CSC10004 – Data Structures and Algorithms

Session 08
Graph Structure

Instructors:
Dr. Lê Thanh Tùng

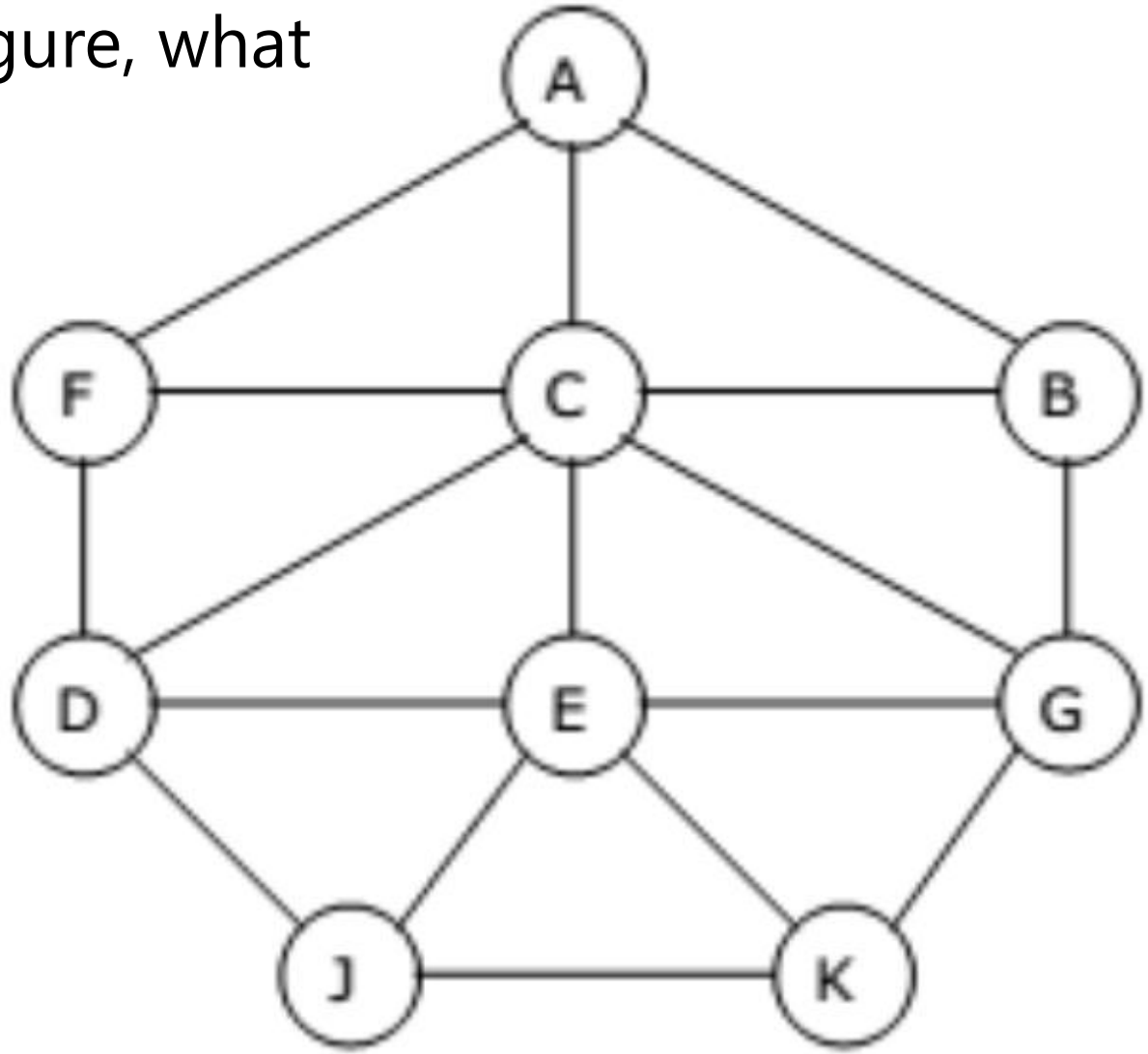
Content

1. Reviews
2. Shortest Path on Unweighted Graphs
3. Dijkstra's Algorithm

Reviews

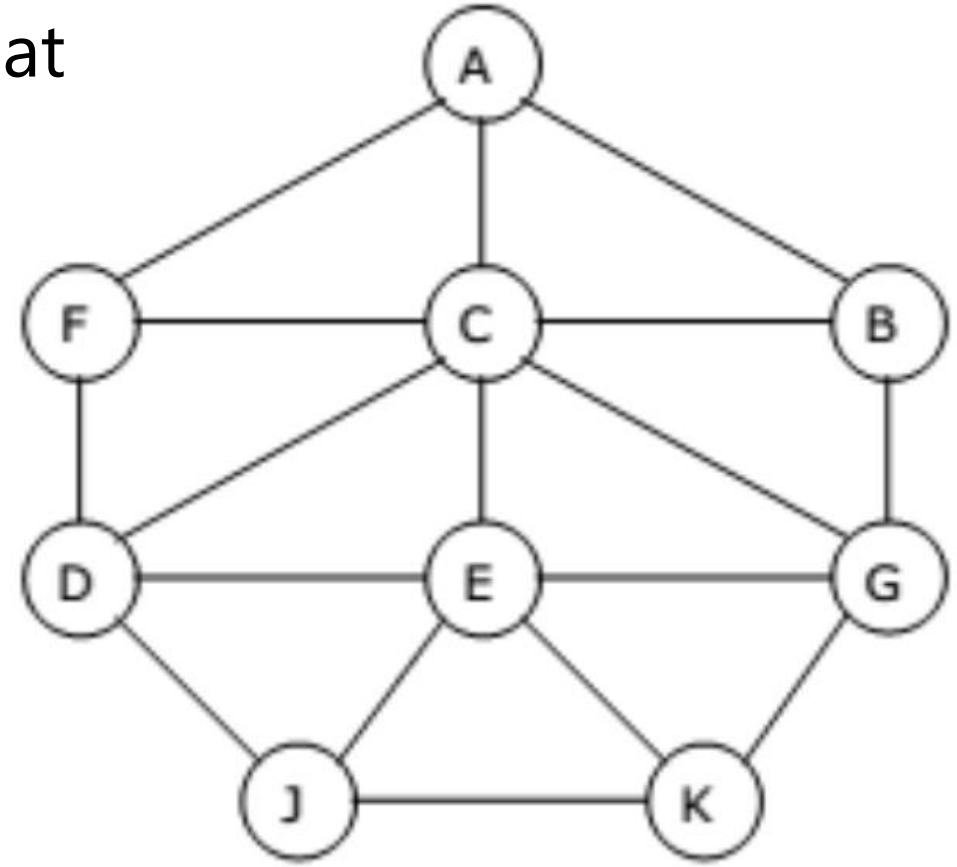
Exercise: Traversal

- Given a graph G in the following figure, what is the result of:
  - DFS from A:
  - BFS from A:



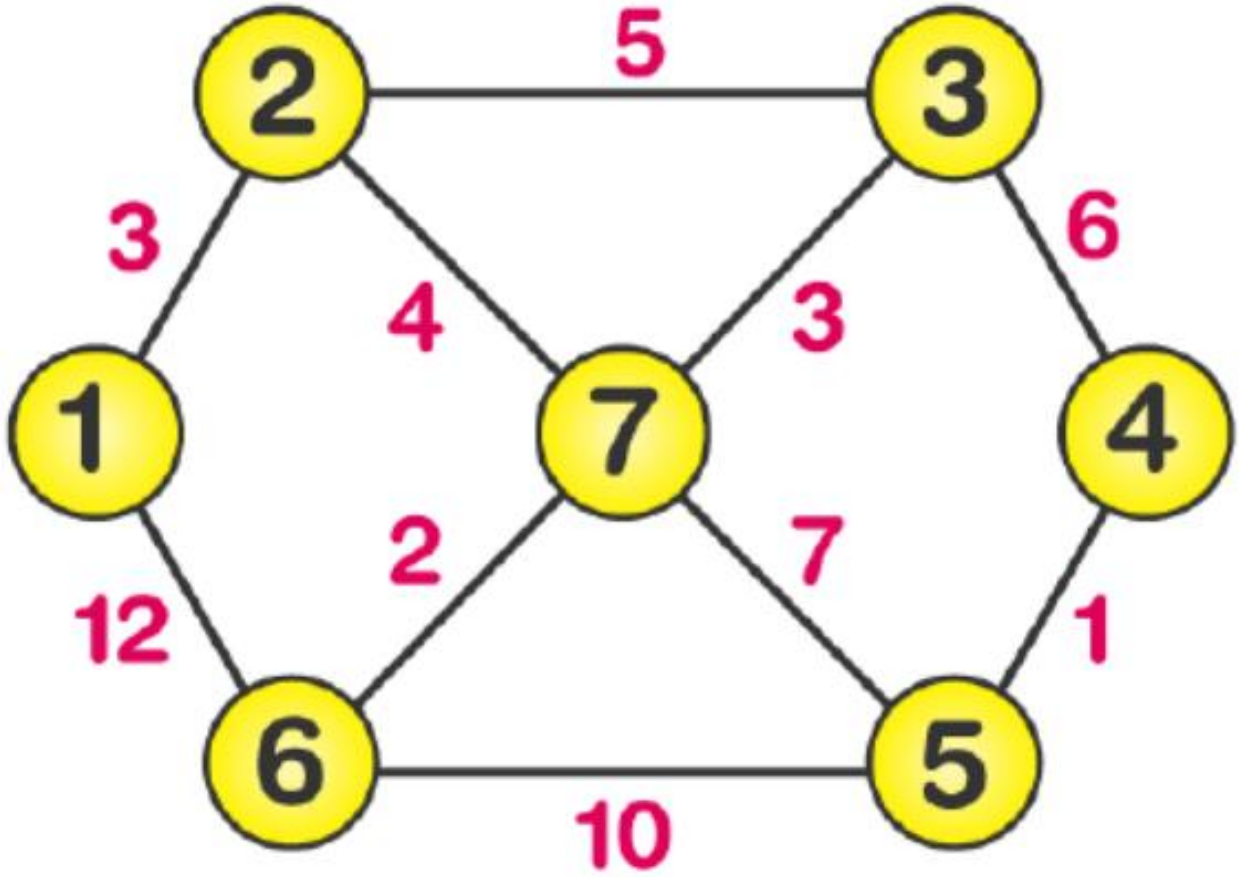
Exercise: Traversal

- Given a graph G in the following figure, what is the result of:
  - DFS from A: A B C D E G K J F
  - BFS from A: A B C F G D E K J



Exercise: Graph Representation

- From a following graph G, write the adjacency matrix of G



Adjacency matrix

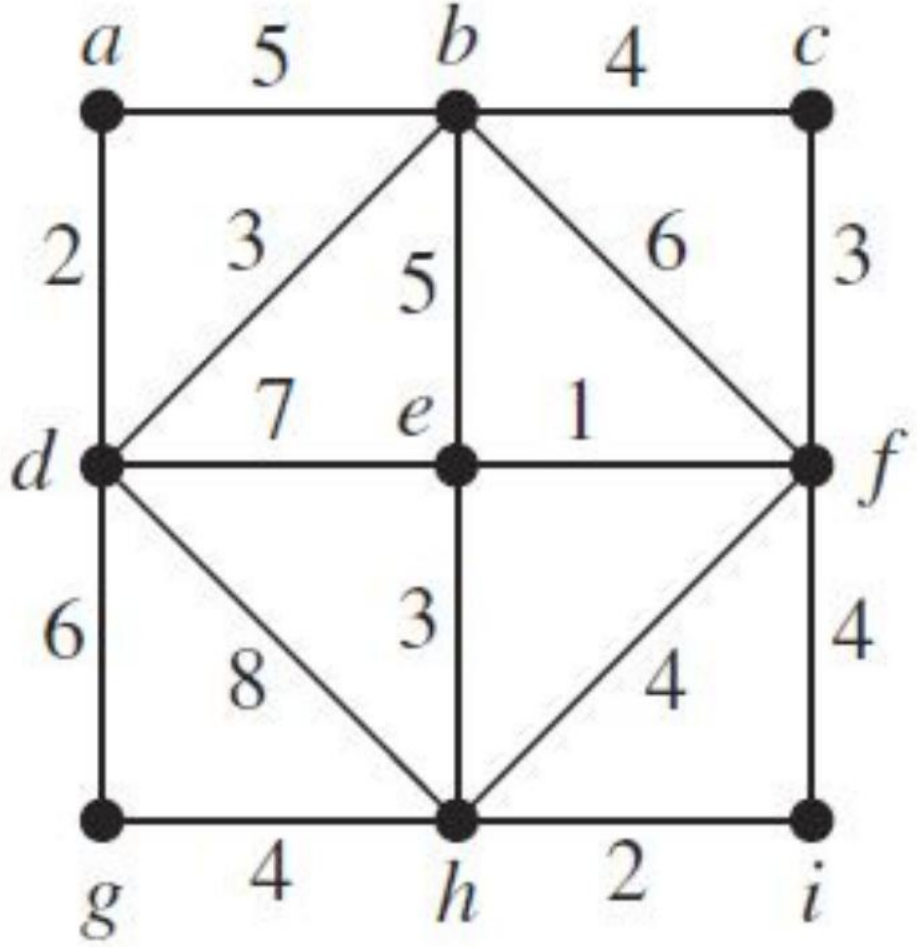
- Advantages:
  - easy to add or remove an edge
  - easy to check if an edge exists
- Disadvantages:
  - the larger the graph is (more nodes), the more memory is needed
  - for a sparse graph, space will be wasted for not storing many edges
  - if a static array is used for the matrix, adding or deleting nodes may not be easy.

Adjacency list

- Advantages:
  - easy to find successors of a node
  - easy to find all neighboring nodes
  - space efficient as it only stores connected nodes
- Disadvantages:
  - it is time consuming to check if an edge is a part of a graph

Exercise

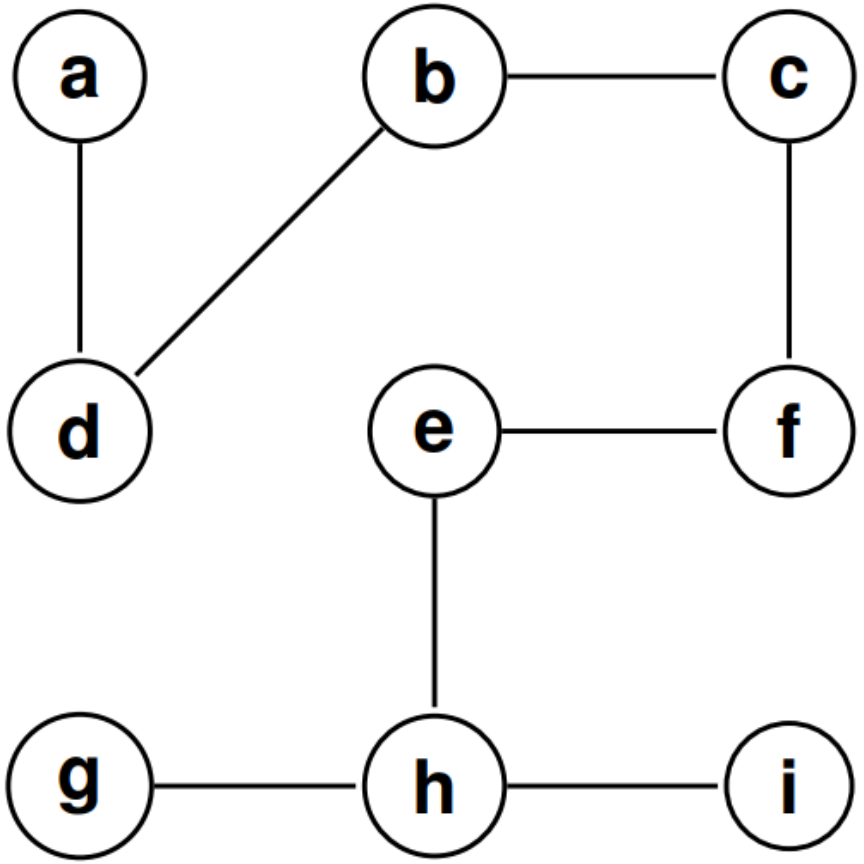
- Starting from e, use Prim's algorithm to find a minimum spanning tree for the given weighted graph



Exercise

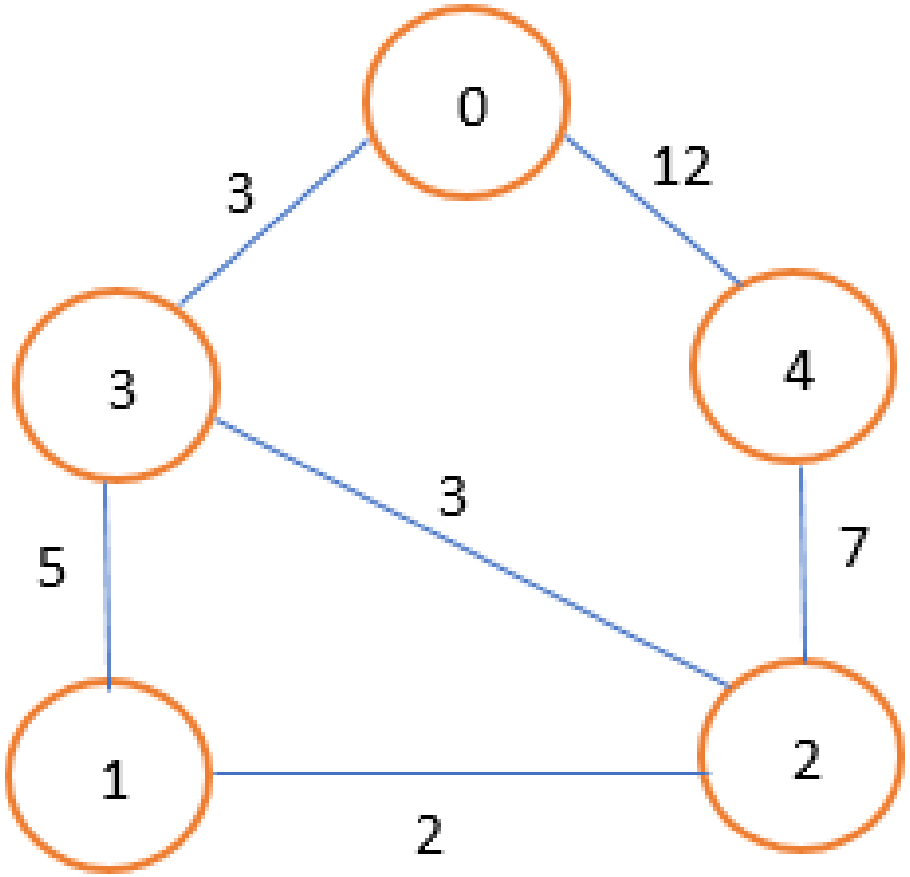
- Use Prim's algorithm to find a minimum spanning tree for the given weighted graph

Choice	Edge	Weight
1	$\{e, f\}$	1
2	$\{c, f\}$	3
3	$\{e, h\}$	3
4	$\{h, i\}$	2
5	$\{b, c\}$	4
6	$\{b, d\}$	3
7	$\{a, d\}$	2
8	$\{g, h\}$	4
		total: 22



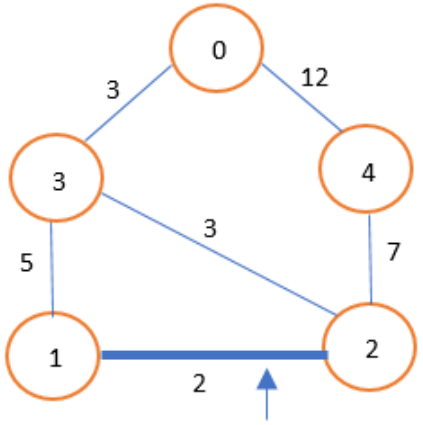
Exercise

- Use Kruskal's algorithm to find a minimum spanning tree for the given weighted graph

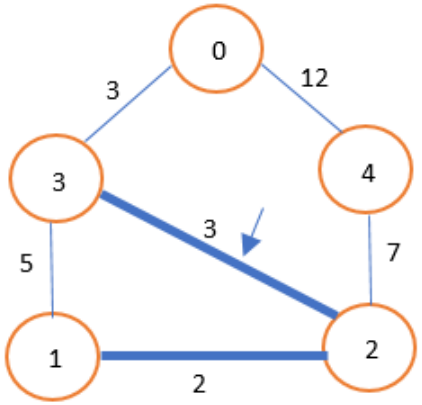


Exercise

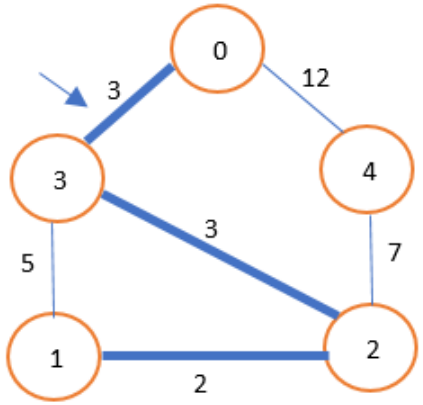
- Use Kruskal's algorithm to find a minimum spanning tree for the given weighted graph



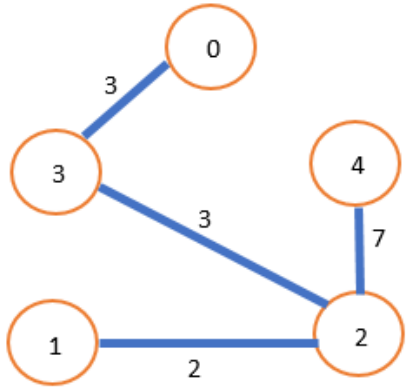
1



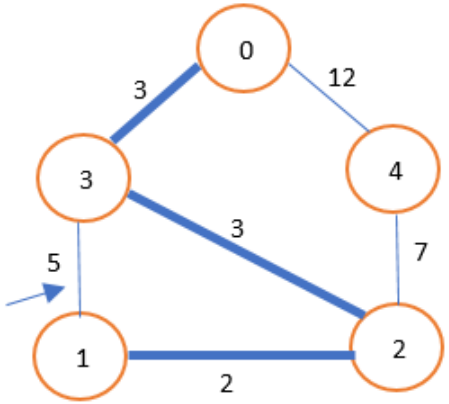
2



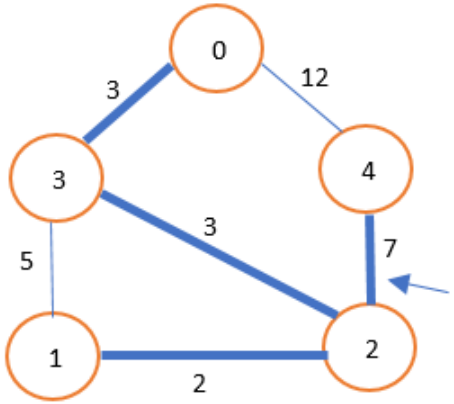
3



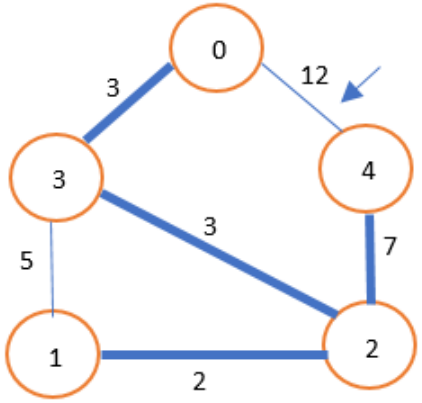
7



4



5



6

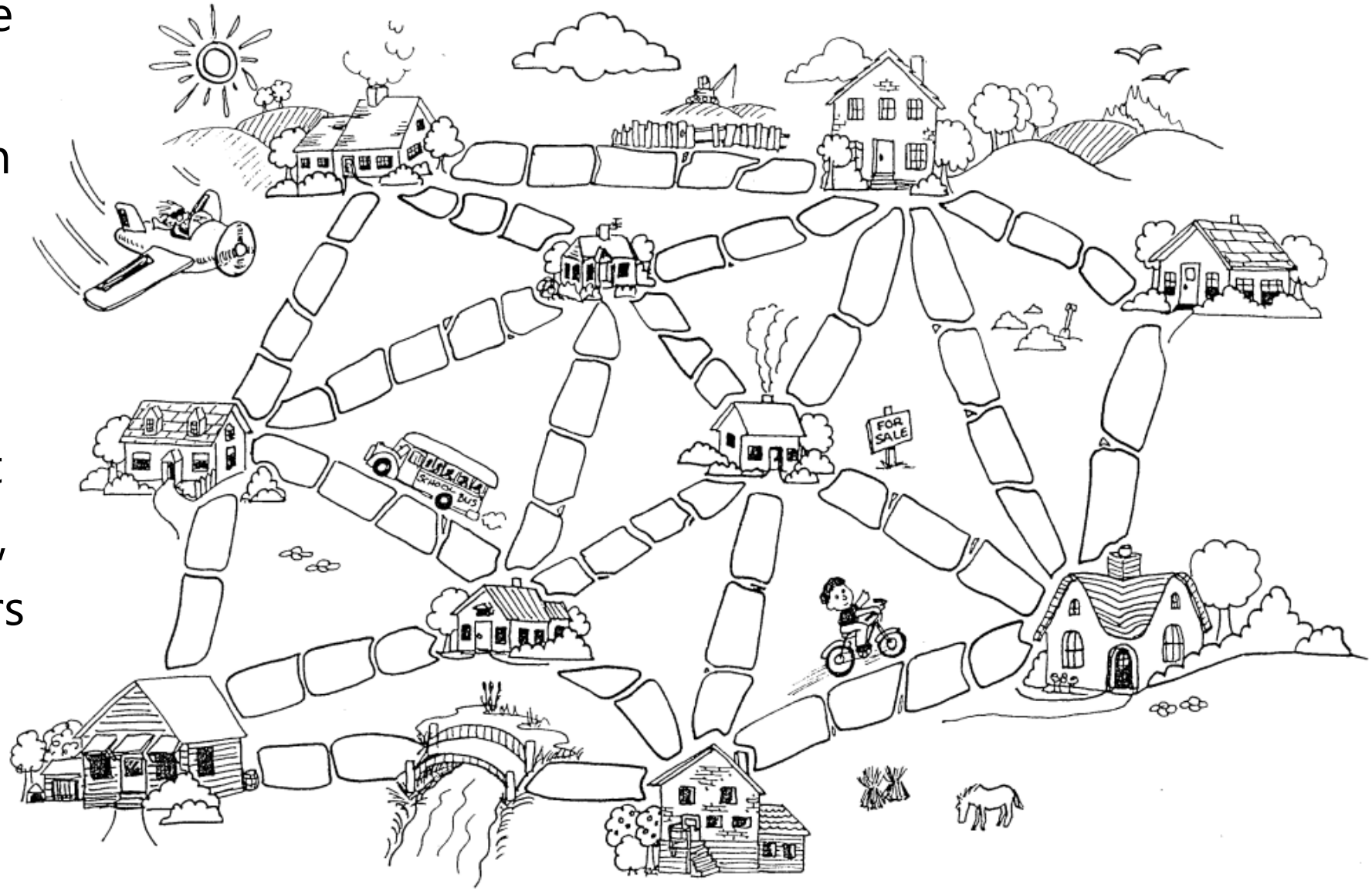
Exercise

- Once upon a time there was a city that had no roads.
- The mayor of the city decided that some of the streets must be paved, but he didn't want to spend more money than necessary. The mayor therefore specified two conditions:
  - enough streets must be paved so that it is possible for everyone to travel from their house to anyone else house only along paved roads,
  - the paving should cost as little as possible

Exercise

Here is the layout of the city. The **number of paving stones** between each house represents the cost of paving that route.

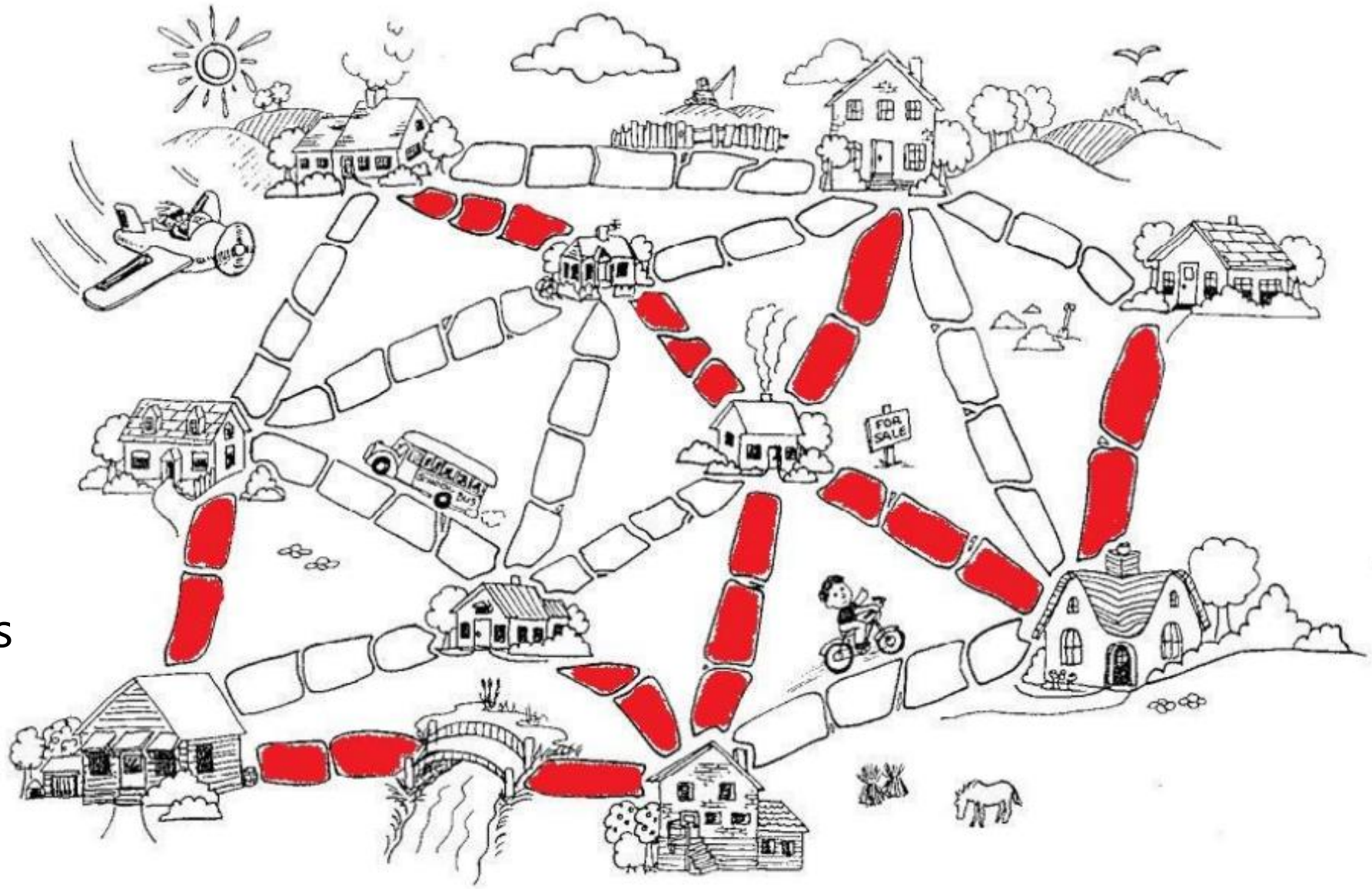
Find the best route that connects all the houses, but uses as few counters (paving stones) as possible.



Exercise

Here is the layout of the city. The **number of paving stones** between each house represents the cost of paving that route.

Find the best route that connects all the houses, but uses as few counters (paving stones) as possible.



Comparison

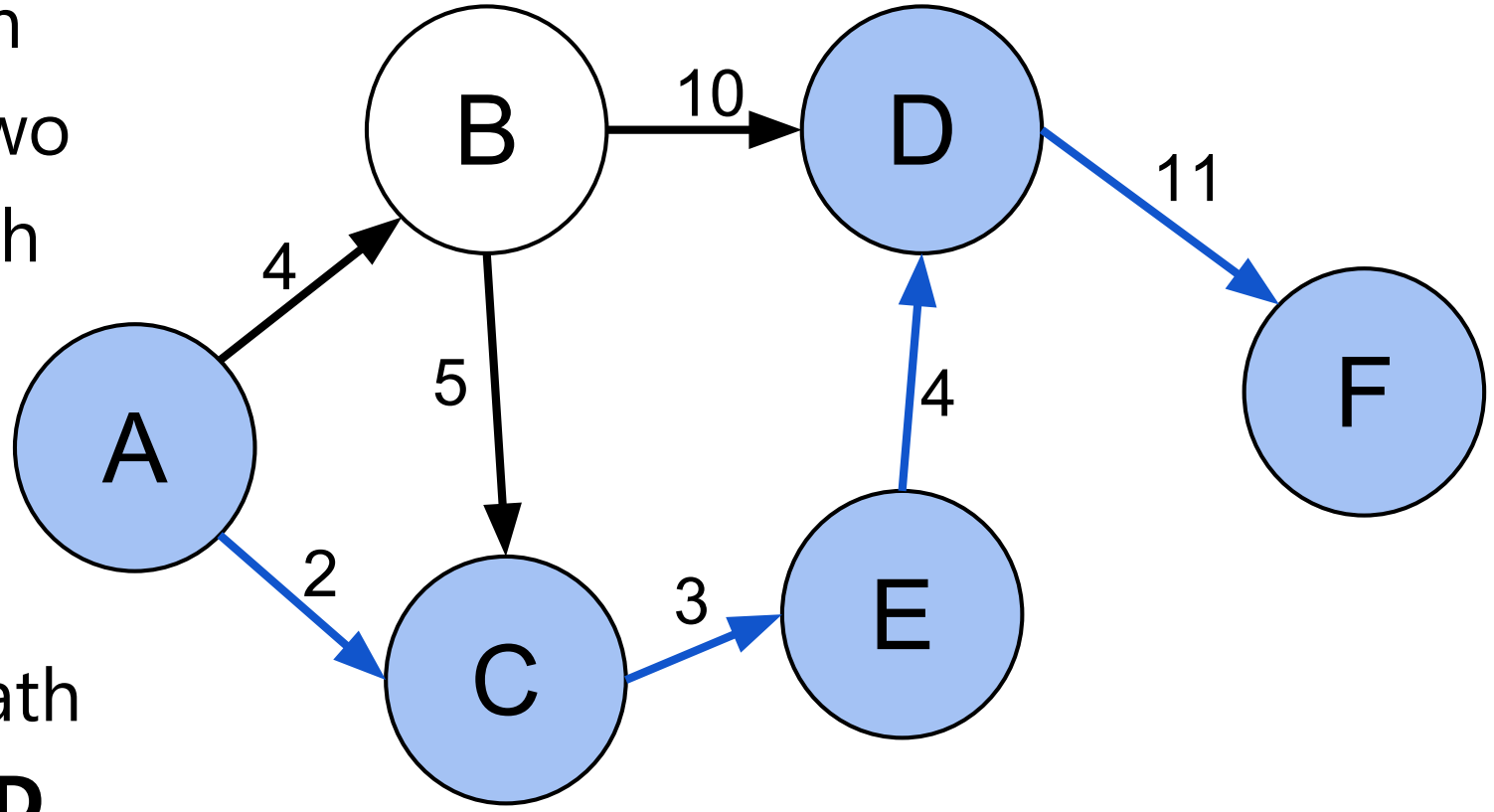
Difference between Prim's and Kruskal's algorithm for MST

Prim's	Kruskal's
Starts from random vertex	Starts from vertex with min weight
Time Complexity: $O(V^2)$	Time Complexity: O(E log(V))
Works only on connected graphs	Can work on disconnected components
Runs faster in dense graphs	Runs faster in sparse graphs
Prefers list data structures	Prefers heap data structures
Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities	LAN connection, TV Network

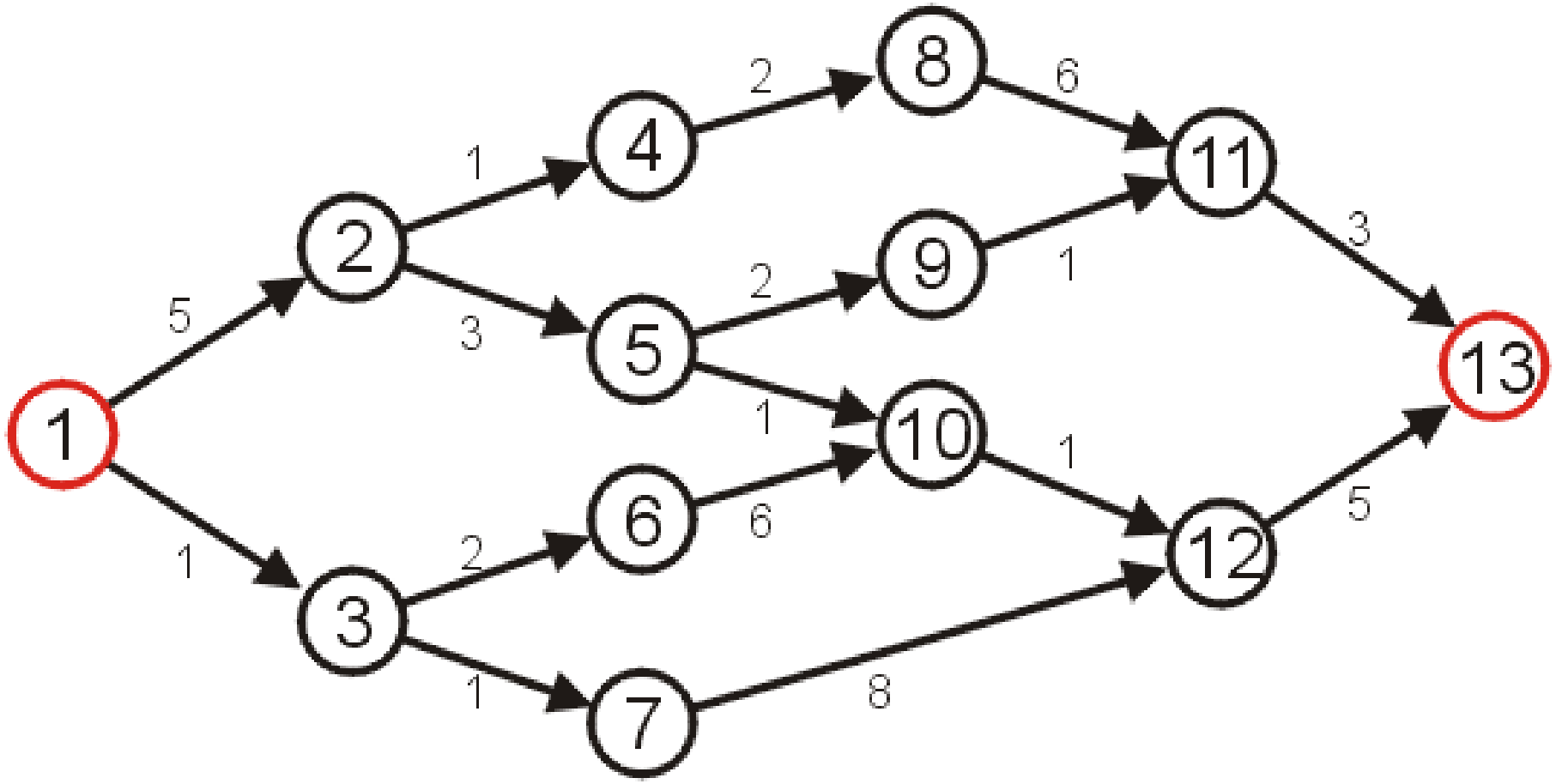
Shortest Path on Unweighted Graphs

Shortest Path

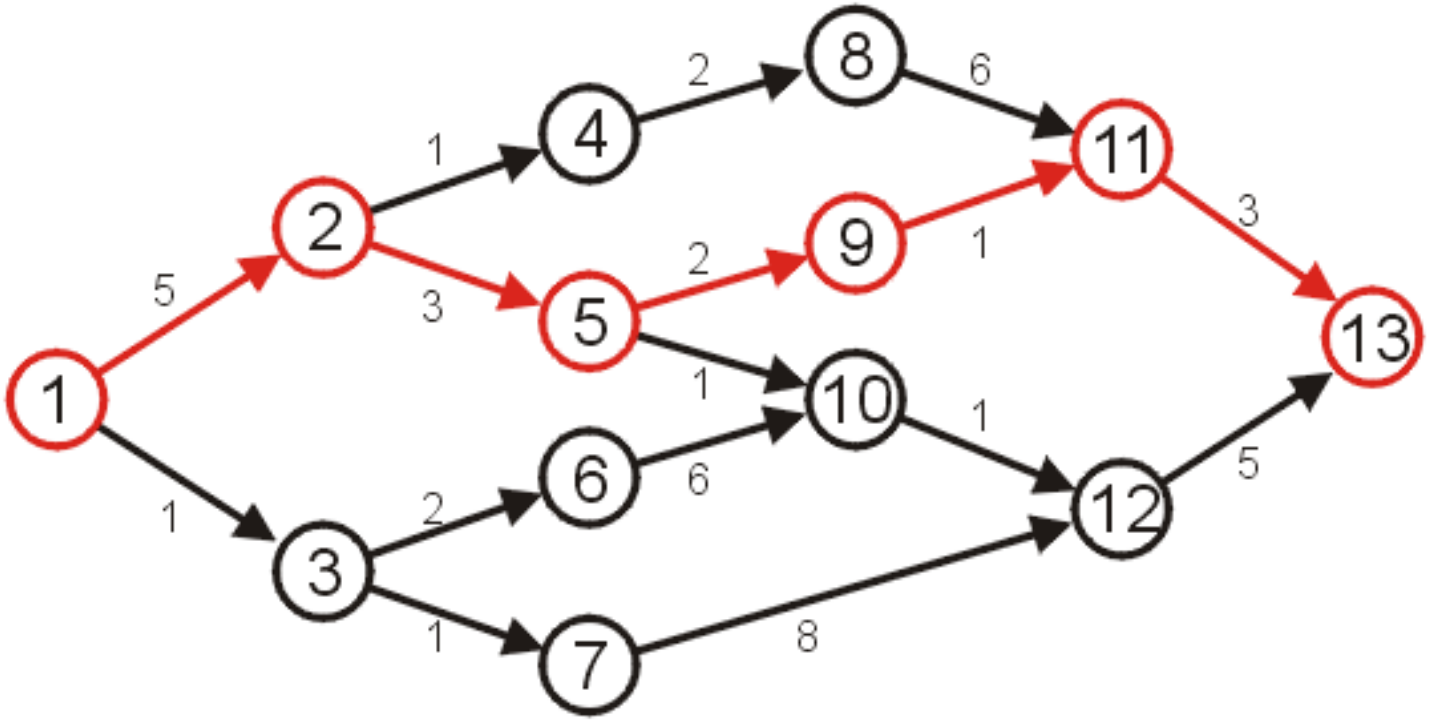
- In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is **minimized**
- For example, the shortest path from S to F is **A** → **C** → **E** → **D** → **F** with the cost **20**



- Given the graph below, suppose we wish to find the shortest path from vertex 1 to vertex 13



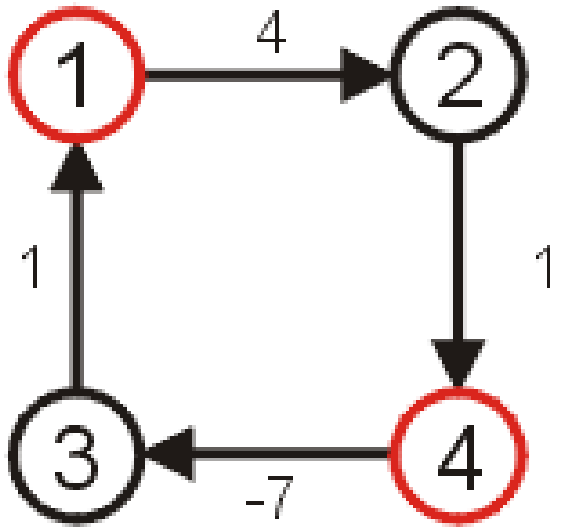
- After some consideration, we may determine that the shortest path is as follows, with length 14



- Other paths may exist, but they are longer

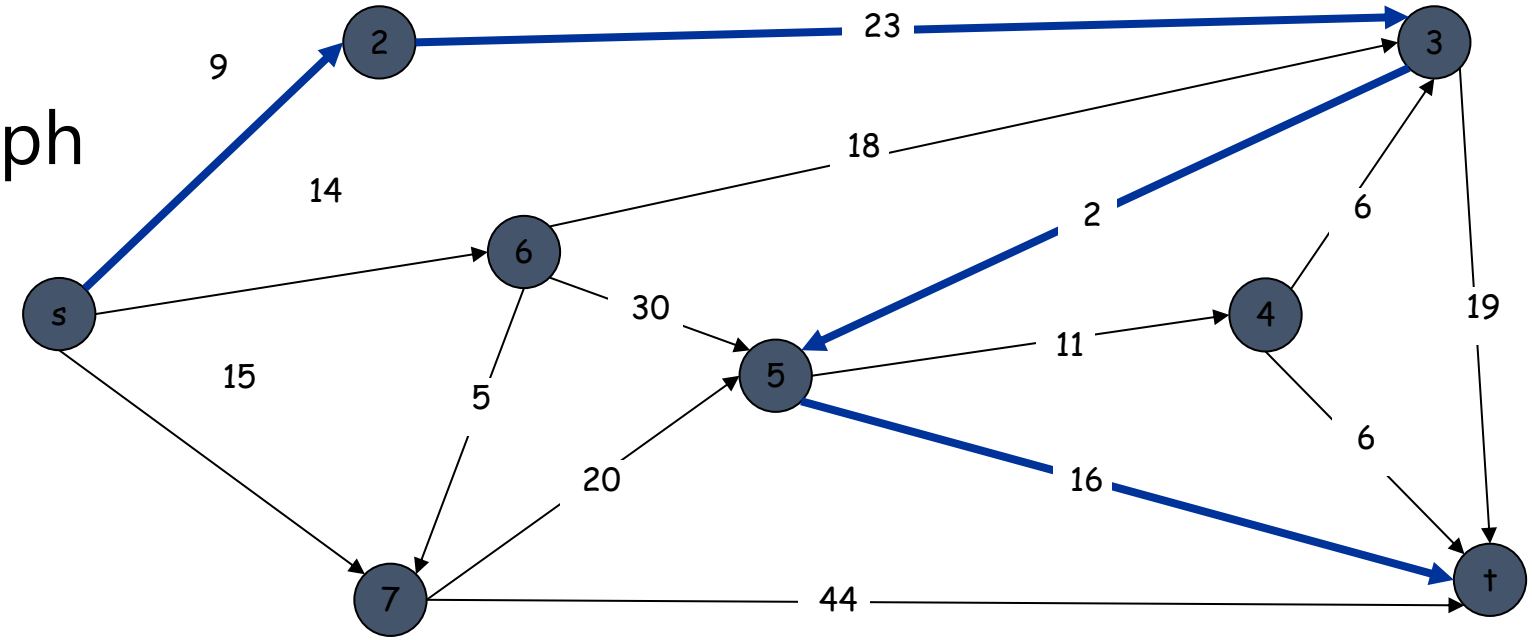
Negative Cycles

- Clearly, if we have negative edges, it may be possible to end up in a cycle whereby each pass through the cycle decreases the total length
- Thus, a shortest length would be undefined for such a graph
- Consider the shortest path from vertex 4 to 1
- We will only consider non-negative weights.



Example

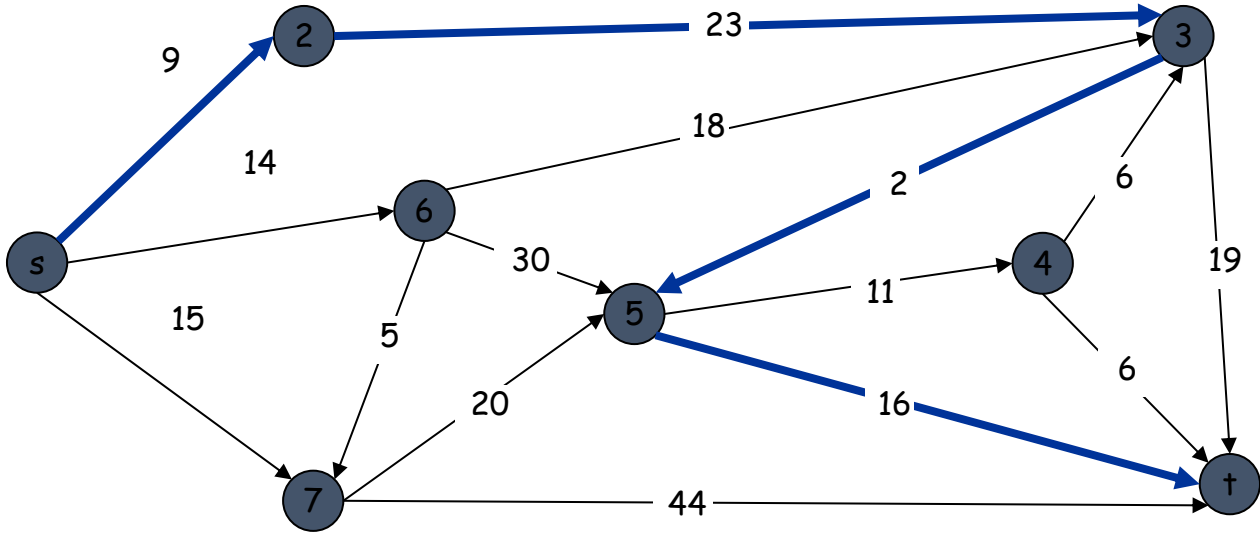
- Given:
  - Weighted Directed graph G = (V, E).
  - Source s
  - Destination t.
- Find the shortest directed path from s to t.



Cost of path s-2-3-5-t = 9 + 23 + 2 + 16 = 48.

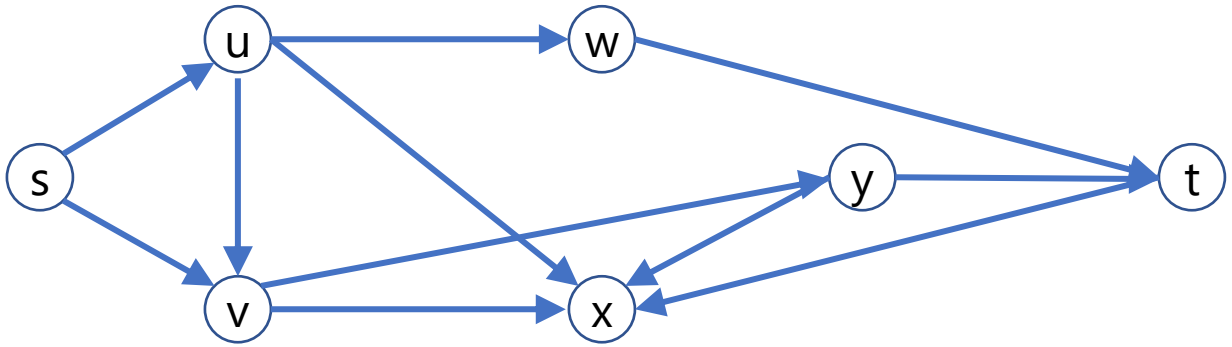
Example

- How many possible paths are there from s to t?
- Can we safely ignore cycles? If so, how?
- Any suggestions on how to reduce the set of possibilities?
- Can we determine a lower bound on the complexity like we did for comparison sorting?



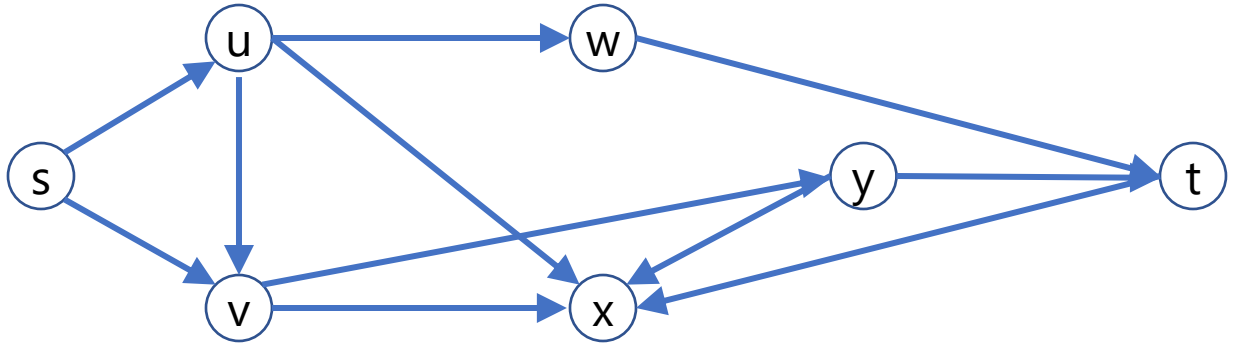
Cost of path s-2-3-5-t = 9 + 23 + 2 + 16 = 48.

- If the graph is unweighted, how do we find a shortest path?



- What's the shortest path from s to s?
  - Well….we're already there.
- What's the shortest path from s to u or v?
  - Just go on the edge from s
- From s to w,x, or y?
  - Can't get there directly from s. If we want a length 2 path, we have to go through u or v.

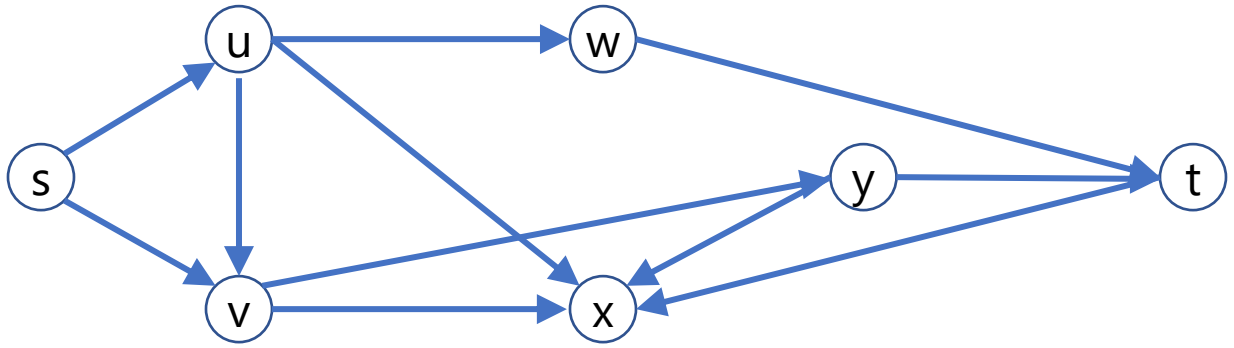
- To find the set of vertices at distance k, just find the set of vertices at distance k-1, and see if any of them have an outgoing edge to an undiscovered vertex.
- Do we already know an algorithm that does something like that?



Unweighted graphs

- To find the set of vertices at distance k, just find the set of vertices at distance k-1, and see if any of them have an outgoing edge to an undiscovered vertex.
- Do we already know an algorithm that does something like that?

→ **breadth-first search (BFS)**



Dijkstra's Algorithm

Dijkstra's Algorithm

- Works when all of the weights are **positive**.
- Provides the shortest paths from **a source** to **all other vertices** in the graph.
- Can be terminated early once the shortest path to t is found if desired

- Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree.

Dijkstra's Algorithm

shortestPath(matrix[N][N], source, length[])

Input:

matrix[N][N]: adjacency matrix of Graph *G* with *N* vertices

source: the *source* vertex

Output:

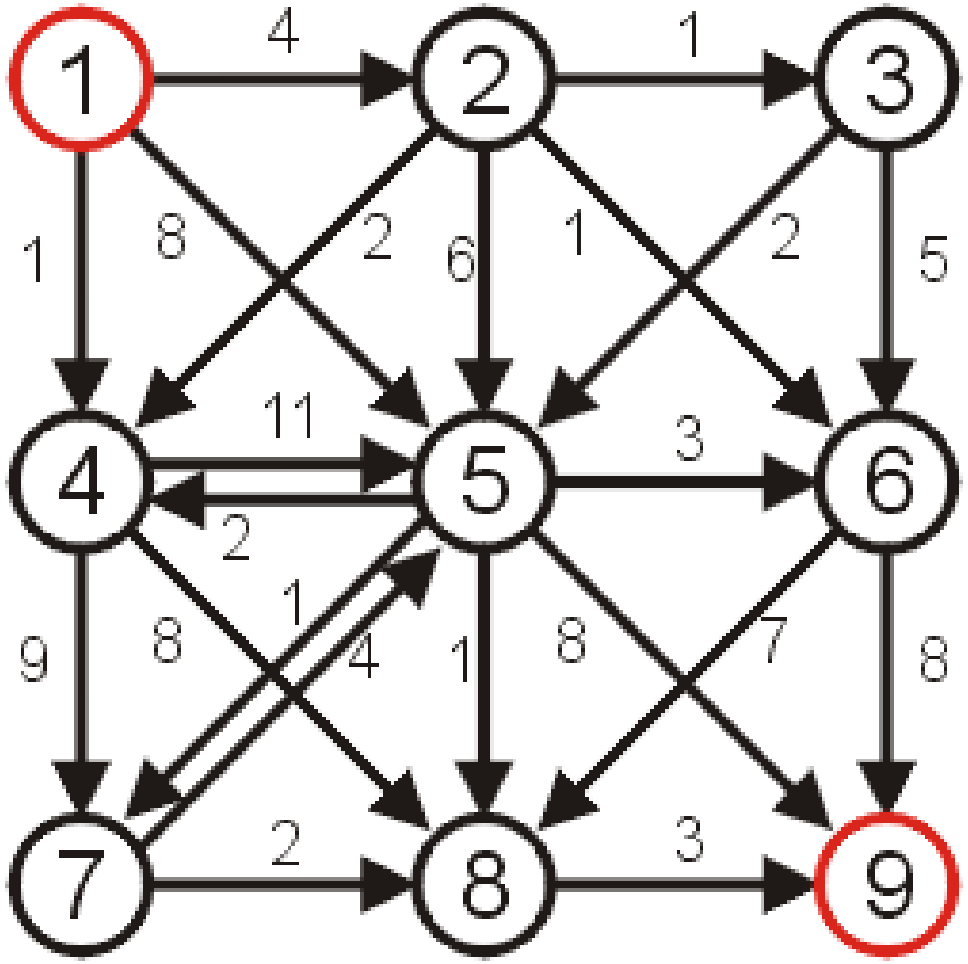
length[]: the length of the shortest path from the *source* to all *vertices* in G.

Dijkstra's Algorithm

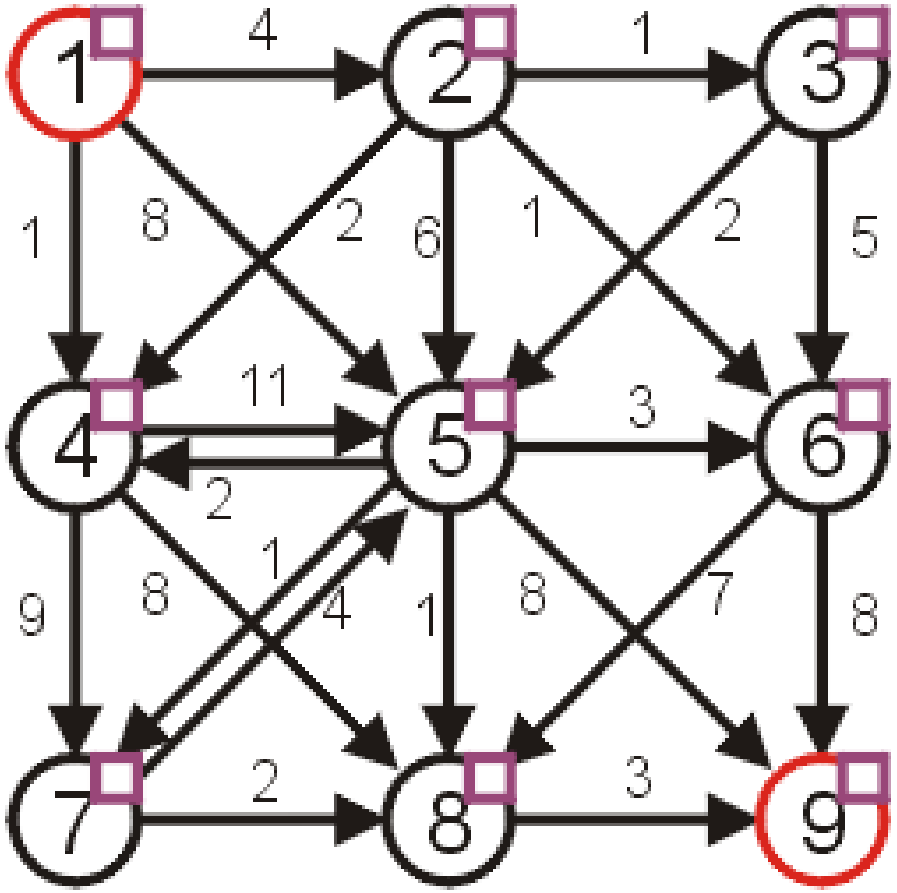
```
shortestPath(matrix[N][N], source, length[]){
   for v = 0 to N-1
      length[v] = ∞
   length[source] = 0 // why?
   for step = 1 to N {
      Find the vertex v such that length[v] is smallest and
         v is not in vertexSet
      Add v to vertexSet
      for all vertices u not in vertexSet
         if (length[u] > length[v] + matrix[v][u]){
            length[u] = length[v] + matrix[v][u]
            parent[u] = v }
   }
}
```

Dijkstra's Algorithm

- Consider the following graph with positive weights and cycles.



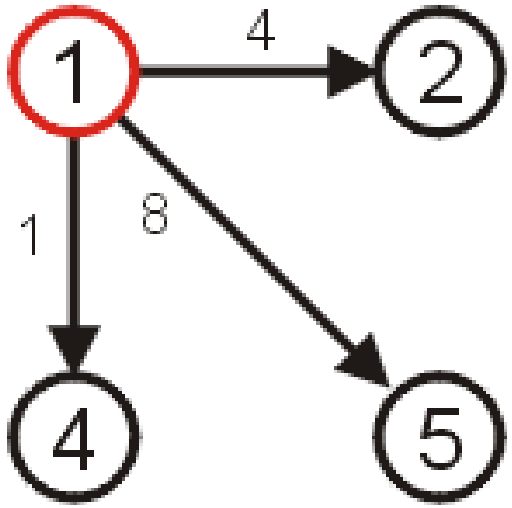
- Graphically, we will denote this with check boxes next to each of the vertices (initially unchecked)



Dijkstra's Algorithm

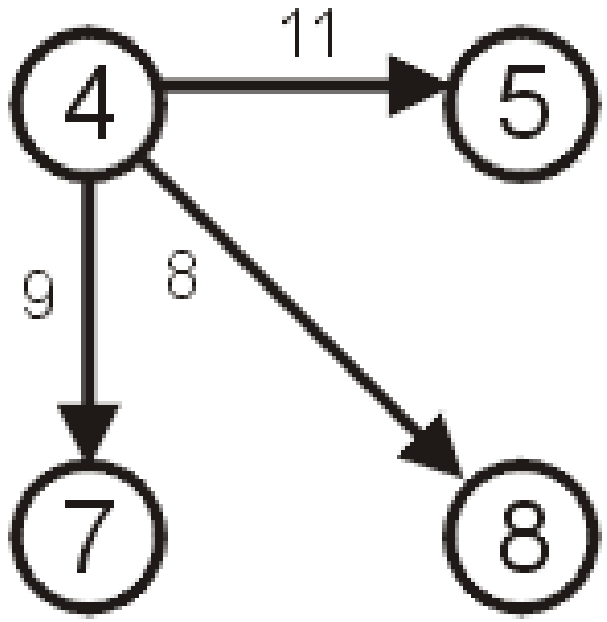
- We will work bottom up.
  - Note that if the starting vertex has any adjacent edges, then there will be one vertex that is the shortest distance from the starting vertex. This is the shortest reachable vertex of the graph.
- We will then try to extend any existing paths to new vertices.
- Initially, we will start with the path of length 0
  - this is the trivial path from vertex 1 to itself

- If we now extend this path, we should consider the paths
  - (1, 2) length 4
  - (1, 4) length 1
  - (1, 5) length 8

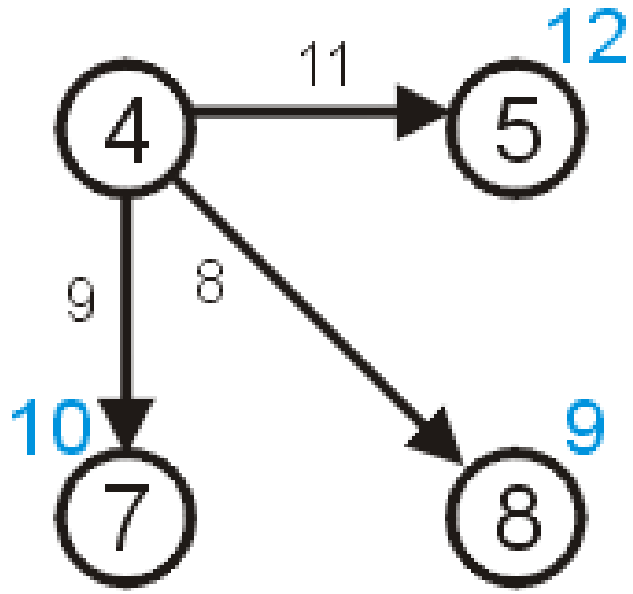


- The shortest path so far is (1, 4) which is of length 1.

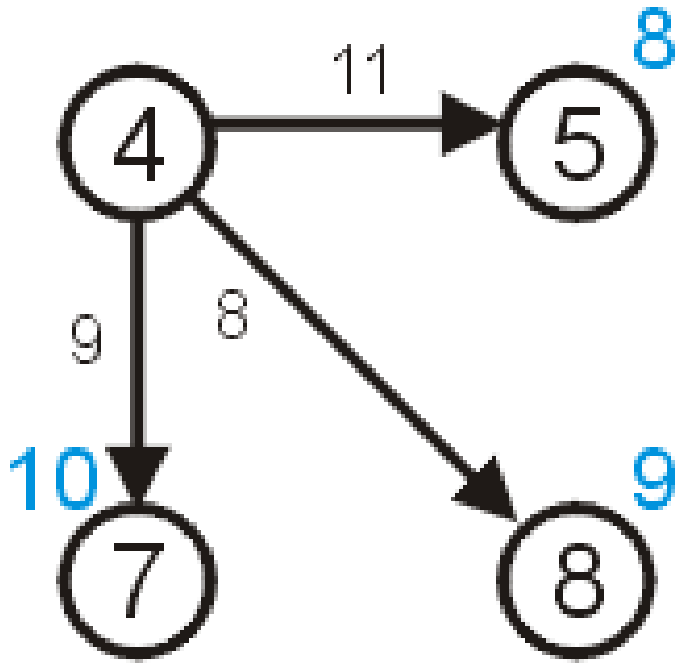
- Thus, if we now examine vertex 4, we may deduce that there exist the following paths:
  - (1, 4, 5) length 12
  - (1, 4, 7) length 10
  - (1, 4, 8) length 9



- We need to remember that the length of that path from node 1 to node 4 is 1
- Thus, we need to store the length of a path that goes through node 4:
  - 5 of length 12
  - 7 of length 10
  - 8 of length 9



- We have already discovered that there is a path of length 8 to vertex 5 with the path (1, 5).
- Thus, we can safely ignore this longer path.



Dijkstra's Algorithm

- We now know that:
  - There exist paths from vertex 1 to vertices {1,2,4,5,7,8}.
  - We know that the shortest path from vertex 1 to vertex 4 is of length 1.
  - We know that the shortest path to the other vertices {2,5,7,8} is at most the length listed in the table to the right.

Vertex	Length
1	0
2	4
4	1
5	8
7	10
8	9

Dijkstra's Algorithm

- There cannot exist a shorter path to either of the vertices 1 or 4, since the distances can only increase at each iteration.
- We consider these vertices to be **visited**

Vertex	Length
1	0
2	4
4	1
5	8
7	10
8	9

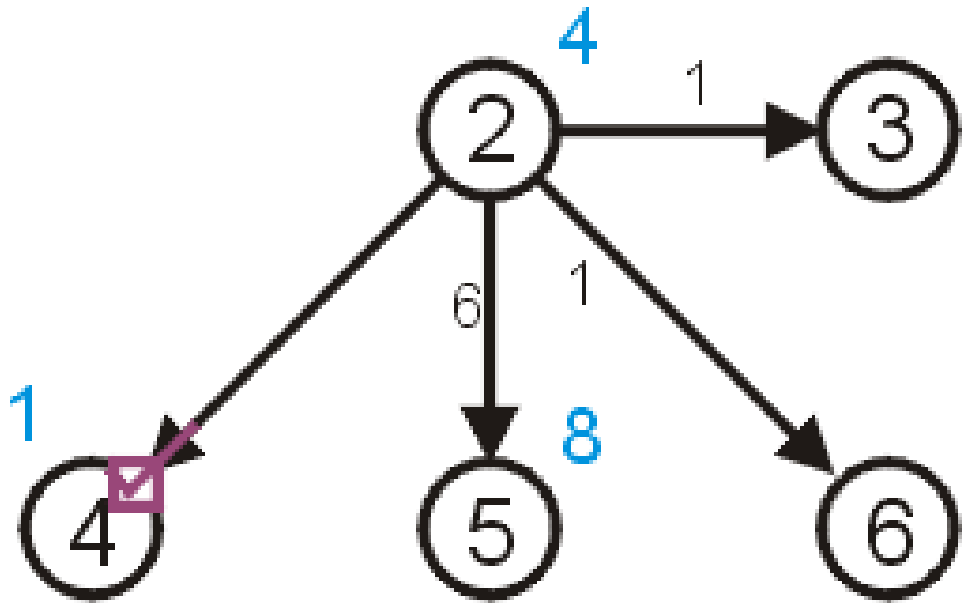
If you only knew this information and nothing else about the graph, what is the possible lengths from vertex 1 to vertex 2?
What about to vertex 7?

- Maintaining this shortest discovered distance d[v] is called relaxation:

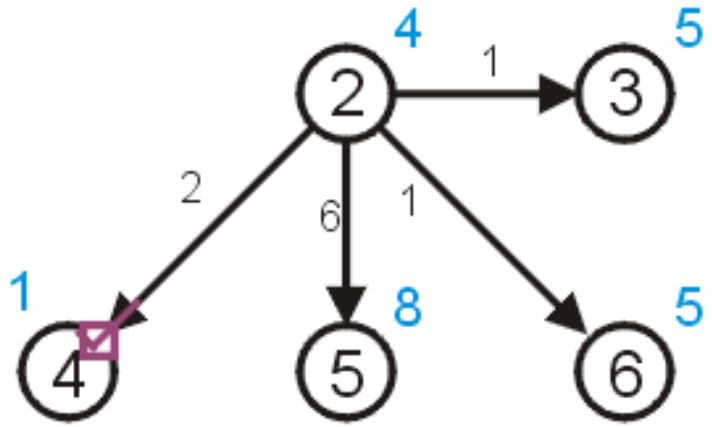
```
Relax(u,v,w) {
   if (d[v] > d[u]+w) then
      d[v]=d[u]+w;
}
```

Dijkstra's Algorithm

- In Dijkstra's algorithm, we always take the next unvisited vertex which has **the current shortest path** from the starting vertex in the table.
- This is vertex 2



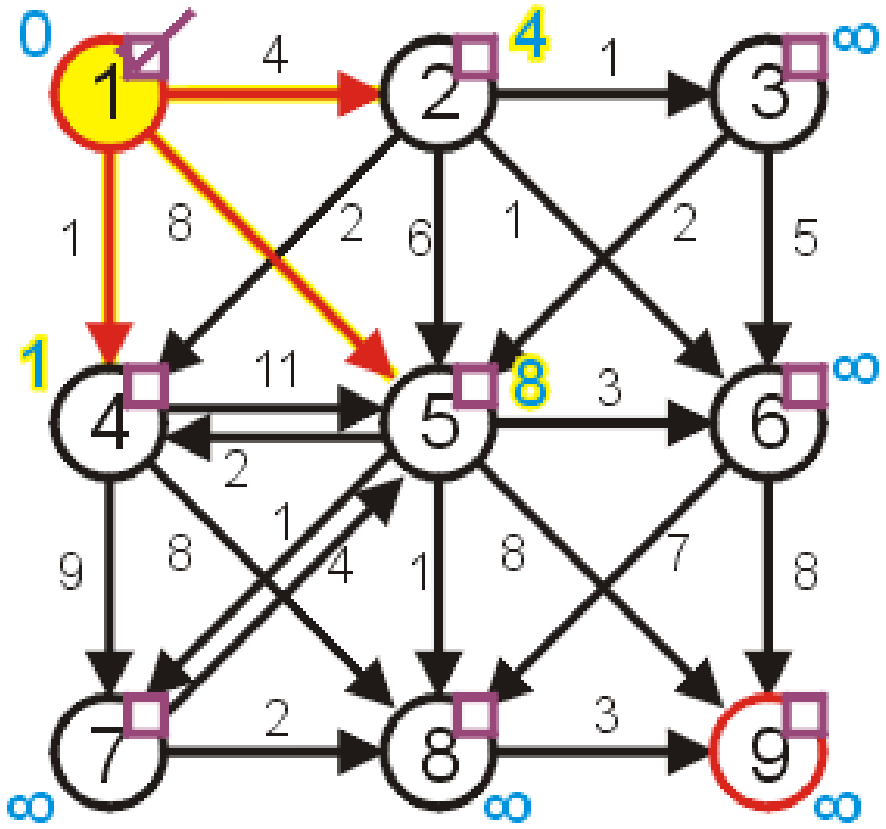
- We can try to update the shortest paths to vertices 3 and 6 (both of length 5) however:
  - there already exists a path of length 8 < 10 to vertex 5 (10 = 4 + 6)
  - we already know the shortest path to 4 is 1



Dijkstra's Algorithm

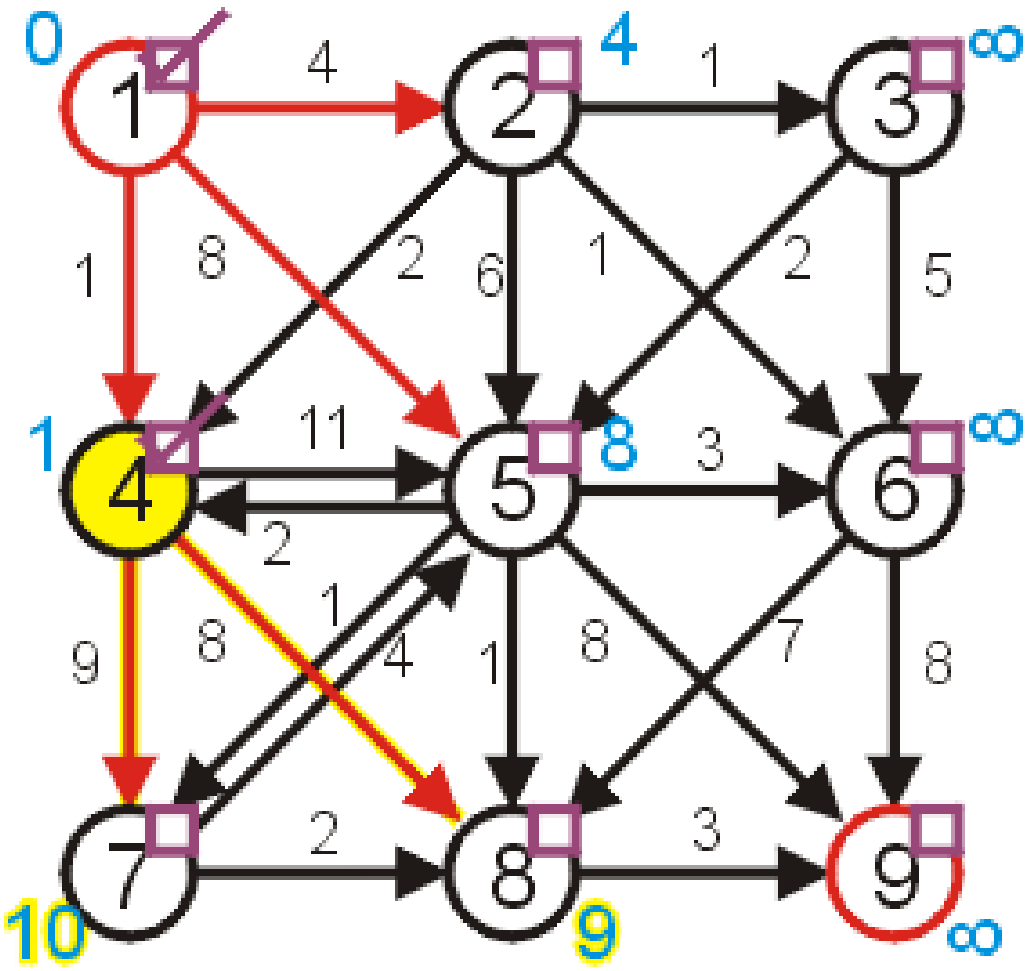
- To keep track of those vertices to which no path has reached, we can assign those vertices an initial distance of either
  - infinity (∞),
  - a number larger than any possible path, or
  - a negative number
- For demonstration purposes, we will use ∞

- Visit vertex 1 and update its neighbors, marking it as visited
  - the shortest paths to 2, 4, and 5 are updated



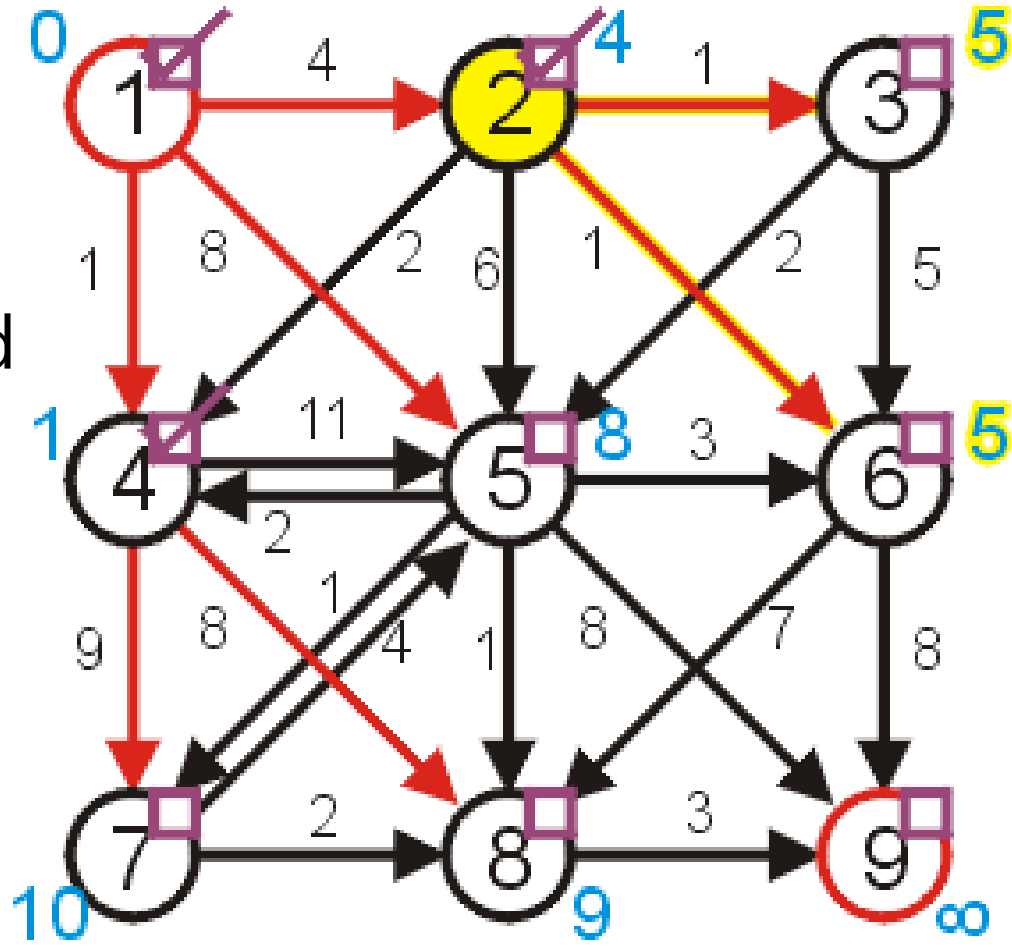
Dijkstra's Algorithm

- The next vertex we visit is vertex 4
  - vertex 5 1 + 11 ≥ 8 don't update
  - vertex 7 1 + 9 < ∞ update
  - vertex 8 1 + 8 < ∞ update



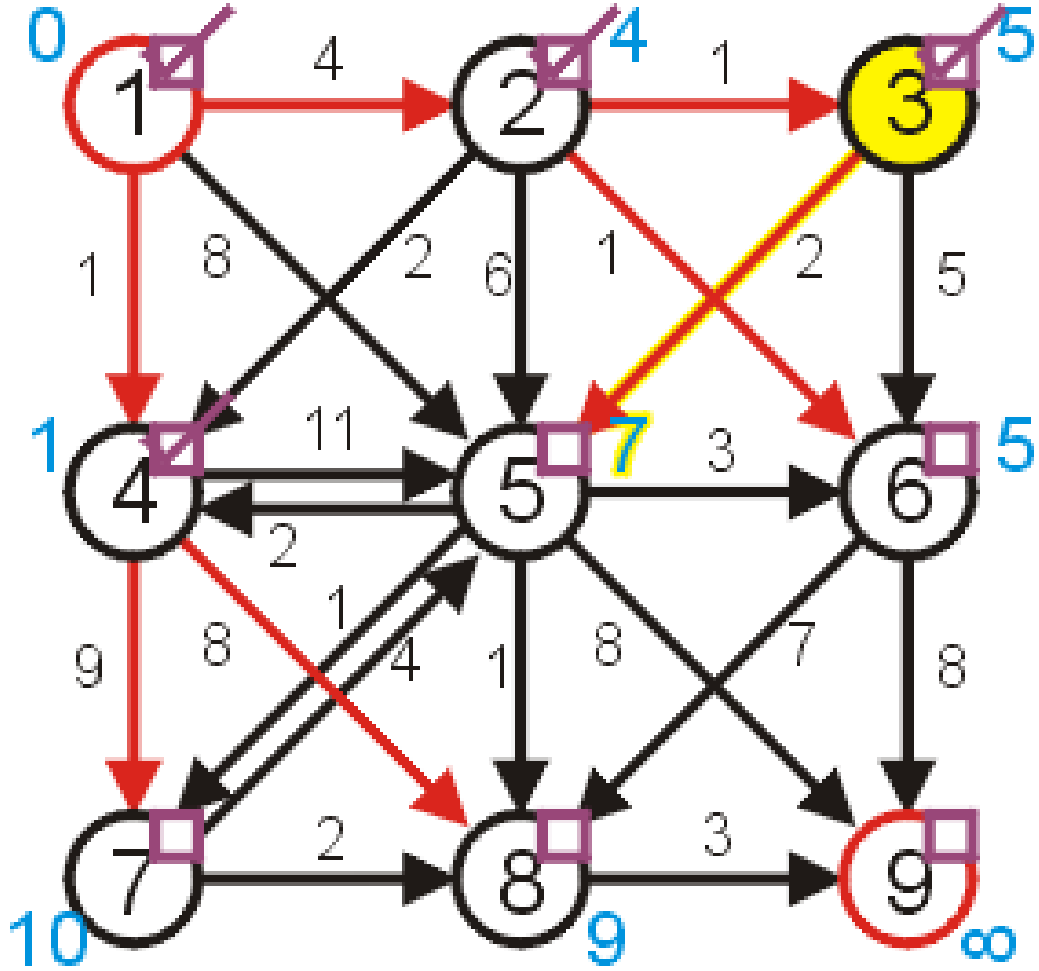
Dijkstra's Algorithm

- Next, visit vertex 2
  - vertex 3 $4 + 1 < \infty$ update
  - vertex 4 already visited
  - vertex 5 $4 + 6 \geq 8$ don't update
  - vertex 6 $4 + 1 < \infty$ update

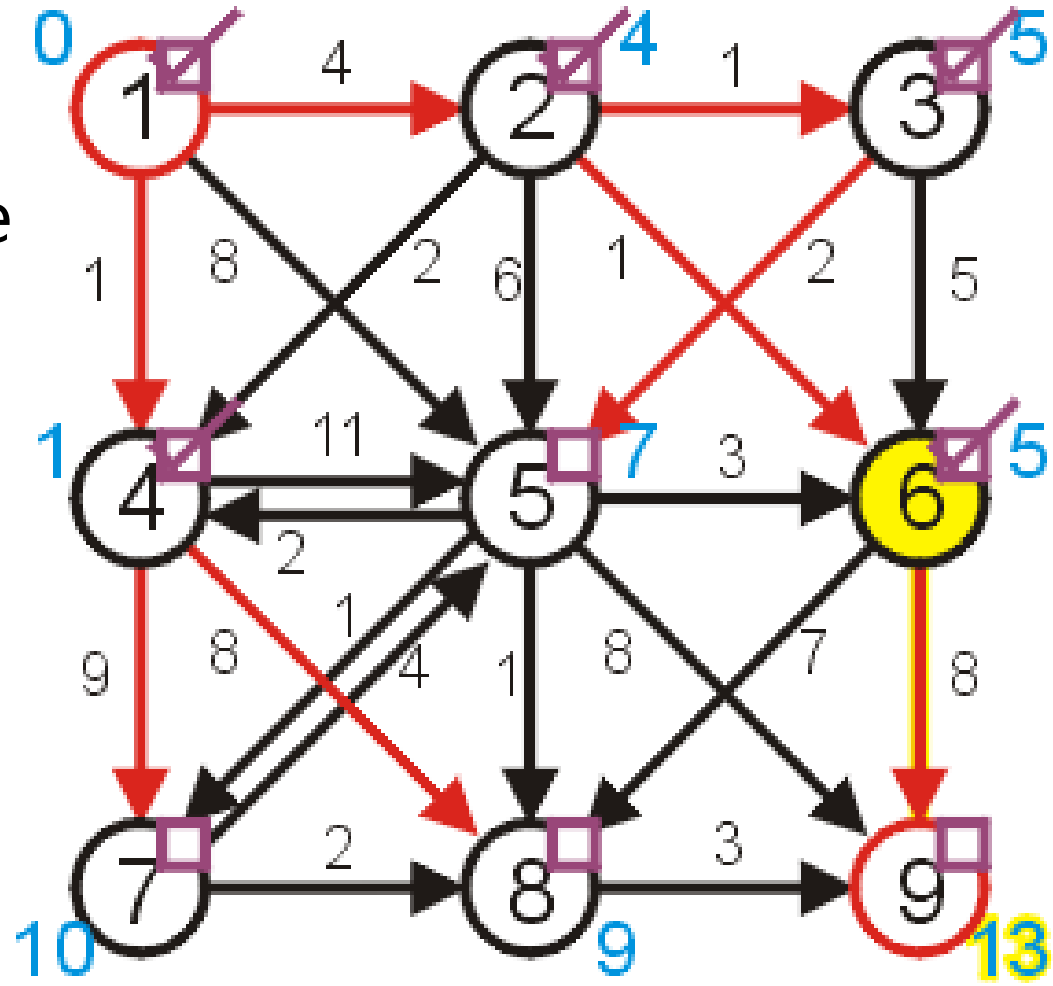


Dijkstra's Algorithm

- Next, we have a choice of either 3 or 6
- We will choose to visit 3
  - vertex 5 5 + 2 < 8 update
  - vertex 6 5 + 5 ≥ 5 don't update

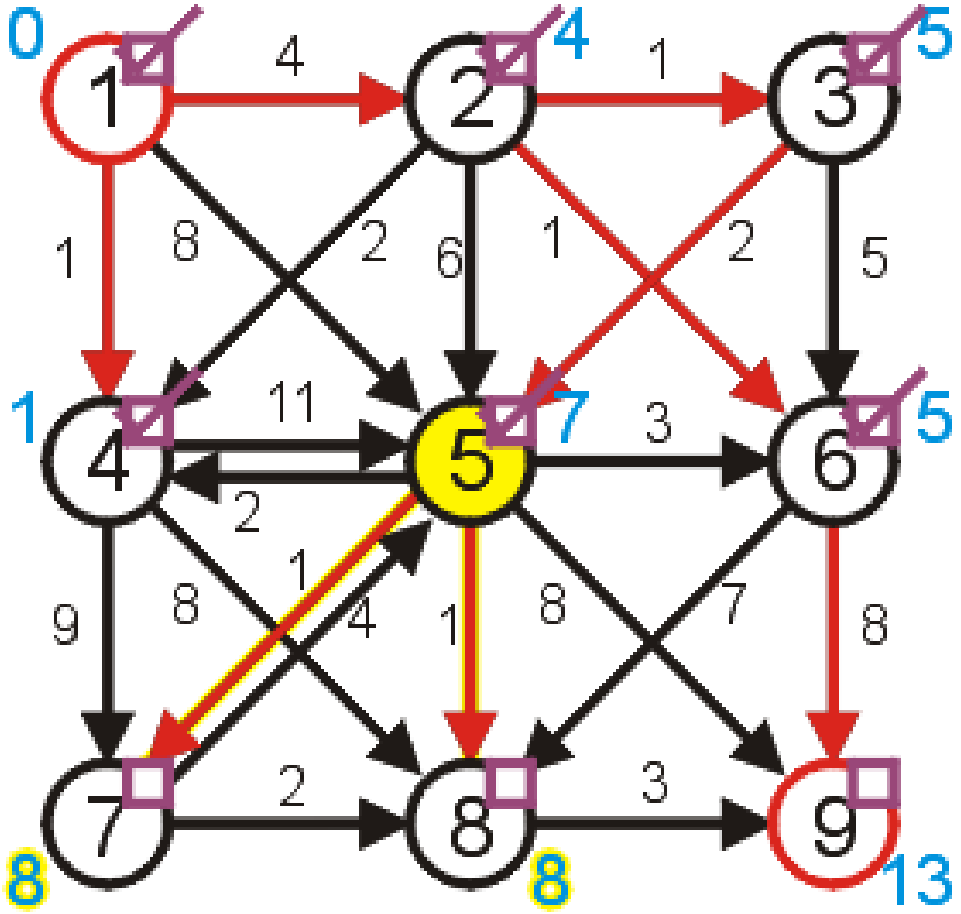


- We then visit 6
  - vertex 8 5 + 7 ≥ 9 don't update
  - vertex 9 5 + 8 < ∞ update



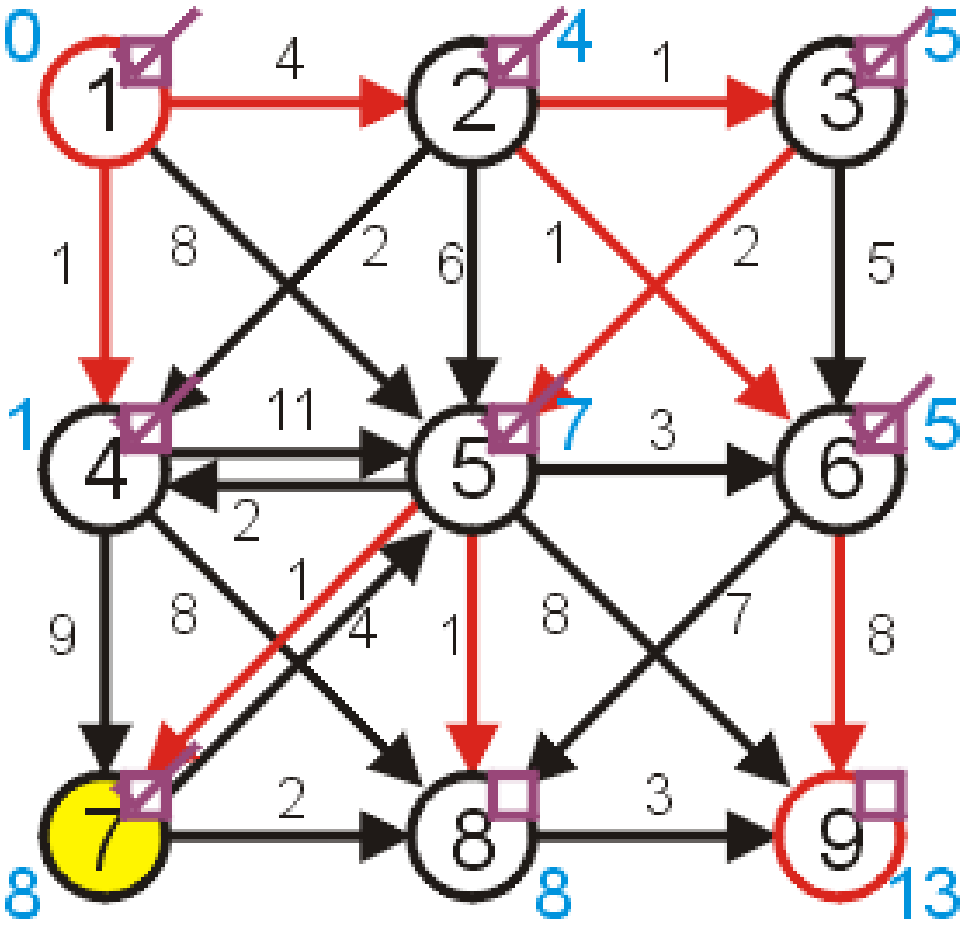
Dijkstra's Algorithm

- Next, we finally visit vertex 5:
  - vertices 4 and 6 have already been visited
  - vertex 7 7 + 1 < 10 update
  - vertex 8 7 + 1 < 9 update
  - vertex 9 7 + 8 ≥ 13 don't update



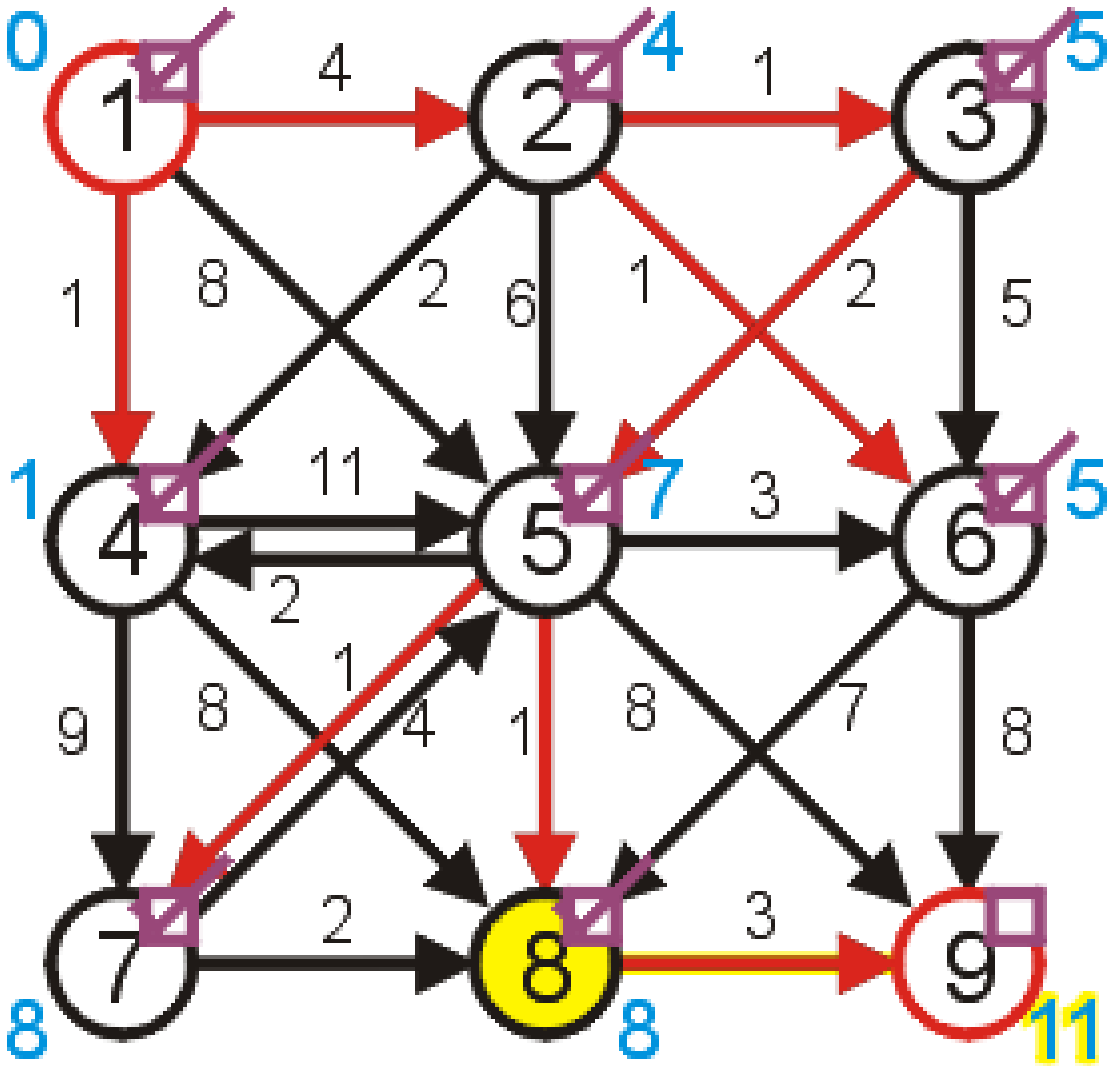
Dijkstra's Algorithm

- Given a choice between vertices 7 and 8,
- we choose vertex 7
  - vertices 5 has already been visited
  - vertex 8 8 + 2 ≥ 8 don't update



Dijkstra's Algorithm

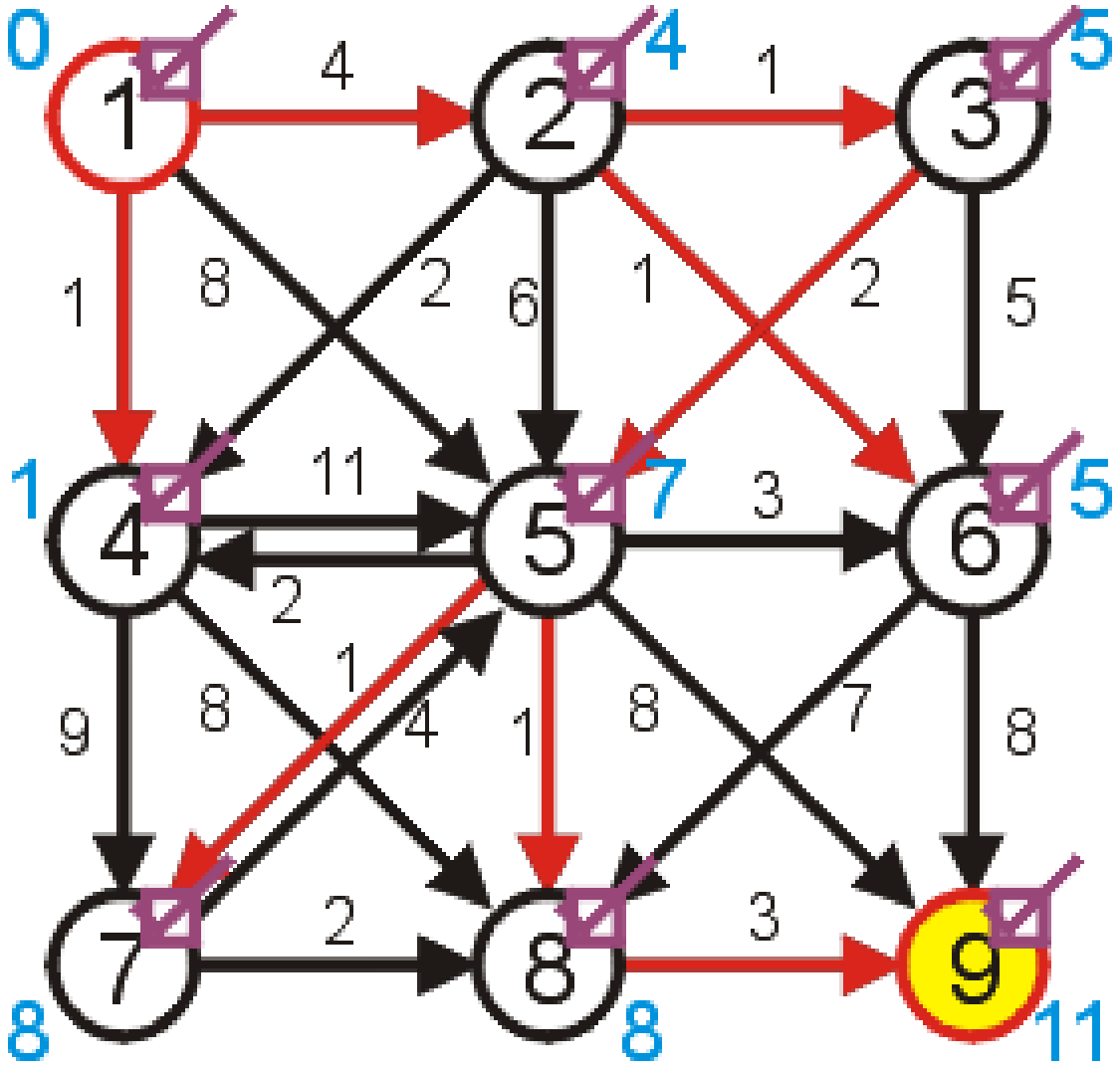
- Next, we visit vertex 8:
  - vertex 9 8 + 3 < 13 update



Dijkstra's Algorithm

- Finally, we visit the end vertex
- Therefore, the shortest path

from 1 to 9 has length 11



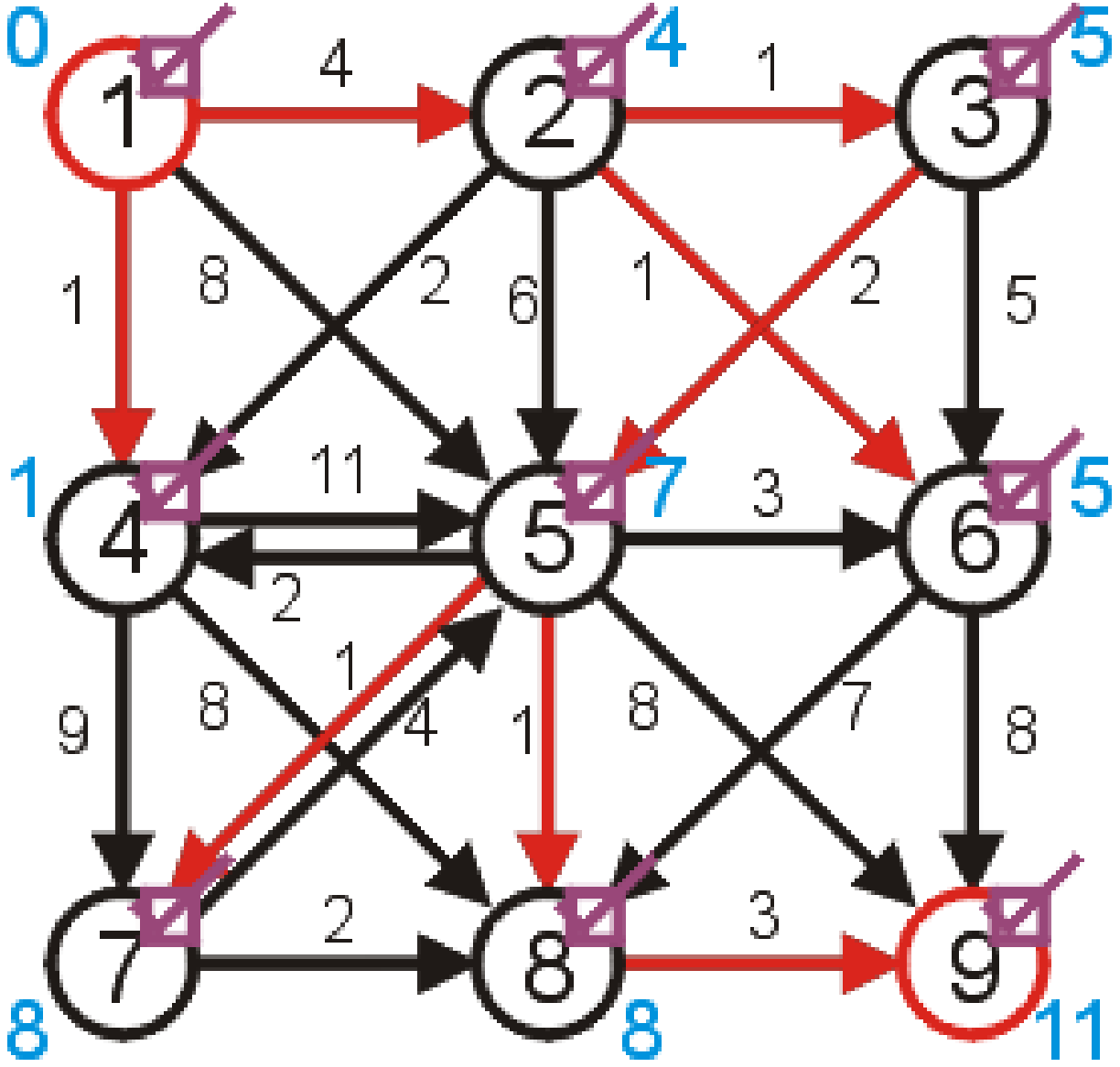
Dijkstra's Algorithm

- We can find the shortest path by working back from the final vertex:

  9, 8, 5, 3, 2, 1

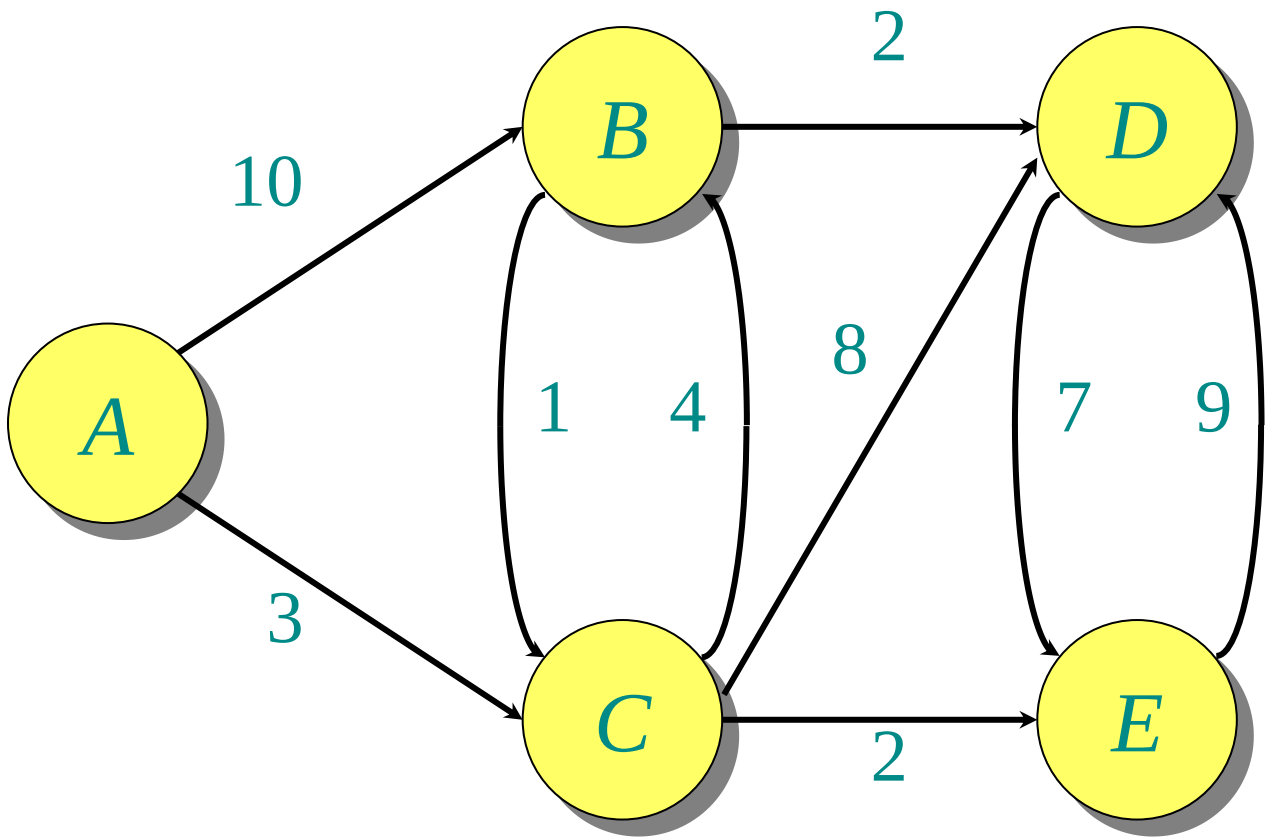
- Thus, the shortest path is

  (1, 2, 3, 5, 8, 9)



Another example

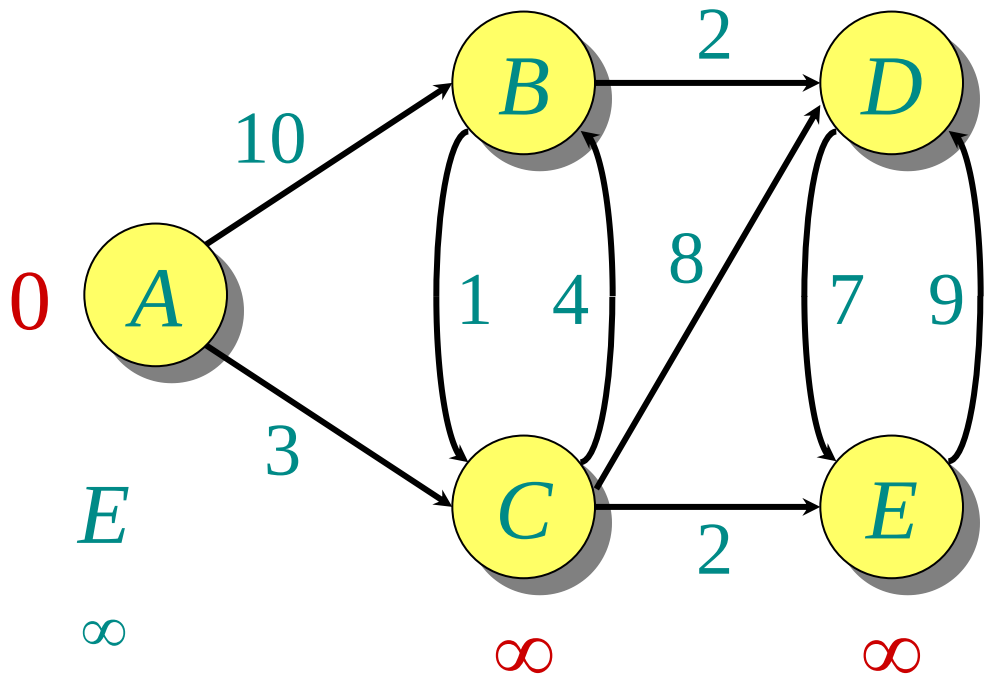
- Demonstrate Dijkstra's Algorithm from node A



Another example

- Demonstrate Dijkstra's Algorithm from node A

Initialize:

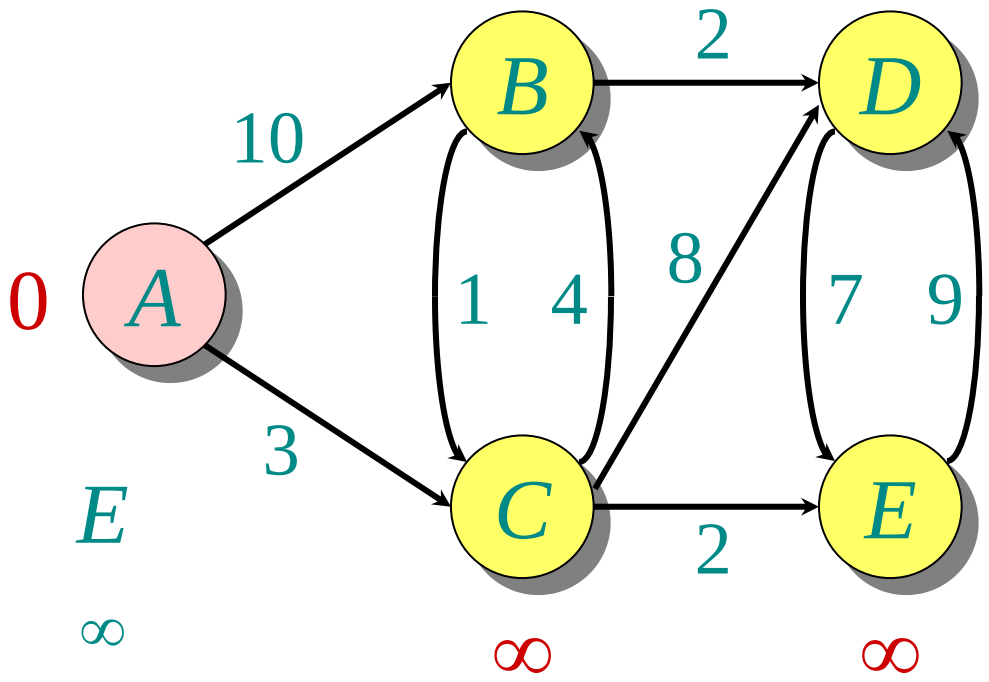


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Another example

- Demonstrate Dijkstra's Algorithm from node A



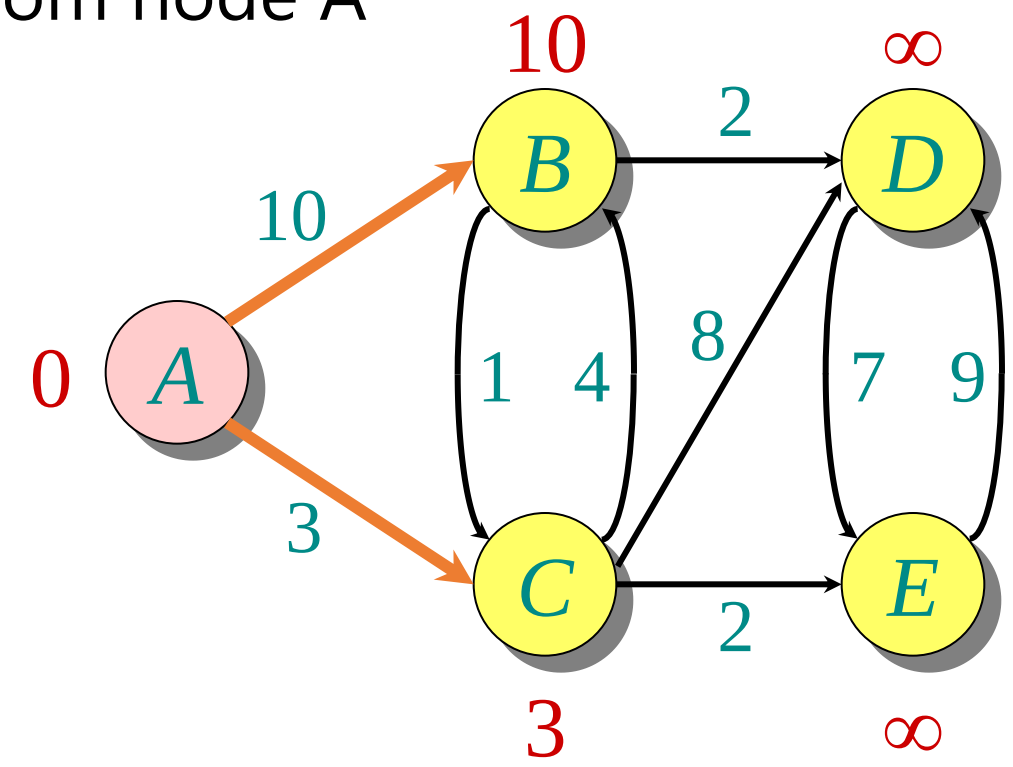
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: { A }

Another example

- Demonstrate Dijkstra's Algorithm from node A

Relax all edges leaving *A*:

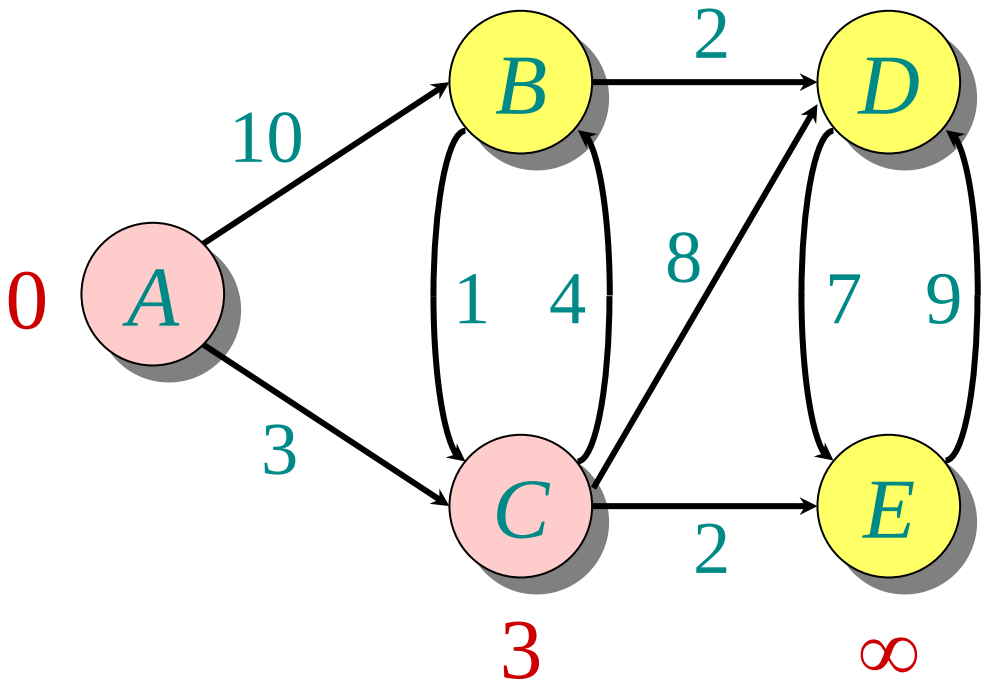


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A* }

Another example

- Demonstrate Dijkstra's Algorithm from node A



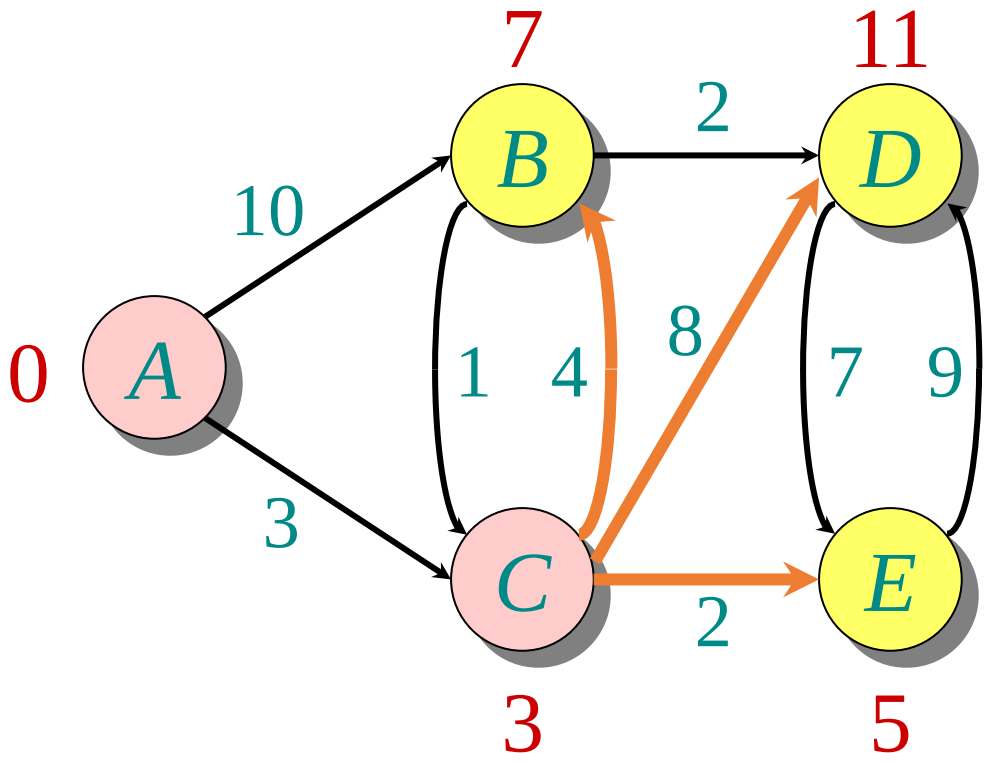
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

Another example

- Demonstrate Dijkstra's Algorithm from node A

Relax all edges leaving *C*:

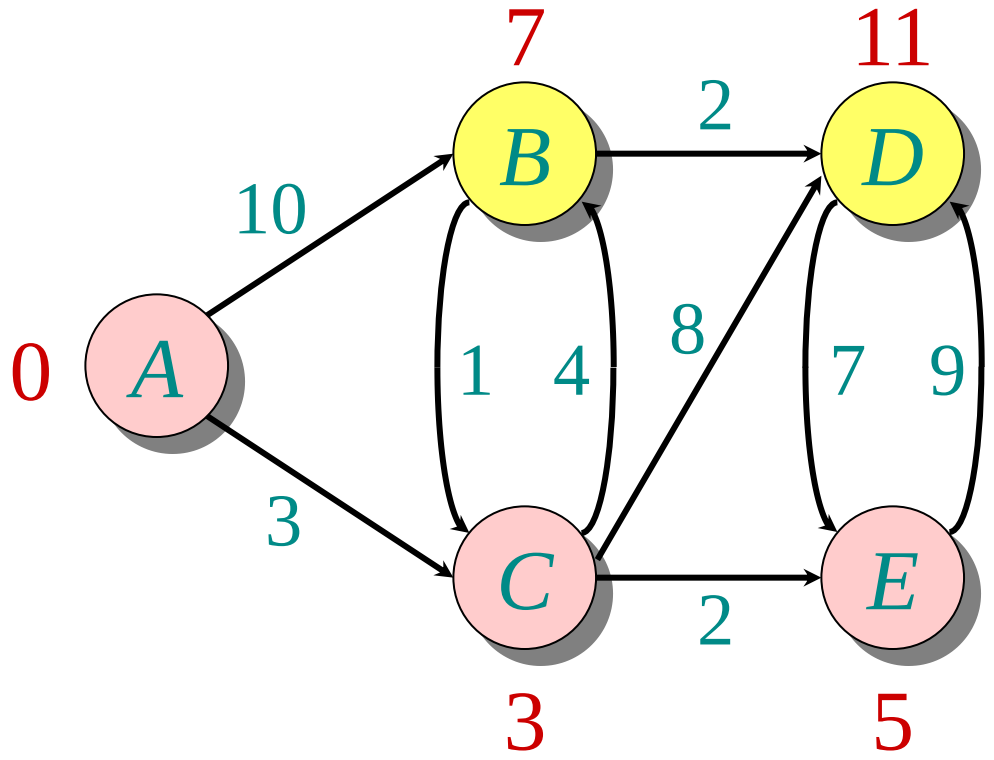


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

Another example

- Demonstrate Dijkstra's Algorithm from node A



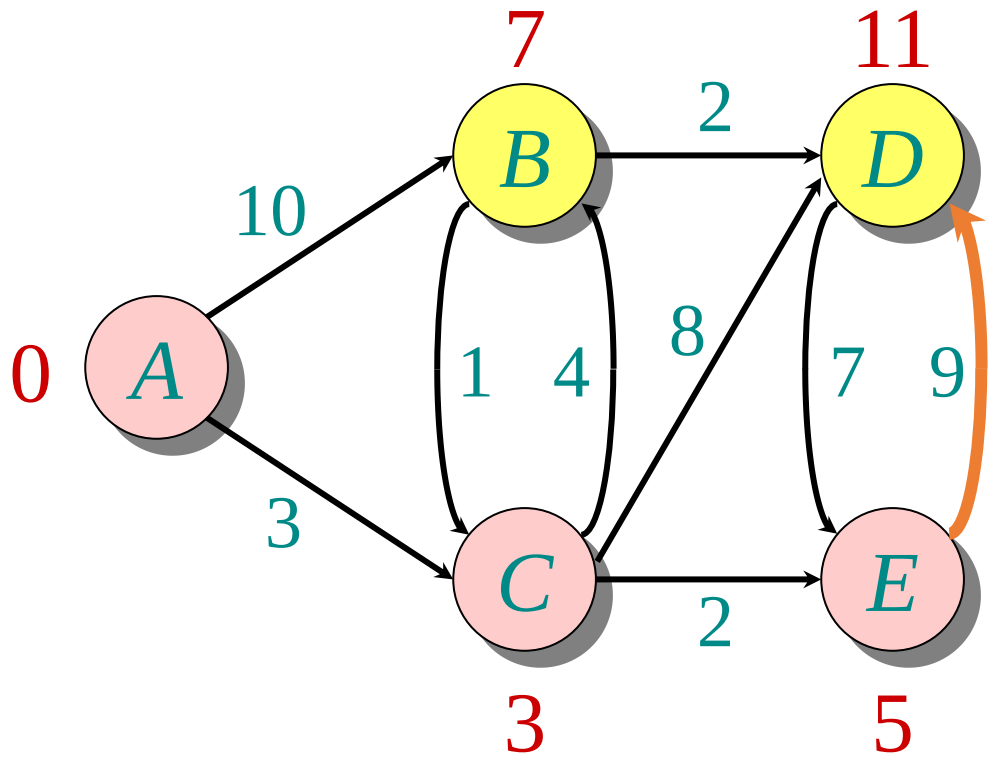
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C*, *E* }

Another example

- Demonstrate Dijkstra's Algorithm from node A

Relax all edges leaving *E*:

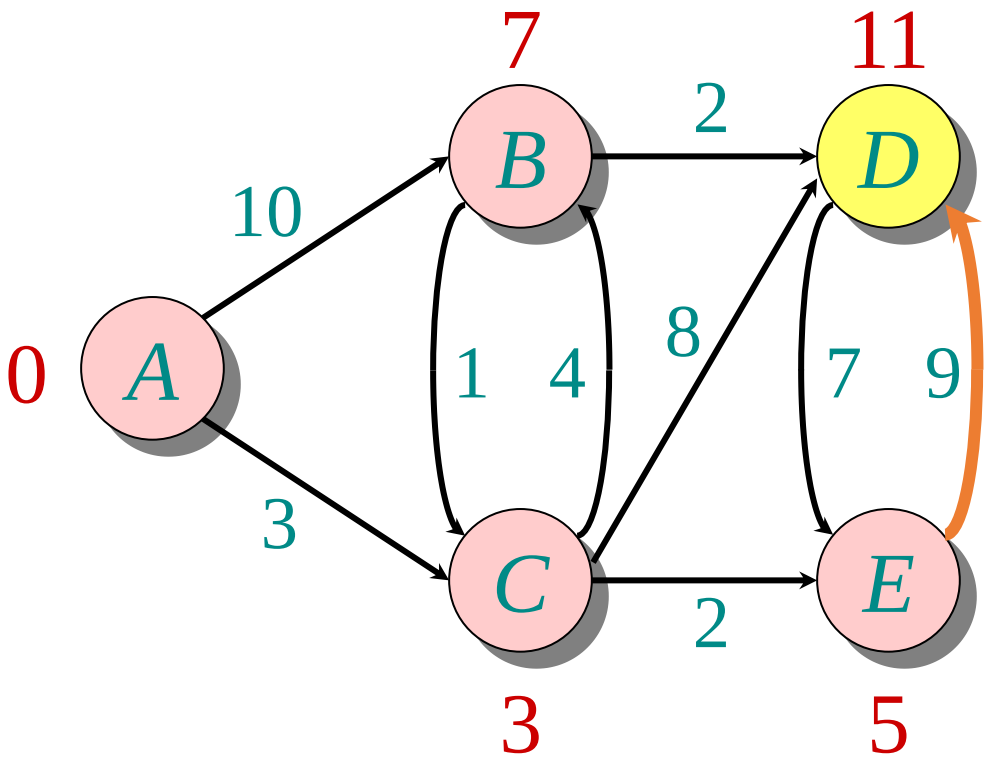


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E* }

Another example

- Demonstrate Dijkstra's Algorithm from node A



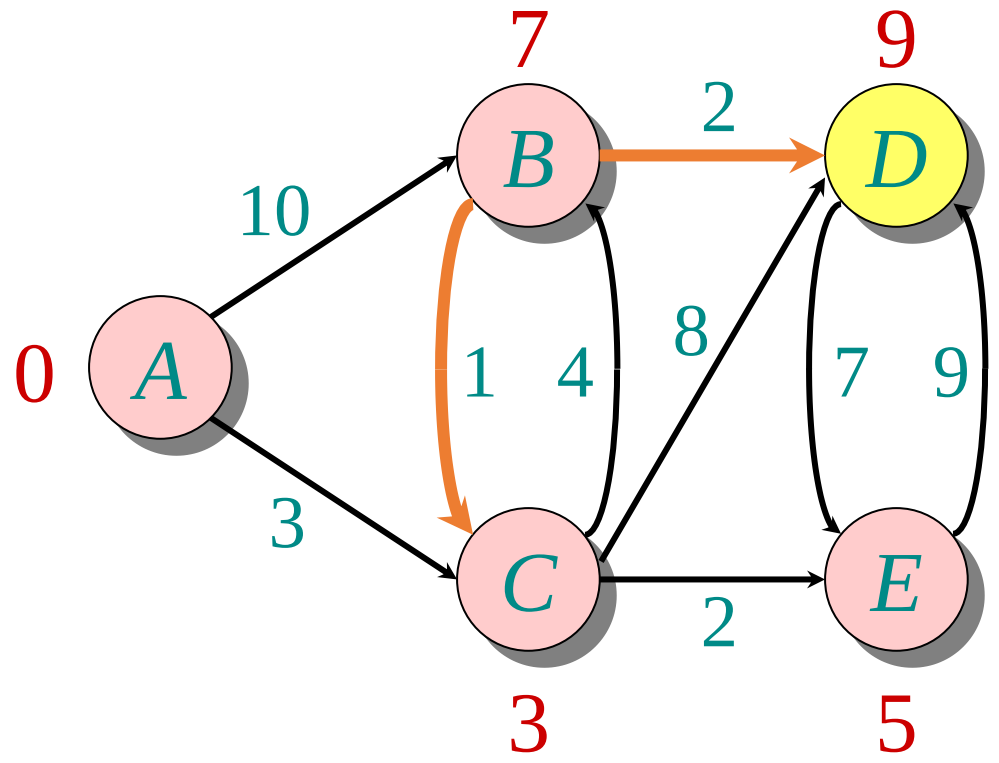
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E*, *B* }

Another example

- Demonstrate Dijkstra's Algorithm from node A

Relax all edges leaving *B*:

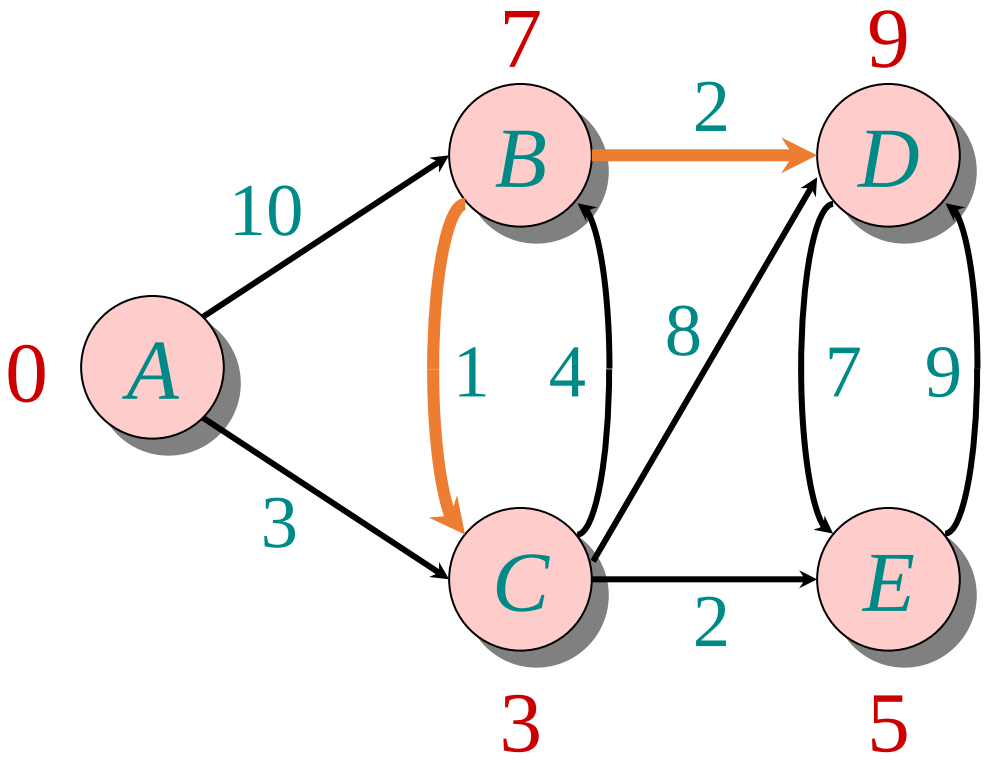


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B* }

Another example

- Demonstrate Dijkstra's Algorithm from node A

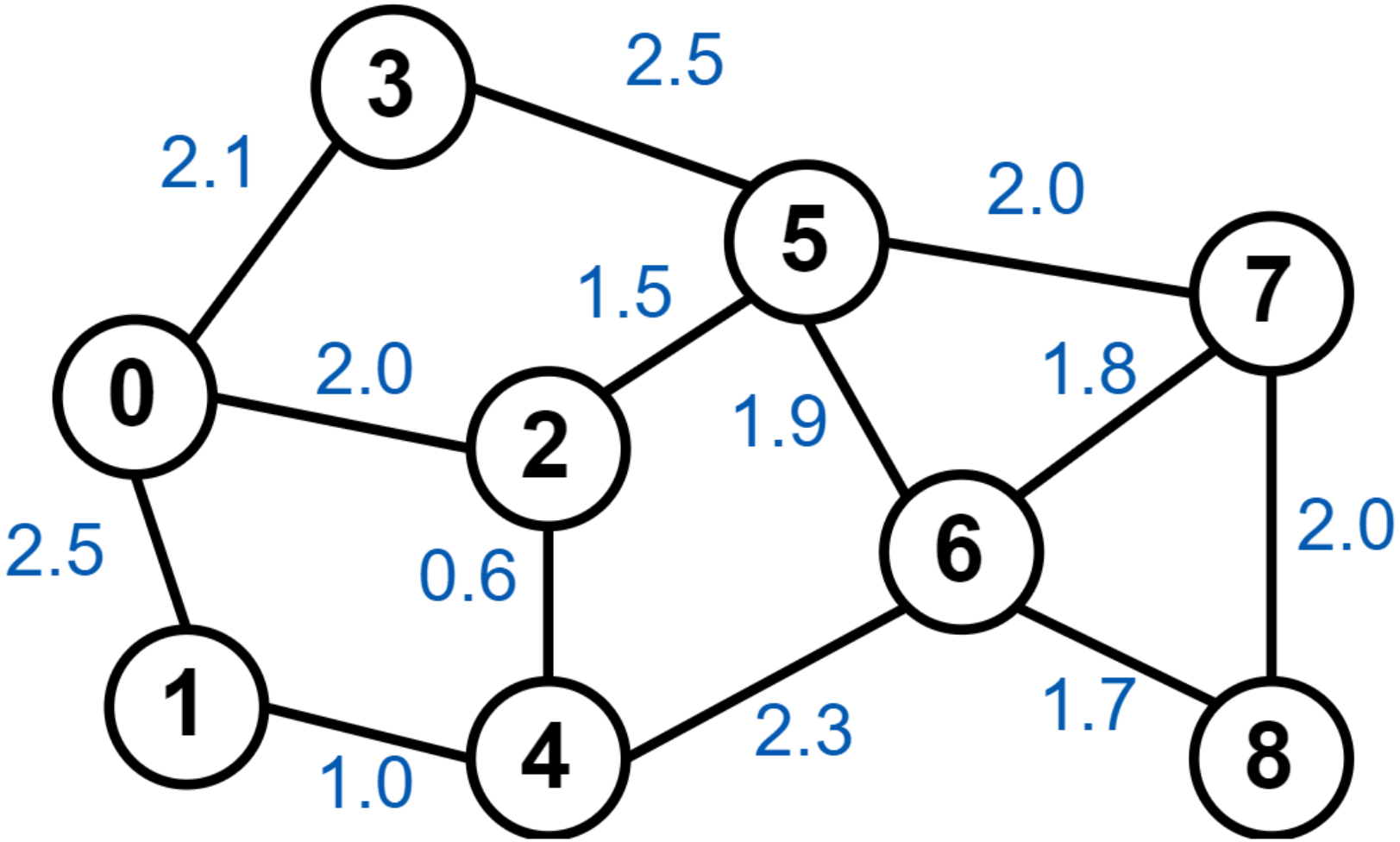


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B*, *D* }

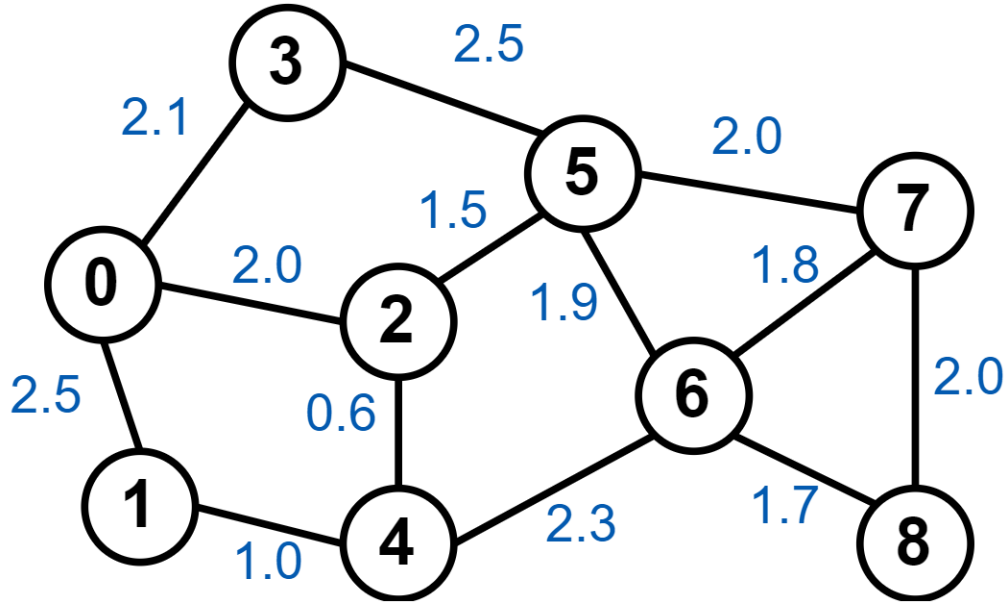
Another example

- Demonstrate Dijkstra's Algorithm from node **0**



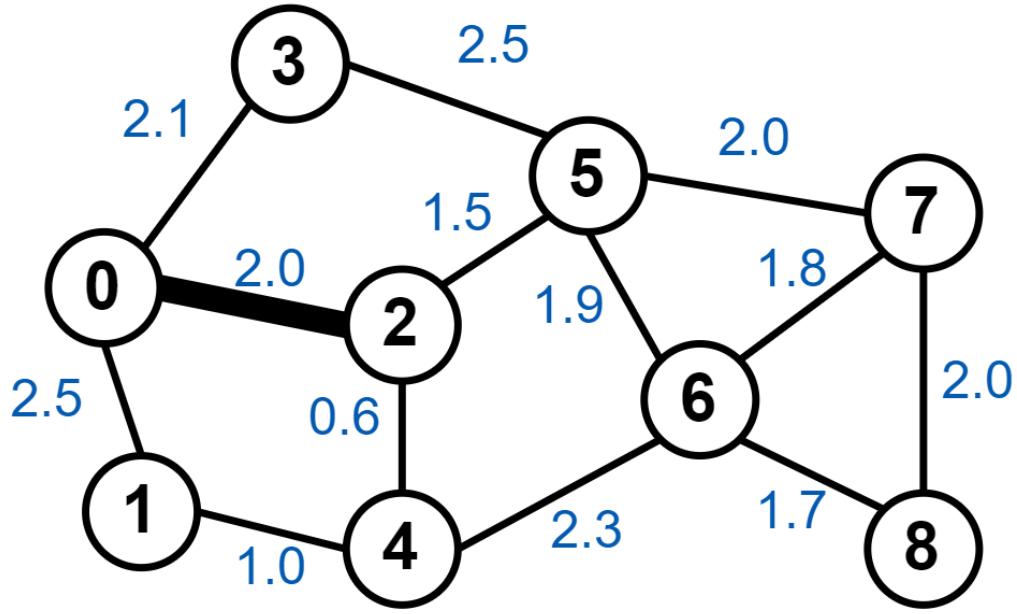
Another example

- Demonstrate Dijkstra's Algorithm from noc



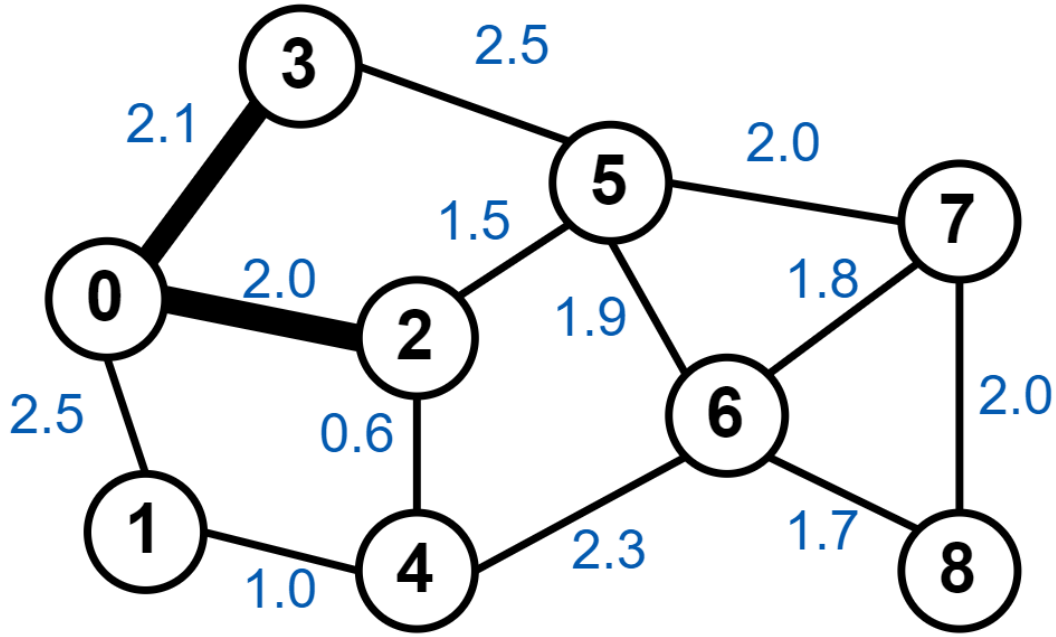
0	1	2	3	4	5	6	7	8
0	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)

Another example



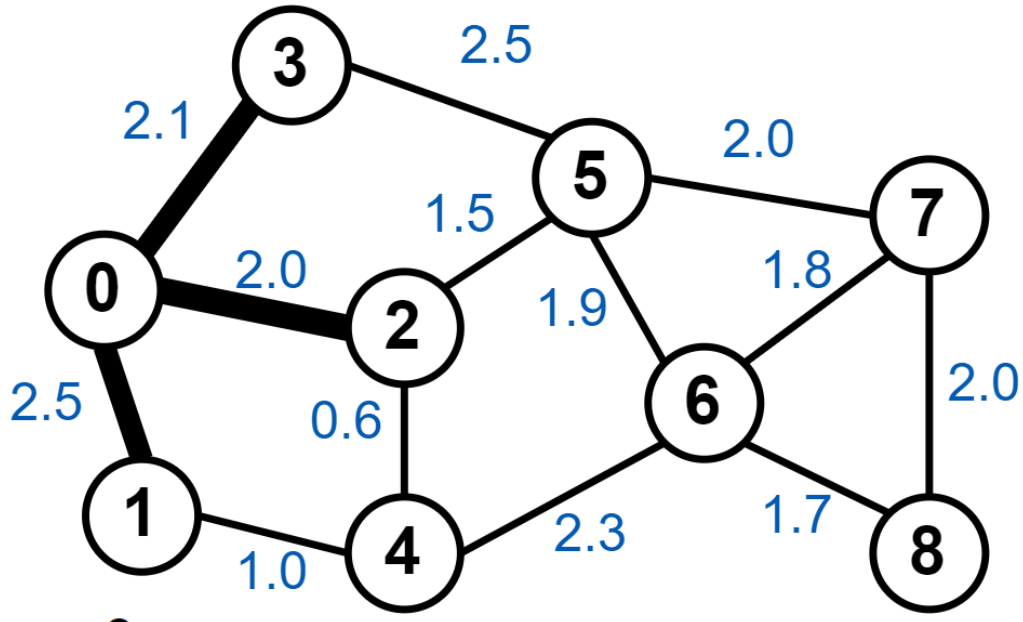
0	1	2	3	4	5	6	7	8
0	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$
–	$(2.5, 0)$	**$(2.0, 0)$**	$(2.1, 0)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$

Another example



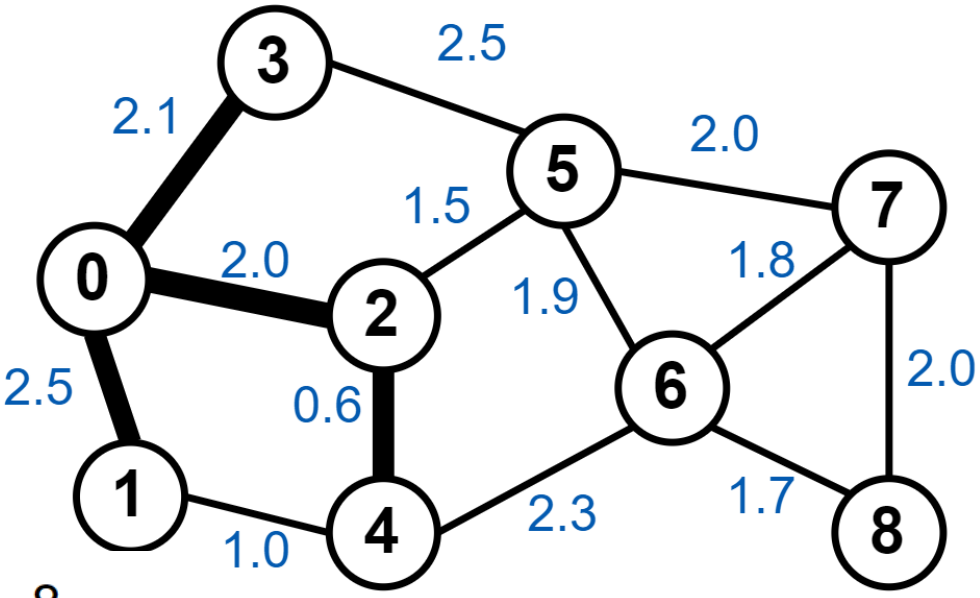
0	1	2	3	4	5	6	7	8
0	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$
–	$(2.5, 0)$	**$(2.0, 0)$**	$(2.1, 0)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$
–	$(2.5, 0)$	–	**$(2.1, 0)$**	$(2.6, 2)$	$(3.5, 2)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$

Another example



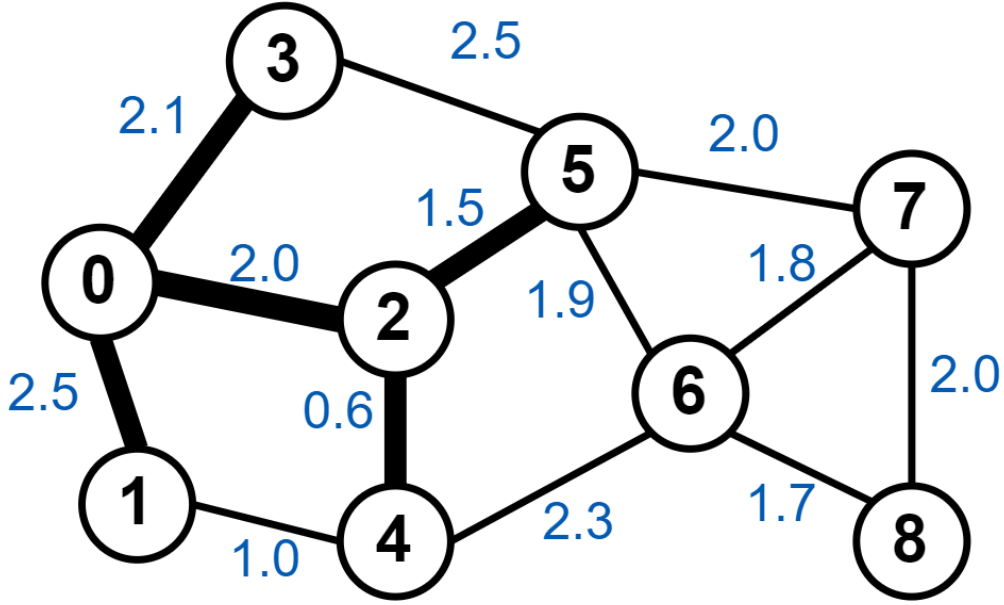
0	1	2	3	4	5	6	7	8
0	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	**(2.0, 0)**	(2.1, 0)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	–	**(2.1, 0)**	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	**(2.5, 0)**	–	–	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)

Another example



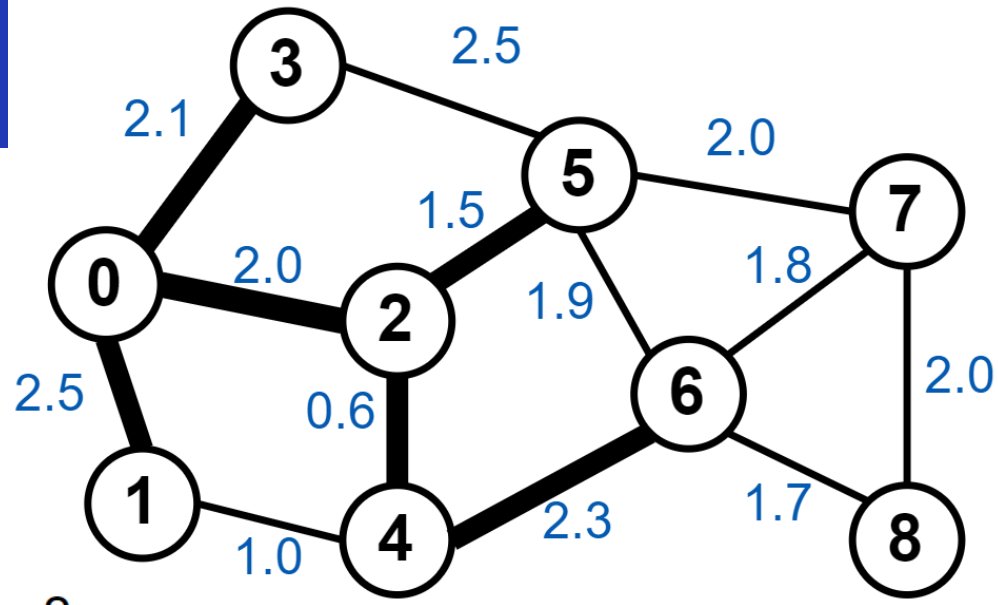
0	1	2	3	4	5	6	7	8
0	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	**(2.0, 0)**	(2.1, 0)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	–	**(2.1, 0)**	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	**(2.5, 0)**	–	–	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	**(2.6, 2)**	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)

Another example



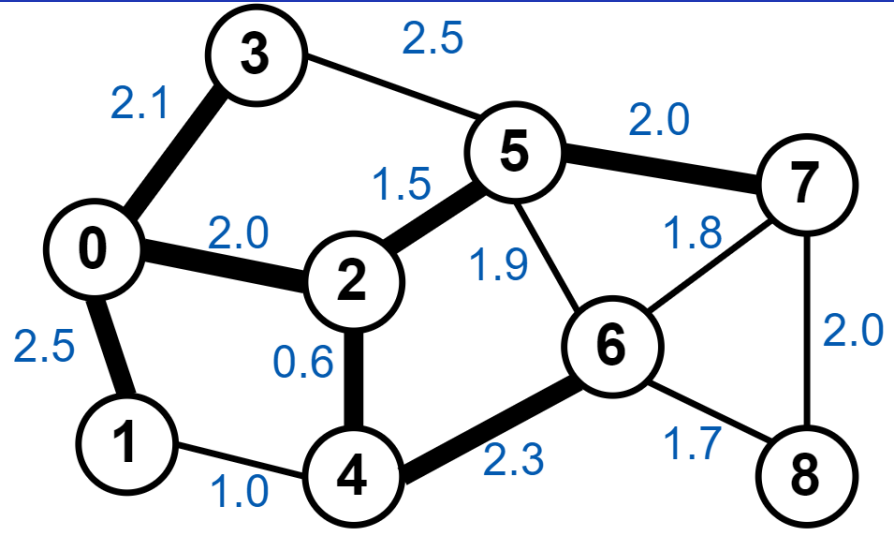
0	1	2	3	4	5	6	7	8
0	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	**(2.0, 0)**	(2.1, 0)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	–	**(2.1, 0)**	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	**(2.5, 0)**	–	–	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	**(2.6, 2)**	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	–	**(3.5, 2)**	(4.9, 4)	(∞,–)	(∞,–)

Another example



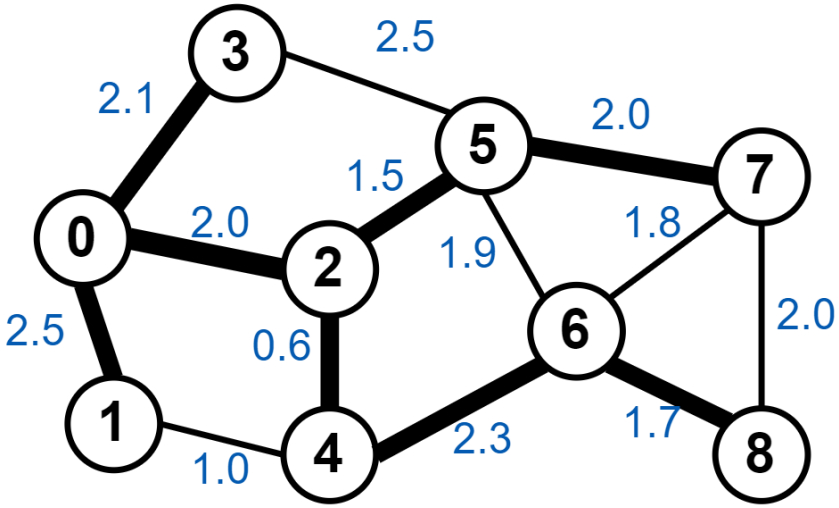
0	1	2	3	4	5	6	7	8
0	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	**(2.0, 0)**	(2.1, 0)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	–	**(2.1, 0)**	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	**(2.5, 0)**	–	–	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	**(2.6, 2)**	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	–	**(3.5, 2)**	(4.9, 4)	(∞,–)	(∞,–)
–	–	–	–	–	–	**(4.9, 4)**	(5.5, 5)	(∞,–)

Another example



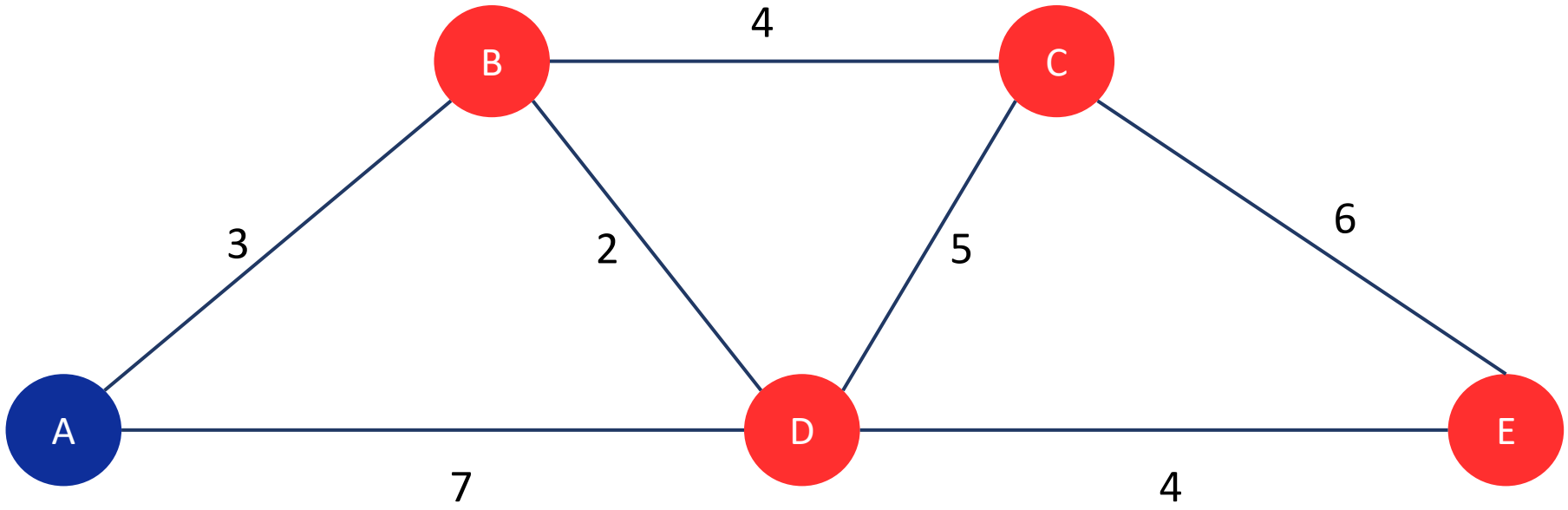
0	1	2	3	4	5	6	7	8
0	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	**(2.0, 0)**	(2.1, 0)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	–	**(2.1, 0)**	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	**(2.5, 0)**	–	–	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	**(2.6, 2)**	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	–	**(3.5, 2)**	(4.9, 4)	(∞,–)	(∞,–)
–	–	–	–	–	–	**(4.9, 4)**	(5.5, 5)	(∞,–)
–	–	–	–	–	–	–	**(5.5, 5)**	(6.6, 6)

Another example

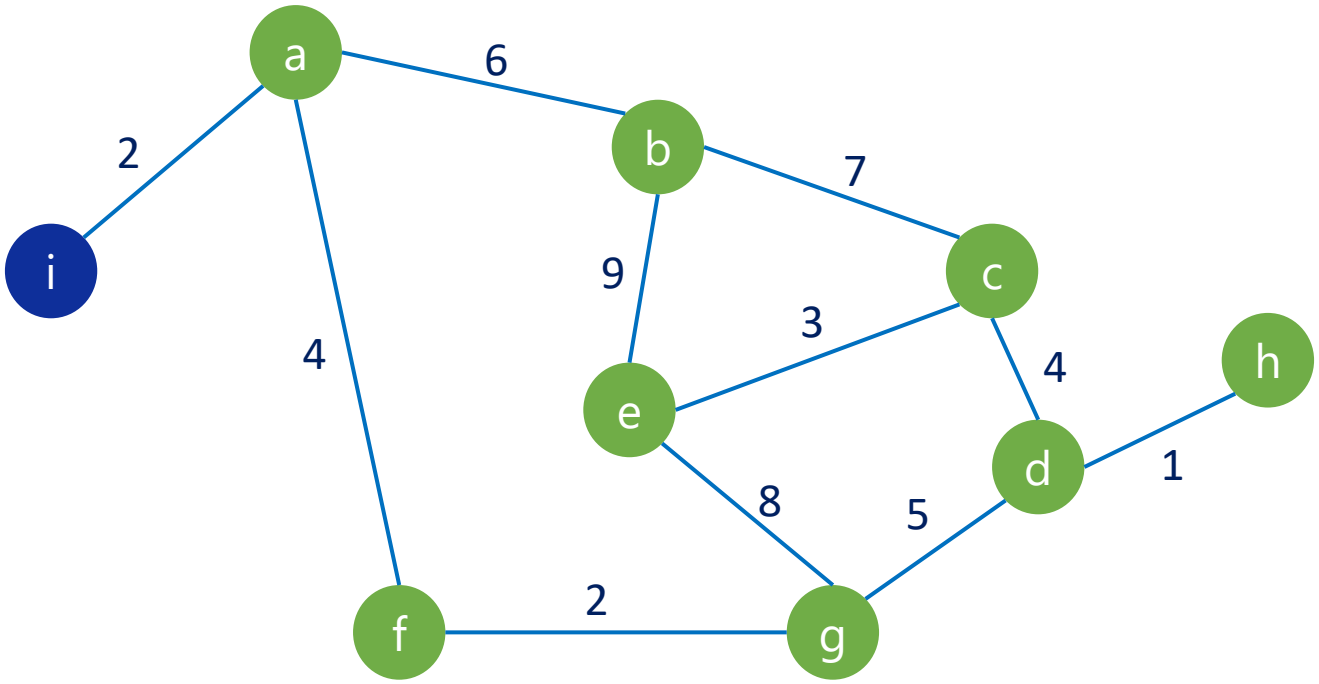


0	1	2	3	4	5	6	7	8
0	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	**(2.0, 0)**	(2.1, 0)	(∞,–)	(∞,–)	(∞,–)	(∞,–)	(∞,–)
–	(2.5, 0)	–	**(2.1, 0)**	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	**(2.5, 0)**	–	–	(2.6, 2)	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	**(2.6, 2)**	(3.5, 2)	(∞,–)	(∞,–)	(∞,–)
–	–	–	–	–	**(3.5, 2)**	(4.9, 4)	(∞,–)	(∞,–)
–	–	–	–	–	–	**(4.9, 4)**	(5.5, 5)	(∞,–)
–	–	–	–	–	–	–	**(5.5, 5)**	(6.6, 6)
–	–	–	–	–	–	–	–	**(6.6, 6)**

- Demonstrate Dijkstra's Algorithm from node A

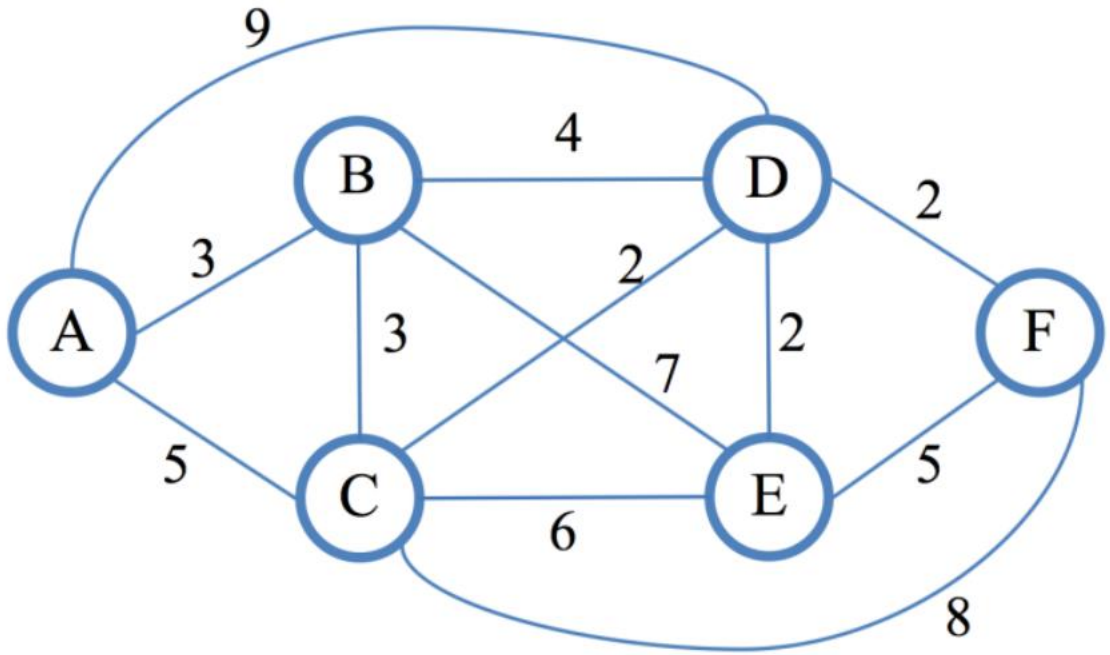


- Demonstrate Dijkstra's Algorithm from node i



Exercise

- Demonstrate Dijkstra's Algorithm from node A



Conclusion

- Dijkstra Algorithm is a graph algorithm for finding the shortest path from **a source node** to **all other nodes** in a graph
- It is a type of greedy algorithm
- It only works on weighted graphs with positive weights
- Adjacency matrix: $O(V^2)$ → **Why ?**
- Adjacency list: $O((V + E)\log V)$ → **Why ?**

THANK YOU
for YOUR ATTENTION