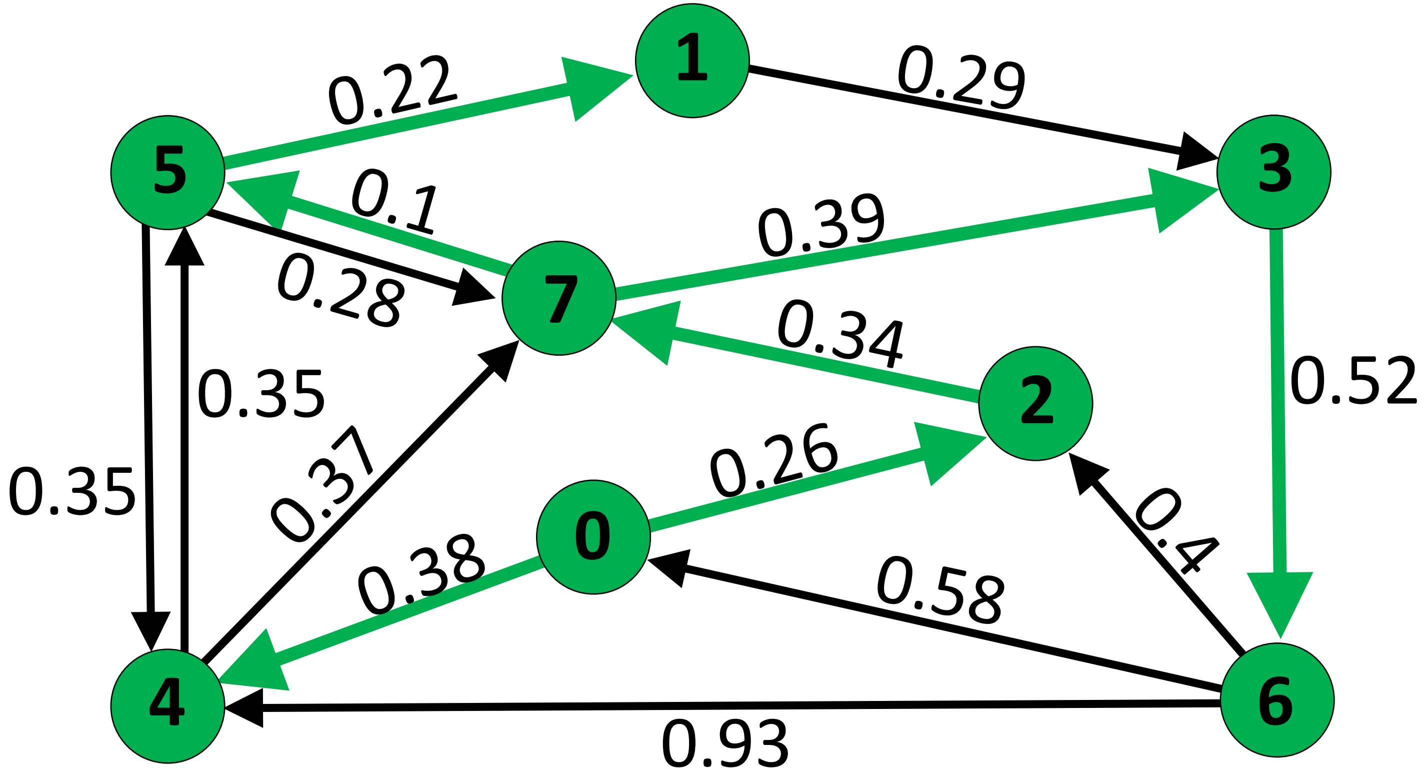
Assumptions:

- Graph is directed.
- Graph is edge-weighted.
- Edge weights are non-negative.
- Graph need not be simple (though our example will be).

What happens if there are parallel edges?

The cheapest one is taken and all others are ignored.

Shortest Path

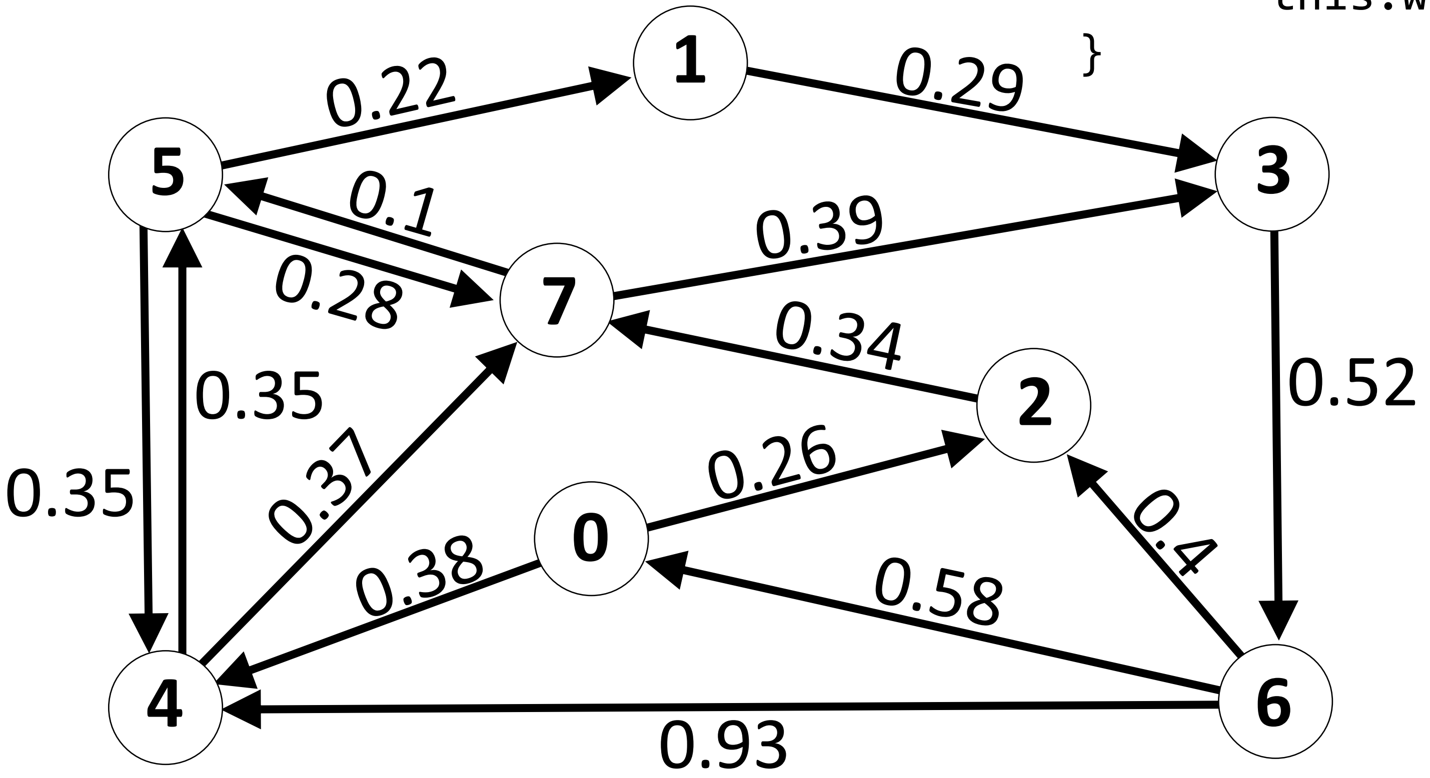


Assumptions:

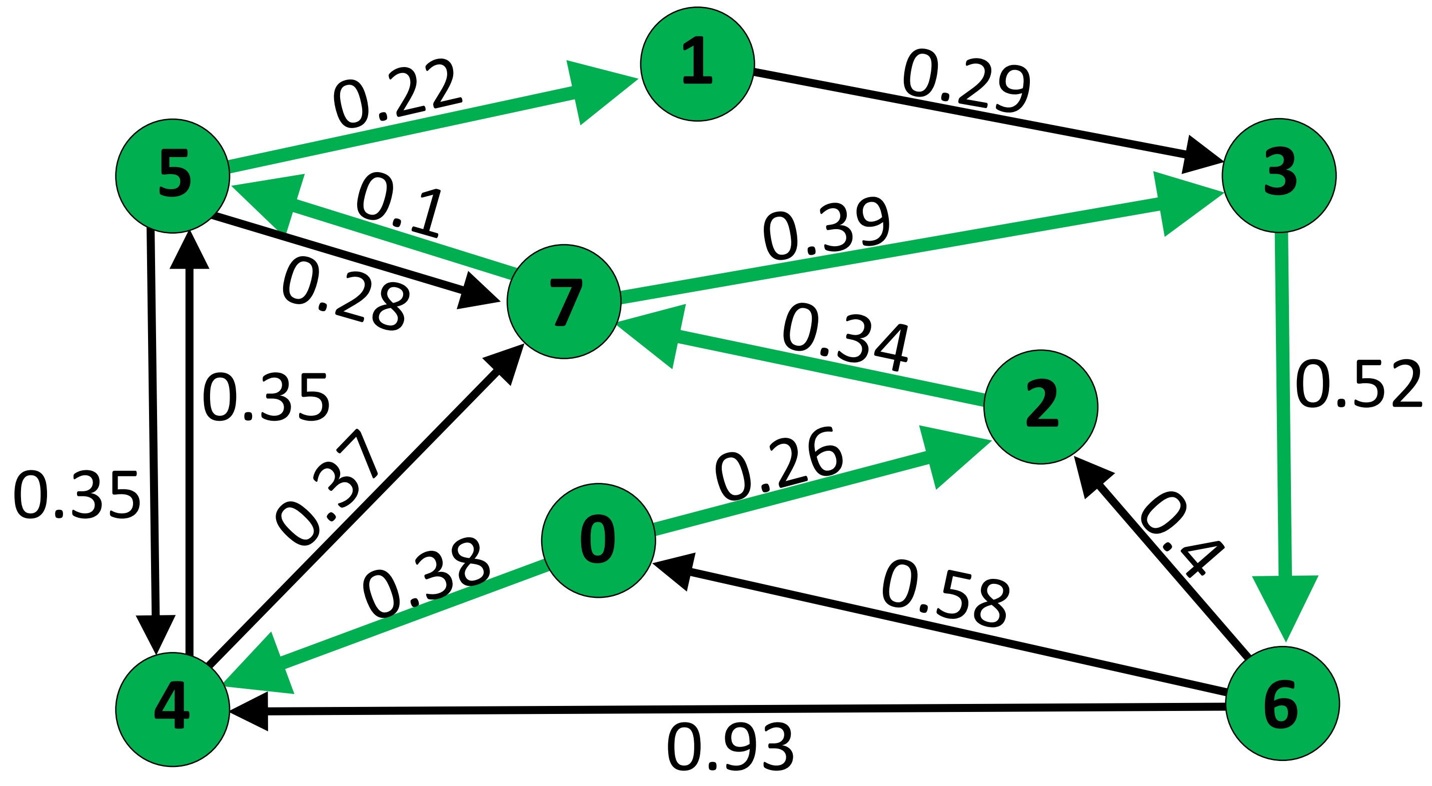
- Graph is directed.
- Graph is edge-weighted.
- Edge weights are non-negative.
- Graph need not be simple (though our example will be).

What happens if there are negative weights?

```java
public class Edge implements Comparable<Edge>{

        private int sourceVertex;
        private int destVertex;
        private double weight;

        public Edge(int vertex1, int vertex2, double weight) {
                this.sourceVertex = vertex1;
                this.destVertex = vertex2;
                this.weight = weight;
        }
```



Dijkstra's Algorithm



Distance from 0

0	0
1	0.92
2	0.26
3	0.99
4	0.38
5	0.70
6	1.51
7	0.60

Previous vertex

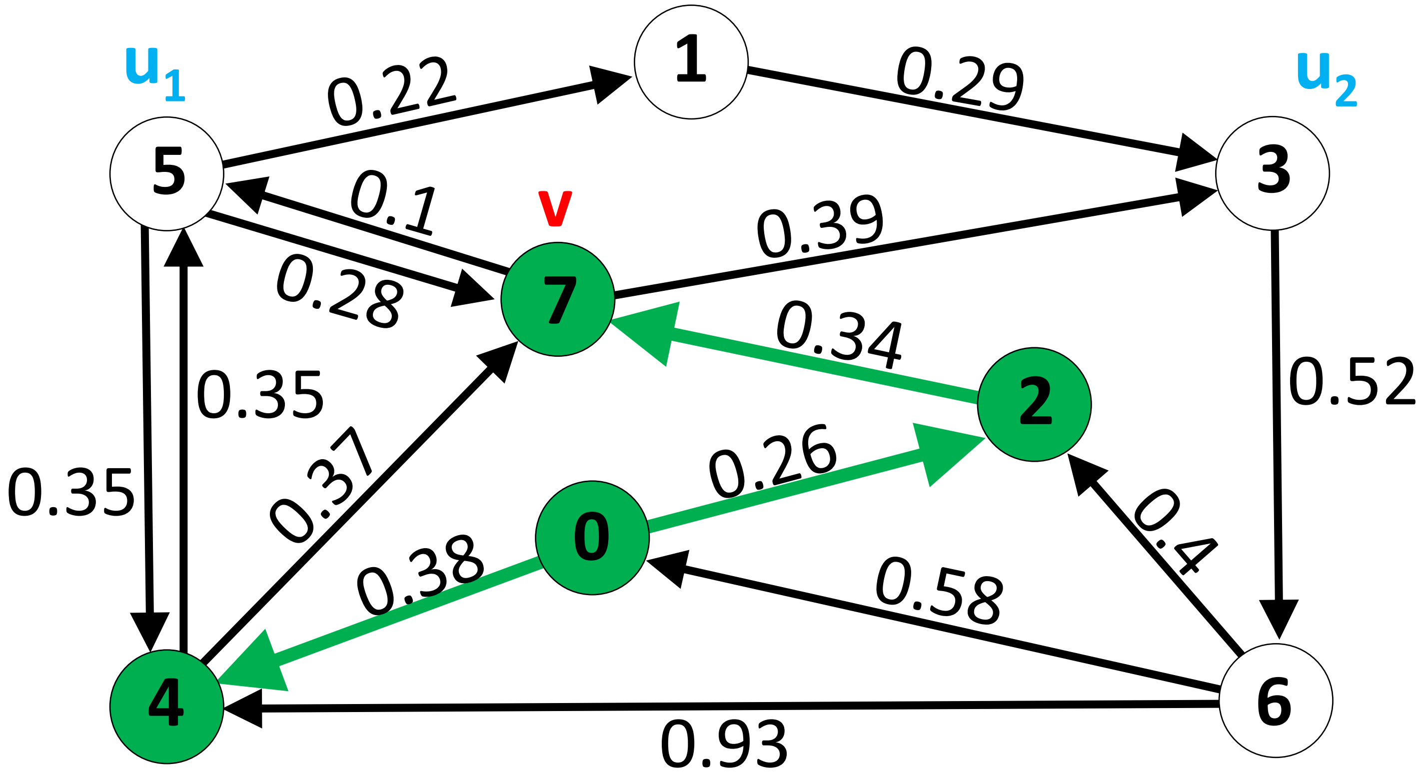
0	-
1	5
2	0
3	7
4	0
5	7
6	3
7	2

Priority queue

vertex (distance)

Rule: When processing vertex v, only add/modify queue for neighbor u if and only if:

`distance[v] + weight(v, u) < distance[u]`



	Distance from 0
0	0
1	∞
2	0.26
3	0.99
4	0.38
5	0.73
6	∞
7	0.60

	Previous vertex
0	-
1	
2	0
3	7
4	0
5	4
6	
7	2

Priority queue

- 5 (0.73)
- 3 (0.99)

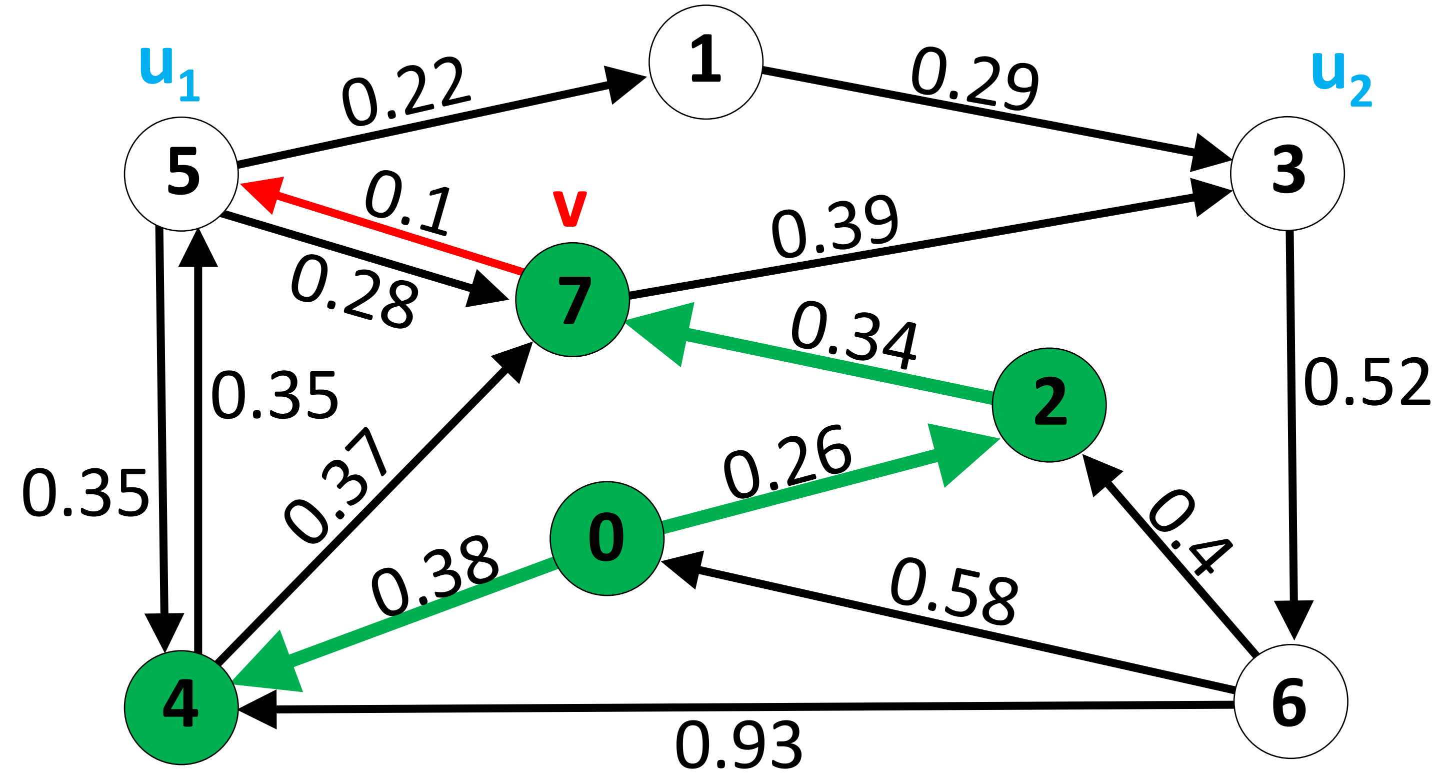
vertex (distance)

`PriortityVertex` Objects

Rule: When processing vertex v, only add/modify queue for neighbor u if and only if:

`distance[v] + weight(v, u) < distance[u]`

0.60 + 0.1 < 0.73



	Distance from 0		Previous vertex	
0	0		-	
1	∞			
2	0.26		0	
3	0.99		7	
4	0.38		0	
5	~~0.73~~	0.70	4	7
6	∞			
7	0.60		2	

Priority queue

- ~~5 (0.73)~~
- 3 (0.99)
- 5 (0.70)

vertex (distance)

PriortityVertex Objects

<u>Rule:</u> When processing vertex v, only add/modify queue for neighbor u if and only if:

`distance[v] + weight(v, u) < distance[u]`

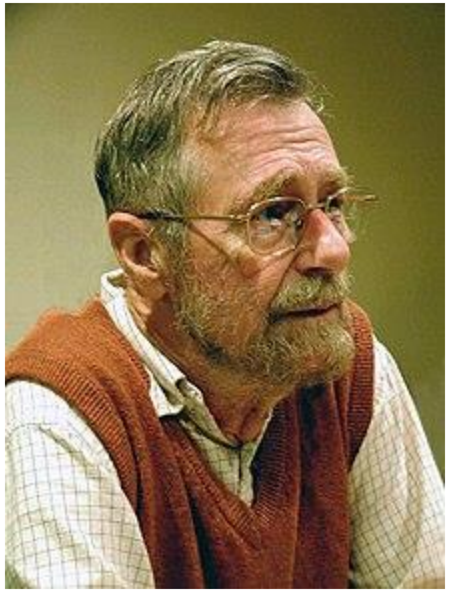
Dijkstra’s Algorithm

Running Time: **O(E · log(V))***

E = # of edges

V = # of vertices

* Varies depending on implementation and representation



Edsger Wybe Dijkstra
11 May 1930 – 6 August 2002

Proposition R. Dijkstra’s algorithm solves the single-source shortest-paths problem in edge-weighted digraphs with nonnegative weights.

Proof: If v is reachable from the source, every edge `v->w` is relaxed exactly once, when v is relaxed, leaving `distTo[w] <= distTo[v] + e.weight()`. This inequality holds until the algorithm completes, since `distTo[w]` can only decrease (any relaxation can only decrease a `distTo[]` value) and `distTo[v]` never changes (because edge weights are nonnegative and we choose the lowest `distTo[]` value at each step, no subsequent relaxation can set any `distTo[]` entry to a lower value than `distTo[v]`). Thus, after all vertices reachable from s have been added to the tree, the shortest-paths optimality conditions hold, and PROPOSITION P applies.

Proposition R (continued). Dijkstra’s algorithm uses extra space proportional to V and time proportional to $E \log V$ (in the worst case) to compute the SPT rooted at a given source in an edge-weighted digraph with E edges and V vertices.

Proof: Same as for Prim’s algorithm (see PROPOSITION N).

Proposition N (continued). Kruskal’s algorithm uses space proportional to E and time proportional to $E \log E$ (in the worst case) to compute the MST of an edge-weighted connected graph with E edges and V vertices.

Proof: The implementation uses the priority-queue constructor that initializes the priority queue with all the edges, at a cost of at most E compares (see SECTION 2.4). After the priority queue is built, the argument is the same as for Prim’s algorithm. The number of edges on the priority queue is at most E, which gives the space bound, and the cost per operation is at most $2 \lg E$ compares, which gives the time bound. Kruskal’s algorithm also performs up to E `find()` and V `union()` operations, but that cost does not contribute to the $E \log E$ order of growth of the total running time (see SECTION 1.5).

A Star

A Star or **A*** is another algorithm that will compute the shortest path in a graph

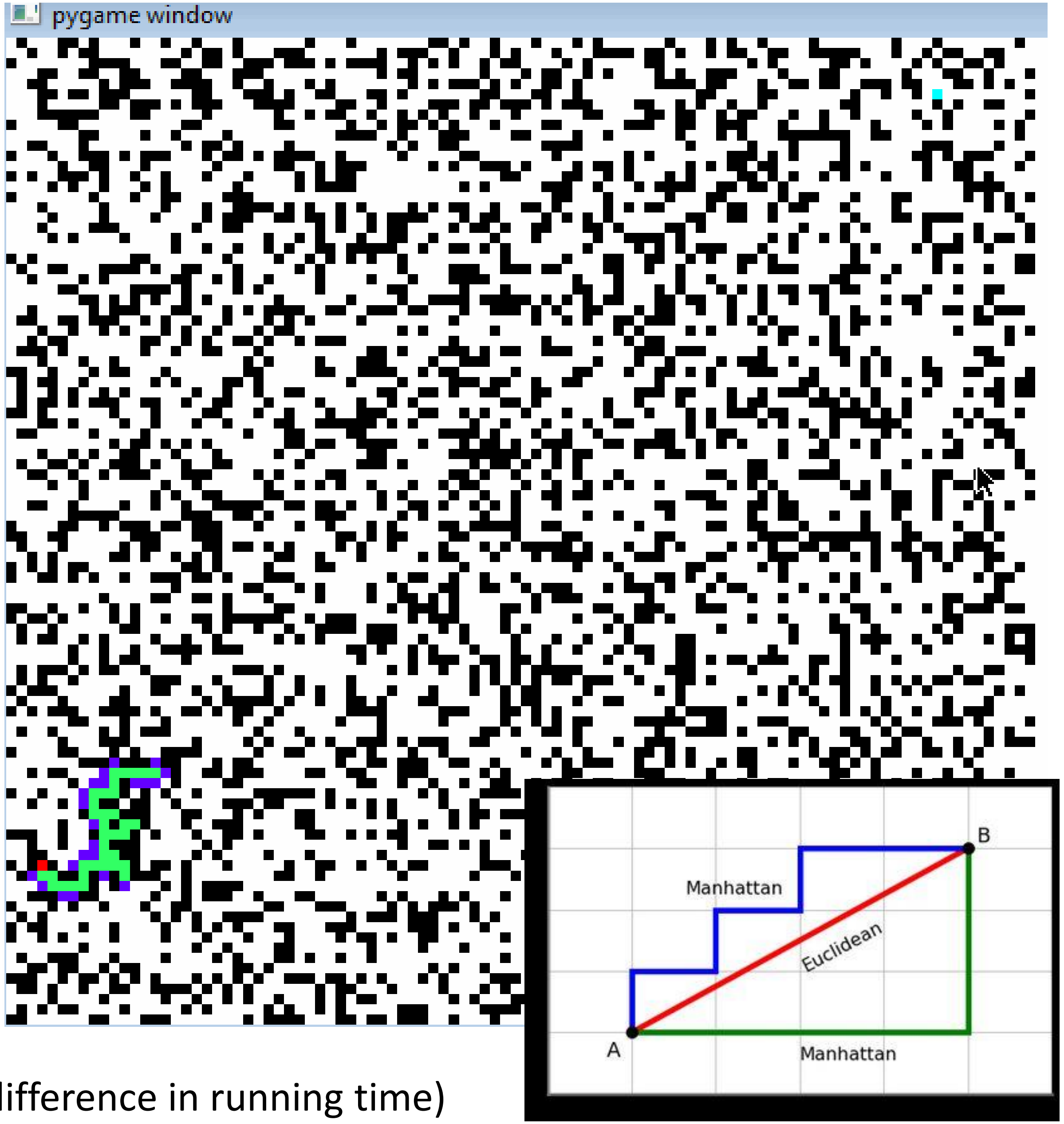
In **Dijkstra's Algorithm** we select the least-cost unvisited node, and we compute the shortest path to all other nodes

In **A***, we select the node that is the shortest distance away from the target, and does not compute the shortest path to all other nodes

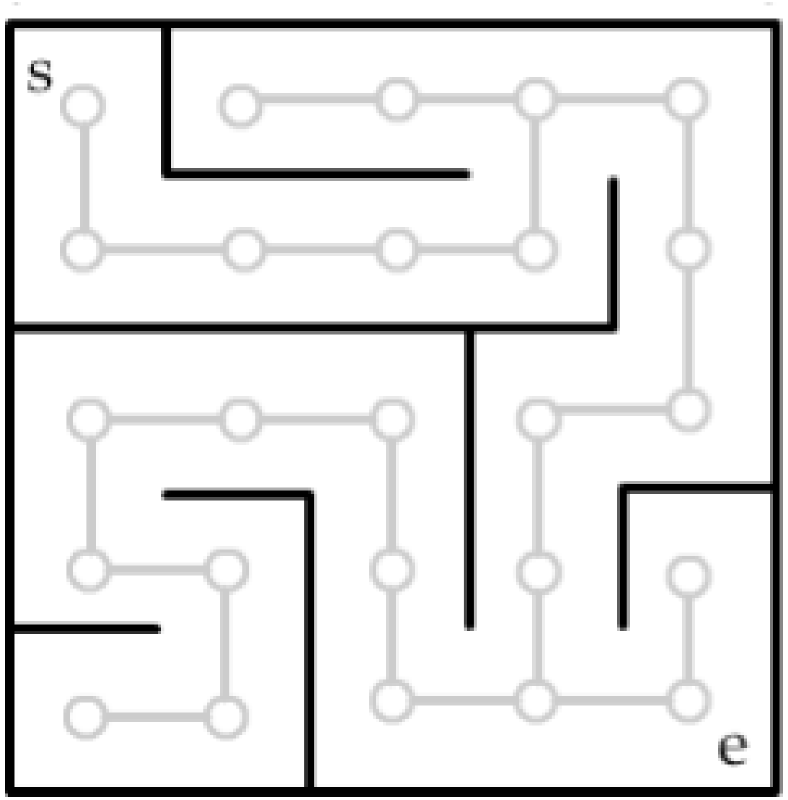
In A* we use a **heuristic** to make decisions

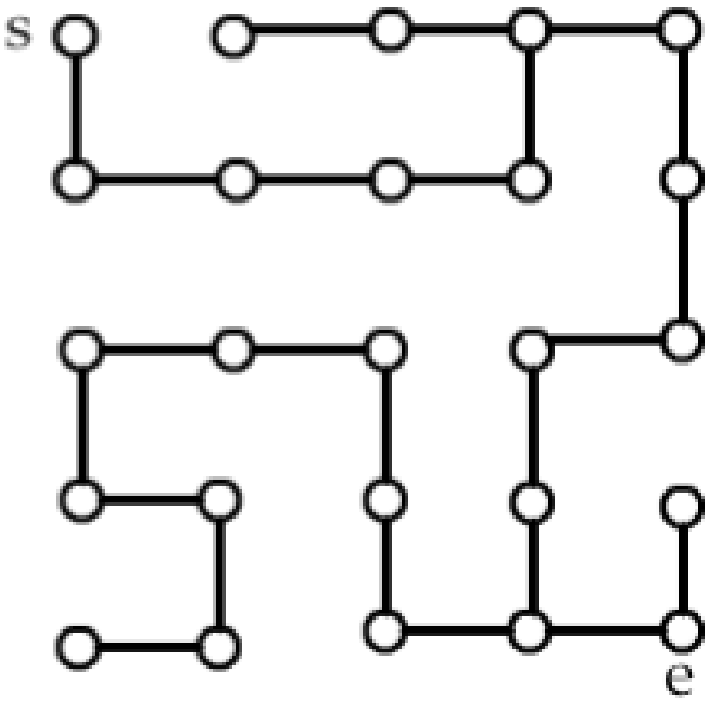
Euclidean Distance heuristic

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$



(No difference in running time)





We can represent a maze using graphs!

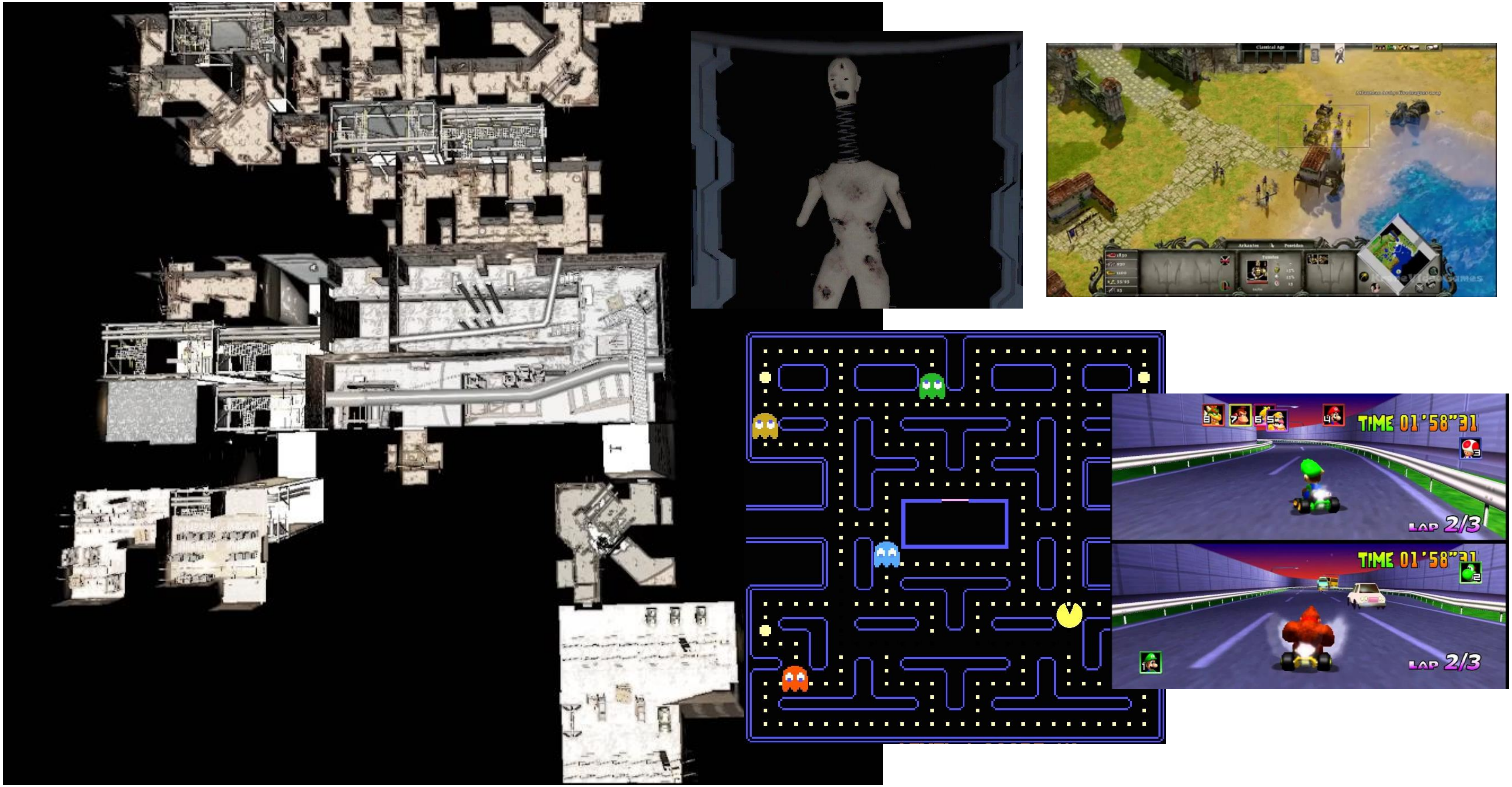
Creating Mazes with Depth First

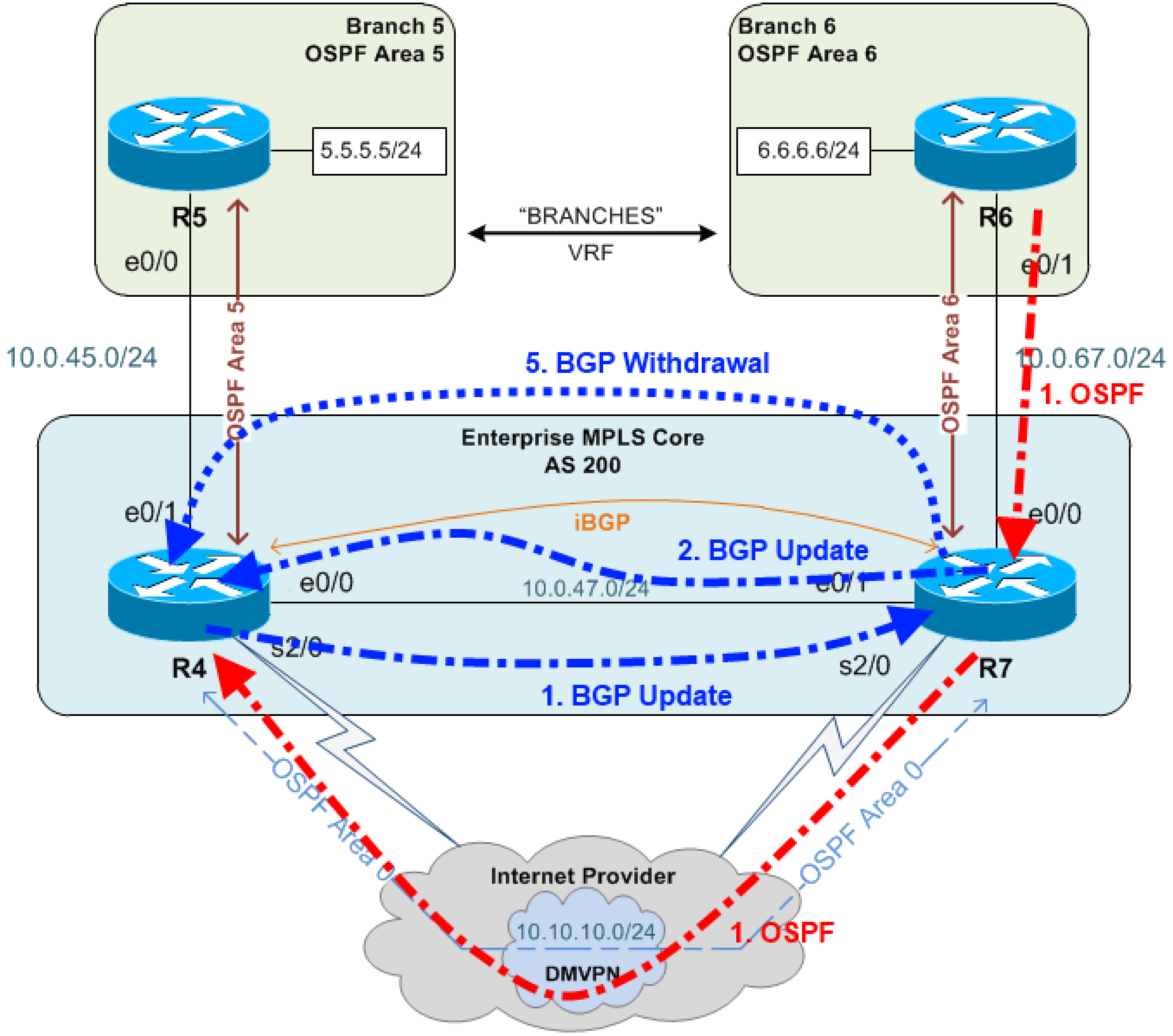
https://www.youtube.com/watch?v=e5zDG4JIsyg

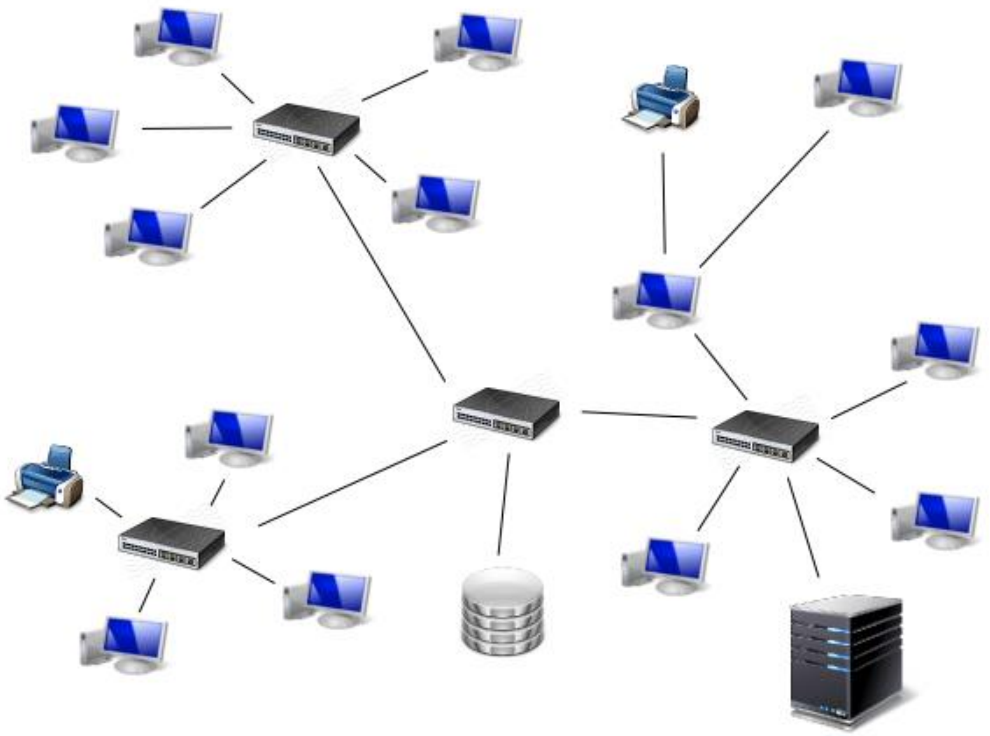
Shortest Path Algorithms on Maze

https://clementmihailescu.github.io/Pathfinding-Visualizer/

Applications of Shortest Path?


TIME 01'58"31
LAP 2/3
LAP 2/3

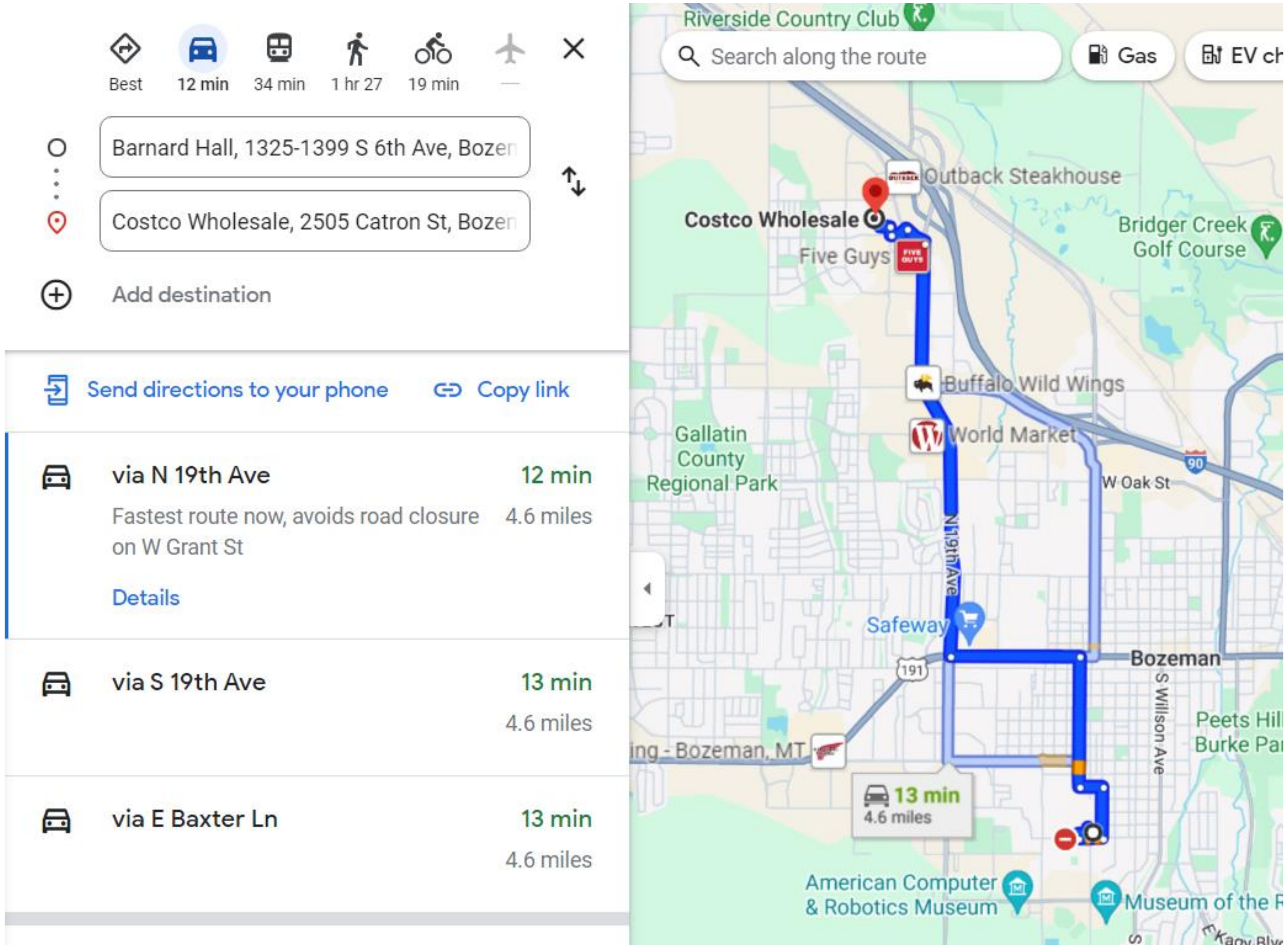




Dijkstra's Algorithm is used for network routing

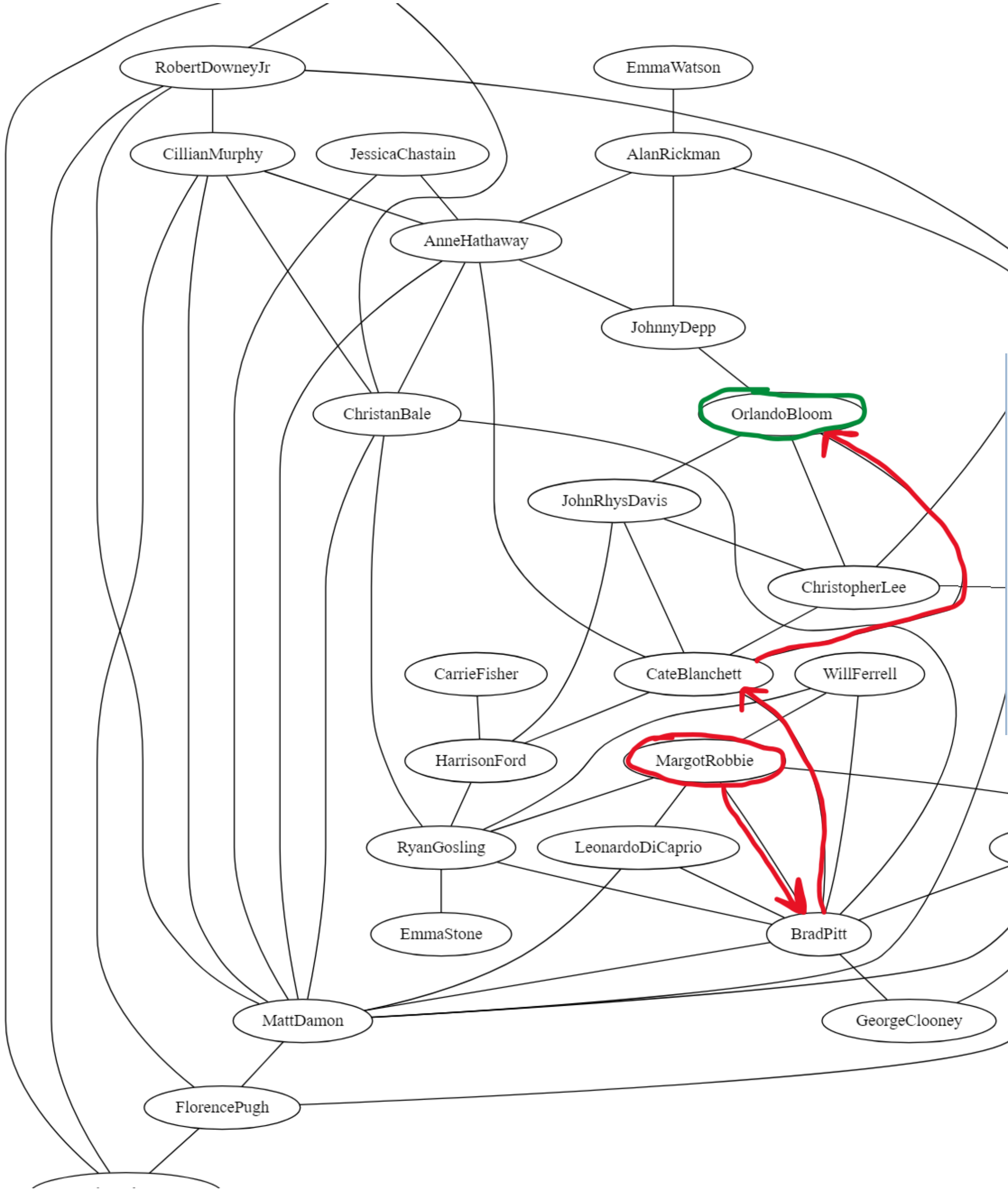
The **OSPF** Protocol

Finding shortest path on a map



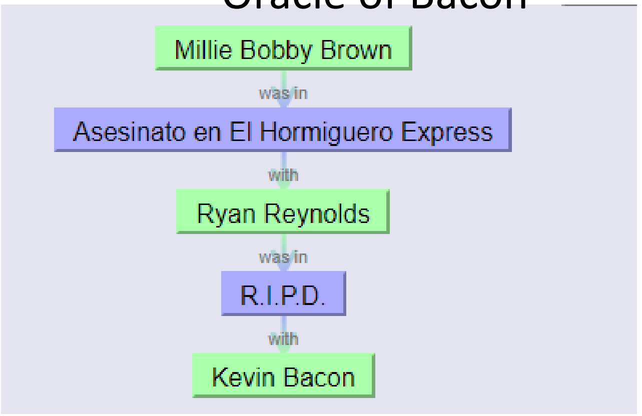
Sending drones or robots on the shortest path

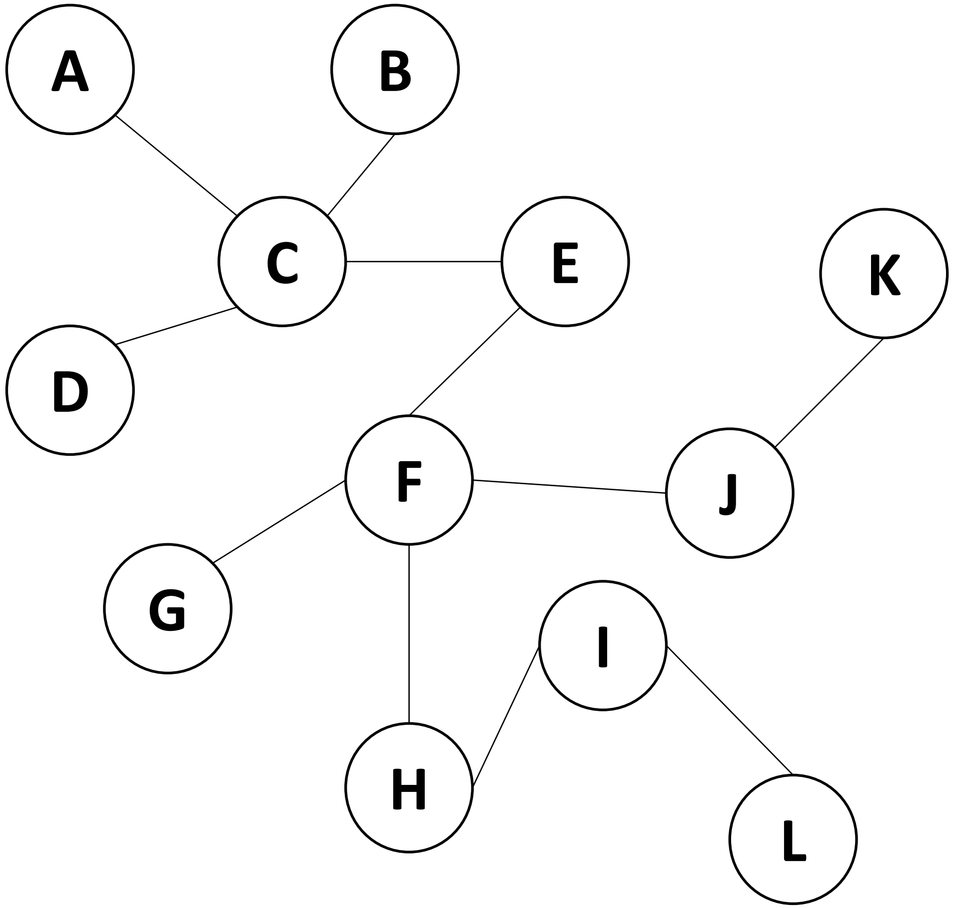
Finding Shortest Path between actors




```
Enter starting actor:
Margot Robbie
Enter destination actor:
Orlando Bloom
Margot Robbie acted with Brad Pitt in Once Upon a Time in Hollywood
Brad Pitt acted with Cate Blanchett in The Curious Case of Benjamin Button
Cate Blanchett acted with Orlando Bloom in Fellowship of the Ring
Number of hops: 3
```

Oracle of Bacon

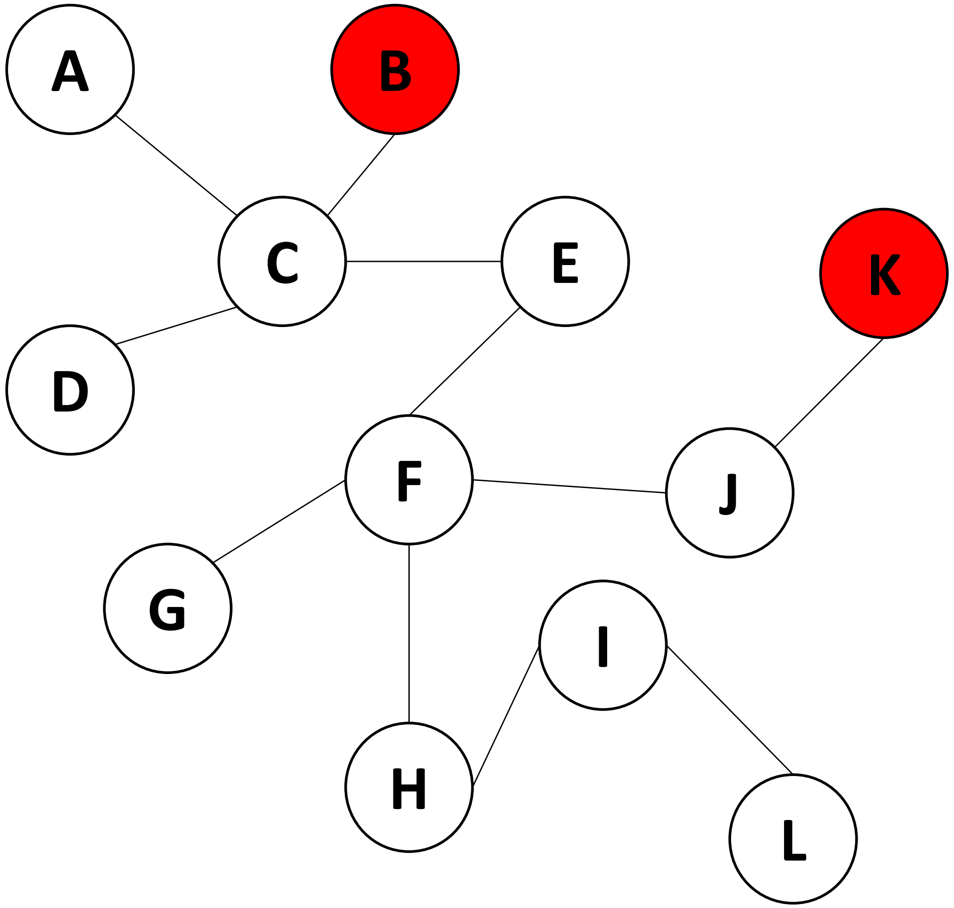




Consider an **Acyclic** graph (a graph with no cycles) (a “tree”)

Observation:

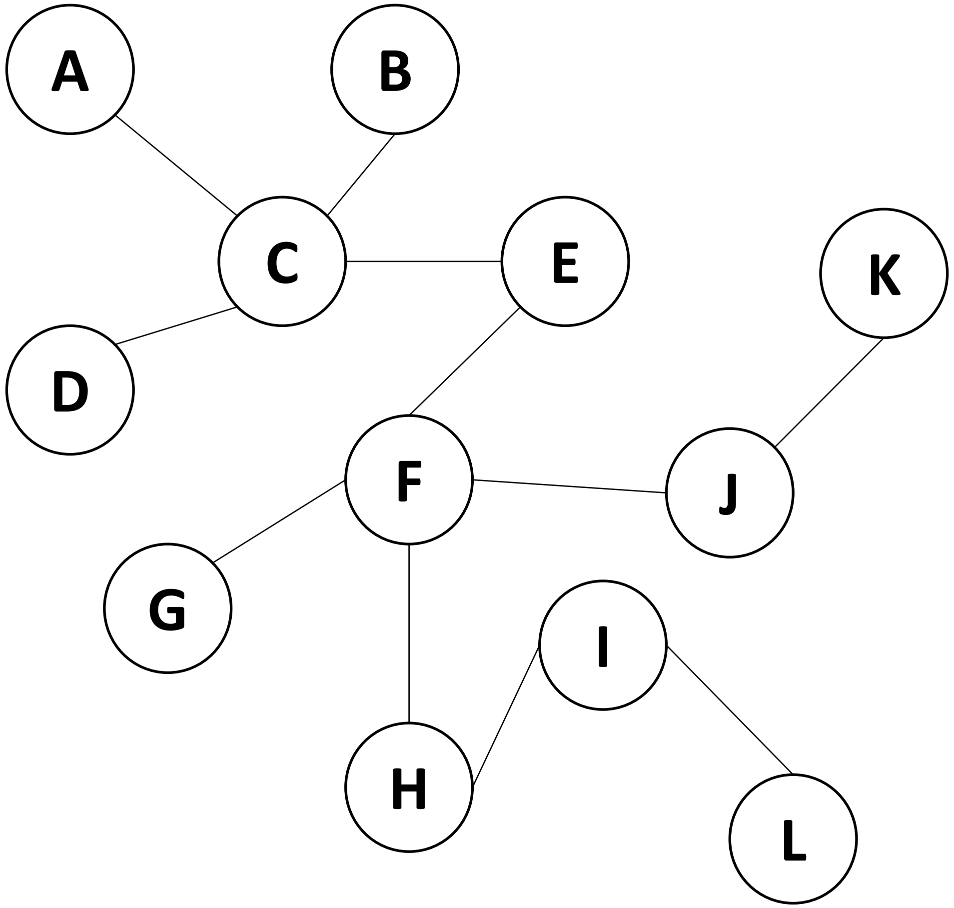
Pick any two vertices (V1, V2). There is **only one possible path** that goes from V1 to V2 (and vice versa)



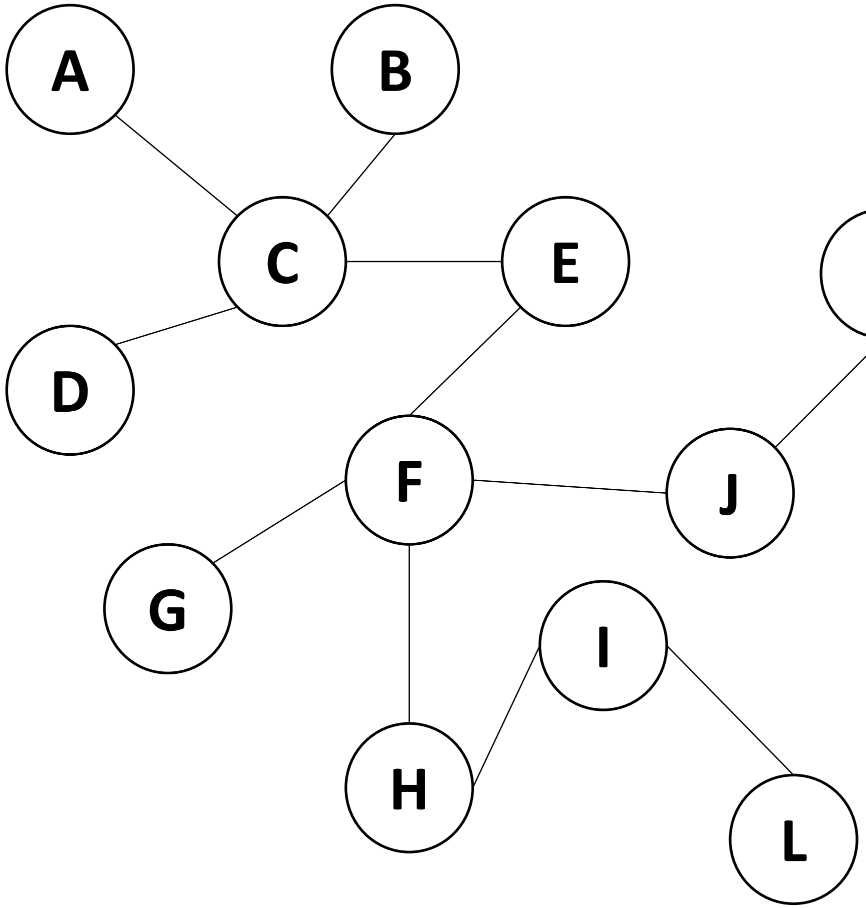
Consider an **Acyclic** graph (a graph with no cycles) (a “tree”)

Observation:

Pick any two vertices (V1, V2). There is **only one possible path** that goes from V1 to V2 (and vice versa)



Longest Path ?

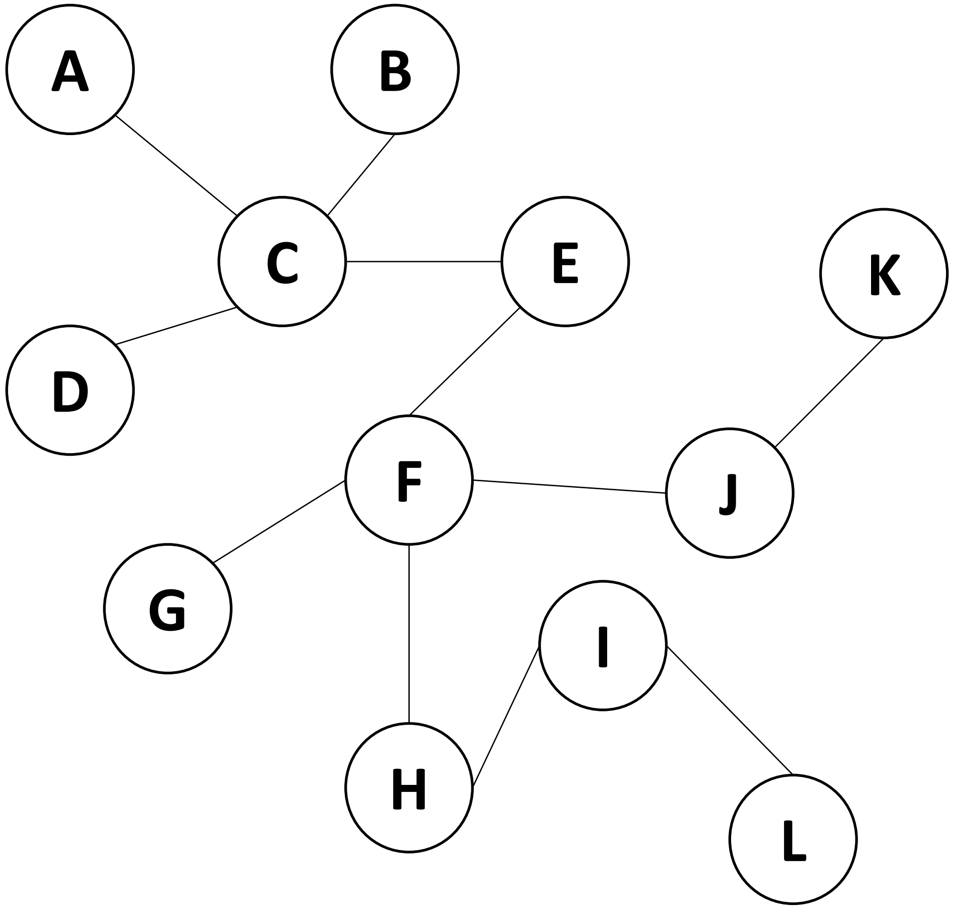


Select any vertex **v1**

```
HashMap<String, LinkedList<String>> adjList
```

```
{ J: [K], A:[C], C:[A,D,B], F: [E, G, H, J], … }
```

HashMaps are unordered, and there is no way to pick a key at an "index"



Select any vertex **v1**

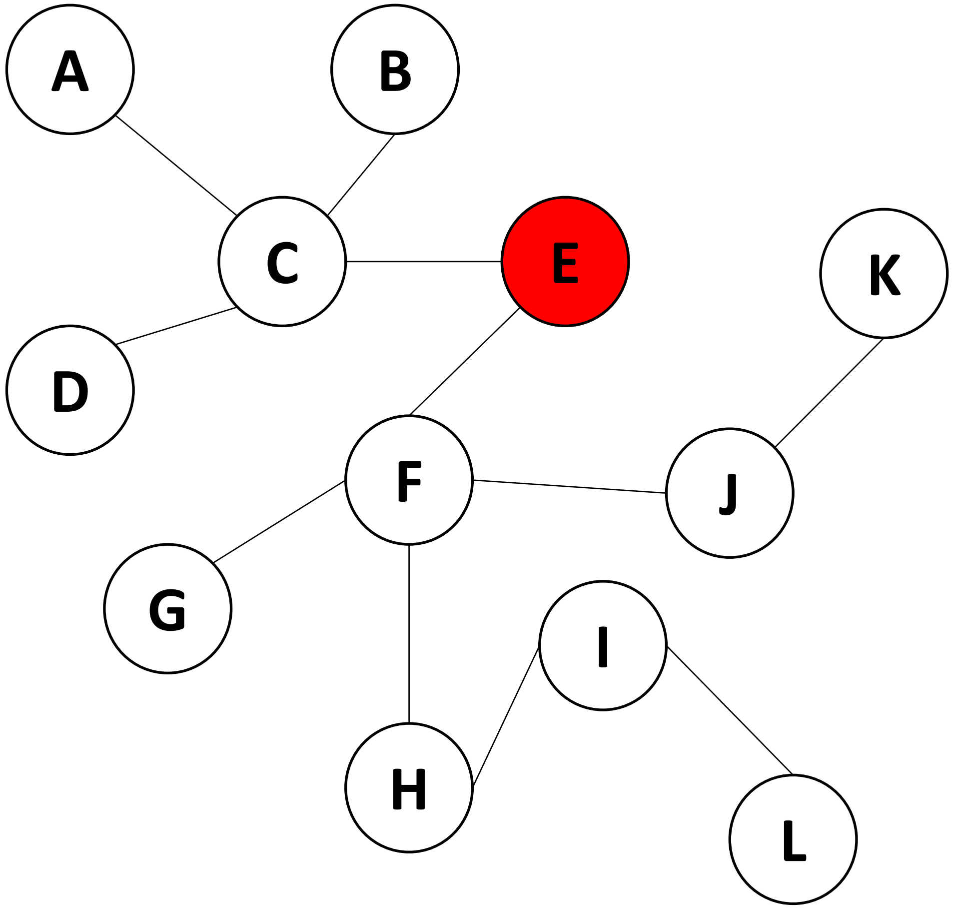
```
HashMap<String, LinkedList<String>> adjList
```

```
{ J: [K], A:[C], C:[A,D,B], F: [E, G, H, J], … }
```

HashMaps are unordered, and there is no way to pick a key at an “index”

Just return the first key when iterating over it

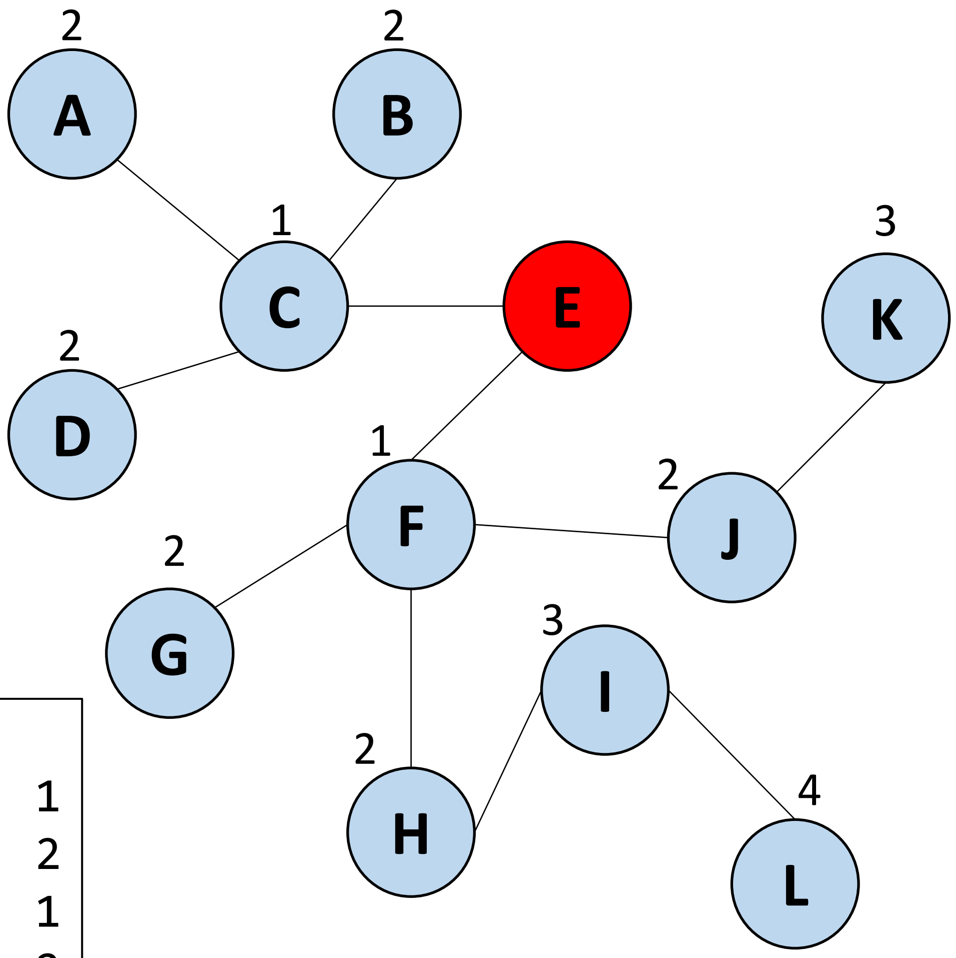
```
for(String key: adjList){
        return key;
}
```



Select any vertex **v1**

Do Breadth First Traversal from vertex **v1** to all other vertices

While doing breadth first, keep track of the distance from vertex **v1**

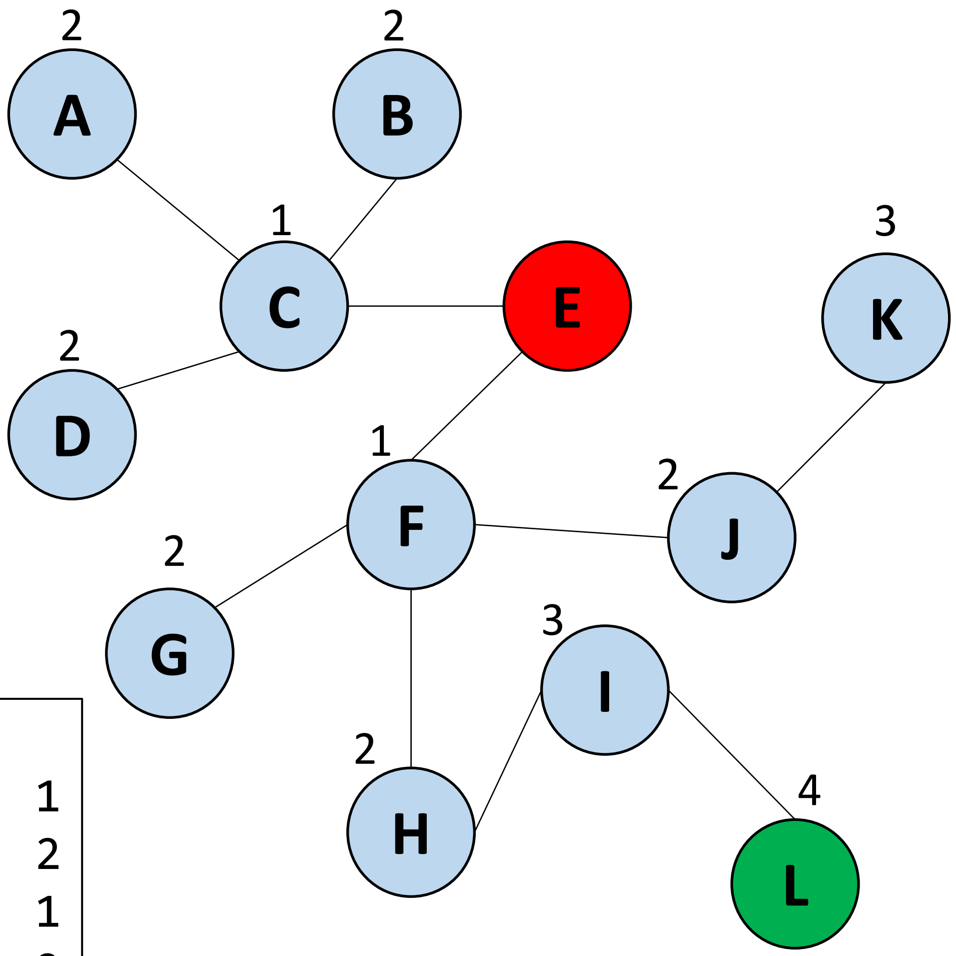



```
{
C: 1
J: 2
F: 1
D: 2
G: 2
L: 4
…

}
```

Select any vertex **v1**

Do Breadth First Traversal from vertex **v1** to all other vertices

While doing breadth first, keep track of the distance from vertex **v1** (store in some kind of data structure)



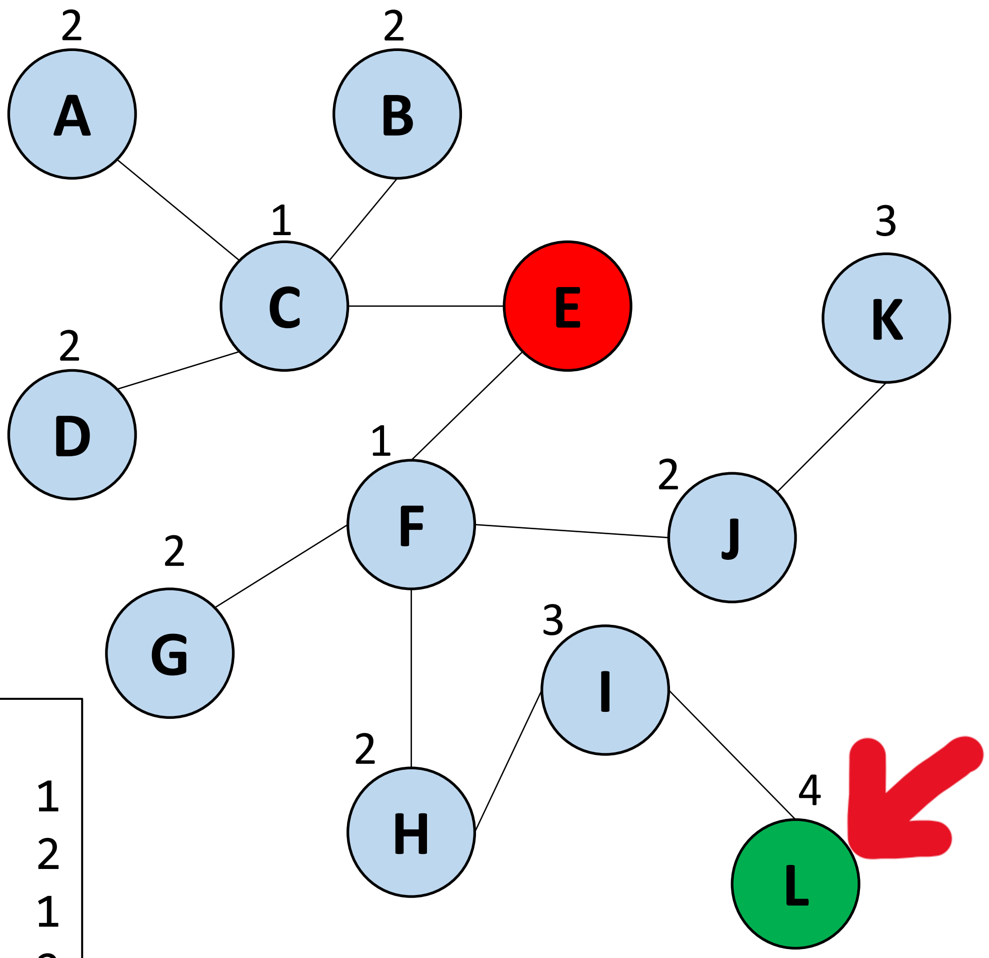

```
{
C: 1
J: 2
F: 1
D: 2
G: 2
L: 4
…

}
```

Select any vertex **v1**

Do Breadth First Traversal from vertex **v1** to all other vertices

While doing breadth first, keep track of the distance from vertex **v1** (store in some kind of data structure)

Select the node that was the furthest away, **v2**



Select any vertex **v1**

Do Breadth First Traversal from vertex **v1** to all other vertices

While doing breadth first, keep track of the distance from vertex **v1** (store in some kind of data structure)

Select the node that was the furthest away, **v2**

```
{
C: 1
J: 2
F: 1
D: 2
G: 2
L: 4
…

}
```

v2 must be an endpoint on the longest path because …

Choose an arbitrary tree node s. Assume $u, v \in V(G)$ are nodes with $d(u,v) = diam(G)$. Assume further that the algorithm finds a node x starting at s first, some node y starting at x next. wlog $d(s,u) \geq d(s,v)$. note that $d(s,x) \geq d(s,y)$ must hold, unless the algorithm's first stage wouldn't end up at x. We will see that $d(x,y) = d(u,v)$.

The most general configuration of all nodes involved can be seen in the following pseudo-graphics (possibly $s = z_{uv}$ or $s = z_{xy}$ or both):

```
(u)                                              (x)
  \                                              /
   \                                            /
    \                                          /
    ( z_uv )---------( s )----------( z_xy )
    /                                          \
   /                                            \
  /                                              \
(v)                                              (y)
```

we know that:

1. $d(z_{uv}, y) \leq d(z_{uv}, v)$. otherwise $d(u,v) < diam(G)$ contradicting the assumption.
2. $d(z_{uv}, x) \leq d(z_{uv}, u)$. otherwise $d(u,v) < diam(G)$ contradicting the assumption.
3. $d(s, z_{xy}) + d(z_{xy}, x) \geq d(s, z_{uv}) + d(z_{uv}, u)$, otherwise stage 1 of the algorithm wouldn't have stopped at x.
4. $d(z_{xy}, y) \geq d(v, z_{uv}) + d(z_{uv}, z_{xy})$, otherwise stage 2 of the algorithm wouldn't have stopped at y.

1) and 2) imply

$$\begin{aligned} d(u,v) &= d(z_{uv}, v) + d(z_{uv}, u) \\ &\geq d(z_{uv}, x) + d(z_{uv}, y) = d(x,y) + 2\,d(z_{uv}, z_{xy}) \\ &\geq d(x,y) \end{aligned}$$

3) and 4) imply

$$\begin{aligned} & d(z_{xy}, y) + d(s, z_{xy}) + d(z_{xy}, x) \\ &\geq d(s, z_{uv}) + d(z_{uv}, u) + d(v, z_{uv}) + d(z_{uv}, z_{xy}) \end{aligned}$$

equivalent to

$$\begin{aligned} d(x,y) &= d(z_{xy}, y) + d(z_{xy}, x) \\ &\geq 2 * d(s, z_{uv}) + d(v, z_{uv}) + d(u, z_{uv}) \\ &\geq d(u,v) \end{aligned}$$

therefore $d(u,v) = d(x,y)$.

analogue proofs hold for the alternative configurations

```
                (u)                       (x)
                  \                       /
                   \                     /
( s )---------( z_uv )----------( z_xy )
                   /                     \
                  /                       \
                (v)                       (y)
```

and

```
                (x)          (u)
                  /            \
                 /              \
( s )---------( z_xy )----------( z_uv )
                 \              /
                  \            /
                (y)          (v)
```

these are all possible configurations. in particular, $x \notin path(s,u), x \notin path(s,v)$ due to the result of stage 1 of the algorithm and $y \notin path(x,u), y \notin path(x,v)$ due to stage 2.

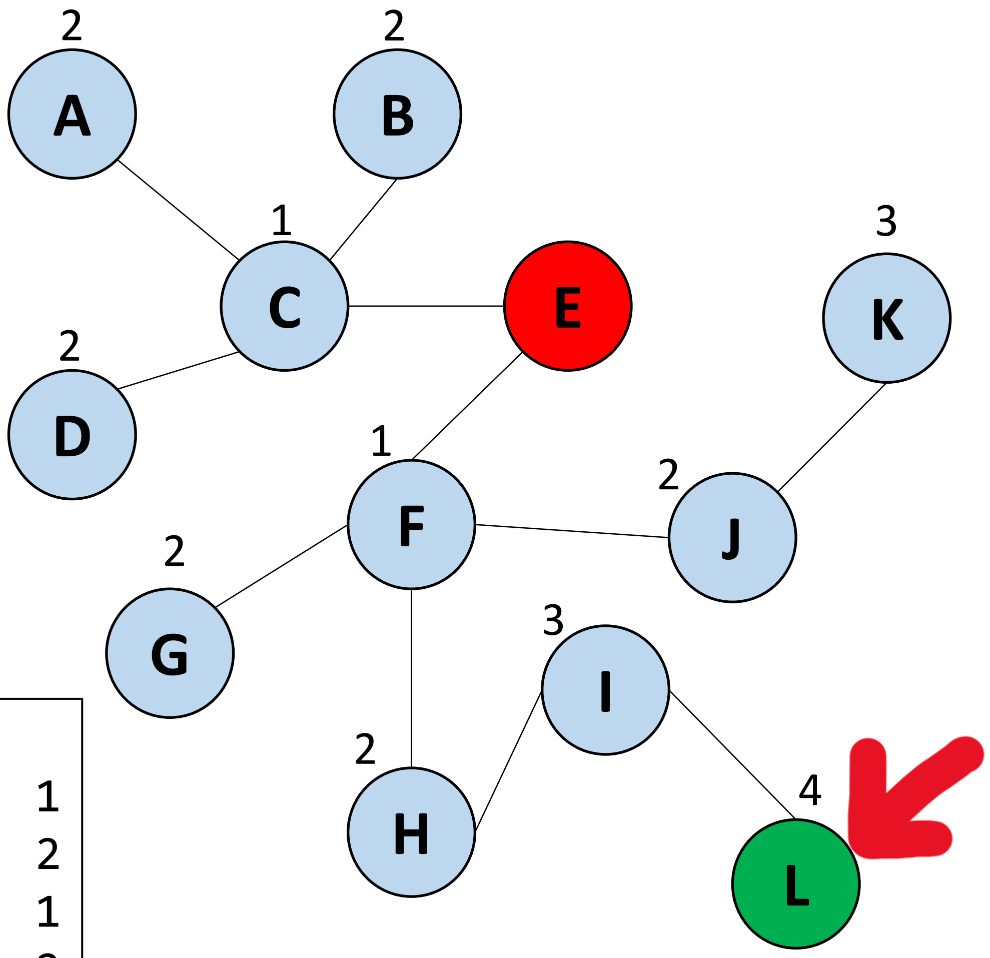
```
{
C: 1
J: 2
F: 1
D: 2
G: 2
L: 4
…

}
```

som

ect t

ay, **v2**

v2 mu … st path because reese told you so




```
{
C: 1
J: 2
F: 1
D: 2
G: 2
L: 4
…

}
```

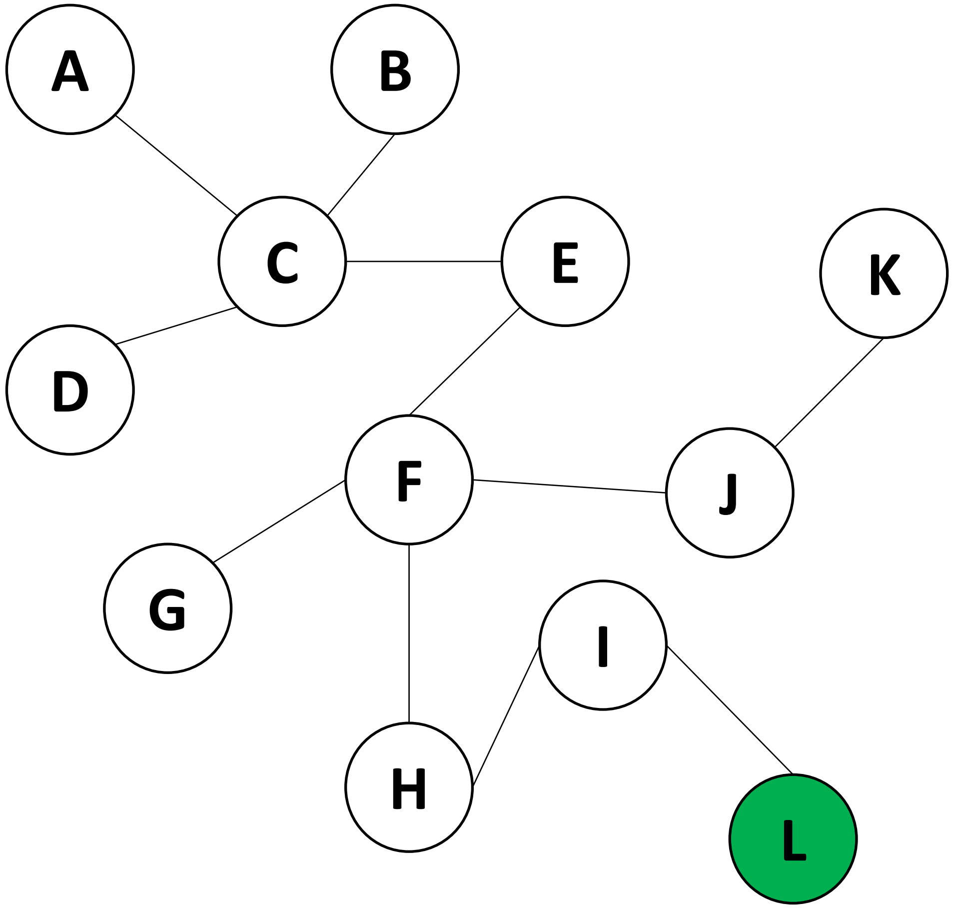
Select any vertex **v1**

Do Breadth First Traversal from vertex **v1** to all other vertices

While doing breadth first, keep track of the distance from vertex **v1** (store in some kind of data structure)

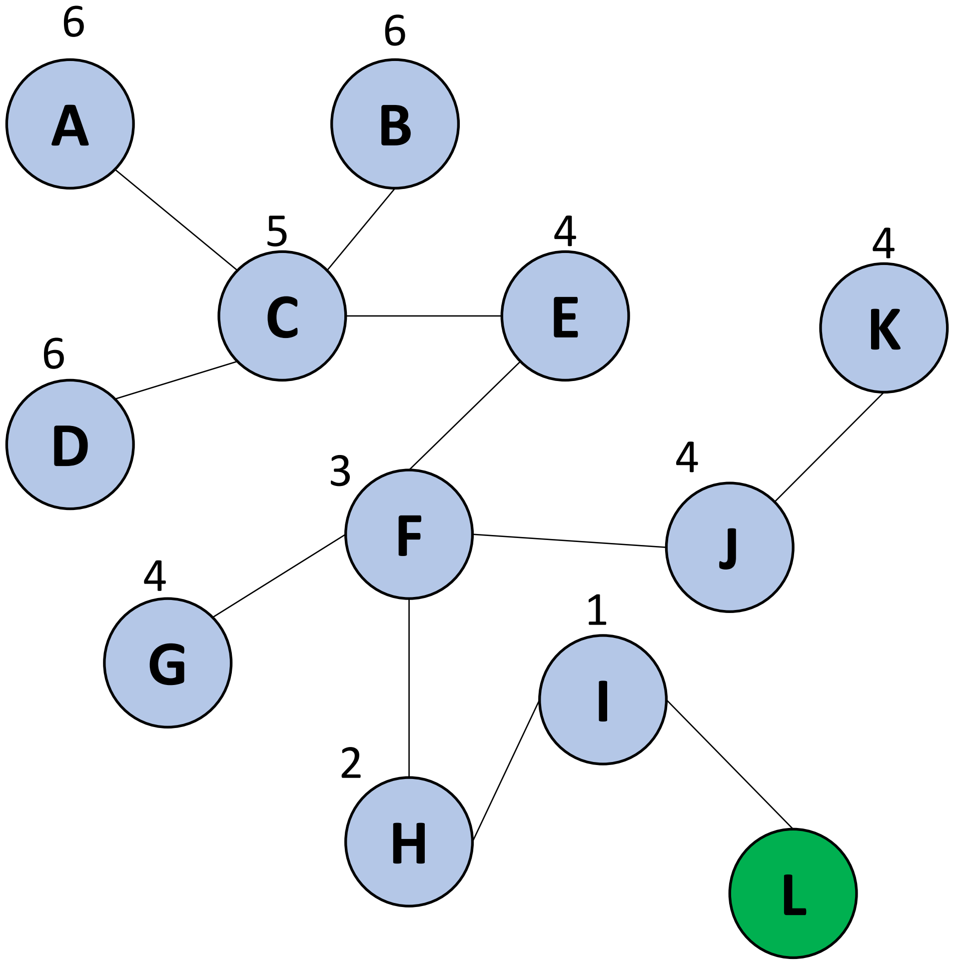
Select the node that was the furthest away, **v2**

v2 must be an endpoint on the longest path because otherwise BFS would have found a deeper vertex



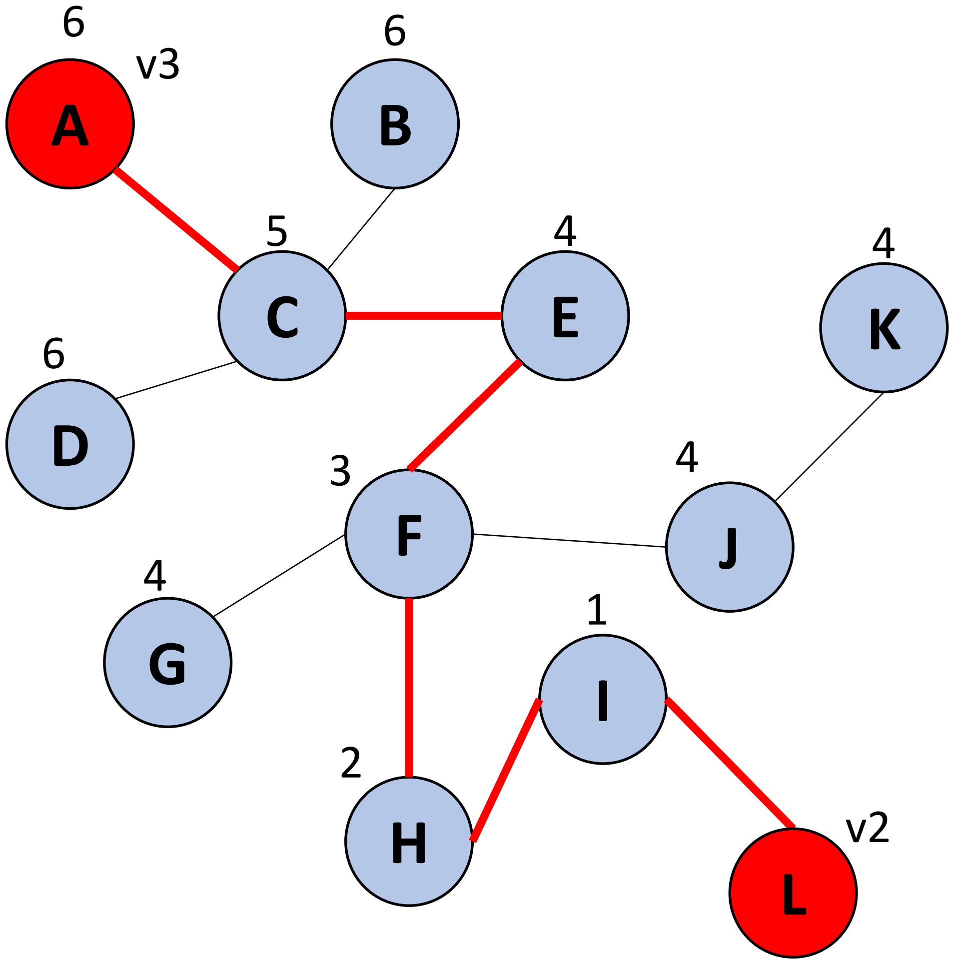
Do breadth first search again, but now starting from **v2**

Keep track of distances from **v2**



Do breadth first search again, but now starting from **v2**

Keep track of distances from **v2**



Do breadth first search again, but now starting from **v2**

Keep track of distances from **v2**

Select the vertex with the longest distance, **v3**

(We have a tie for longest path, so just select one of them)

Breadth First will visit every node, and will always find the farthest away node from some starting point

"Double pass BFS"

Will only work on an acyclic graph

[V2, V3] is the longest path