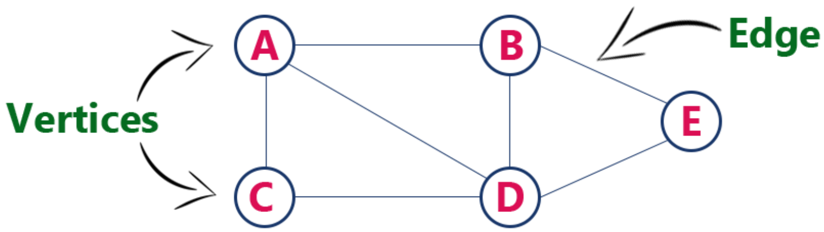
Graphs

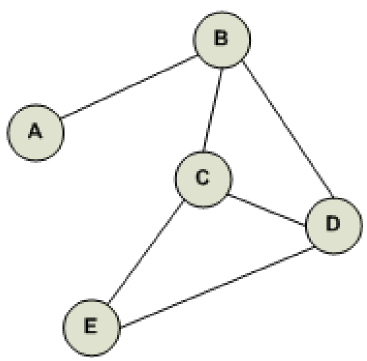
Department of Computer Science & Engineering
The Pennsylvania State University

Graph

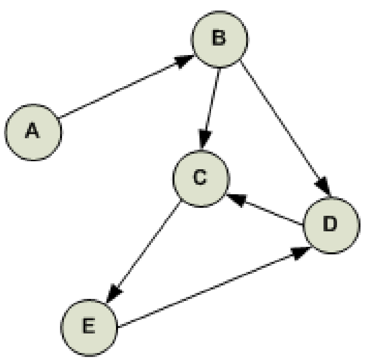
- A graph is a data structure with nodes (also called vertices) that are connected by edges



- Graphs can be directed or undirected



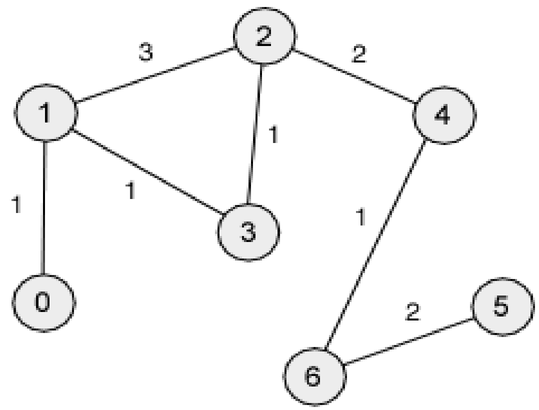
Undirected Graph



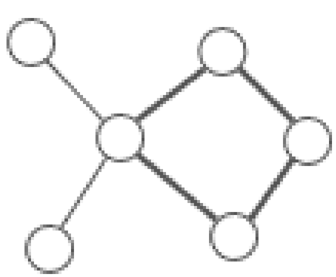
Directed Graph

Graph

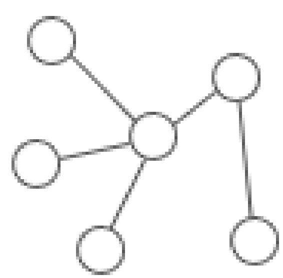
- The edges could represent distance or weight



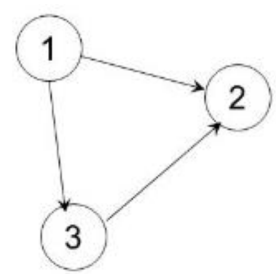
- A graph might contain cycles



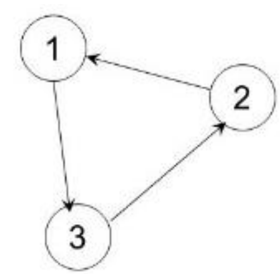
CYCLIC GRAPH



ACYCLIC GRAPH



acyclic



cyclic

Key Terms

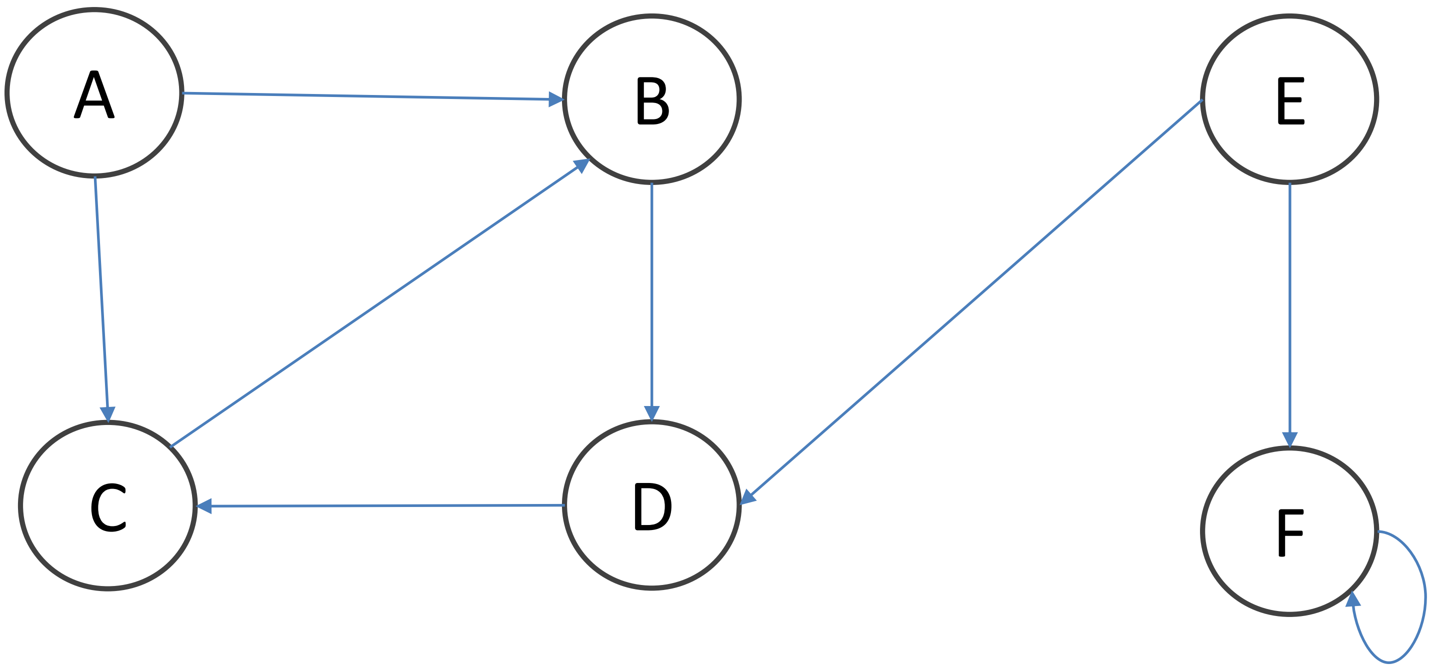
- **Vertex** (Node): a fundamental part of a graph. It can have a name and additional information
- **Edge**: connects two vertices to show that there is a relationship between them. Edges may be one-way or two-way
- **Weight**: edges may be weighted to show that there is a cost to go from one vertex to another
- **Path**: sequence of vertices that are connected by edges
- **Cycle**: A cycle in a directed graph is a path that starts and ends at the same vertex. A graph with no cycles is called an acyclic graph. A directed graph with no cycles is called a directed acyclic graph (DAG)

A graph can be represented as $G=(V,E)$, where V is a set of vertices and E is a set of edges. Each edge is a tuple (v,w) where $w,v \in V$.

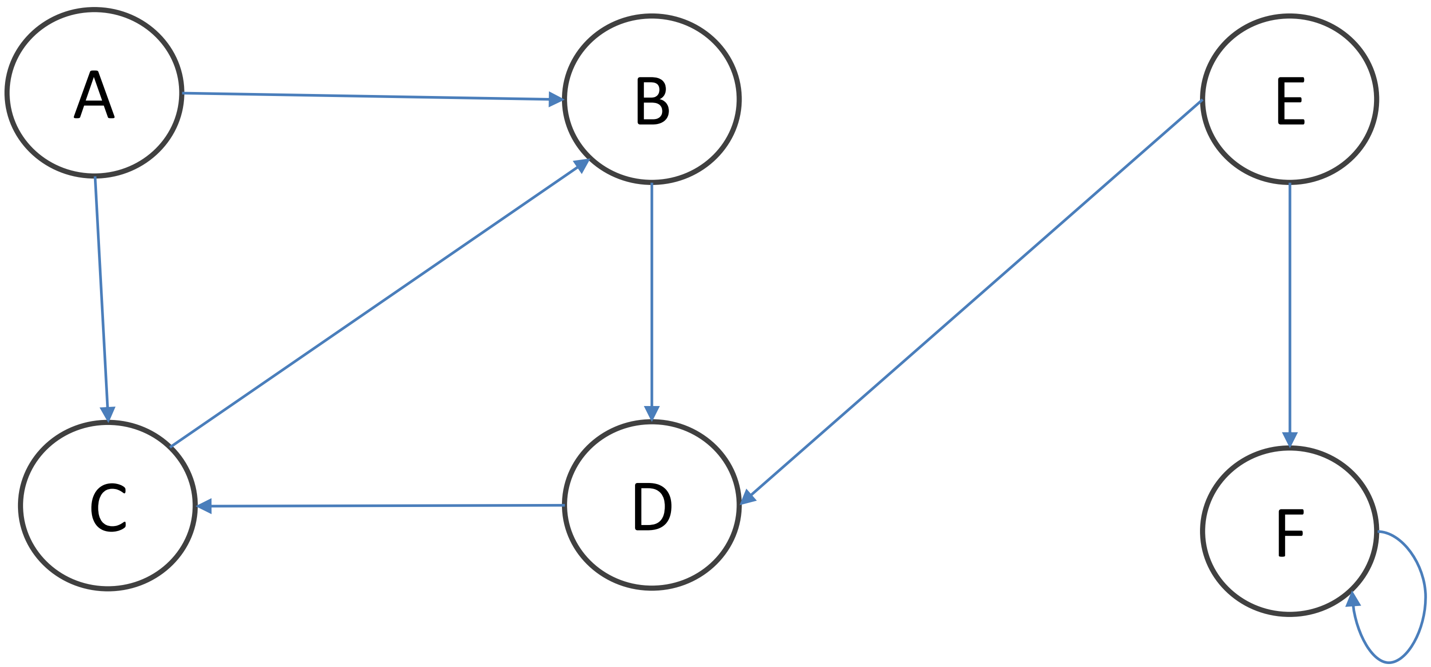
Edges

- Tree Edge: edges that we encounter while down one path of a graph
- Non-tree edge:
  - ✓ Forward Edge: it allows us to move “forward” through the graph, and could potentially be part of another path down the tree.
  - ✓ Backward edge: connects a node in a graph “back up” to one of its ancestors (its parent, grandparent or itself).
  - ✓ Cross edge: connects to sibling nodes that don’t necessarily share an ancestor in a tree path, but connects them anyways.
- In an undirected graph, there are no forward edges or cross edges. Every single edge must be either a tree edge or a back edge

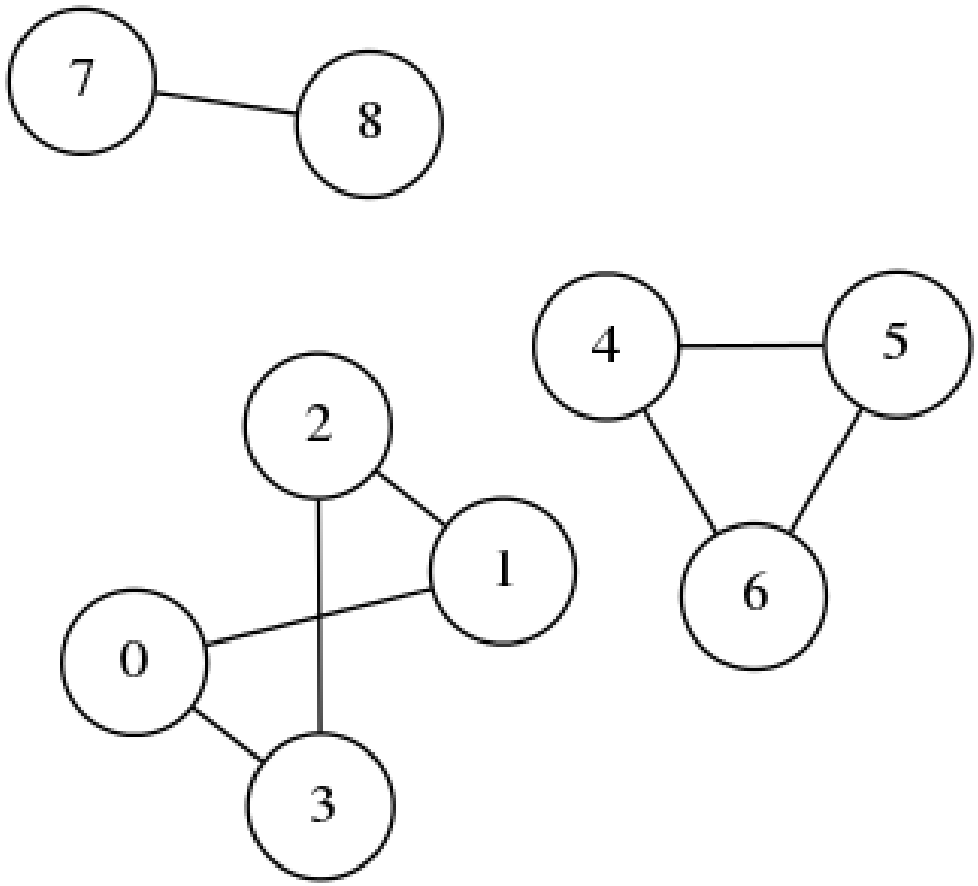
Edges



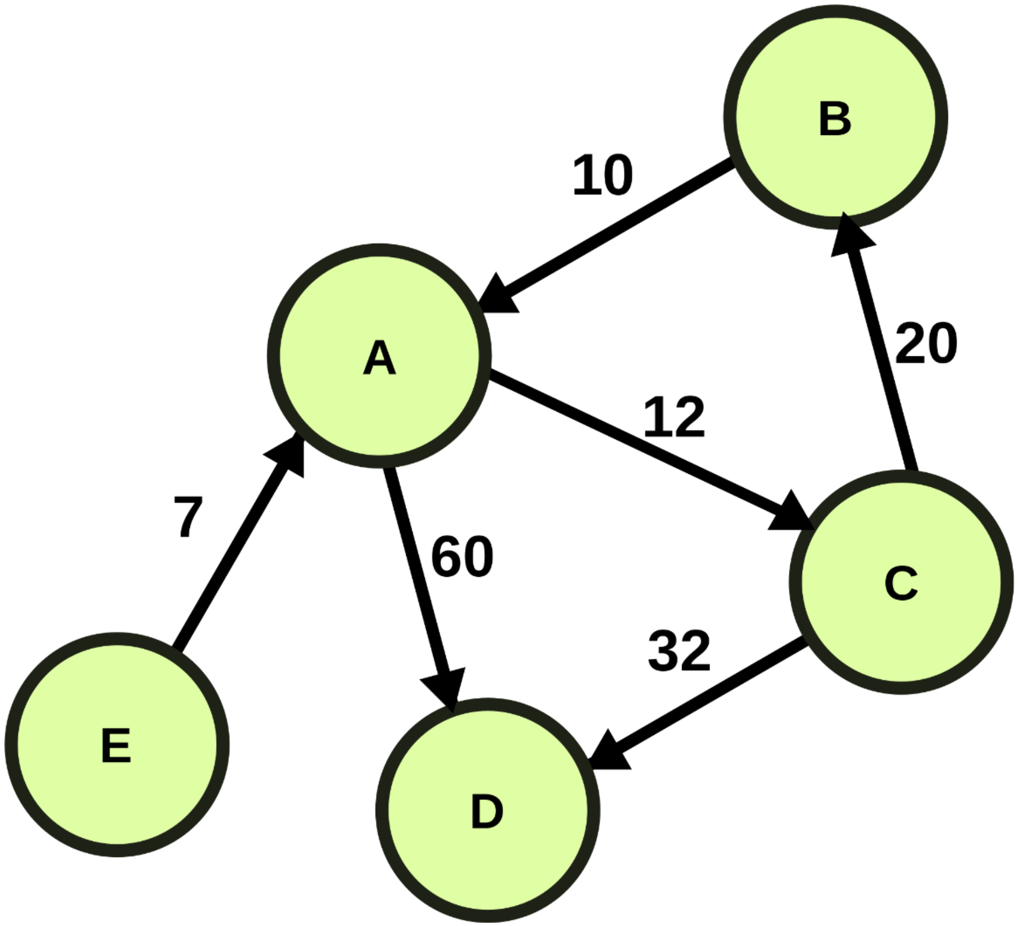
Edges



Connected Components



Graph



G=(V,E) where:

V = {A, B, C, D, E}
E = {(A,C,12), (A,D,60), (B,A,10),
(C,B,20), (C,D,32), (E,A,7)}

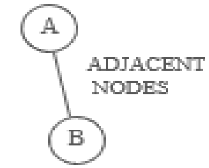
Path from A to B = (A, C, B)

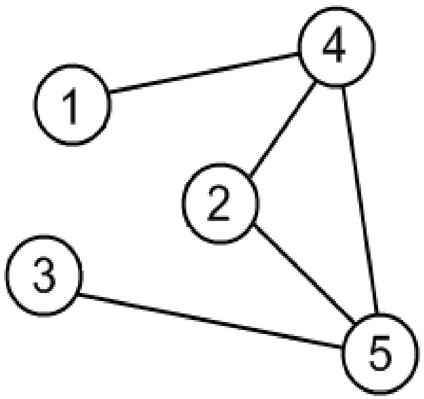
Cycle = (A, C, B, A)

Graph Implementation

Adjacency Matrix

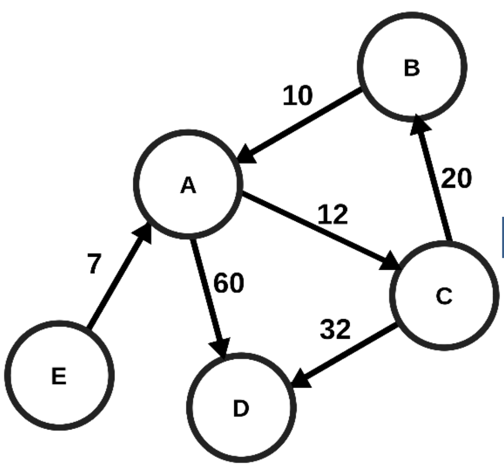
When two vertices are connected by an edge, we say that they are adjacent





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

```
graph = [
   [0, 0, 0, 1, 0],
   [0, 0, 0, 1, 1],
   [0, 0, 0, 0, 1],
   [1, 1, 0, 0, 1],
   [0, 1, 1, 1, 0],
   ]
```



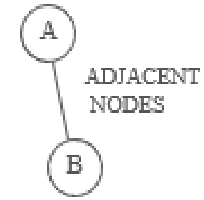
	A	B	C	D	E
A			12	60	
B	10				
C		20		32	
D					
E	7				

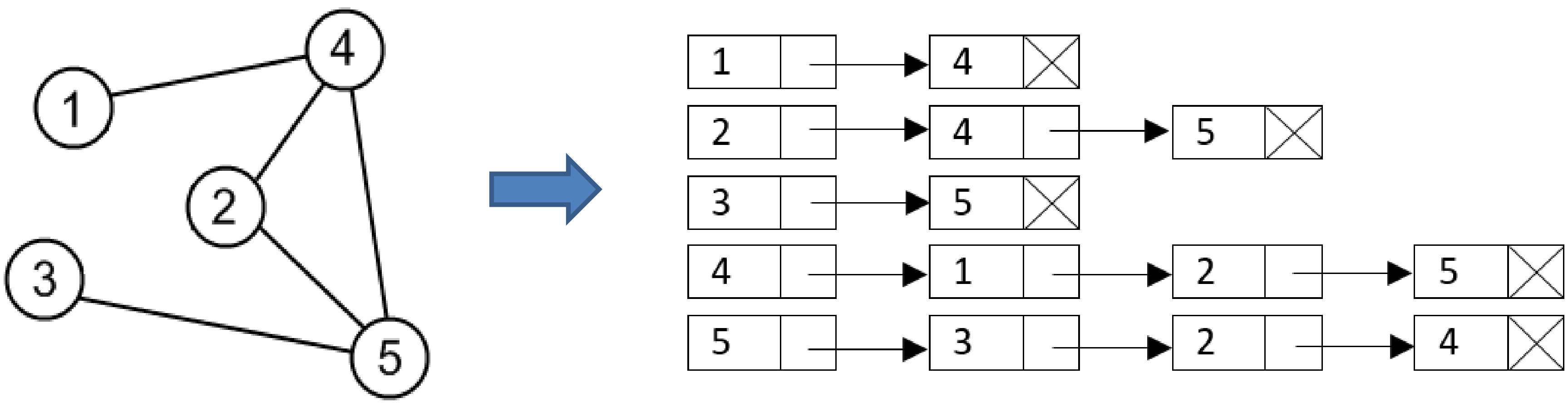
```
graph = [
   [0, 0, 12, 60, 0],
   [10, 0, 0, 0, 0],
   [0, 20, 0, 32, 0],
   [0, 0, 0, 0, 0],
   [7, 0, 0, 0, 0],
   ]
```

Graph Implementation

Adjacency List

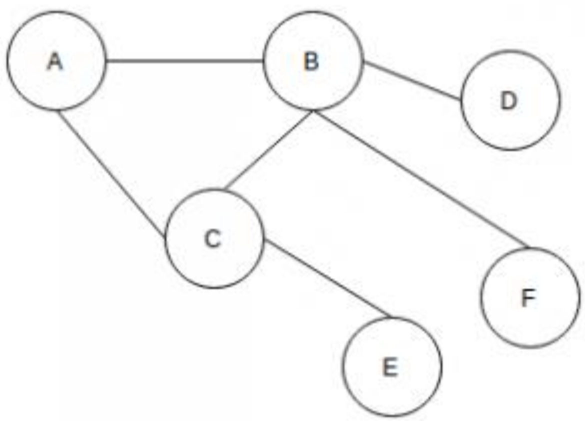
When two vertices are connected by an edge, we say that they are adjacent






```
graph = {
    1: [4],
    2: [4, 5],
    3: [5],
    4: [1, 2, 5],
    5: [3, 2, 4],
}
```

Graph Representation

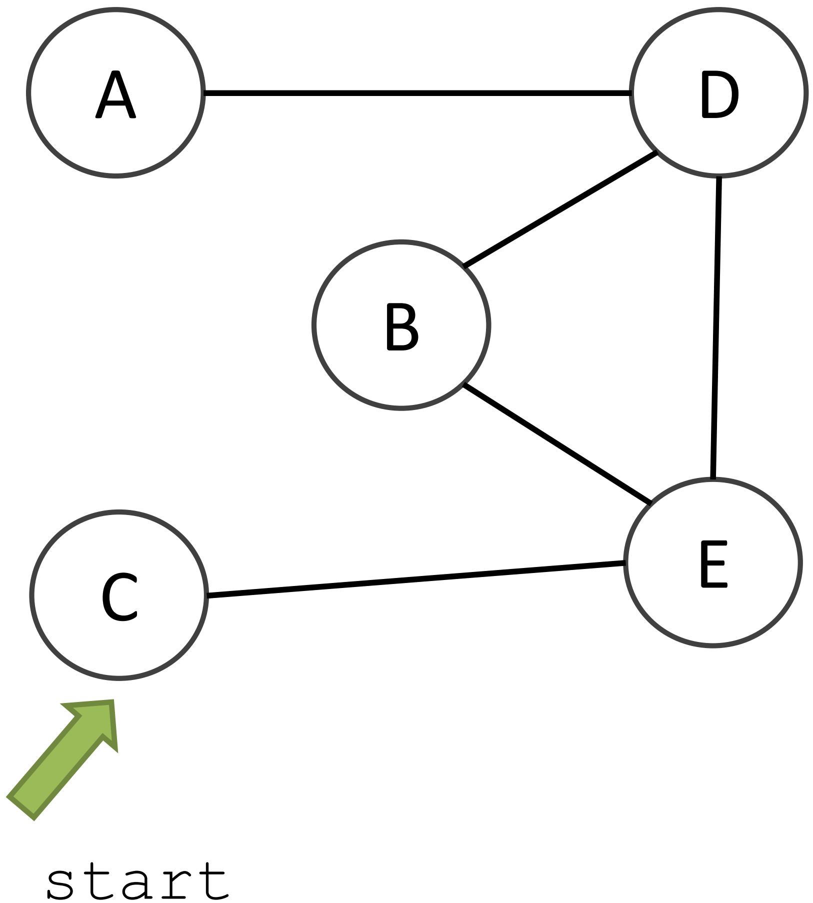


Graph traversals

- Breadth First Search: It starts at some arbitrary node in a graph and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

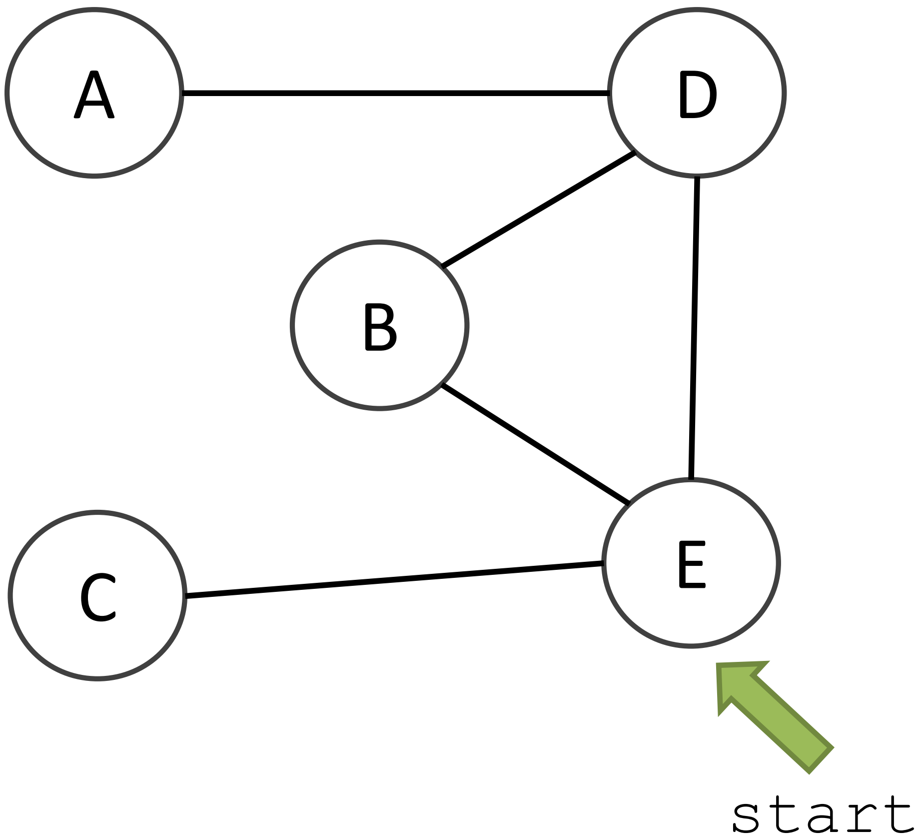
- Depth First Search: It starts at some arbitrary node in a graph and explores as far as possible along each branch before backtracking.

BFS for a graph



BFS:

DFS for a graph



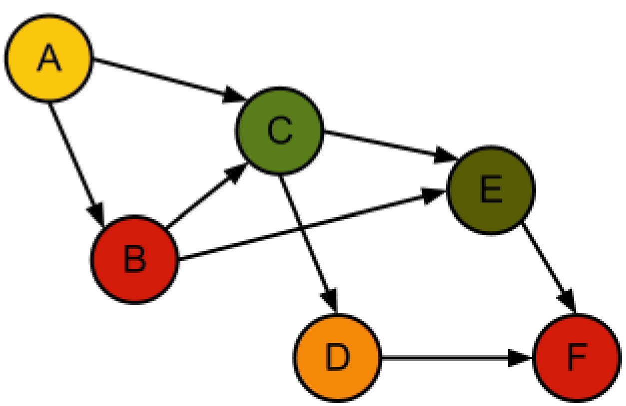
DFS:

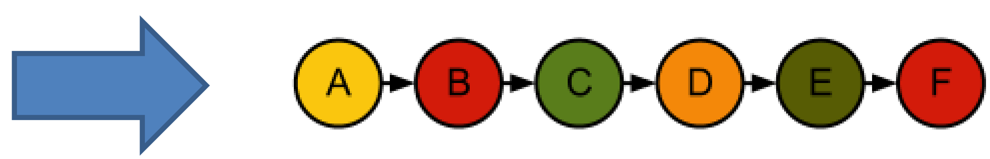
Topological Sorting

- A topological sort takes DAG and produces a linear ordering of all its vertices such that if the graph *G* contains an edge (*v*,*w*) then the vertex *v* comes before the vertex *w* in the ordering

- Any linear ordering in which all the arrows go to the right is a valid solution

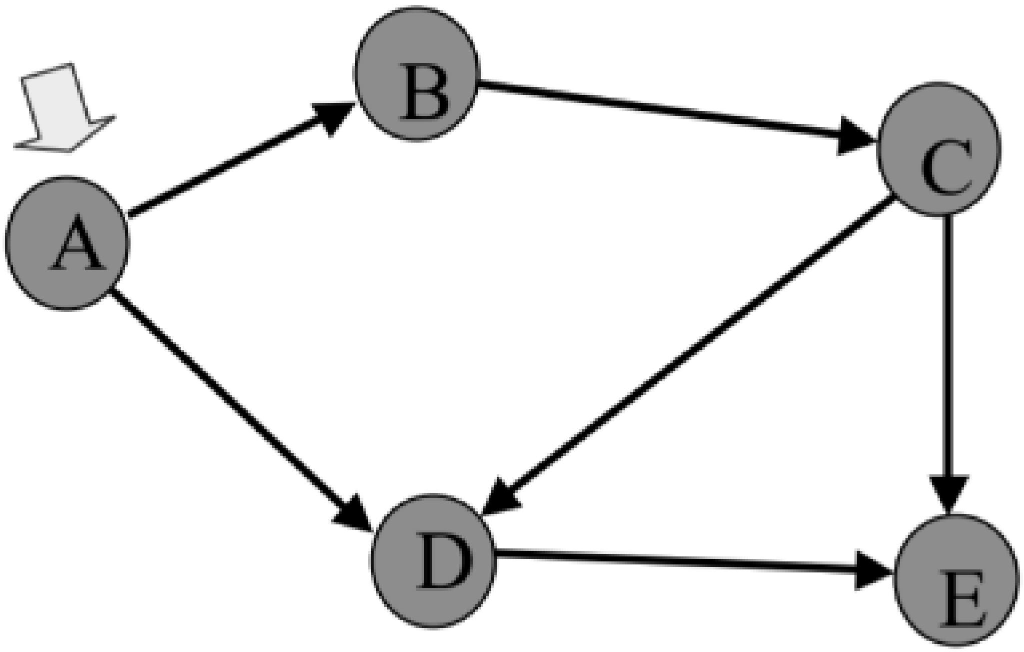
- Topological sorting for a graph is not possible if the graph is not a DAG





Topological Sorting using DFS

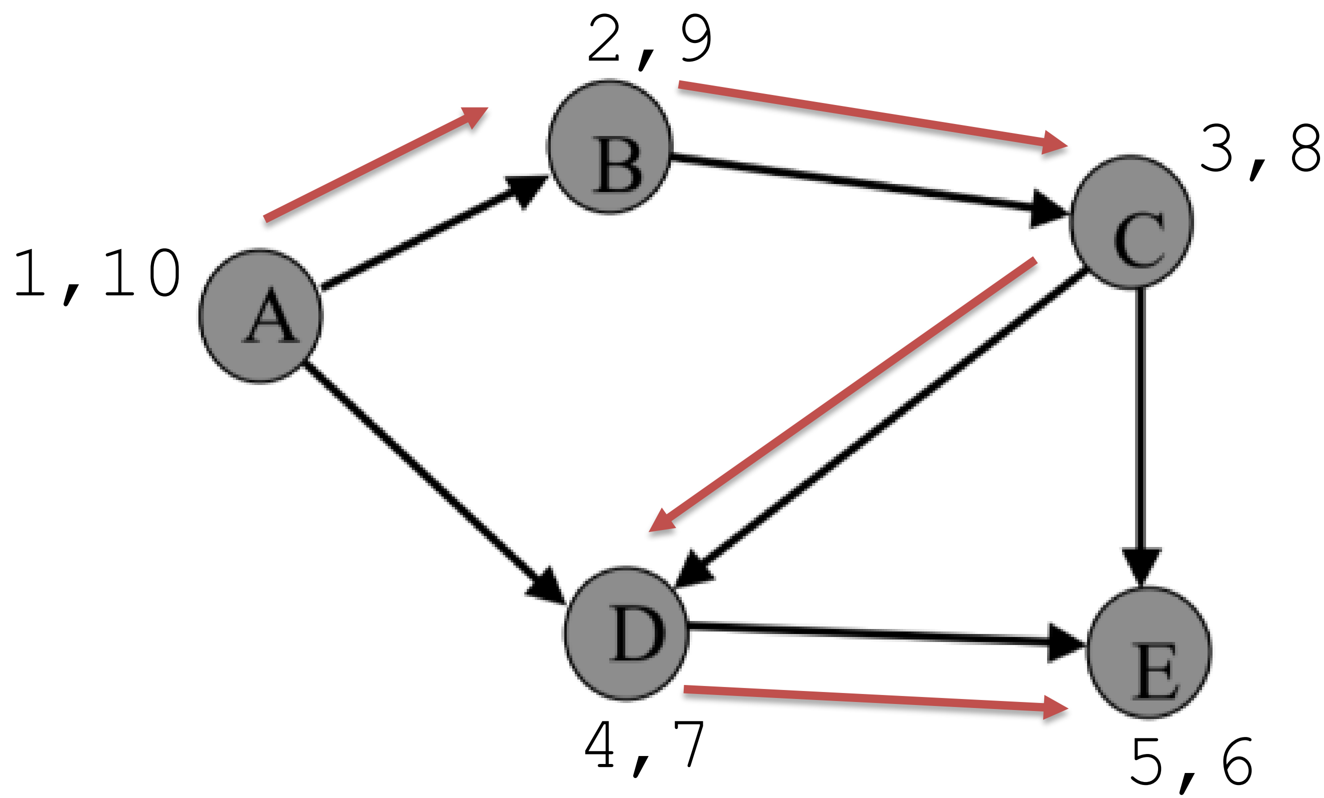
Step 1: Perform DFS for graph *G,* keeping track of starting and ending times



DFS:

Topological Sorting using DFS

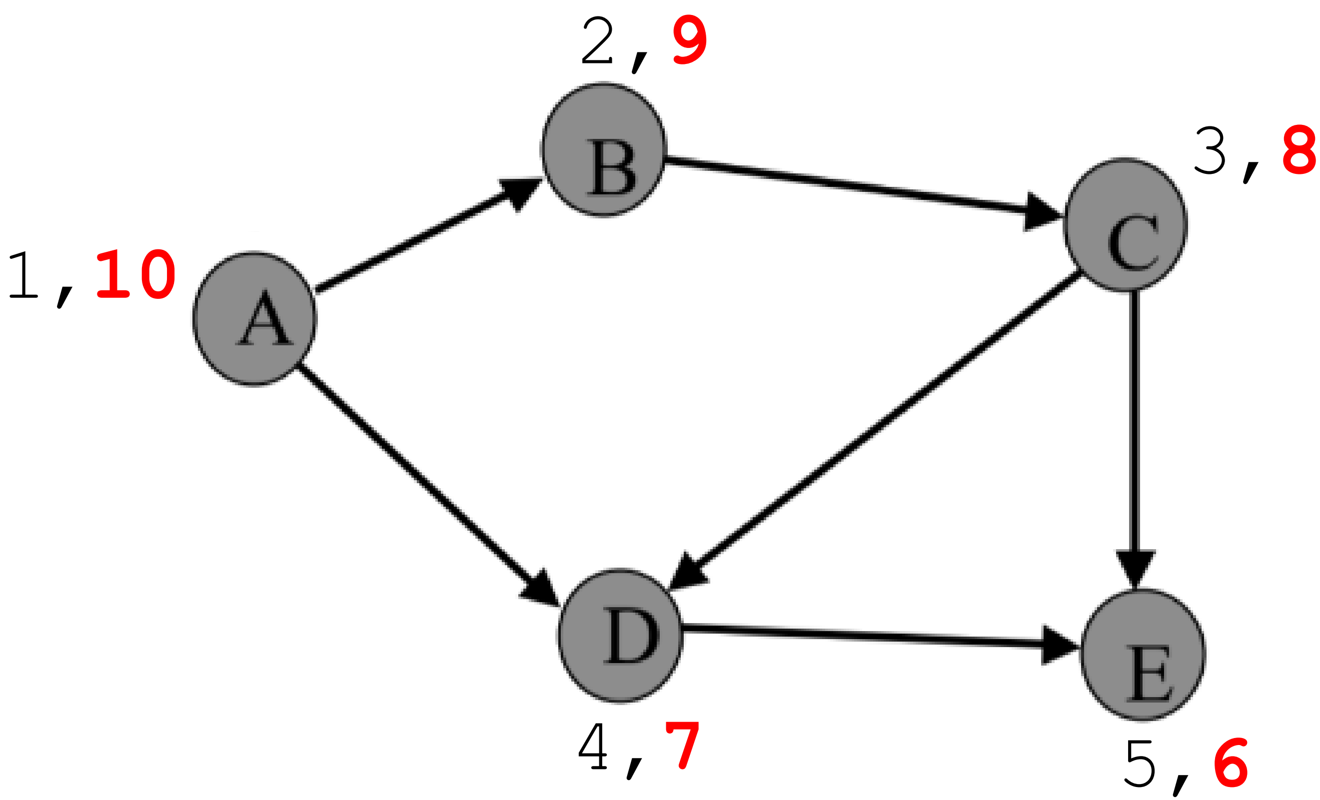
Step 1: Perform DFS for graph *G*



DFS: A, B, C, D, E

Topological Sorting using DFS

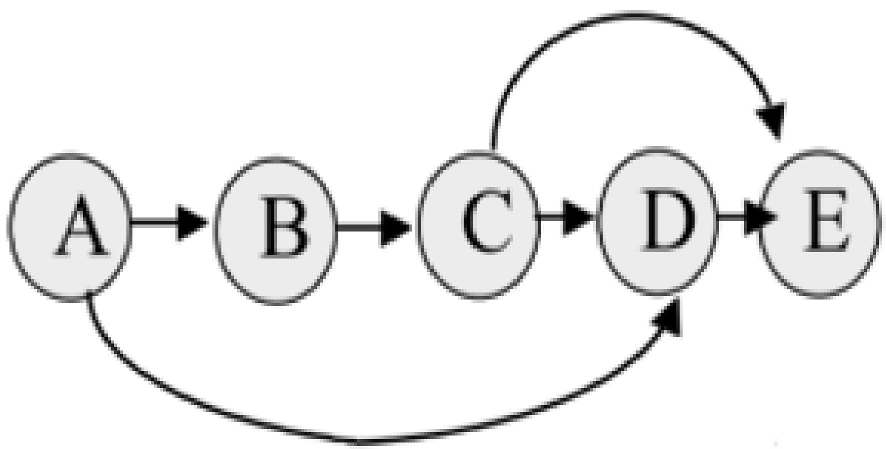
Step 2: Store the vertices in a list in decreasing order of finish time



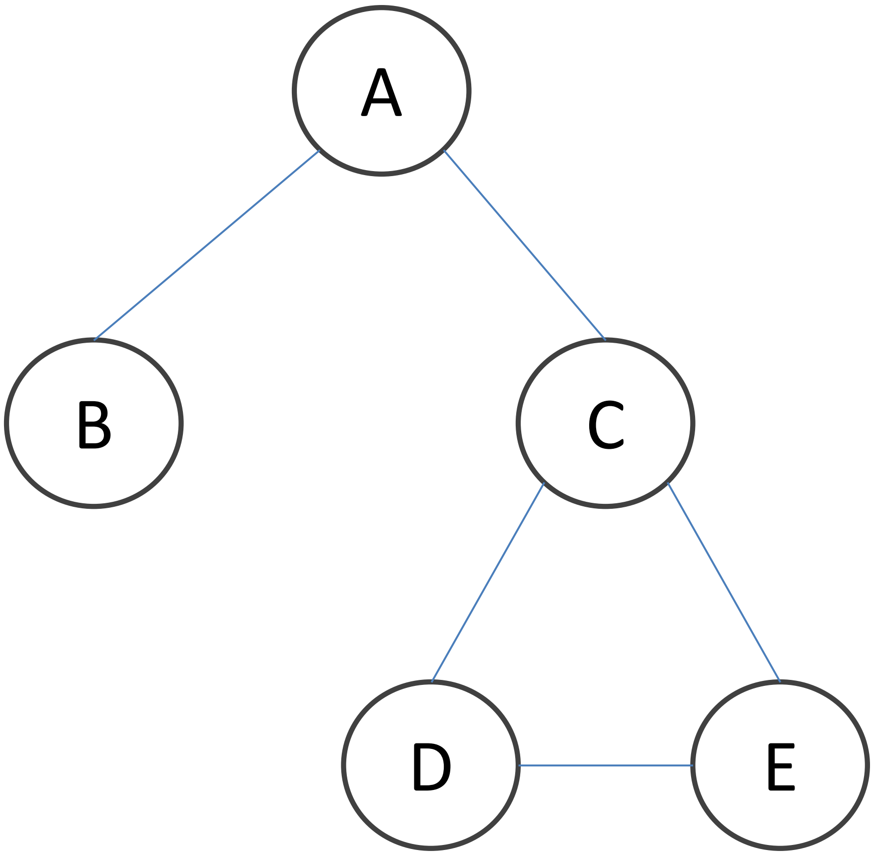
Topological Sorting using DFS

Step 3: Return the ordered list as the result of the topological sort

```
list = ['A', 'B', 'C', 'D', 'E' ]
```



Cycle Detection using DFS



Dijkstra's Algorithm

- Use to determine the shortest path from one node (or vertex) in a graph to every other node within the same graph data structure.

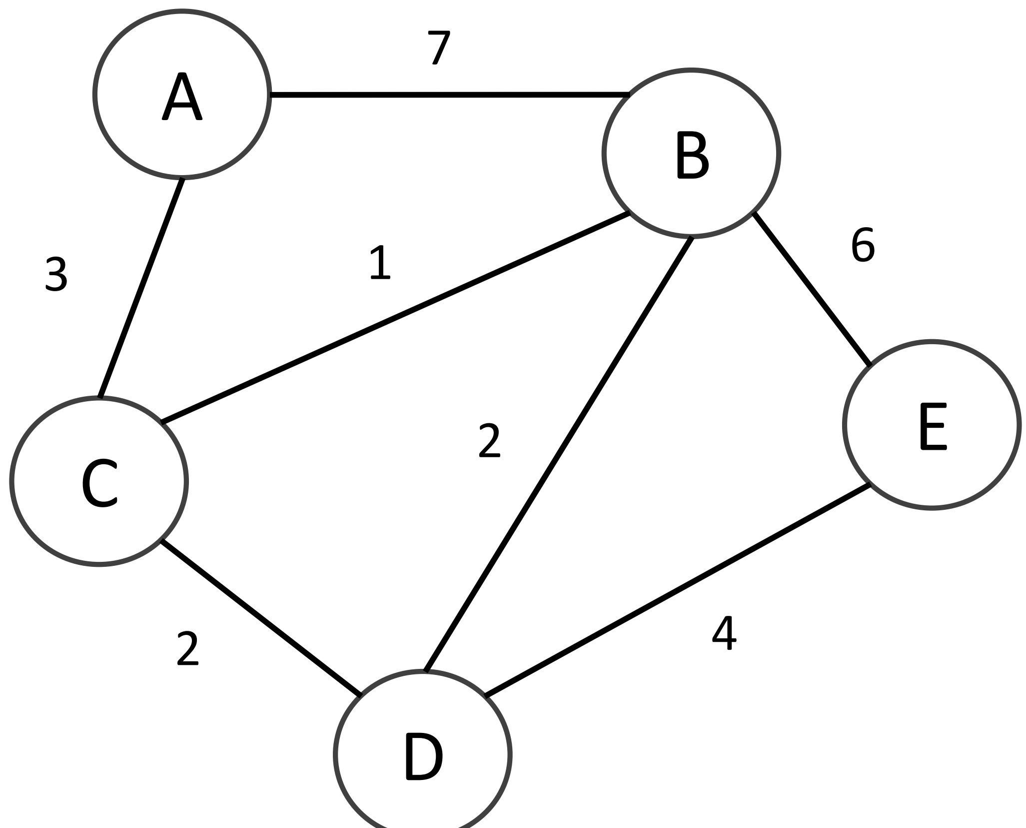
- The algorithm will run until all nodes in the graph have been visited, thus, the shortest path between any 2 nodes can be saved and look up after

Dijkstra's Algorithm

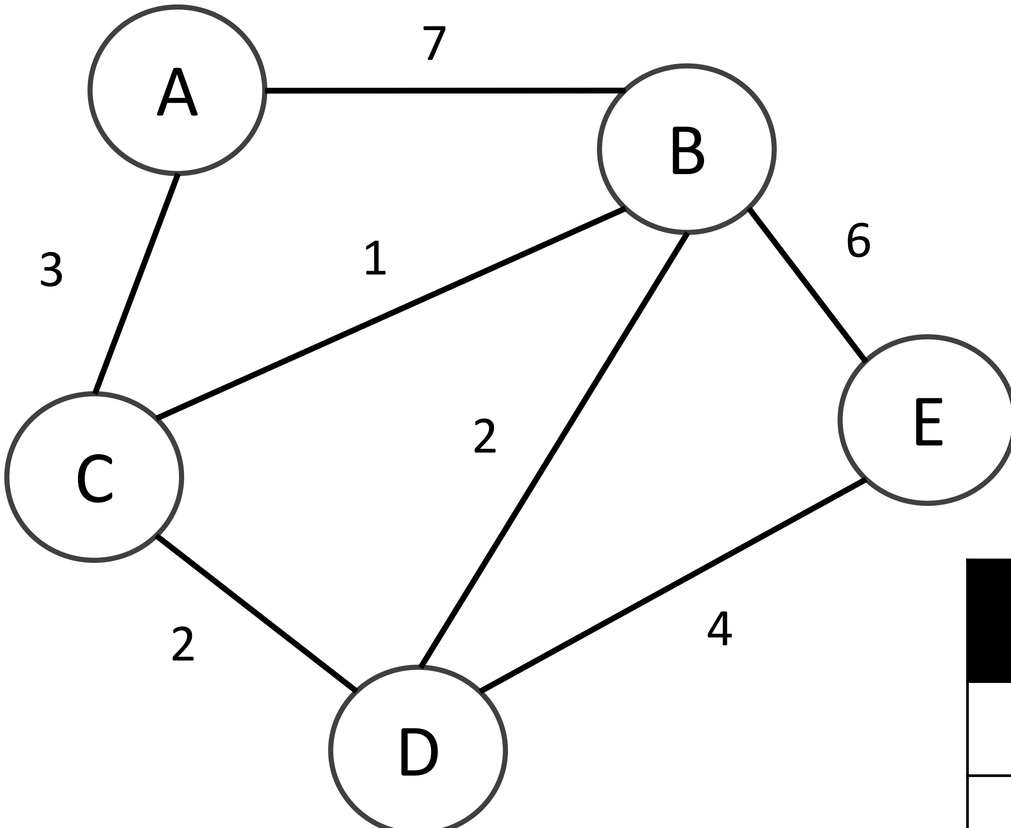
- From the starting node, visit the node with the smallest known distance
- Once you have moved to the smallest-distance node. Check each of its neighboring nodes
- For each neighboring node, compute the distance for the neighboring nodes by summing the distance of the edges leading from the start node
- If the distance to a node is less than a known distance, update the shortest distance for that node

Dijkstra's Algorithm

What is the shortest path between node A and E in the following weighted, undirected graph?



Dijkstra's Algorithm Initialization

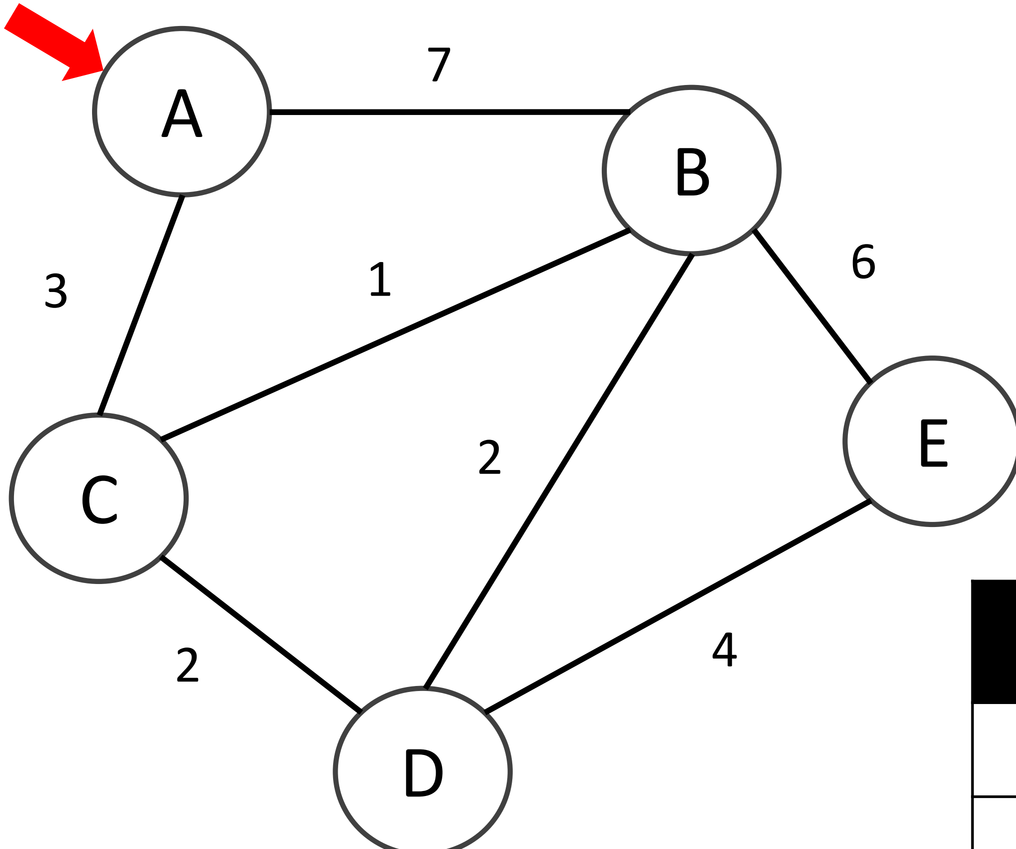


visited = []

unvisited = [A, B, C, D, E]

Node	Shortest distance from A	Previous node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Dijkstra's Algorithm Initialization

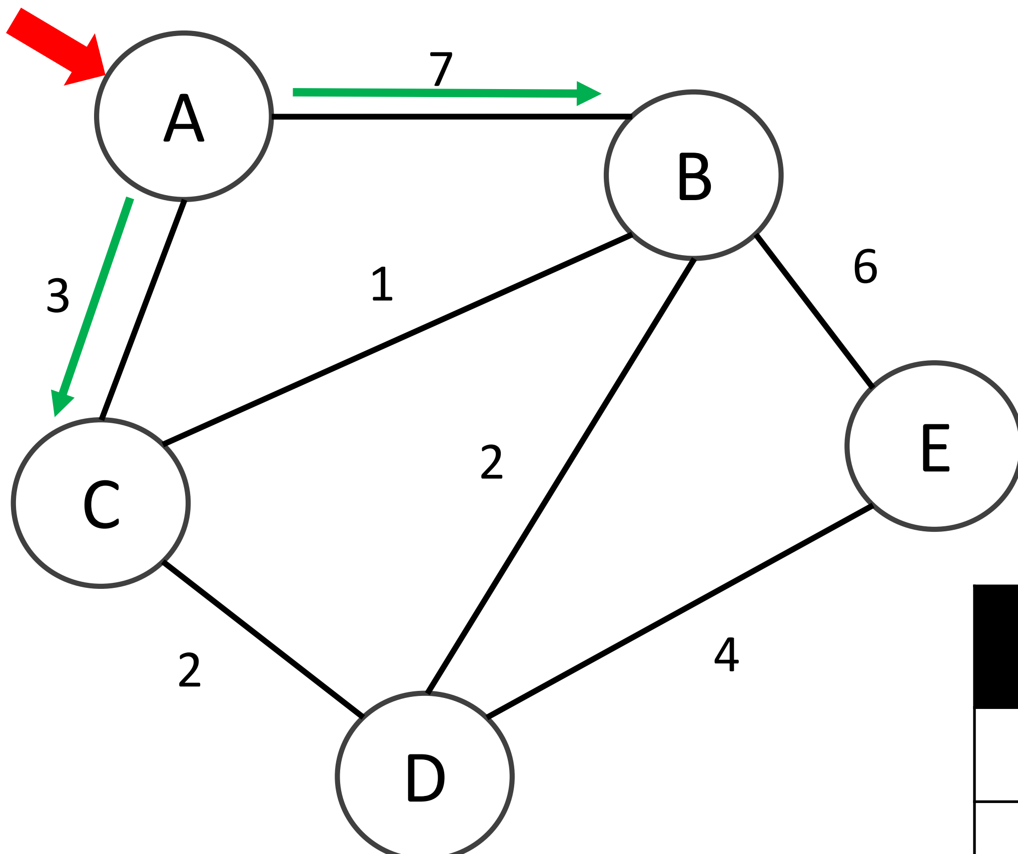


visited = []

unvisited = [A, B, C, D, E]

Node	Shortest distance from A	Previous node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Shortest Path



visited = []

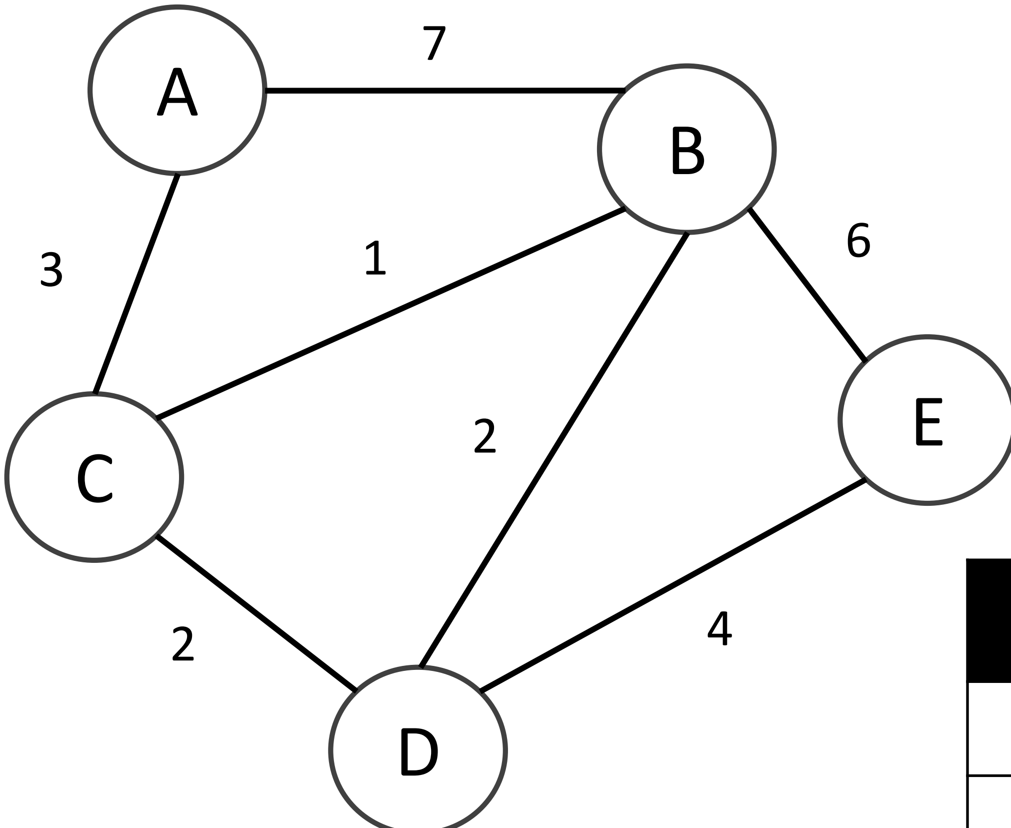
unvisited = [A, B, C, D, E]

Node	Shortest distance from A	Previous node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

distance to B: 0+7=7 $7 < \infty$?

distance to C: 0+3=3 $3 < \infty$?

Shortest Path

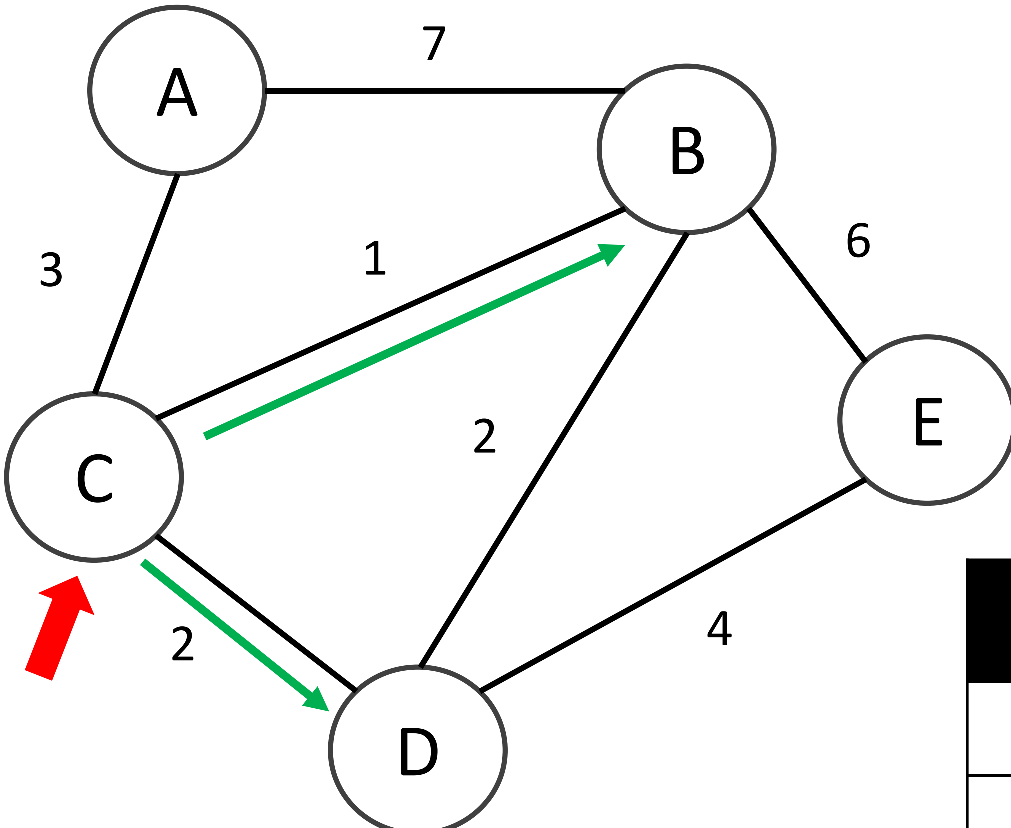


visited = [A]

unvisited = [B, C, D, E]

Node	Shortest distance from A	Previous node
A	0	
B	7	A
C	3	A
D	∞	
E	∞	

Shortest Path



visited = [A]

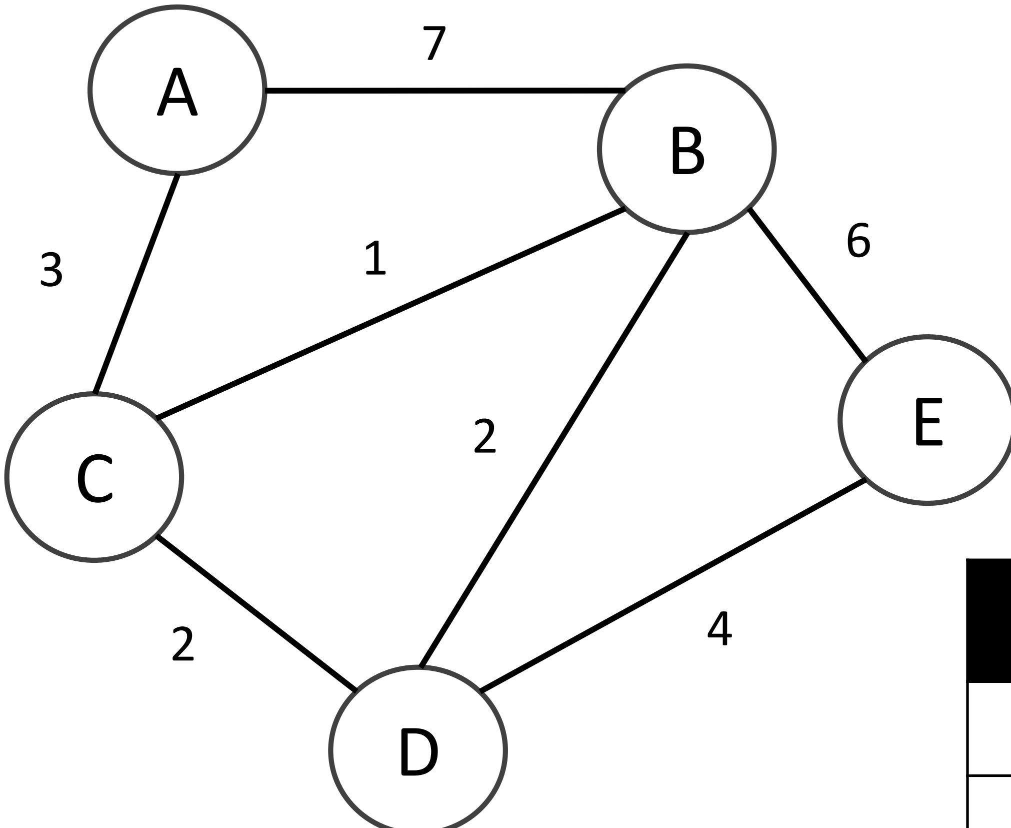
unvisited = [B, C, D, E]

distance to B: 3+1=4 4 < 7?

distance to D: 3+2=5 5 < ∞?

Node	Shortest distance from A	Previous node
A	0	
B	7	A
C	3	A
D	∞	
E	∞	

Shortest Path

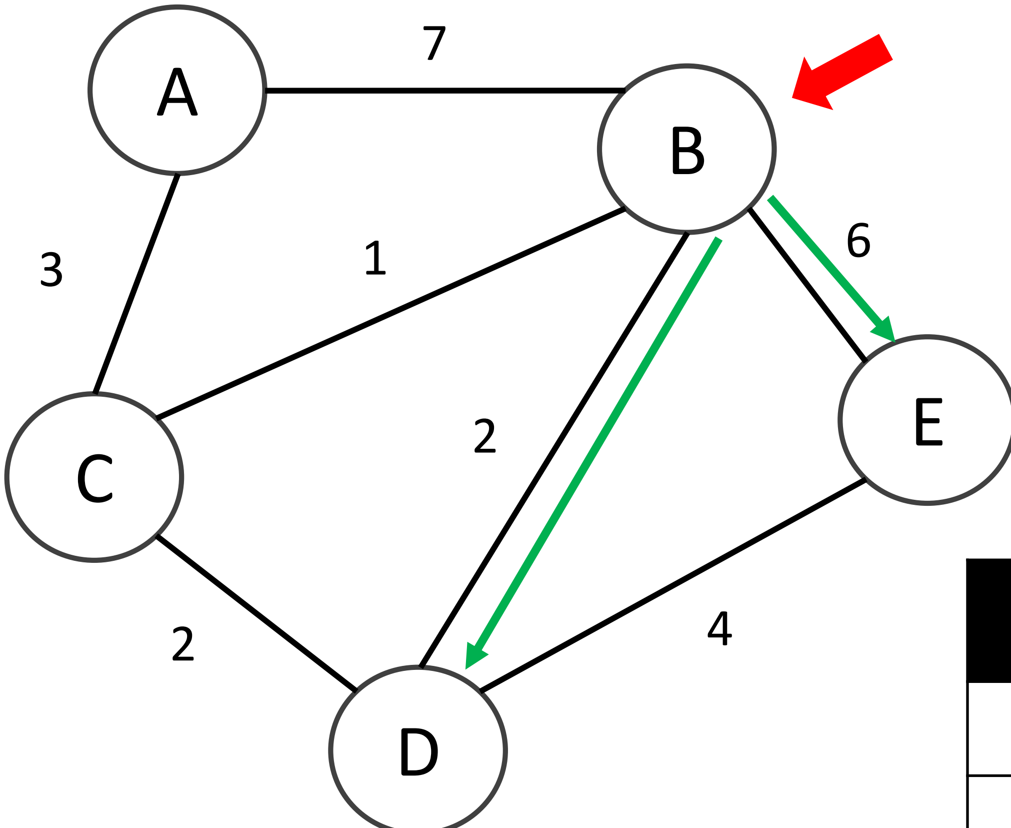


visited = [A, C]

unvisited = [B, D, E]

Node	Shortest distance from A	Previous node
A	0	
B	4	C
C	3	A
D	5	C
E	∞	

Shortest Path



visited = [A, C]

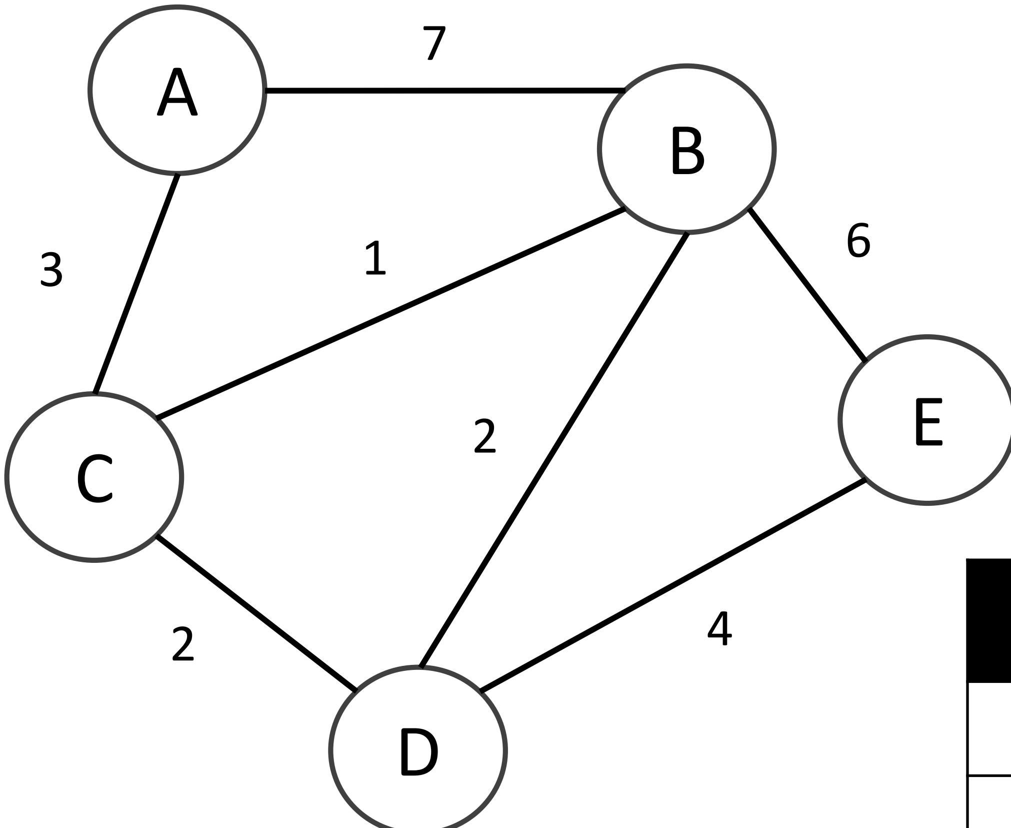
unvisited = [B, D, E]

distance to D: 4+2=6 6 < 5?

distance to E: 4+6=10 10 < ∞?

Node	Shortest distance from A	Previous node
A	0	
B	4	C
C	3	A
D	5	C
E	∞	

Shortest Path

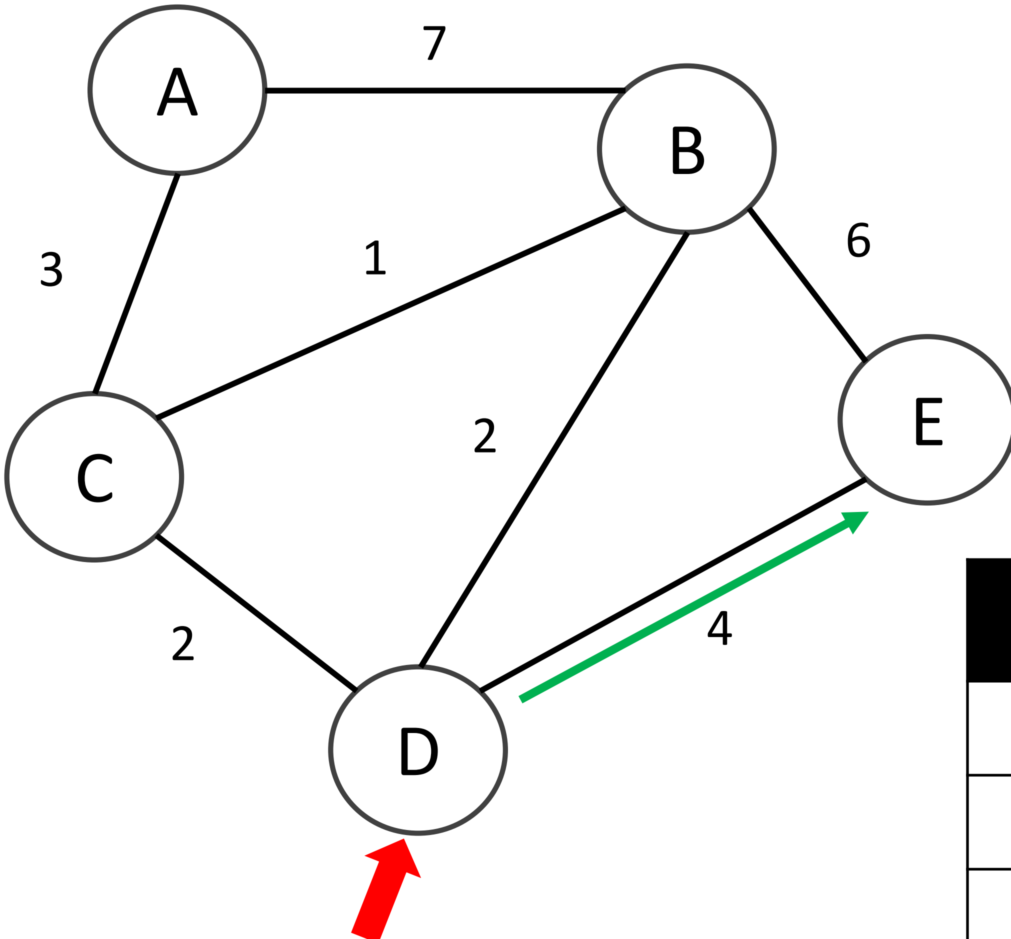


visited = [A, C, B]

unvisited = [D, E]

Node	Shortest distance from A	Previous node
A	0	
B	4	C
C	3	A
D	5	C
E	10	B

Shortest Path



visited = [A, C, B]

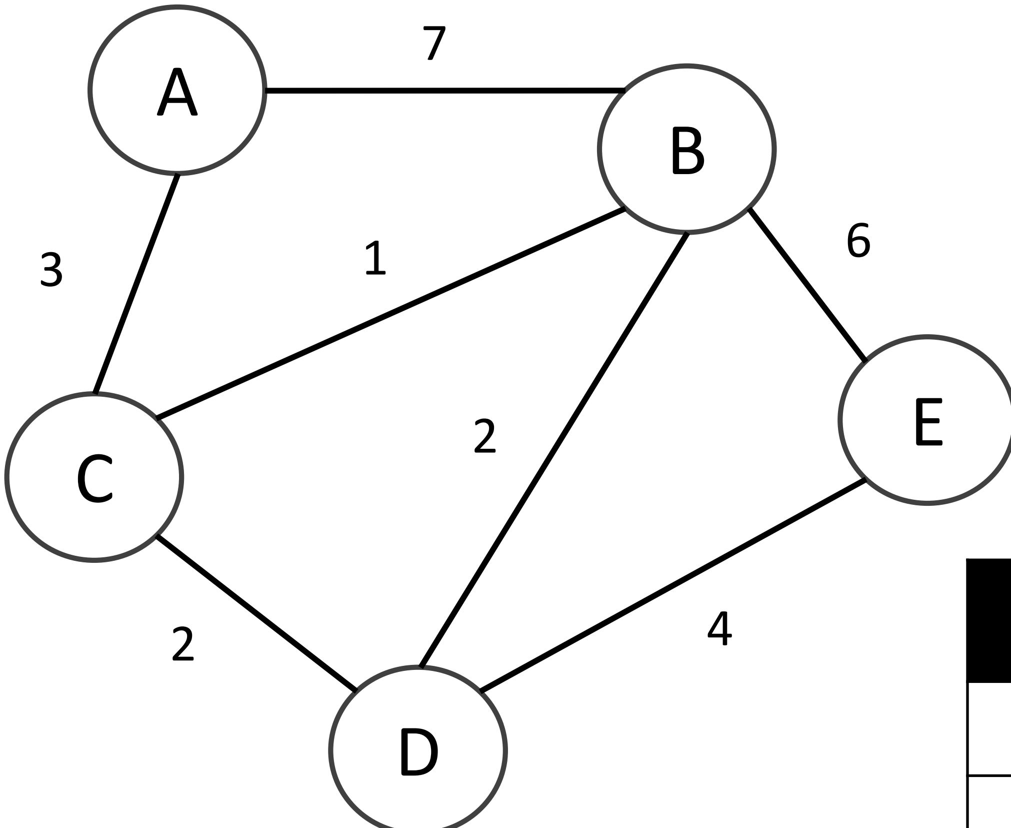
unvisited = [D, E]

Node	Shortest distance from A	Previous node
A	0	
B	4	C
C	3	A
D	5	C
E	10	B

distance to E: 5+4=9

9 < 10?

Shortest Path

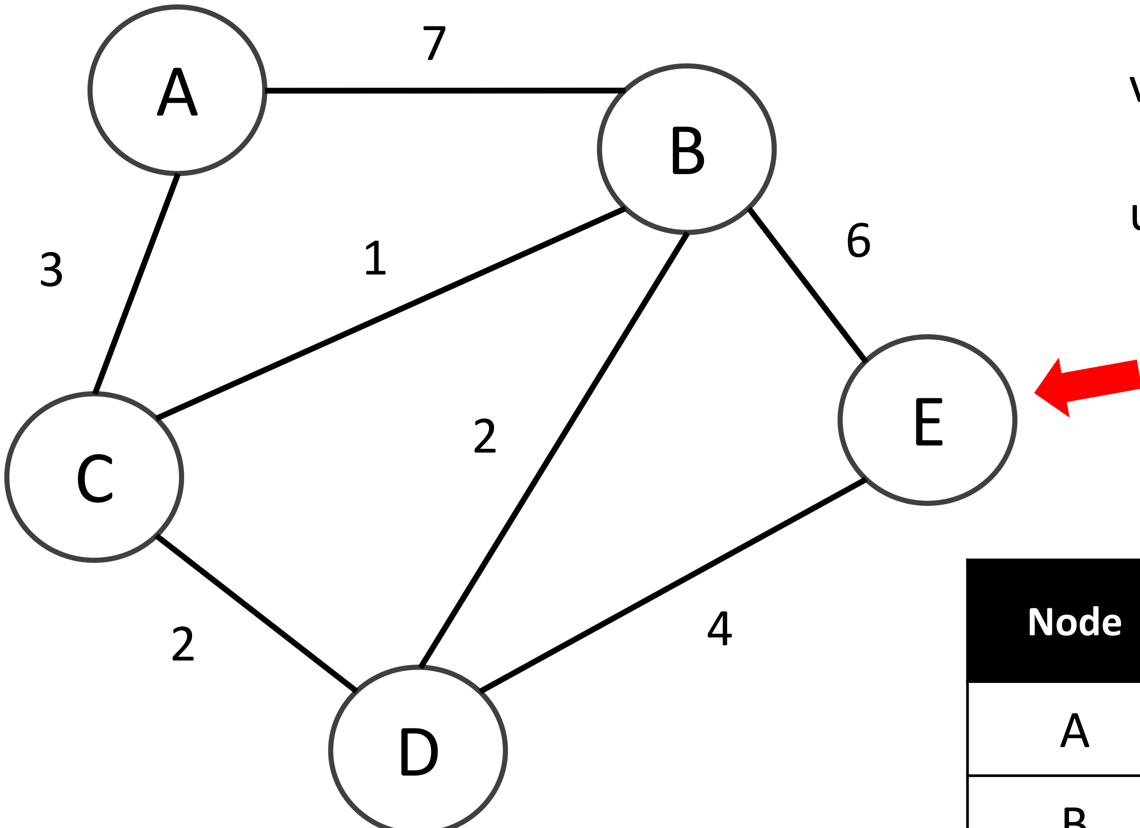


visited = [A, C, B, D]

unvisited = [E]

Node	Shortest distance from A	Previous node
A	0	
B	4	C
C	3	A
D	5	C
E	9	D

Shortest Path

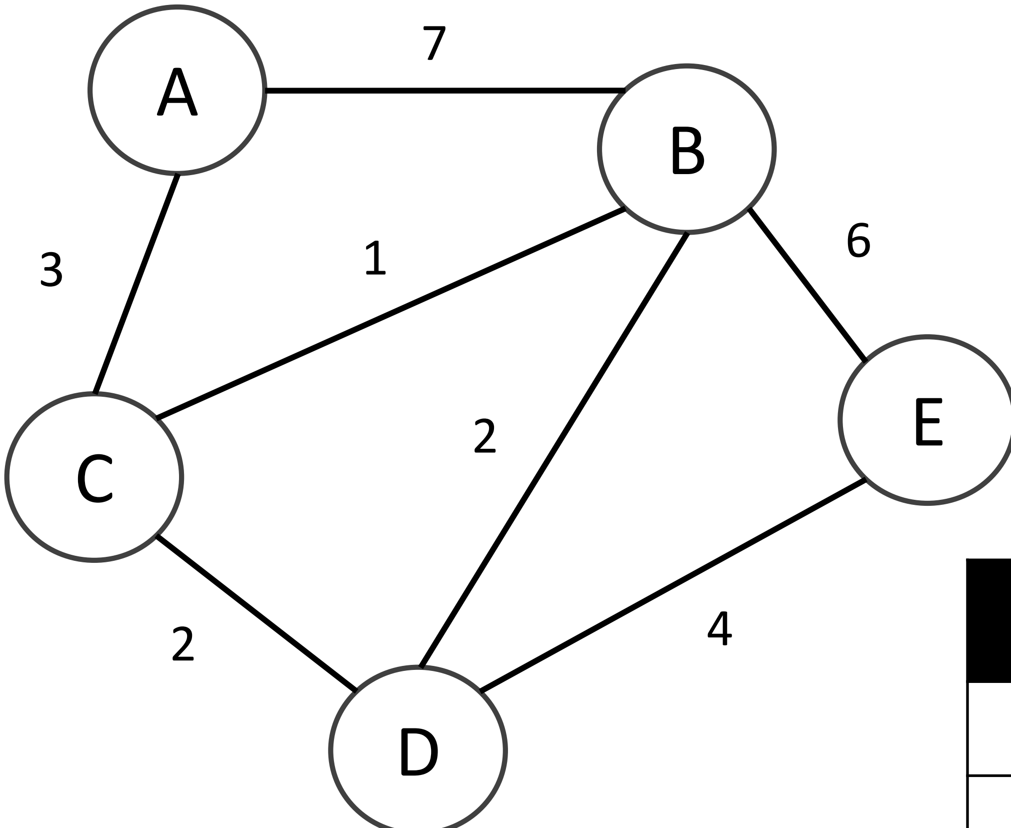


visited = [A, C, B, D]

unvisited = [E]

Node	Shortest distance from A	Previous node
A	0	
B	4	C
C	3	A
D	5	C
E	9	D

Shortest Path



visited = [A, C, B, D, E]

unvisited = []

Node	Shortest distance from A	Previous node
A	0	
B	4	C
C	3	A
D	5	C
E	9	D