Graph

Charles Lin

Agenda

- **Graph Representation**
- **DFS**
- **BFS**
- **Dijkstra**
- **A* Search**
- **Bellman-Ford**
- **Floyd-Warshall**
- **Iterative? Non-iterative?**
- **MST**
- **Flow – Edmond-Karp**

Graph Representation

- **Adjacency Matrix**
  - **`bool way[100][100];`**
  - **`cin >> i >> j;`**
  - **`way[i][j] = true;`**

Graph Representation

- **Adjacency Linked List**
  - **`vector<int> v[100];`**
  - **`cin >> i >> j;`**
  - **`v[i].push_back(j);`**

Graph Representation

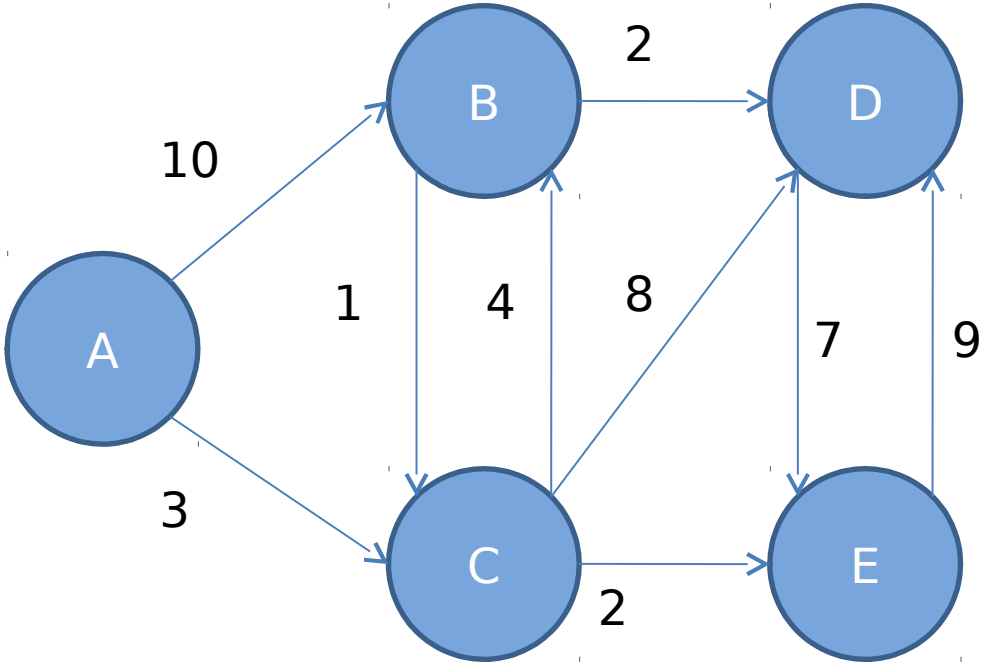
- **Edge List**

```
struct Edge {
  int Start_Vertex, End_Vertex;
} edge[10000];
cin >> edge[i].Start_Vertex
  >> edge[i].End_Vertex;
```

Graph Representation

	Adjacency Matrix	Adjacency Linked List	Edge List
Memory Storage	**$O(V^2)$**	**O(V+E)**	**O(V+E)**
Check whether (u,v) is an edge	**O(1)**	**O(deg(u))** O(log deg(u)) if sorted	**O(E)** O(log E log deg(u)) if sorted
Find all adjacent vertices of a vertex u	**O(V)**	**O(deg(u))**	**O(E)** O(log E deg(u)) if sorted
deg(u): the number of edges connecting vertex *u*			

Graph Theory



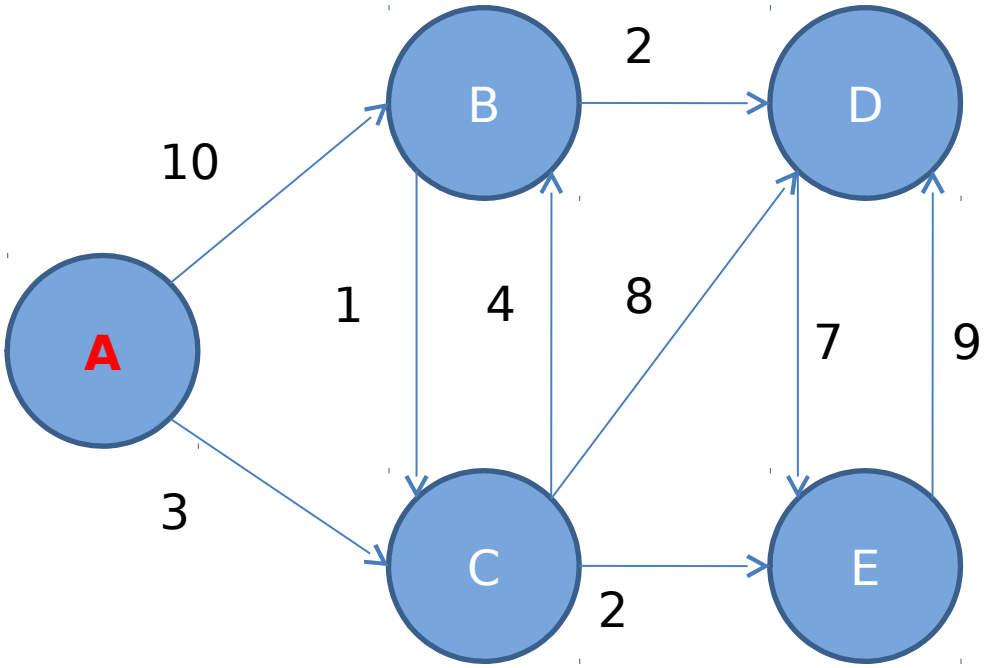
Graph Theory

- **Mission: To go from Point A to Point E**
- **How?**

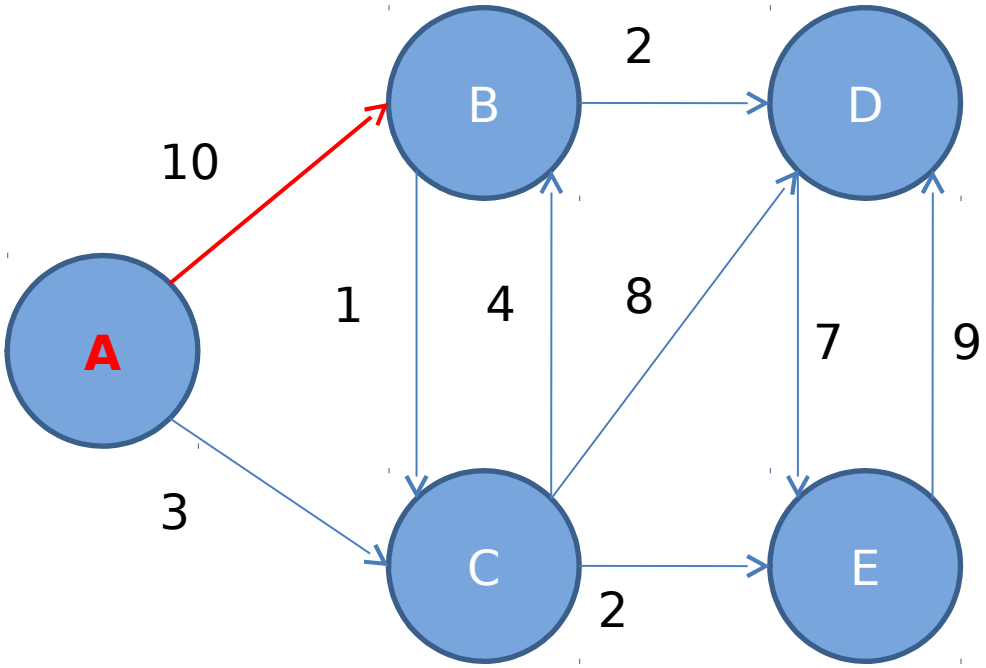
Depth First Search (DFS)

- **Structure to use: Stack**

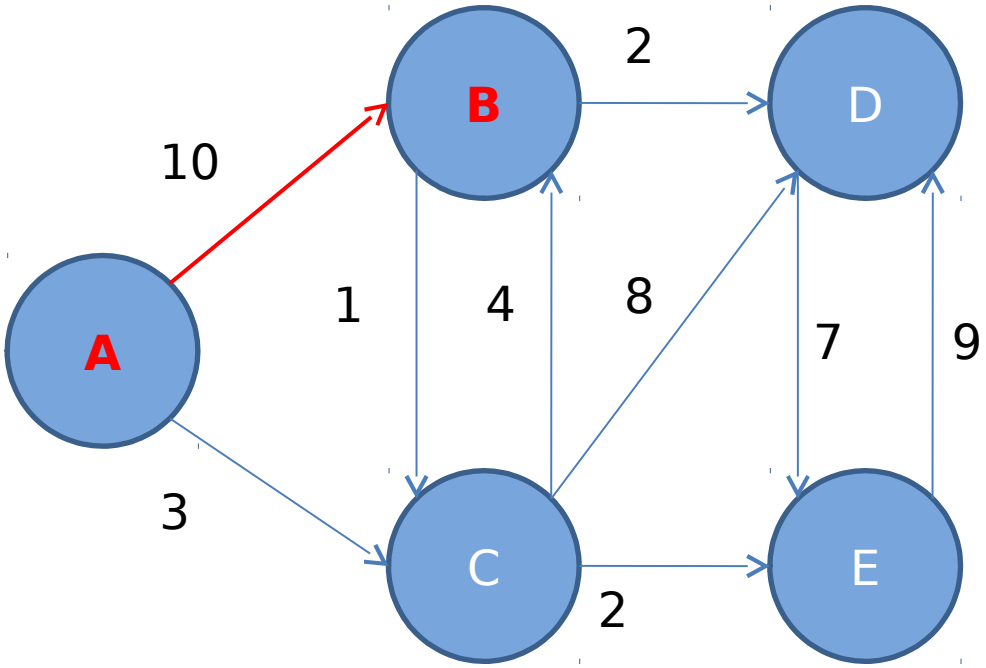
Depth First Search (DFS)



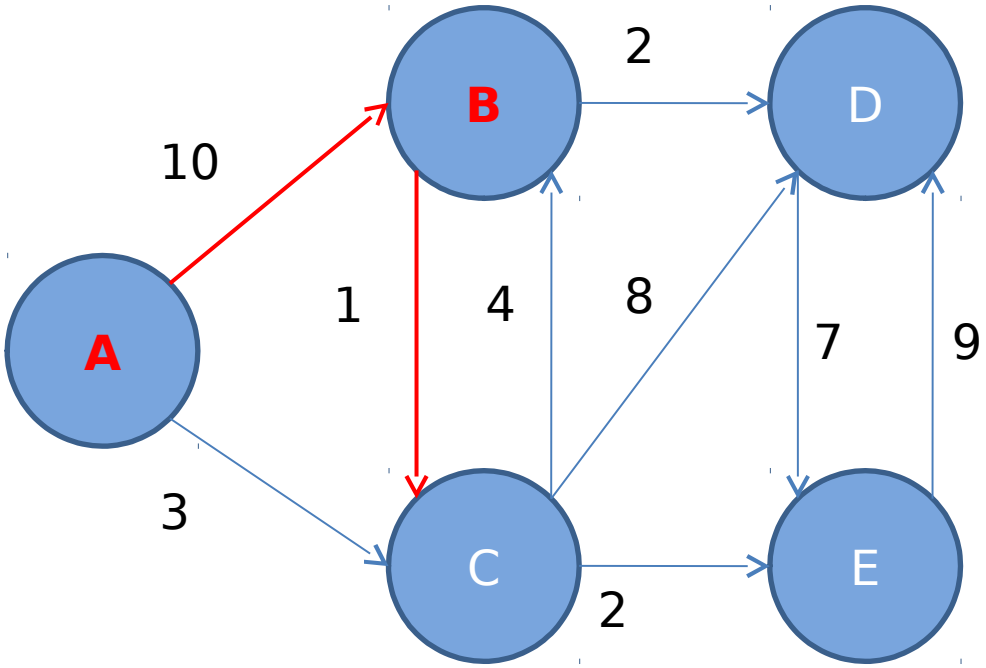
Depth First Search (DFS)



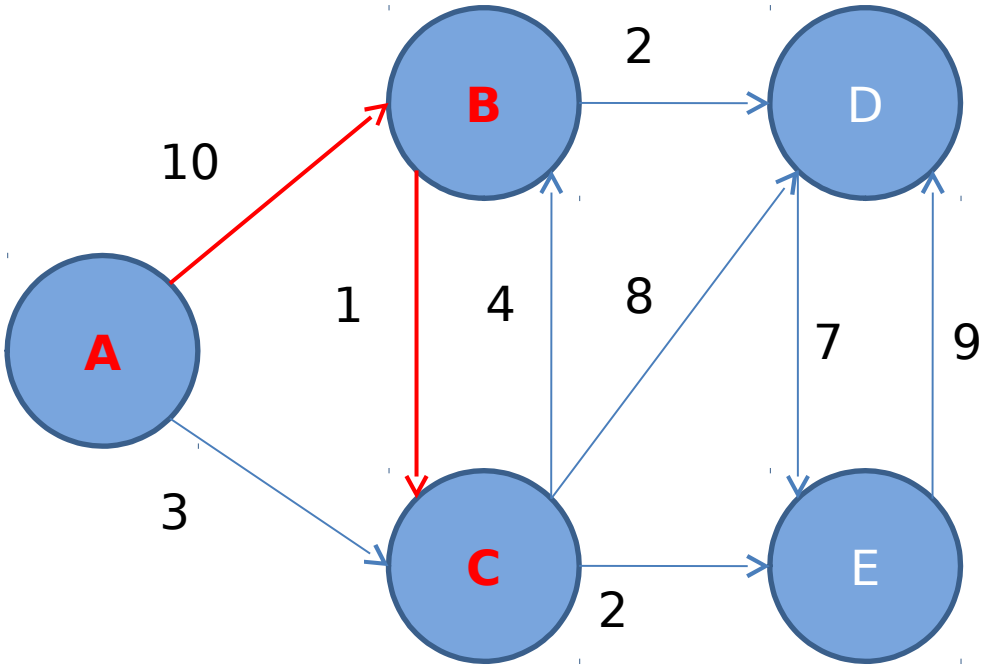
Depth First Search (DFS)



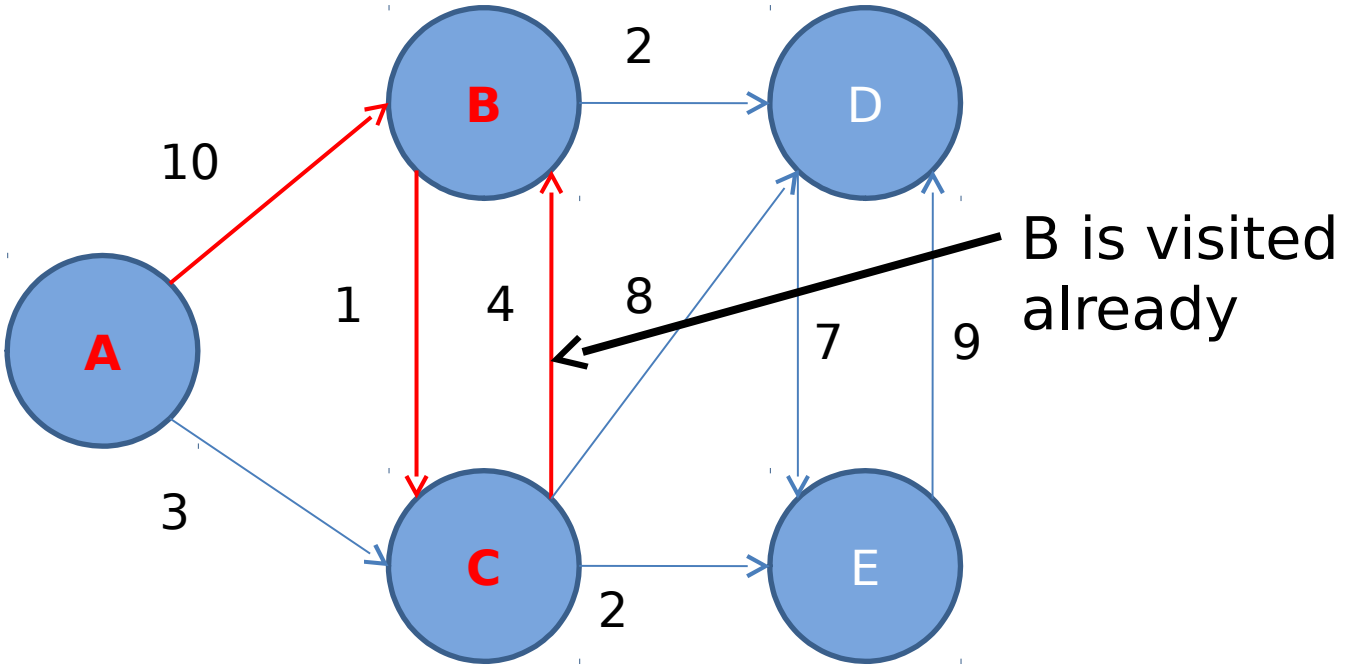
Depth First Search (DFS)



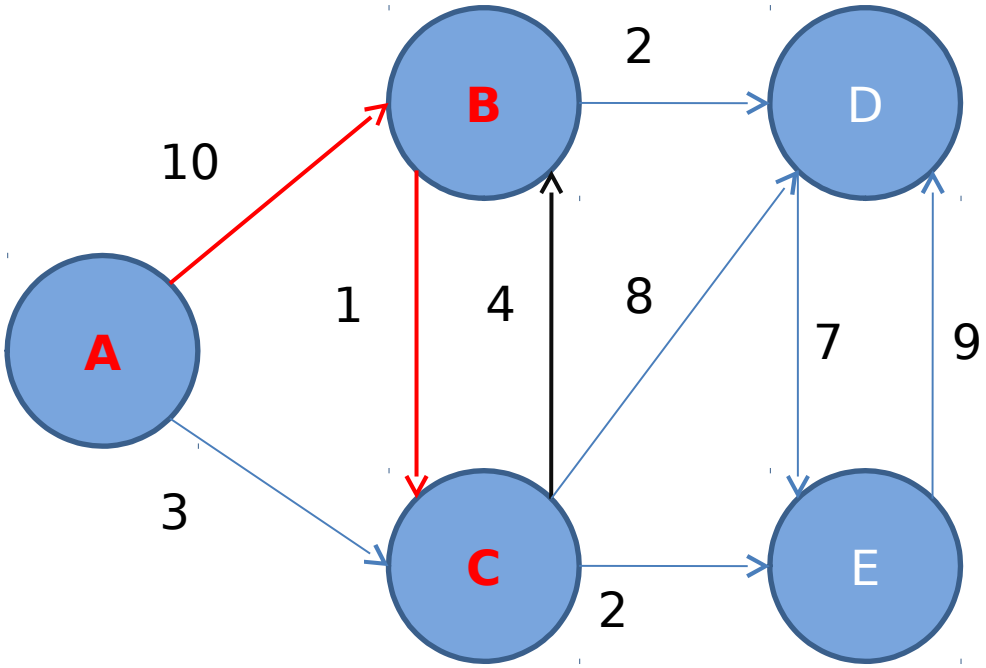
Depth First Search (DFS)



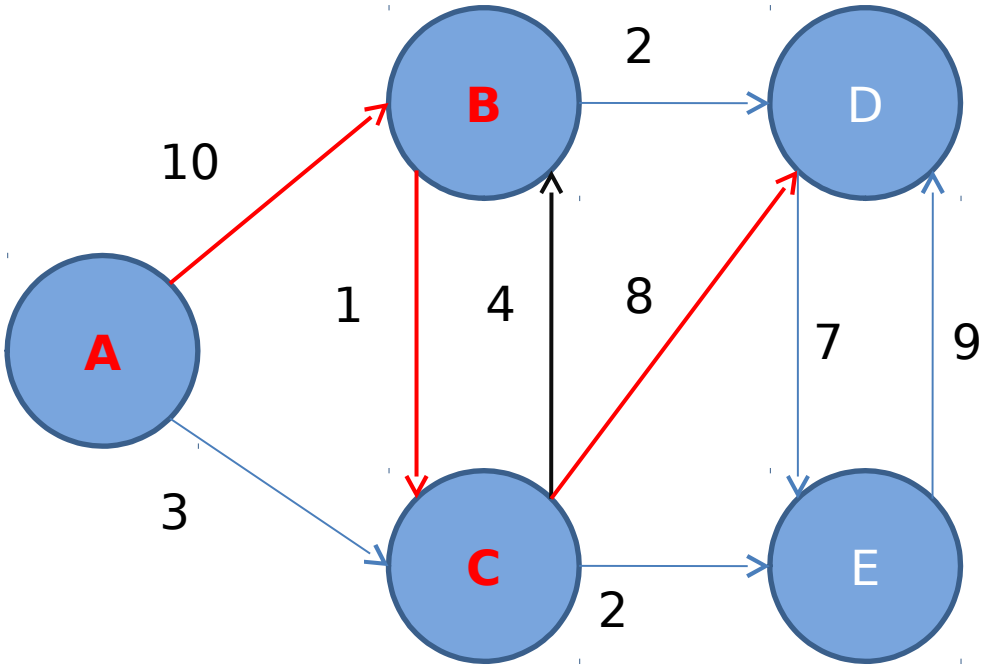
Depth First Search (DFS)



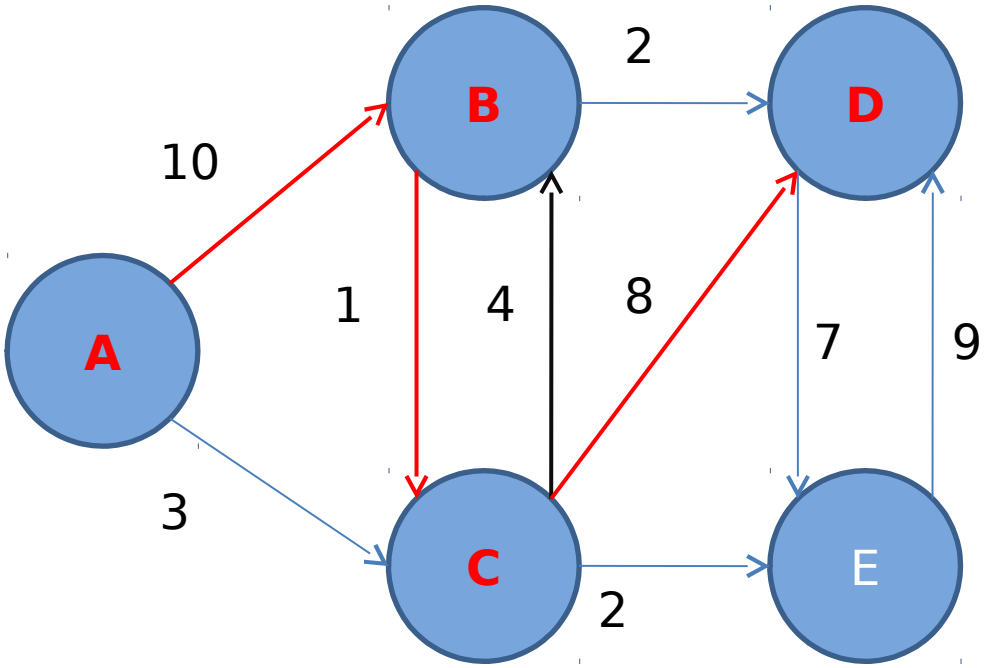
Depth First Search (DFS)



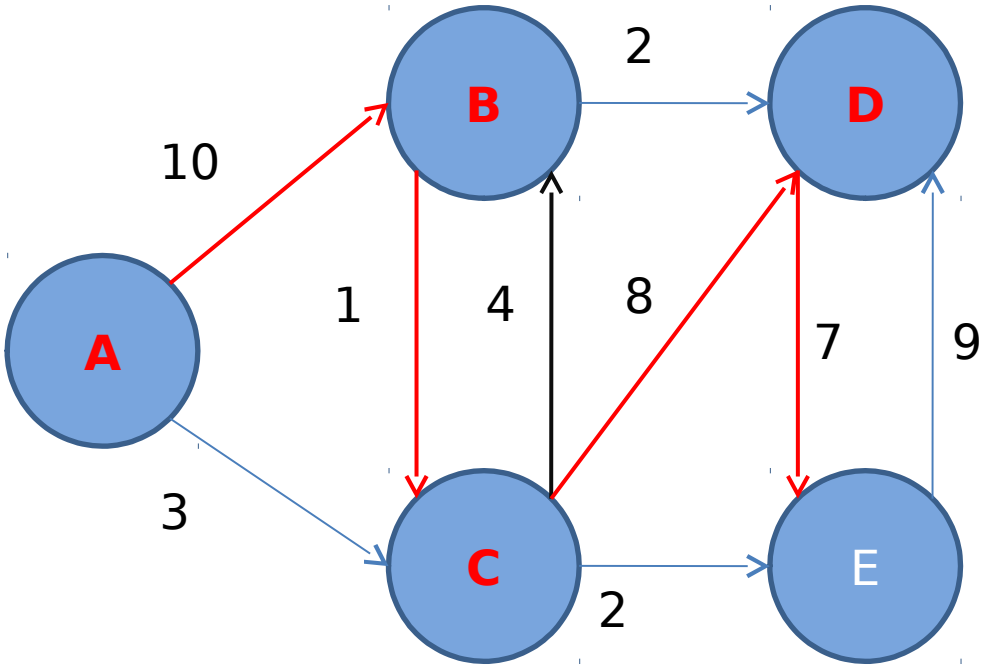
Depth First Search (DFS)



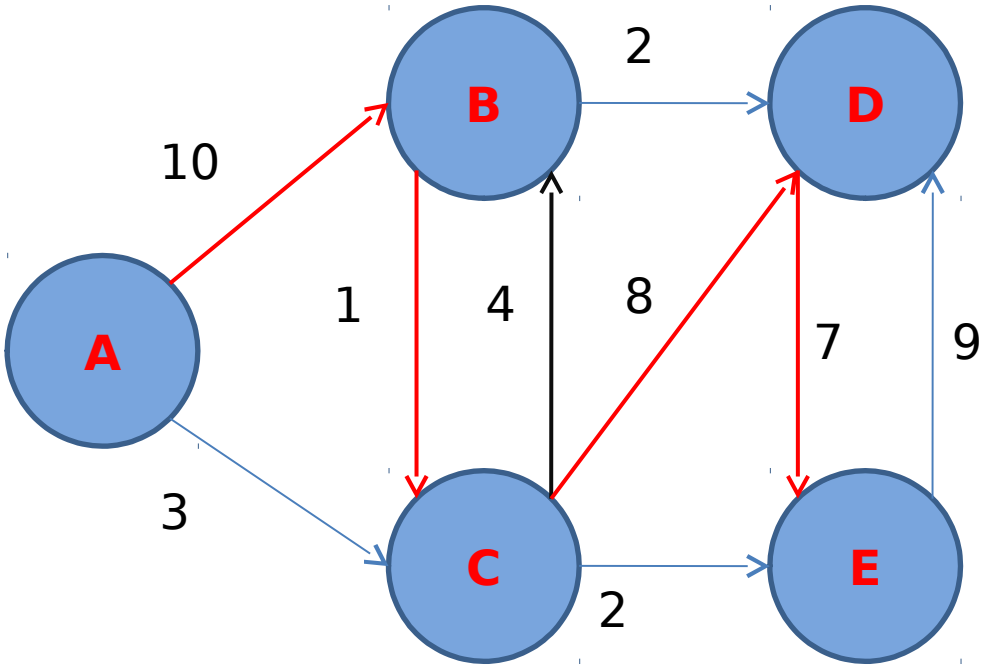
Depth First Search (DFS)



Depth First Search (DFS)



Depth First Search (DFS)



We find a path from Point A to Point E, but …

Depth First Search (DFS)

```
stack<int> s;
bool Visited[MAX];

void DFS(int u) {
    s.push(u);
    if (u == GOAL) {
        for (int x = 0; x < s.size(); x++)
            printf("%d ", s[x]);
        return ;
    }
    for (int x = 0; x < MAX; x++)
        if (!Visited[x]) {
            Visited[x] = true;
            DFS(x);
            Visited[x] = false;
        }
    s.pop();
}
```

```
int main() {
    memset(Visited, 0,
                sizeof(Visited));
    // Input Handling (GOAL = ?)
    DFS(0);
}
```

Very Important
It needs to be restored for other points to traverse, or the path may not be found

Depth First Search (DFS)

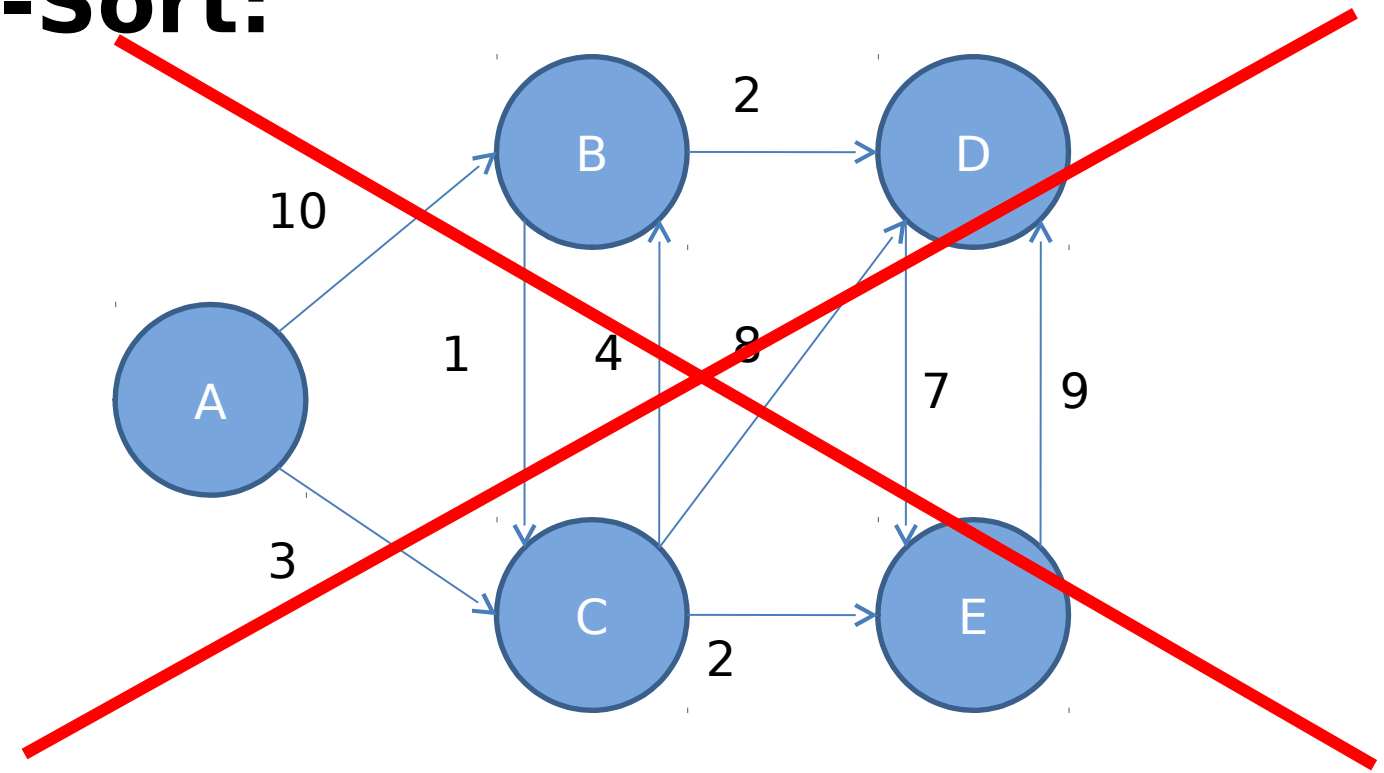
- **Usage:**
  - **Finding a path from start to destination (NOT RECOMMENDED – TLE)**
  - **Topological Sort (T-Sort)**
  - **Strongly-Connected Components (SCC)**
  - **Detect cycles**

Depth First Search (DFS)

- **Topological Sort**
  - **Directed Acyclic Graph (DAG)**
  - **Find the order of nodes such that for each node, every parent is before that node on the list**
  - **Dumb method: Check for root of residual graph (Do it n times)**
  - **Better Method: Reverse of finishing time**
  - **Start from all vertices!**

Depth First Search (DFS)

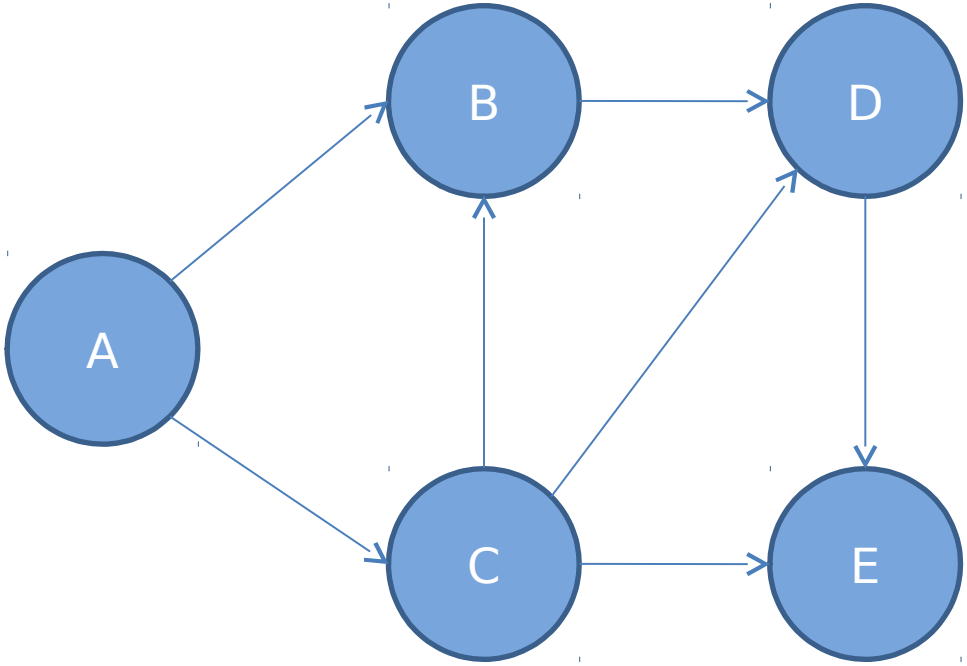
T-Sort:



Cycle Exists !!!

Depth First Search (DFS)

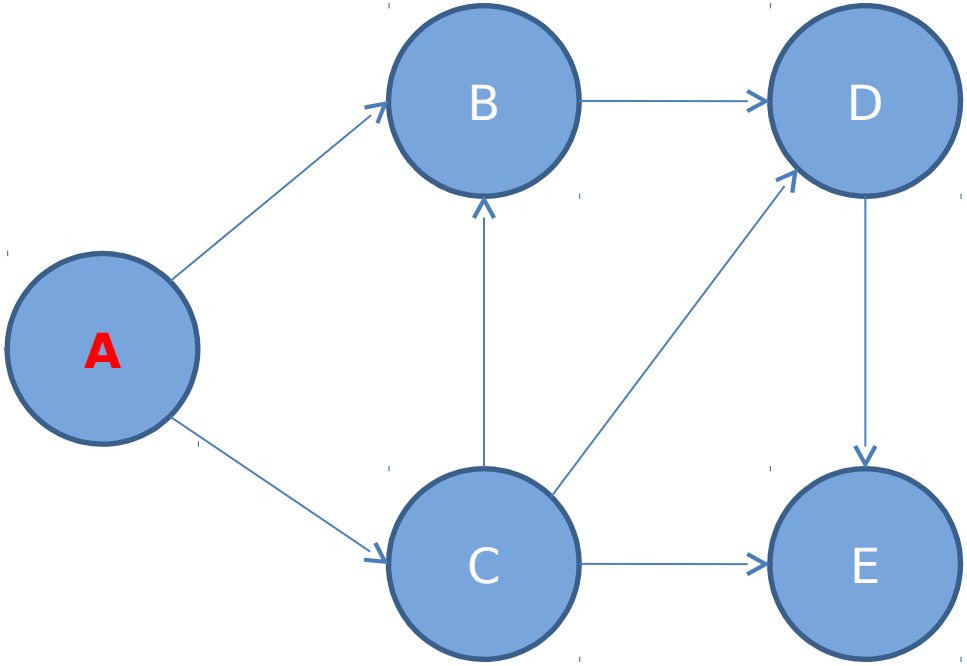
T-Sort:



OK

Depth First Search (DFS)

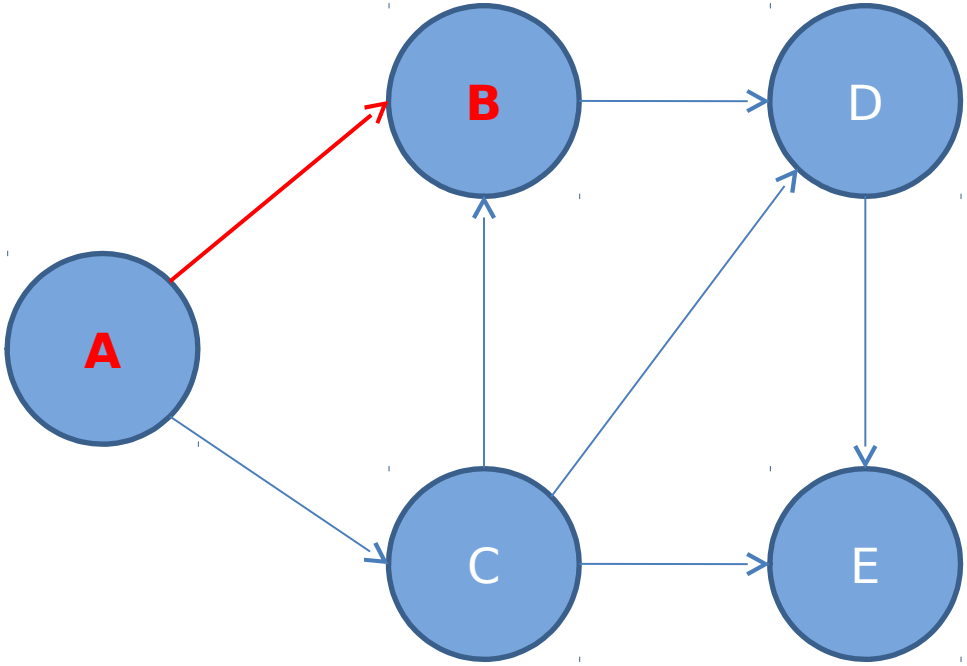
T-Sort:



Things in output stack:
NONE

Depth First Search (DFS)

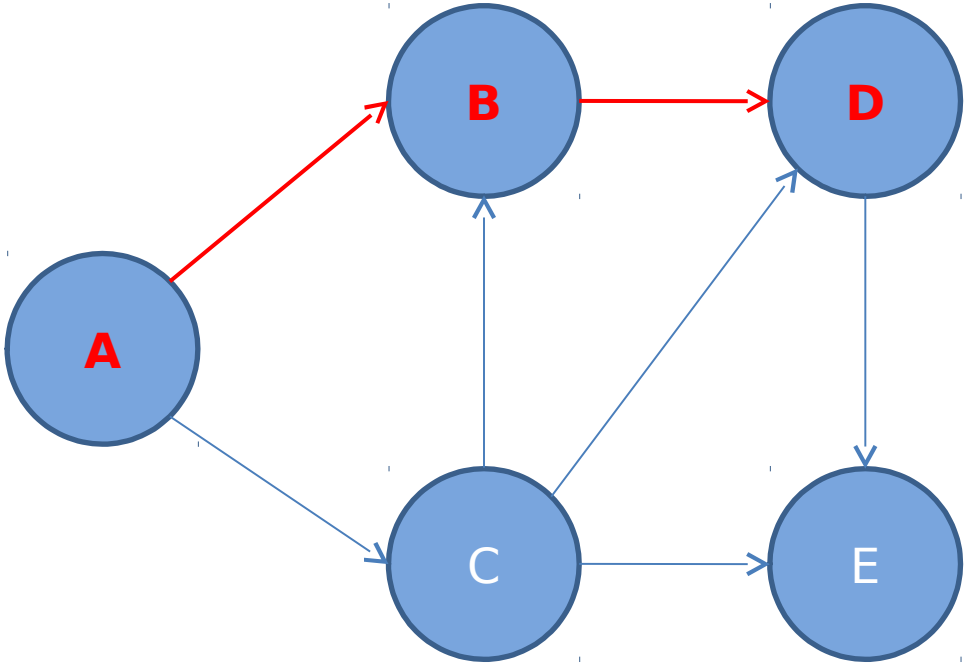
T-Sort:



Things in output stack:
NONE

Depth First Search (DFS)

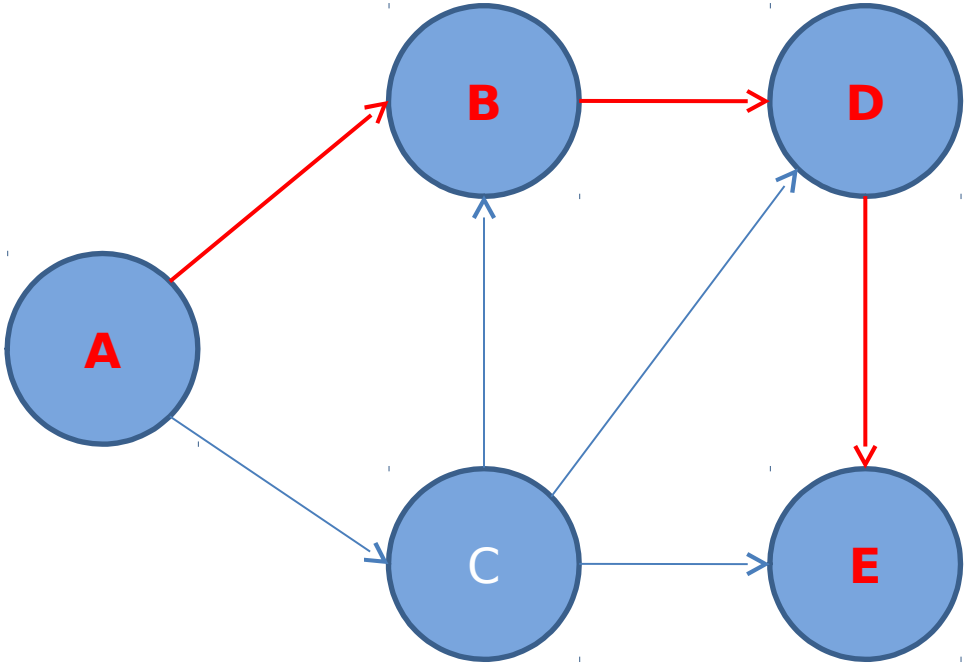
T-Sort:



Things in output stack:
NONE

Depth First Search (DFS)

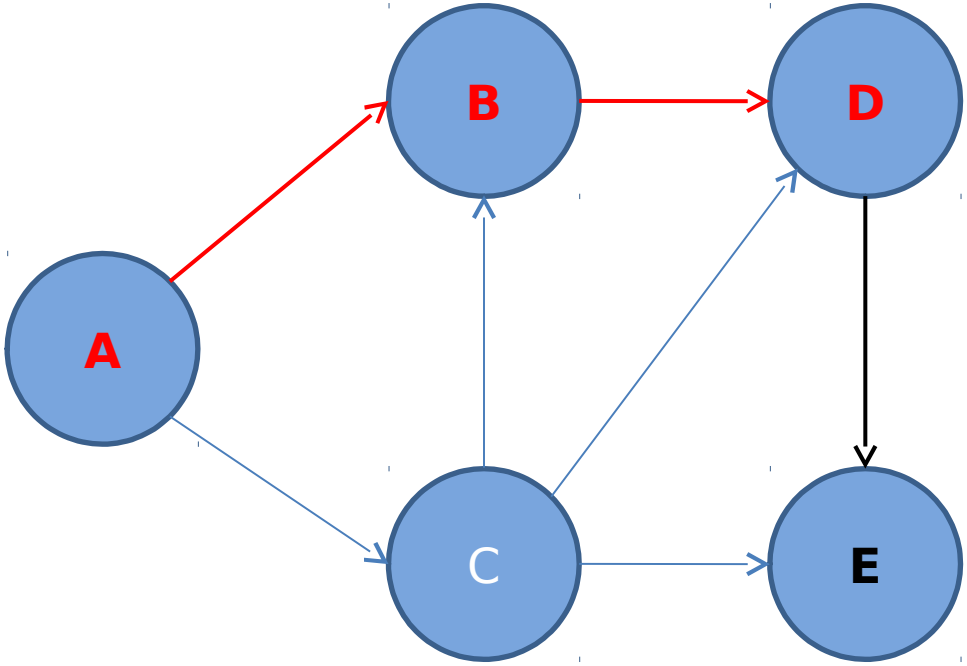
T-Sort:



Things in output stack:
NONE

Depth First Search (DFS)

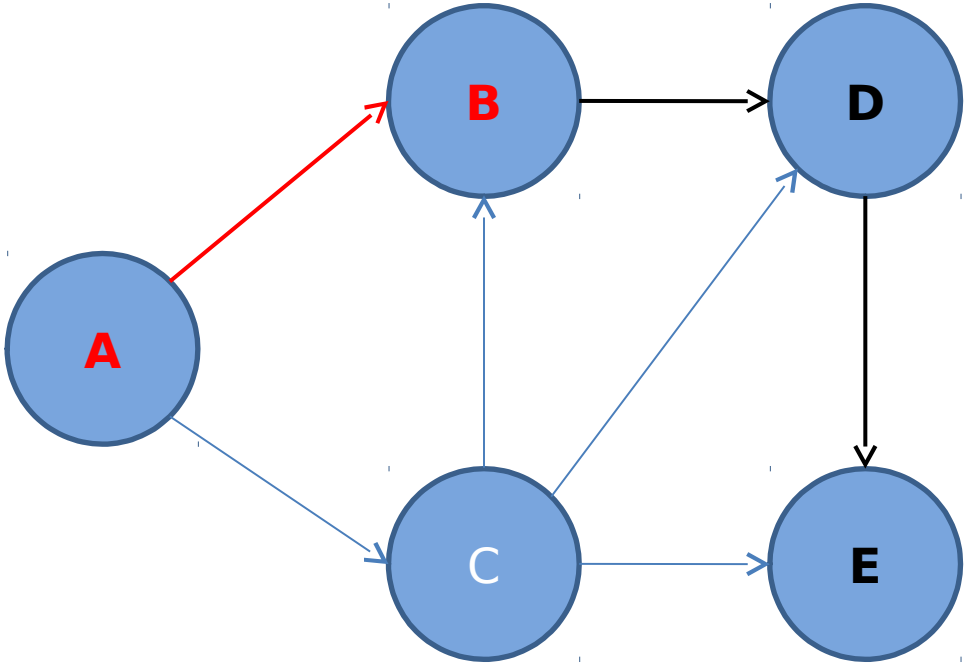
T-Sort:



Things in output stack: E

Depth First Search (DFS)

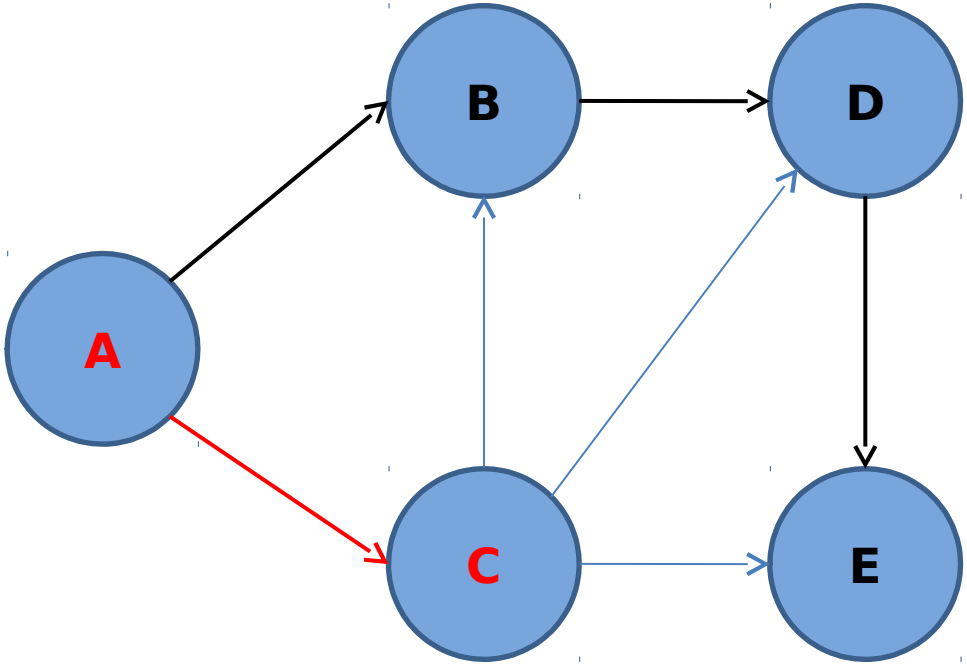
T-Sort:



Things in output stack: E, D

Depth First Search (DFS)

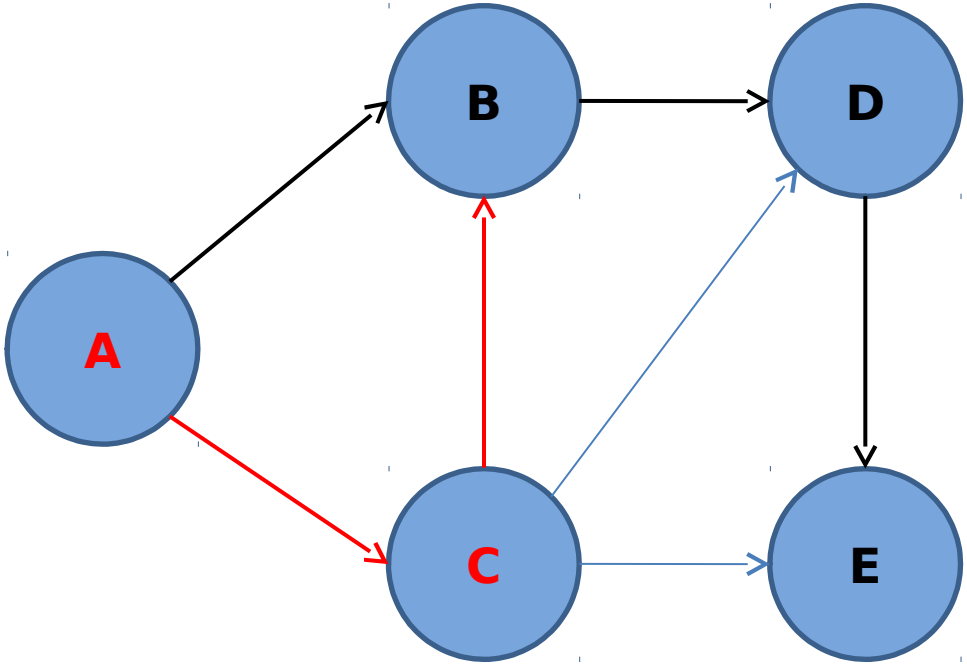
T-Sort:



Things in output stack: E, D, B

Depth First Search (DFS)

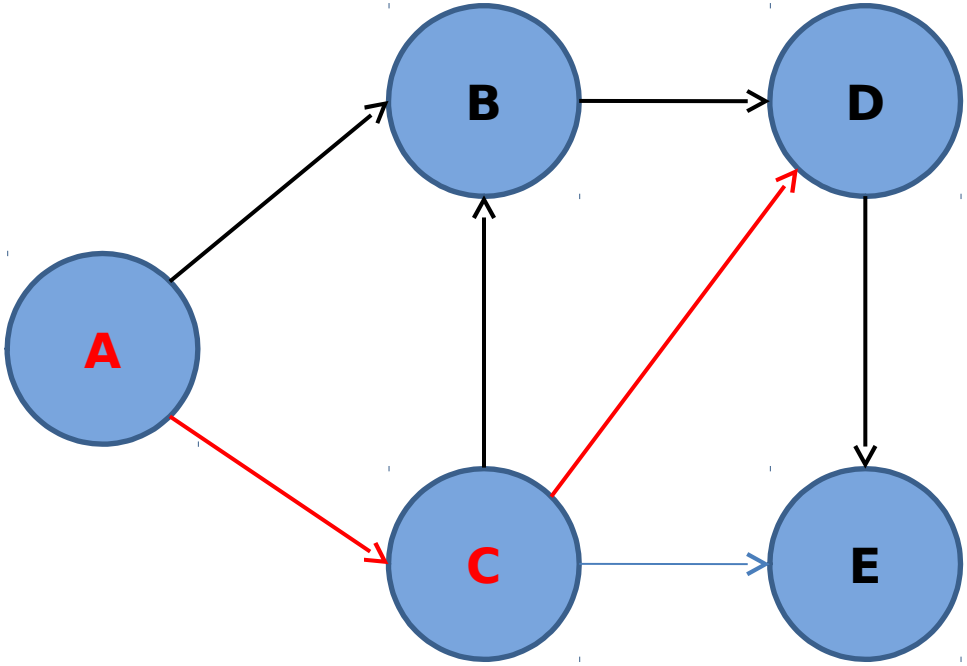
T-Sort:



Things in output stack: E, D, B

Depth First Search (DFS)

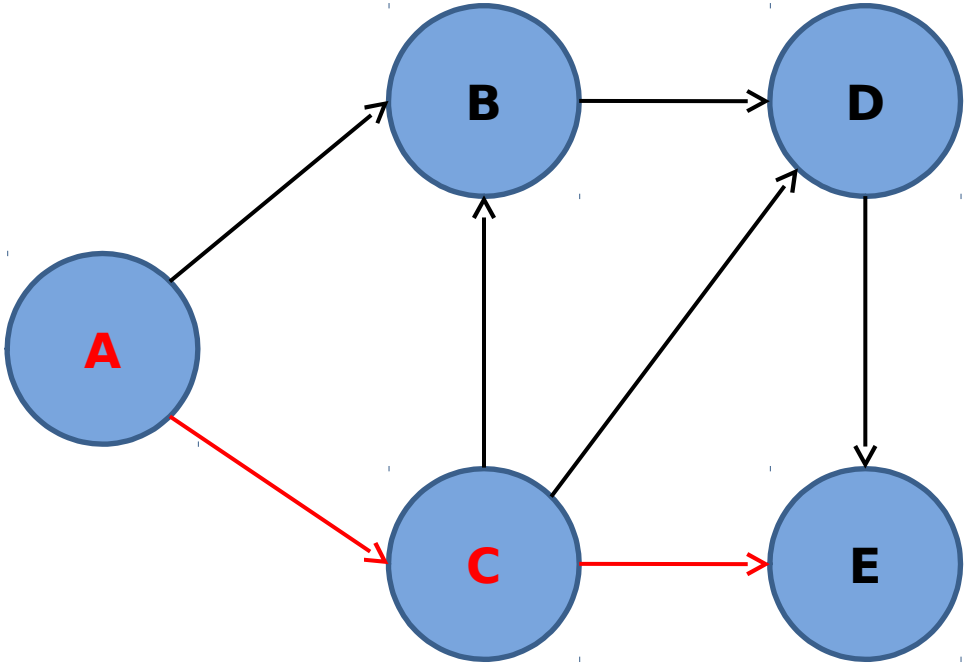
T-Sort:



Things in output stack: E, D, B

Depth First Search (DFS)

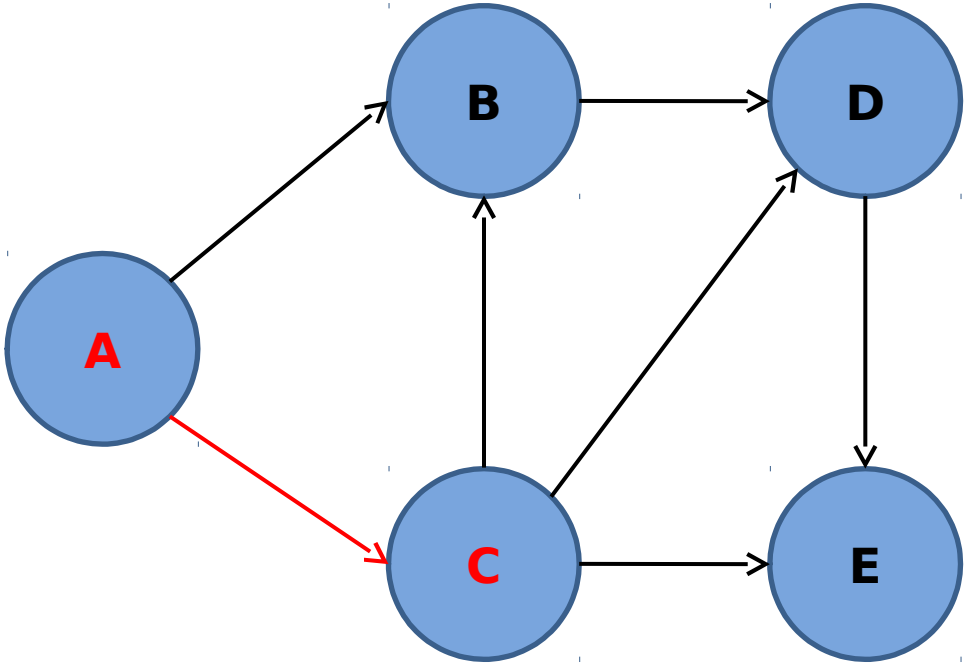
T-Sort:



Things in output stack: E, D, B

Depth First Search (DFS)

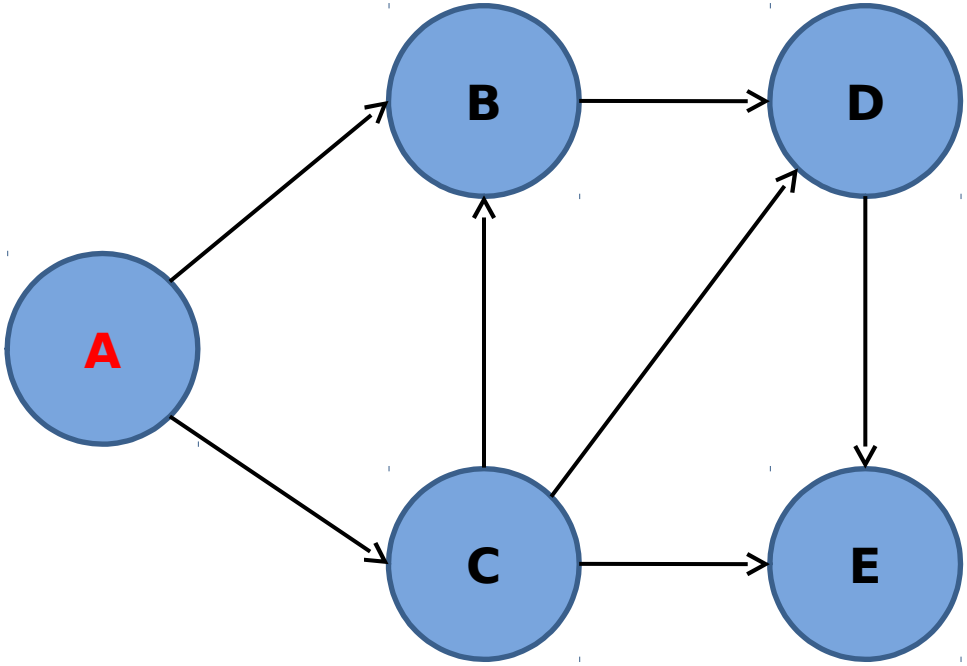
T-Sort:



Things in output stack: E, D, B

Depth First Search (DFS)

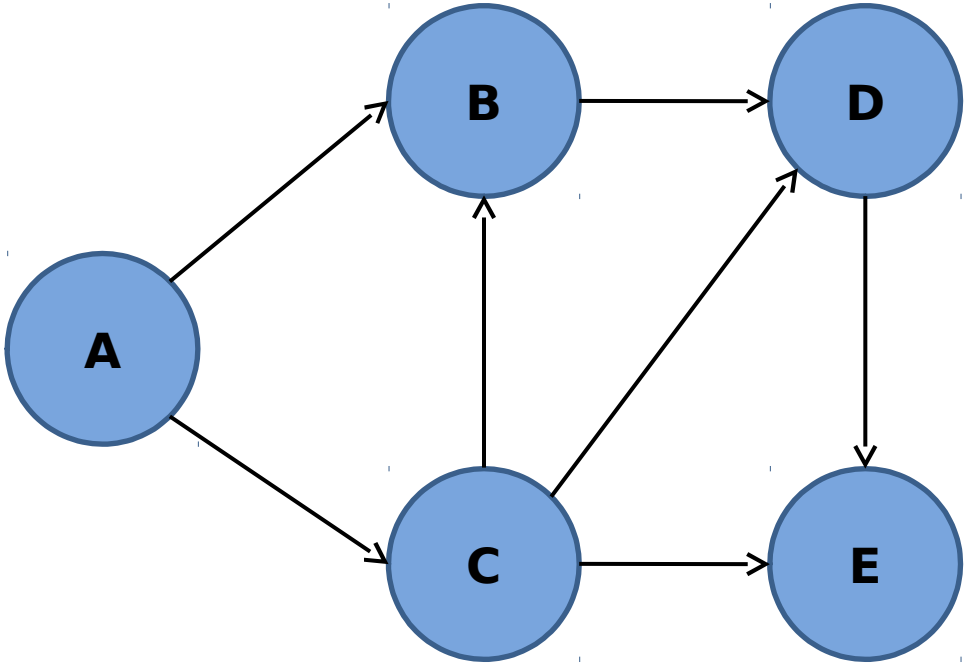
T-Sort:



Things in output stack: E, D, B, C

Depth First Search (DFS)

T-Sort:



Things in output stack: E, D, B, C, A
T-Sort Output: A, C, B, D, E

Depth First Search (DFS)

- **Strongly Connected Components**
  - **Directed Graph**
  - **Find groups of nodes such that in each group, every node has a path to every other node**

Depth First Search (DFS)

- **Strongly Connected Components**
  - **Method: DFS twice (Kosaraju)**
  - **Do DFS on graph G from all vertices, record finishing time of node**
  - **Reverse direction of edges**
  - **Do DFS on Graph G from all vertices sorted by decreasing finishing time**
  - **Each DFS tree is an SCC**

Depth First Search (DFS)

- **Strongly Connected Components**
- **Note: After grouping the vertices, the new nodes form a Directed Acyclic Graph**

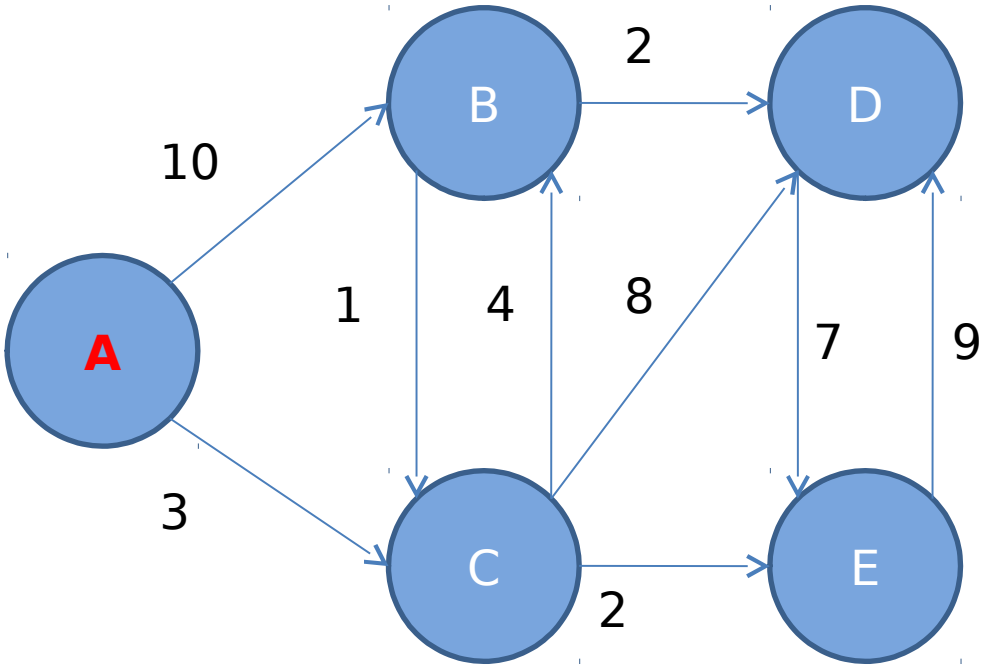
Breadth First Search (BFS)

- **Structure to use: Queue**

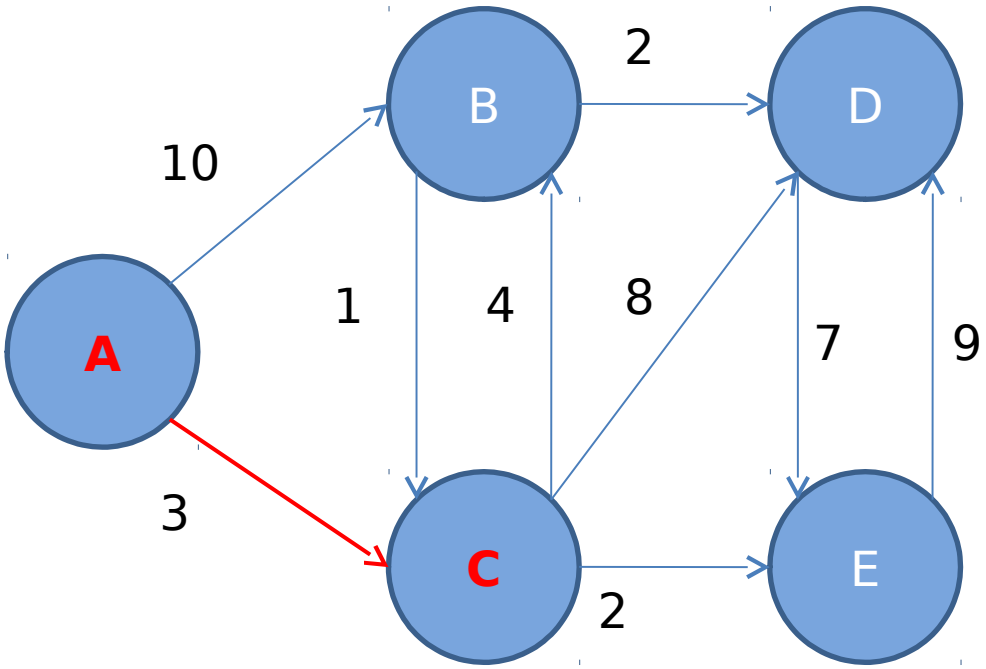
Breadth First Search (BFS)

Things in queue:

A, C	3
A, B	10



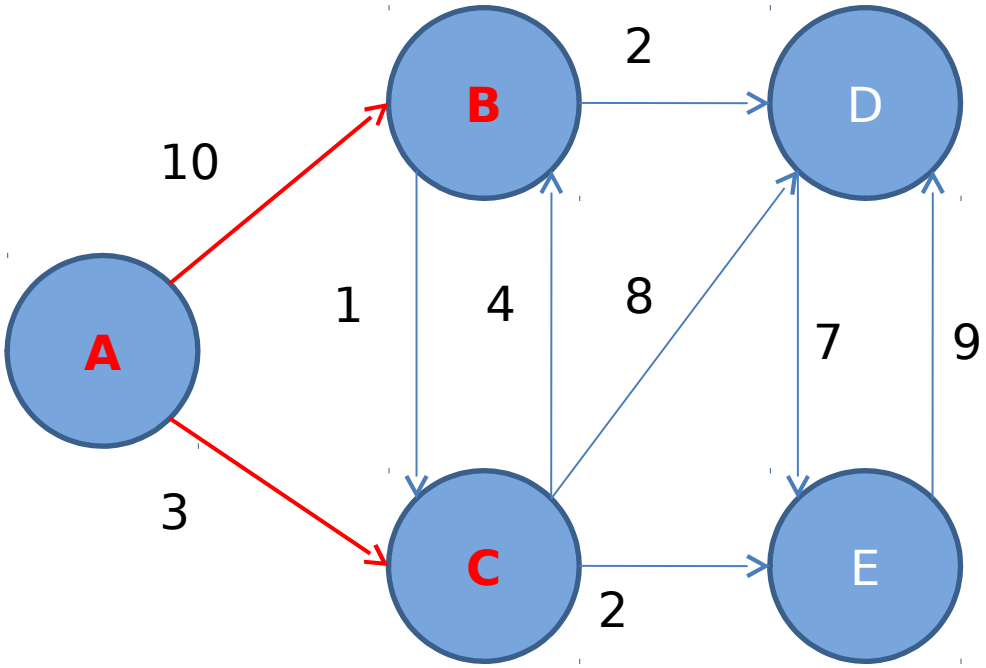
Breadth First Search (BFS)



Things in queue:

A, B	10
A, C, E	5
A, C, B	7
A, C, D	8

Breadth First Search (BFS)



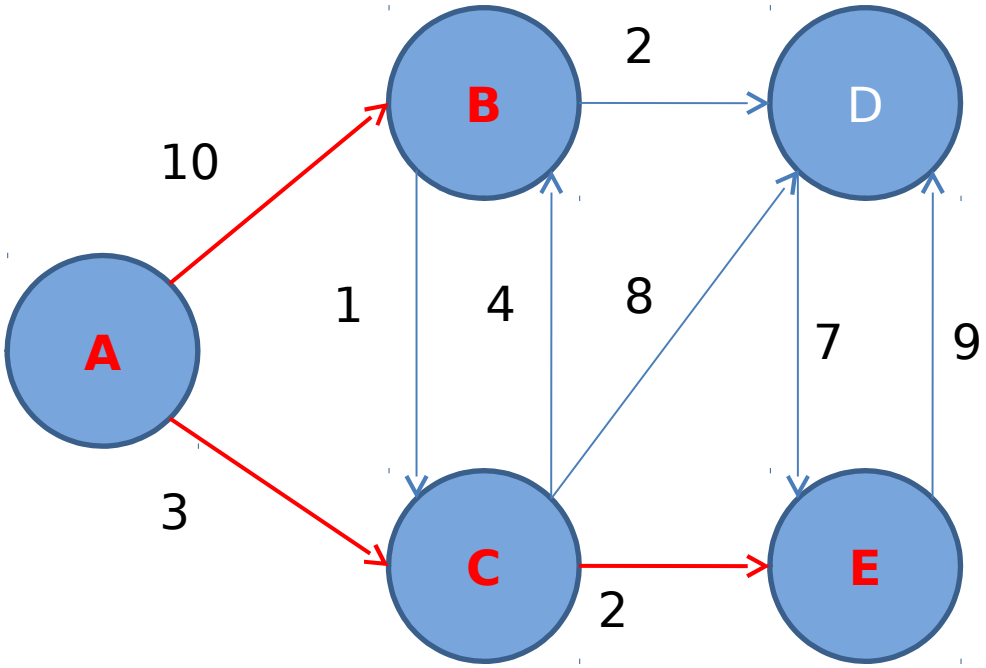
Things in queue:

A, C, E 5
A, C, B 7
A, C, D 8
A, B, C 11
A, B, D 12

Breadth First Search (BFS)

Things in queue:

A, C, B 7
A, C, D 8
A, B, C 11
A, B, D 12



We are done !!!

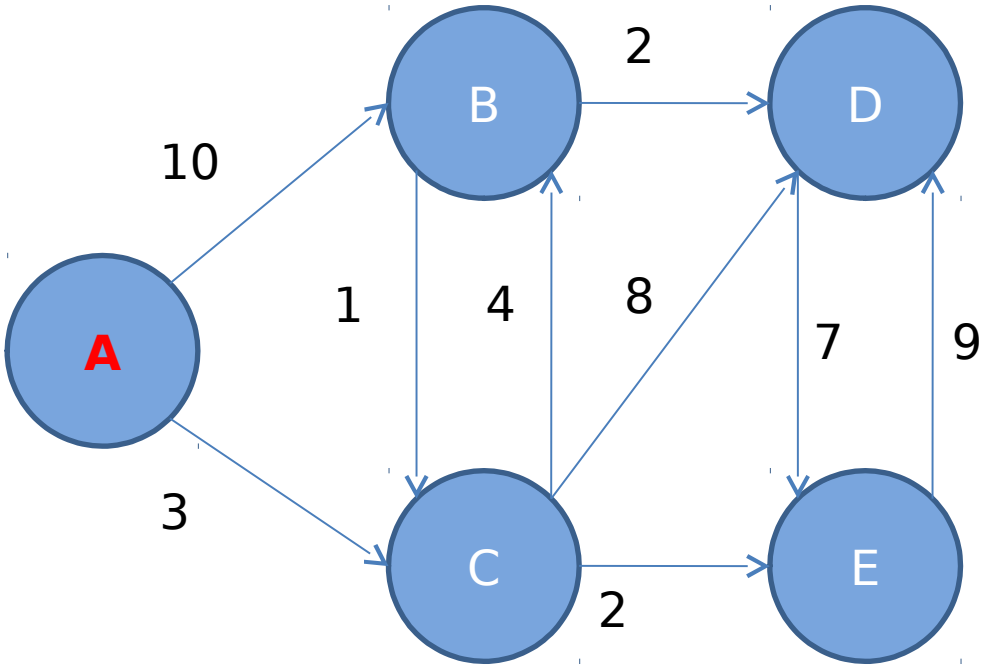
Breadth First Search (BFS)

- **Application:**
  - **Finding shortest path**
  - **Flood-Fill**

Dijkstra

- **Structure to use: Priority Queue**
- **Consider the past**
  - **The length of path visited so far**
- **No negative edges allowed**
- **Maintain known vertices**
- **At each step:**
  - **Take the unknown vertex with smallest overall estimate**

Dijkstra



Things in priority queue:

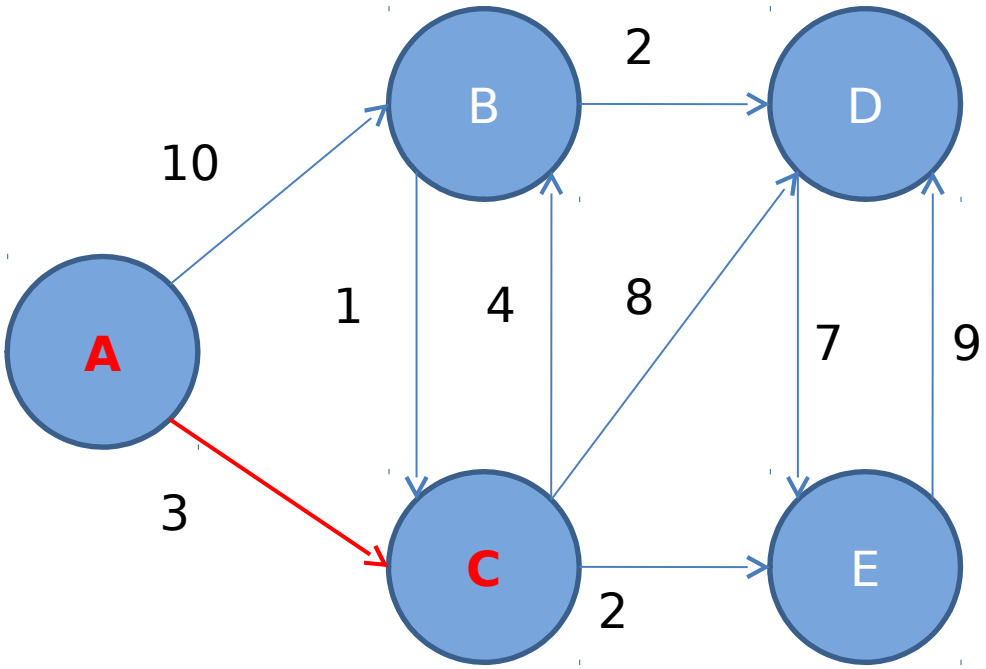
A, C 3

A, B 10

Current Route:

A

Dijkstra



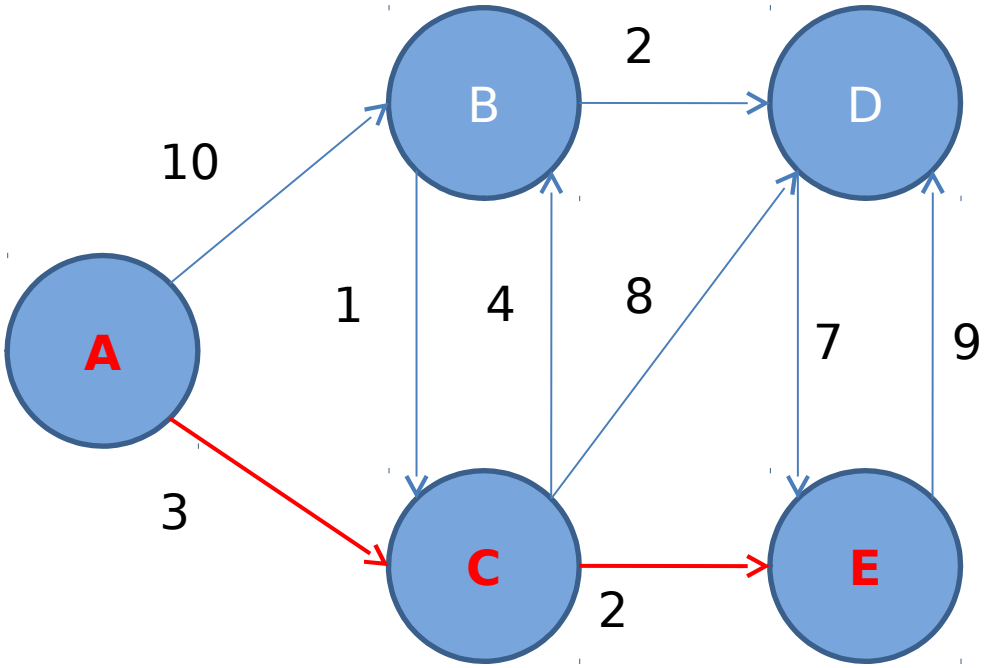
Things in priority queue:

A, C, E 5
A, C, B 7
A, B 10
A, C, D 11

Current Route:

A, C

Dijkstra



Things in priority queue:

A, C, B 7
A, B 10
A, C, D 11

Current Route:

A, C, E

We find the shortest path already !!!

A* Search

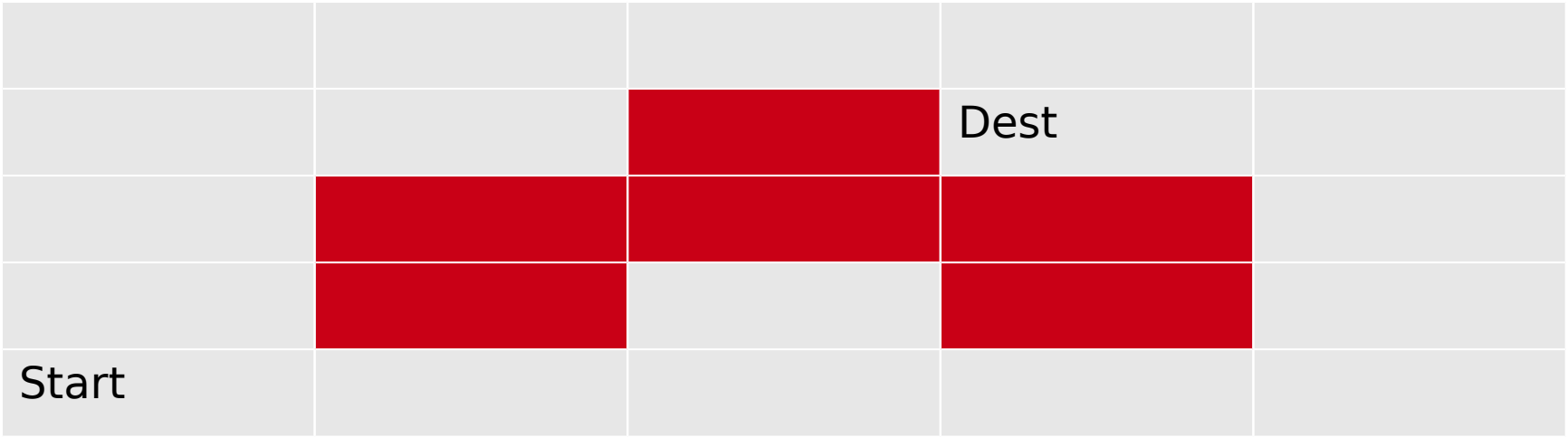
- **Structure to use: Priority Queue**
- **Consider the past + future**
- **How to consider the future?**
  - **Admissible Heuristic**

    (Best-Case Prediction: must never overestimate)
  - **How to predict?**

A* Search

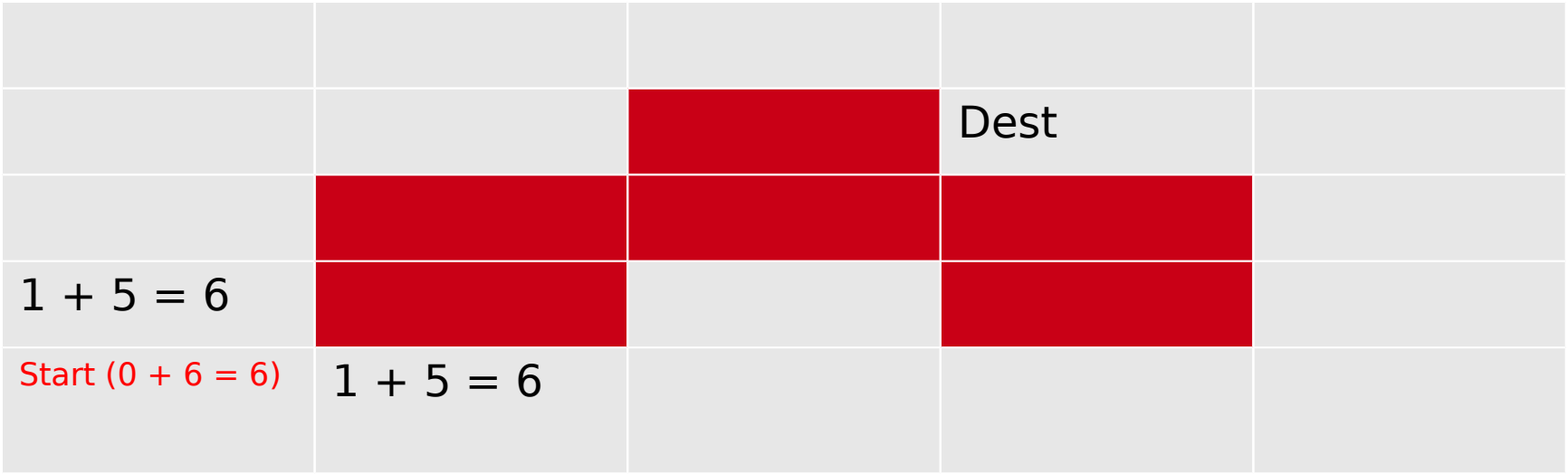
- **Prediction 1: Displacement**
  - **Given coordinates, calculate the displacement of current node to destination**
  - **$\text{sqrt}((x2 - x1)^2 + (y2 - y1)^2)$**
- **Prediction 2: Manhattan Distance**
  - **Given grid, calculate the horizontal and vertical movement**
  - **(x2 – x1) + (y2 – y1)**

A* Search



- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search



- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

			Dest	
2 + 4 = 6				
1 + 5 = 6				
Start (0 + 6 = 6)	1 + 5 = 6			

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

3 + 3 = 6			Dest	
2 + 4 = 6				
1 + 5 = 6				
Start (0 + 6 = 6)	1 + 5 = 6			

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8				
3 + 3 = 6	4 + 2 = 6		Dest	
2 + 4 = 6				
1 + 5 = 6				
Start (0 + 6 = 6)	1 + 5 = 6			

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8	5 + 3 = 8			
3 + 3 = 6	4 + 2 = 6		Dest	
2 + 4 = 6				
1 + 5 = 6				
Start (0 + 6 = 6)	1 + 5 = 6			

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8	5 + 3 = 8			
3 + 3 = 6	4 + 2 = 6		Dest	
2 + 4 = 6				
1 + 5 = 6				
Start (0 + 6 = 6)	1 + 5 = 6	2 + 4 = 6		

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8	5 + 3 = 8			
3 + 3 = 6	4 + 2 = 6		Dest	
2 + 4 = 6				
1 + 5 = 6		3 + 3 = 6		
Start (0 + 6 = 6)	1 + 5 = 6	2 + 4 = 6	3 + 3 = 6	

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8	5 + 3 = 8			
3 + 3 = 6	4 + 2 = 6		Dest	
2 + 4 = 6				
1 + 5 = 6		3 + 3 = 6		
Start (0 + 6 = 6)	1 + 5 = 6	2 + 4 = 6	3 + 3 = 6	

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8	5 + 3 = 8			
3 + 3 = 6	4 + 2 = 6		Dest	
2 + 4 = 6				
1 + 5 = 6		3 + 3 = 6		
Start (0 + 6 = 6)	1 + 5 = 6	2 + 4 = 6	3 + 3 = 6	4 + 4 = 8

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8	5 + 3 = 8	6 + 2 = 8		
3 + 3 = 6	4 + 2 = 6		Dest	
2 + 4 = 6				
1 + 5 = 6		3 + 3 = 6		
Start (0 + 6 = 6)	1 + 5 = 6	2 + 4 = 6	3 + 3 = 6	4 + 4 = 8

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8	5 + 3 = 8	6 + 2 = 8	7 + 1 = 8	
3 + 3 = 6	4 + 2 = 6		Dest	
2 + 4 = 6				
1 + 5 = 6		3 + 3 = 6		
Start (0 + 6 = 6)	1 + 5 = 6	2 + 4 = 6	3 + 3 = 6	4 + 4 = 8

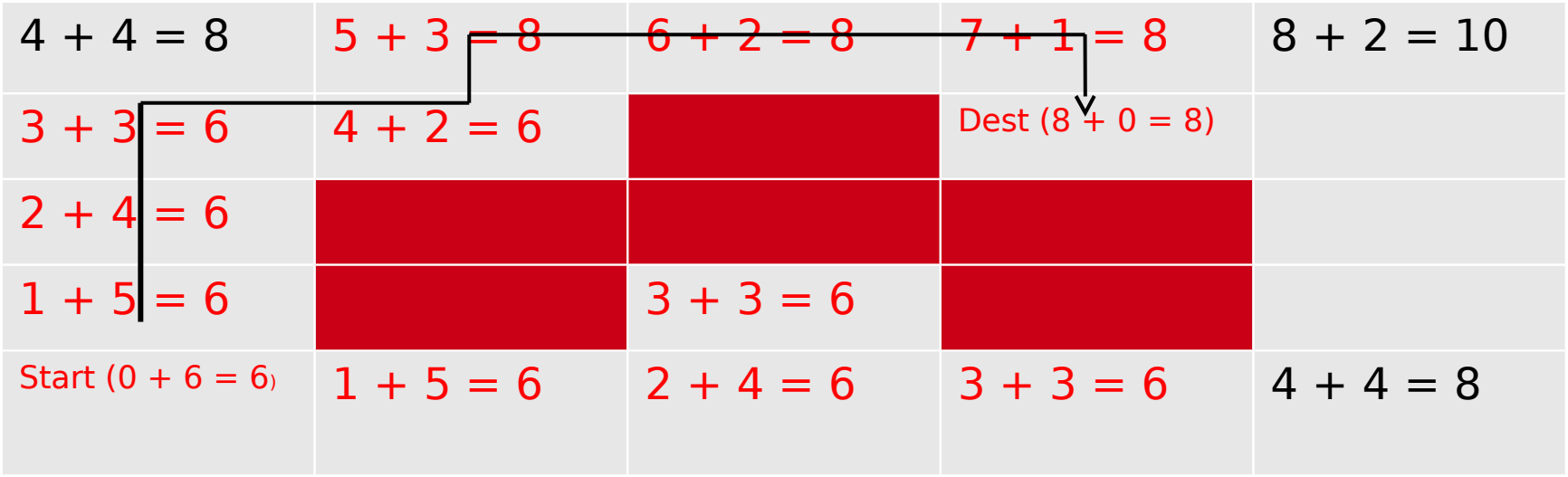
- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

4 + 4 = 8	5 + 3 = 8	6 + 2 = 8	7 + 1 = 8	8 + 2 = 10
3 + 3 = 6	4 + 2 = 6		Dest (8 + 0 = 8)	
2 + 4 = 6				
1 + 5 = 6		3 + 3 = 6		
Start (0 + 6 = 6)	1 + 5 = 6	2 + 4 = 6	3 + 3 = 6	4 + 4 = 8

- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search



- Manhattan Distance is used
- If there is a tie, consider the one with lowest heuristic first
- If there is still a tie, consider in Up, Down, Left, Right order

A* Search

- **A* Search = Past + Future**
- **A* Search – Future = ?**
  Dijkstra
- **A* Search – Past = ?**
  Greedy

Graph Theory

- **How about graph with negative edges?**
  - **Bellman-Ford**

Bellman-Ford Algorithm

- **Consider the source as weight 0, others as infinity**
- **Count = 0, Improve = true**
- **While (Count < n and Improve) {**
  - **Improve = false**
  - **Count++**
  - **For each edge uv**
    - **If u.weight + uv distance < v.weight {**
      - **Improve = true**
      - **v.weight = u.weight + uv.distance**
    - **}**
- **}**
- **If (Count == n)**
  print “Negative weight cycle detected”
- **Else print shortest path distance**

Bellman-Ford Algorithm

- **Time Compleity: O(VE)**

Graph Theory

- **All we have just dealt with is single source**
- **How about multiple sources (all sources)?**
  - **Floyd-Warshall Algorithm**

Floyd-Warshall Algorithm

- **1. Compute on subgraph {1}**
- **2. Compute on subgraph {1 ,2}**
- **3. Compute on subgraph {1, …, 3}**
- **4. …**
- **N. Compute on entire graph**
- **Done !**

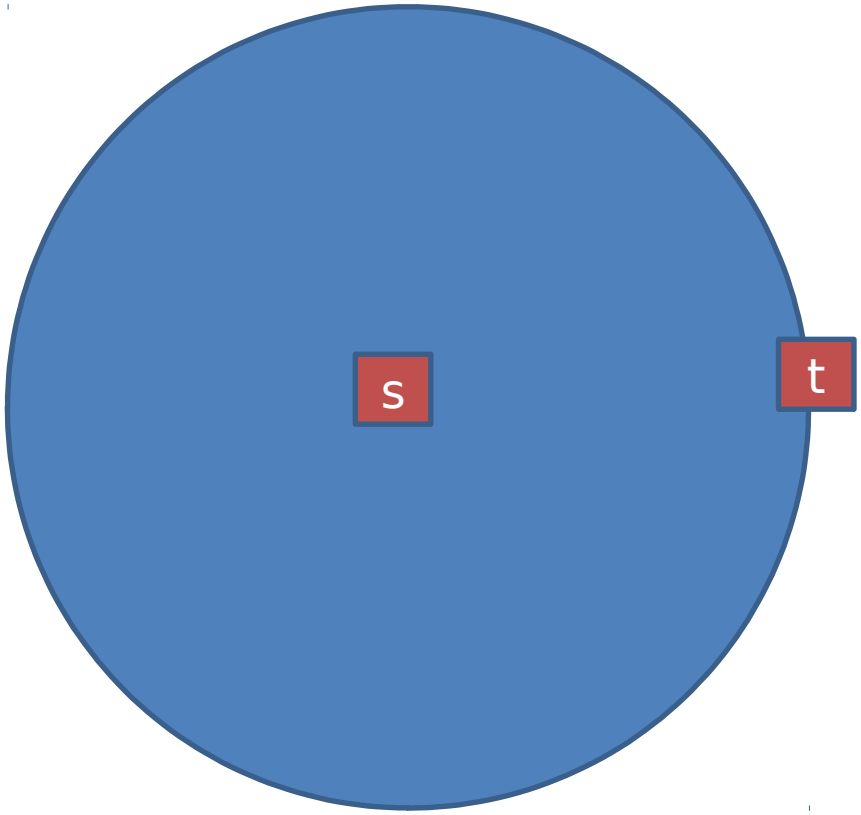
Floyd-Warshall Algorithm

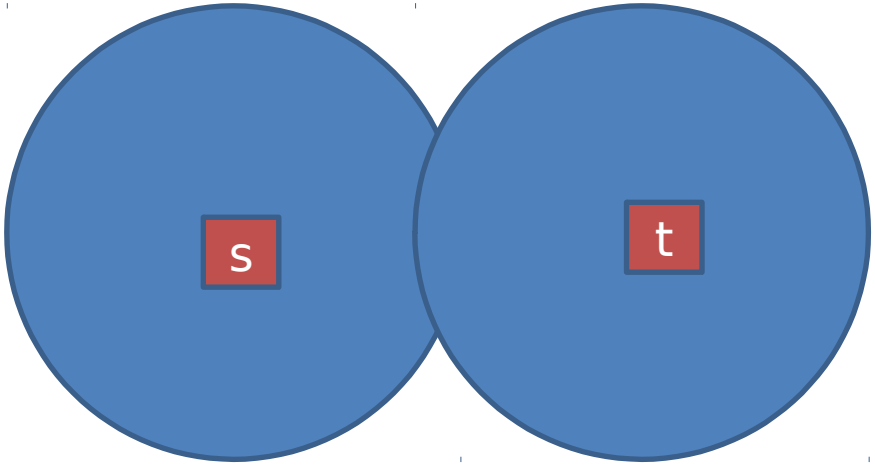
- **For (int k = 0; k < MAX; k++) {**
  - **For (int i = 0; i < MAX; i++) {**
    - **For (int j = 0; j < MAX; j++) {**
      - **Path[i][j] = min(Path[i][j], Path[i][k] + Path[k][j]);**
    - **}**
  - **}**
- **}**

Iterative? Non-iterative?

- **What is iterative-deepening?**
- **Given limited time, f nd the current most optimal solution**
- **Dijkstra (Shortest 1-step Solution)**
- **Dijkstra (Shortest 2-steps Solution)**
- **Dijkstra (Shortest 3-steps Solution)**
- **…**
- **Dijkstra (Shortest n-steps Solution)**

- **Suitable in bi-directional search (Faster than 1-way search): only for DFS, tricky otherwise**


s
t


s
t

Minimum Spanning Tree (MST)

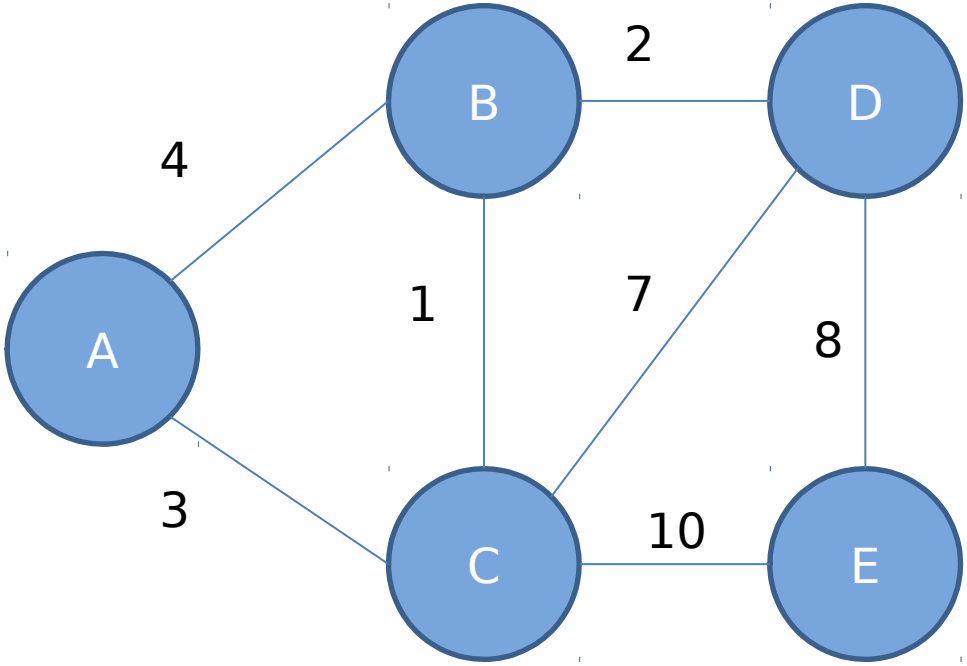
- **Given an undirected graph, find the minimum cost so that every node has path to other nodes**

- **Kruskal’s Algorithm**
- **Prim’s Algorithm**

Kruskal's Algorithm

- **Continue Finding Shortest Edge**
- **If no cycle is formed**
  - **add it into the list**
  - **Construct a parent-child relationship**
- **If all nodes have the same root, we are done (path compression)**
- **General idea: union-find, may be useful in other situations**

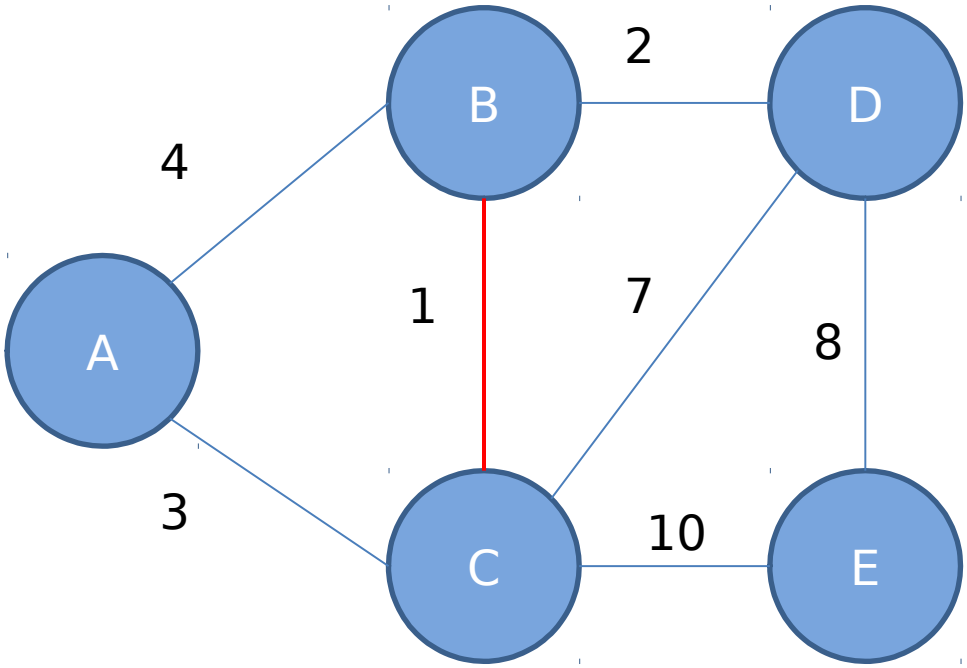
Kruskal’s Algorithm



Node	A	B	C	D	E
Parent	A	B	C	D	E

Kruskal's Algorithm

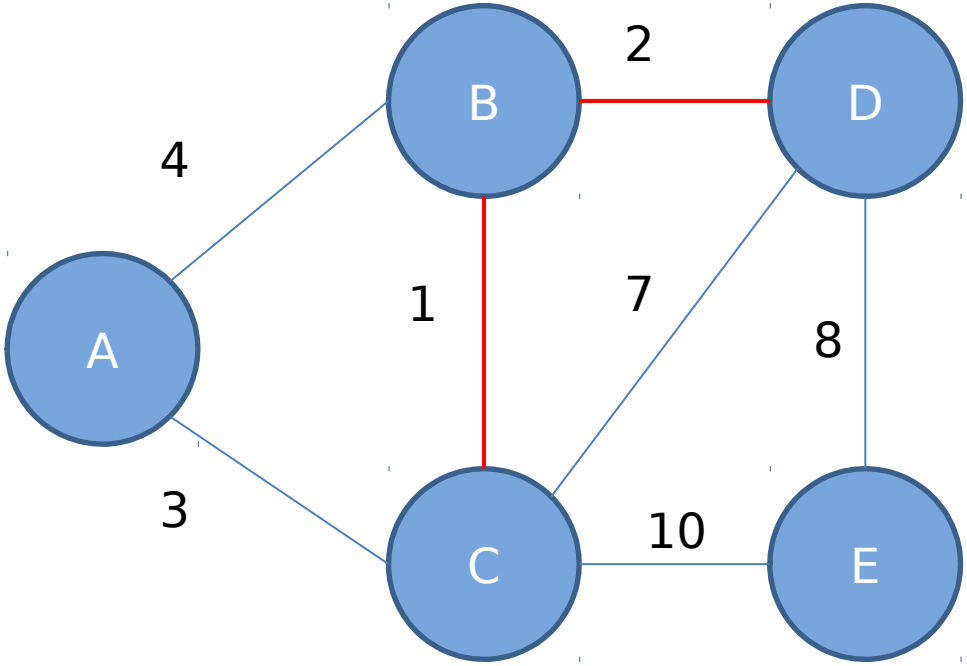
Edge List:
BC 1



Node	A	B	C	D	E
Parent	A	B	B	D	E

Kruskal's Algorithm

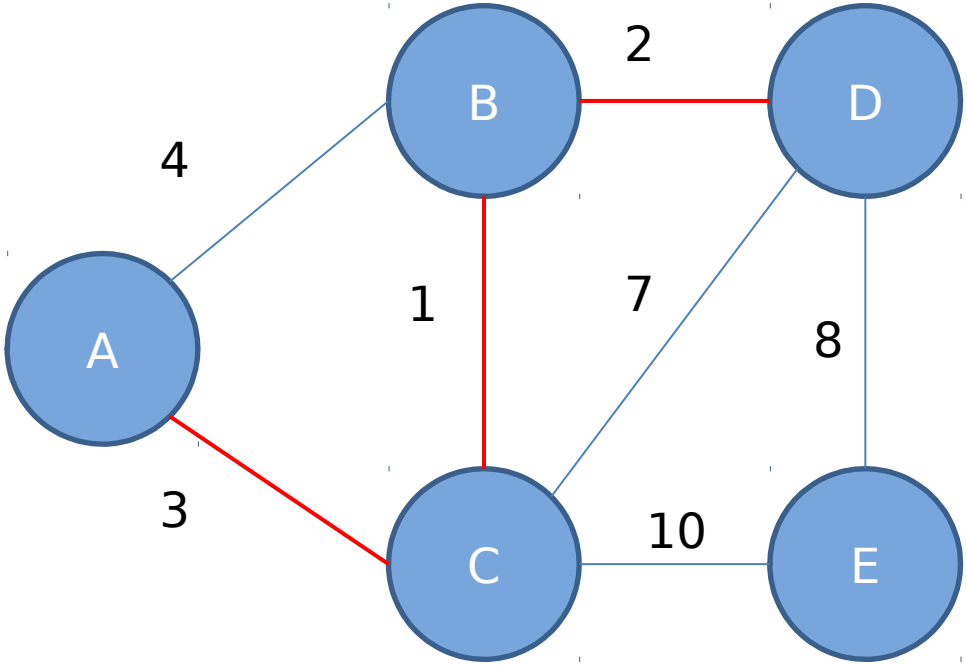
Edge List:
BC 1



Node	A	B	C	D	E
Parent	A	B	B	B	E

Kruskal's Algorithm

Edge List:
BC 1

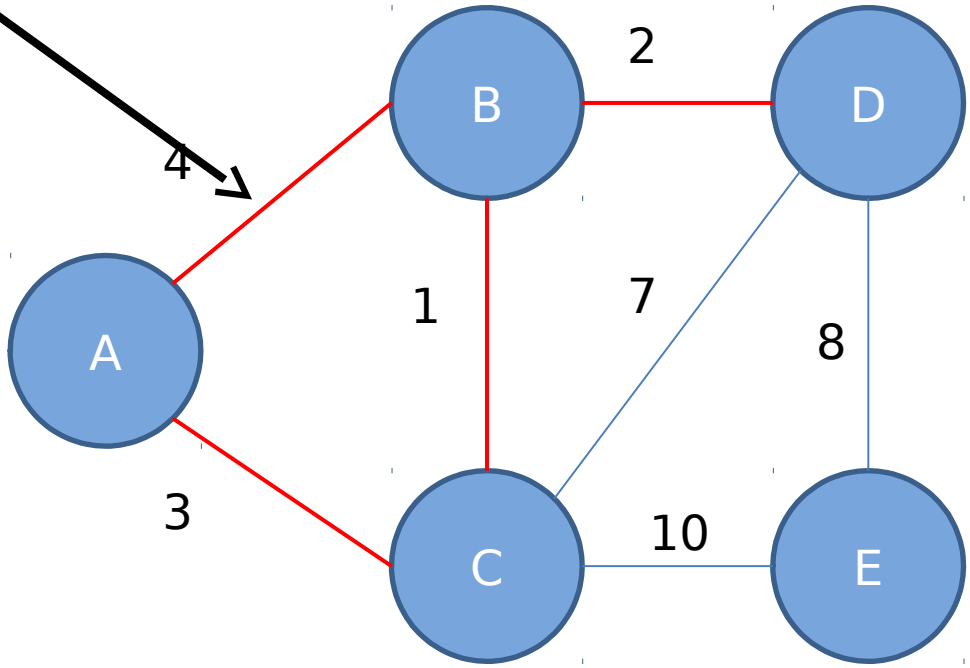


Node	A	B	C	D	E
Parent	A	A	A	A	E

Kruskal's Algorithm

Edge List:
BC 1

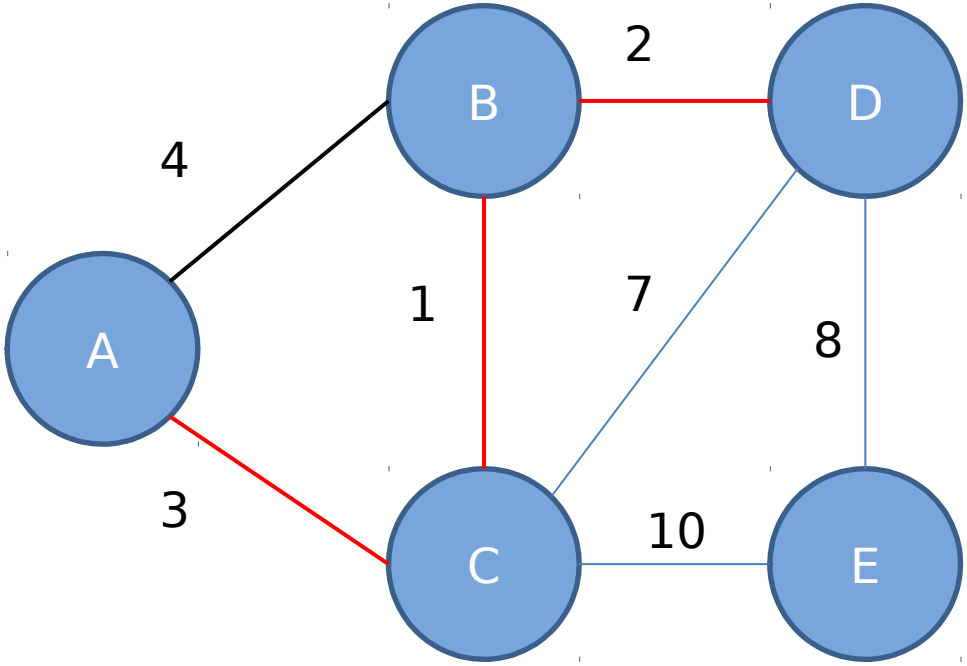
Cycle formed



Node	A	B	C	D	E
Parent	A	A	A	A	E

Kruskal's Algorithm

Edge List:
BC 1

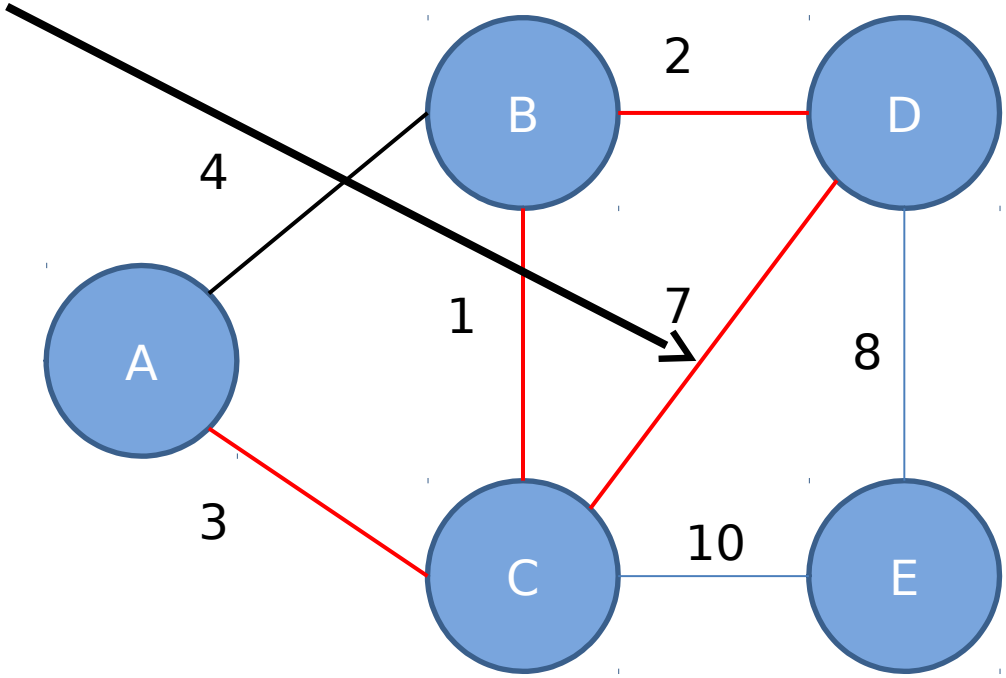


Node	A	B	C	D	E
Parent	A	A	A	A	E

Kruskal's Algorithm

Edge List:
BC 1

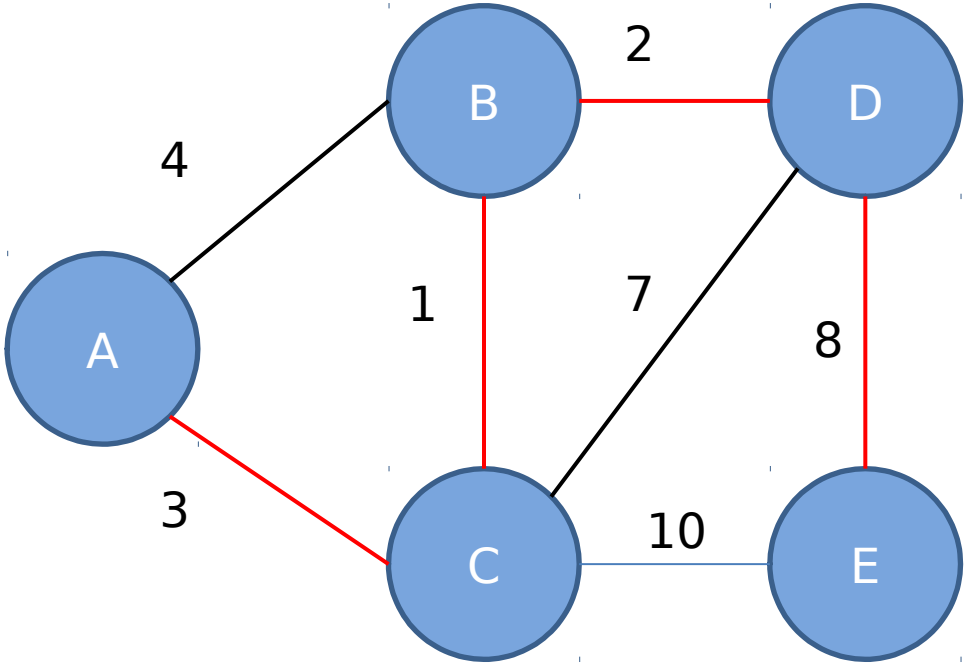
Cycle formed



Node	A	B	C	D	E
Parent	A	A	A	A	E

Kruskal's Algorithm

Edge List:
BC 1



Node	A	B	C	D	E
Parent	A	A	A	A	A

Kruskal’s Algorithm

- **Path Compression can be done when we find root**

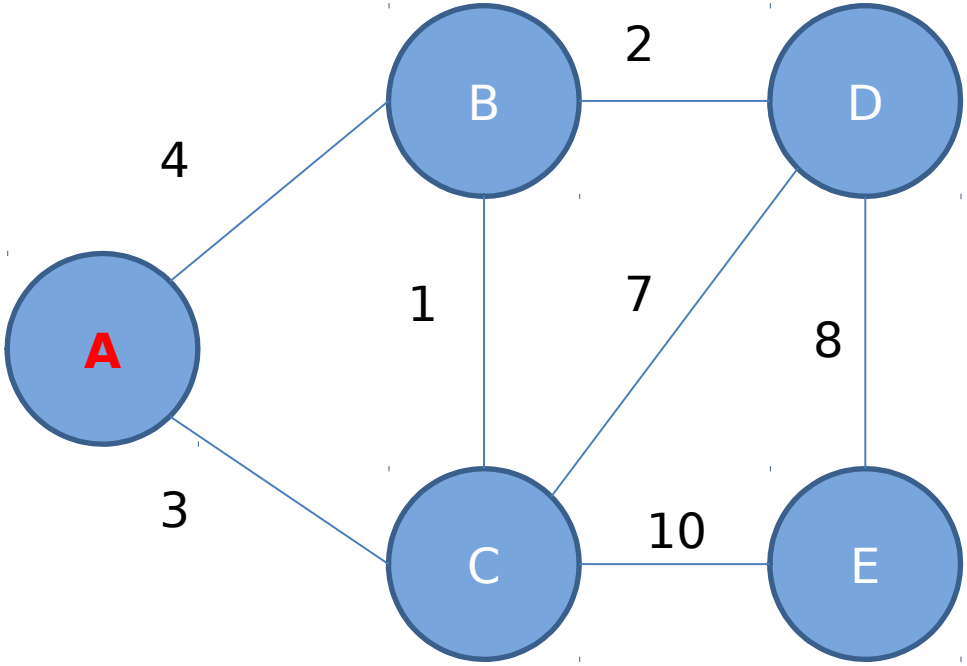
```
int find_root(int x) {
    return (parent[x] == x)? x:
    return (parent[x] = find_root(parent[x]));
}
```

Prim’s Algorithm

- **While not all nodes are visited**
  - **While the list is not empty and the minimum edge connects to visited node**
    - **Pop out the edge**
  - **If the list is empty, print Impossible and return**
  - **Else**
    - **Consider popped edge**
    - **Mark its endpoint as visited**
    - **Add all its unvisited edges to the list**
- **Print the tree**

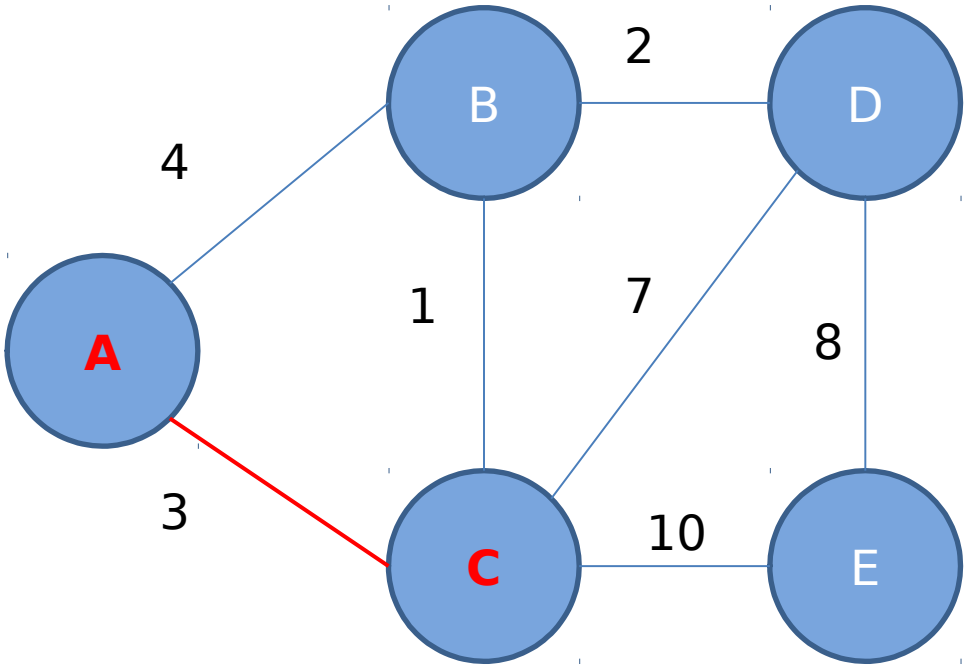
Prim’s Algorithm

Edge List:
AC 3
AB 4



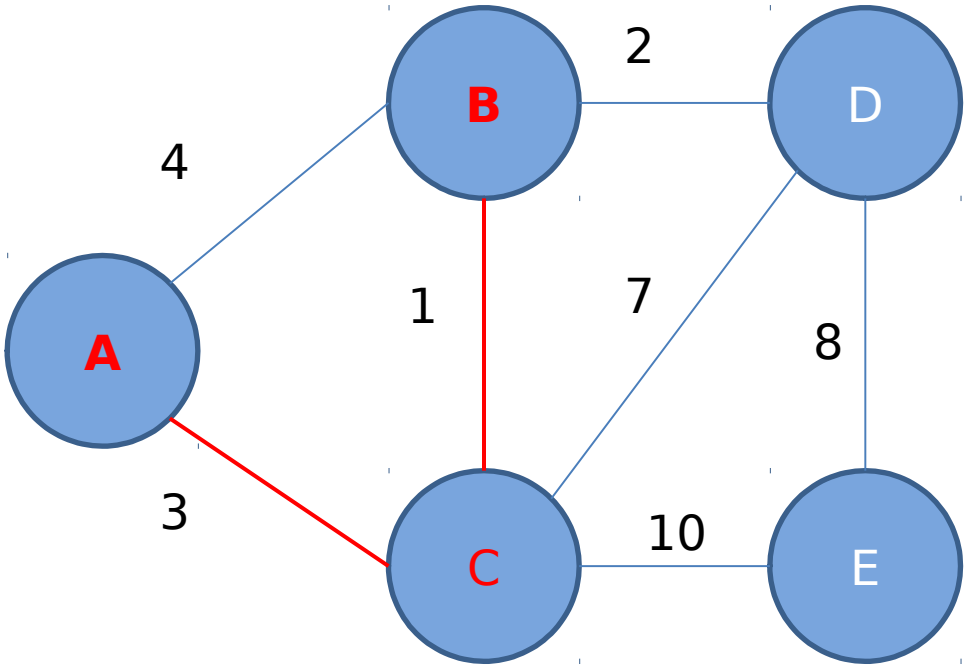
Prim’s Algorithm

Edge List:
CB 1
AB 4
CD 7
CE 10



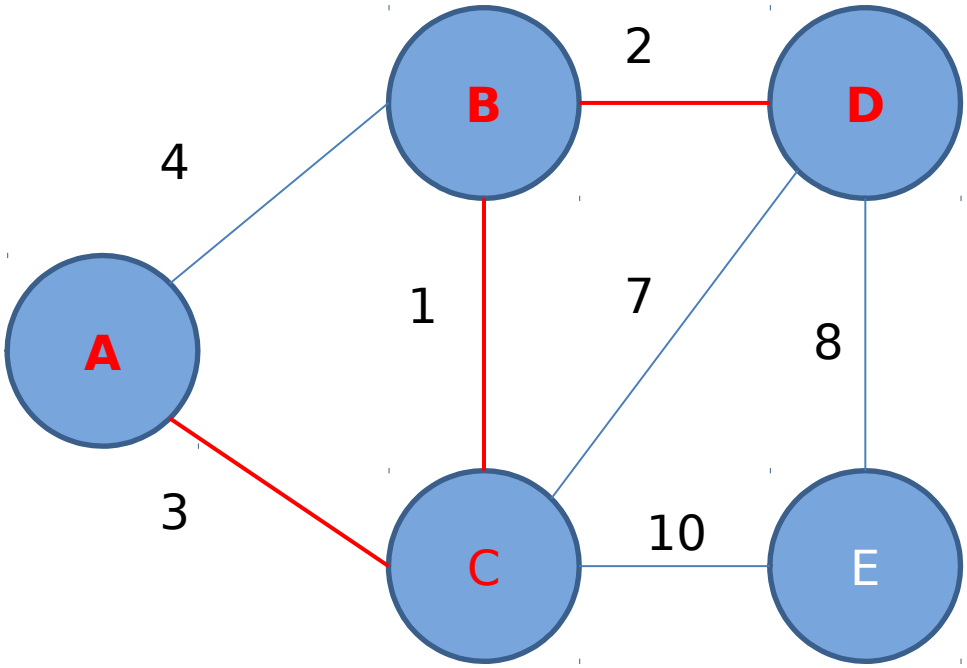
Prim’s Algorithm

Edge List:
BD 2
AB 4
CD 7
CE 10



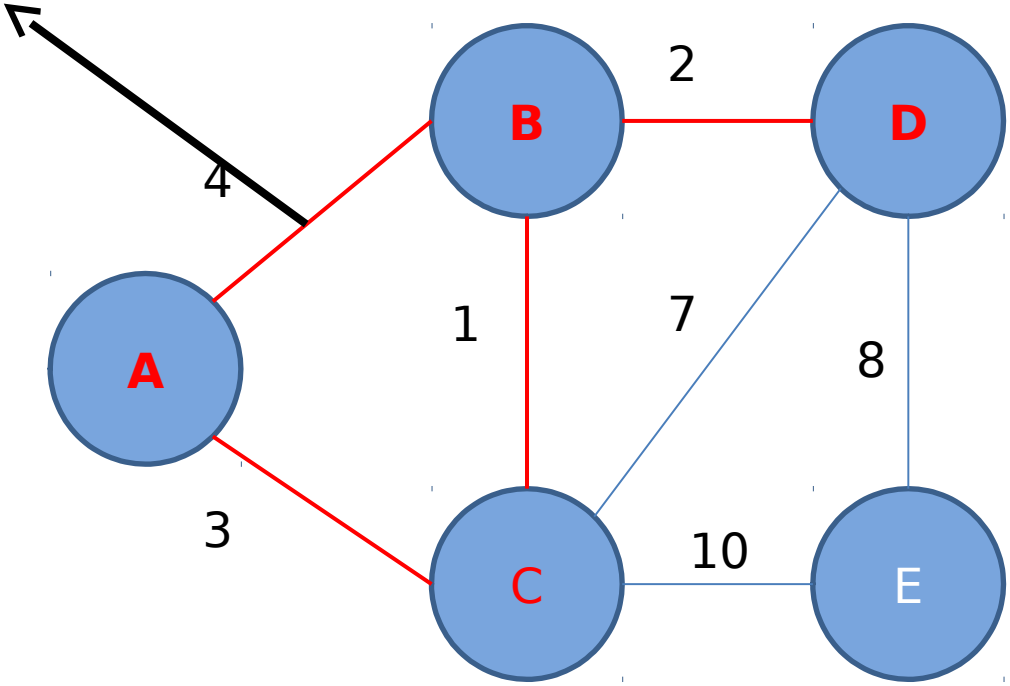
Prim’s Algorithm

Edge List:
AB 4
CD 7
DE 8
CE 10



Prim’s Algorithm

Cycle formed



Edge List:

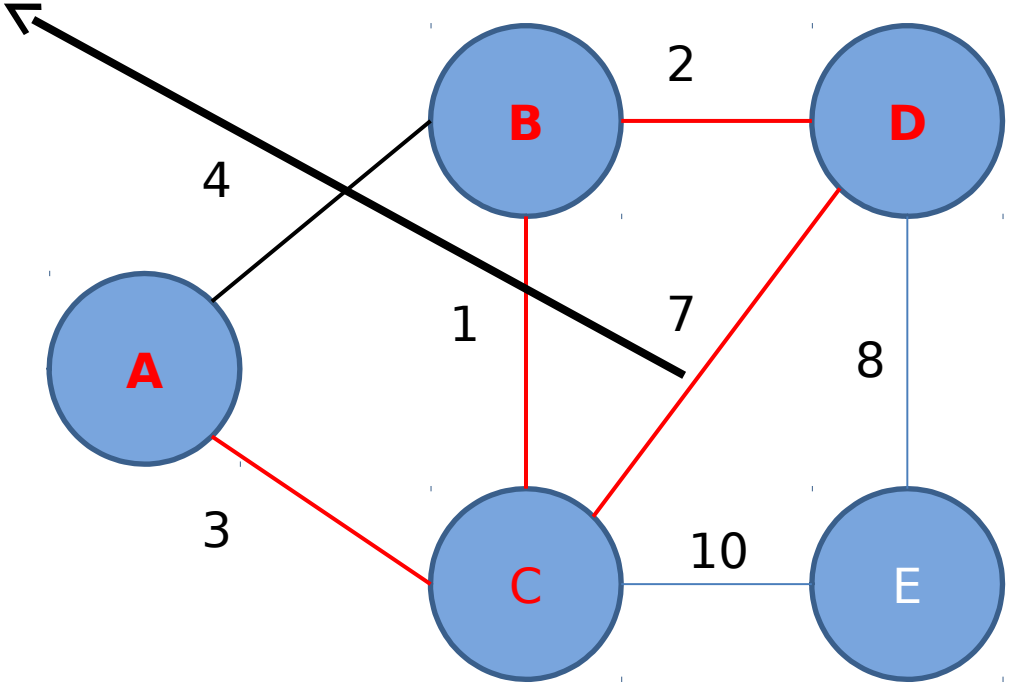
CD 7

DE 8

CE 10

Prim’s Algorithm

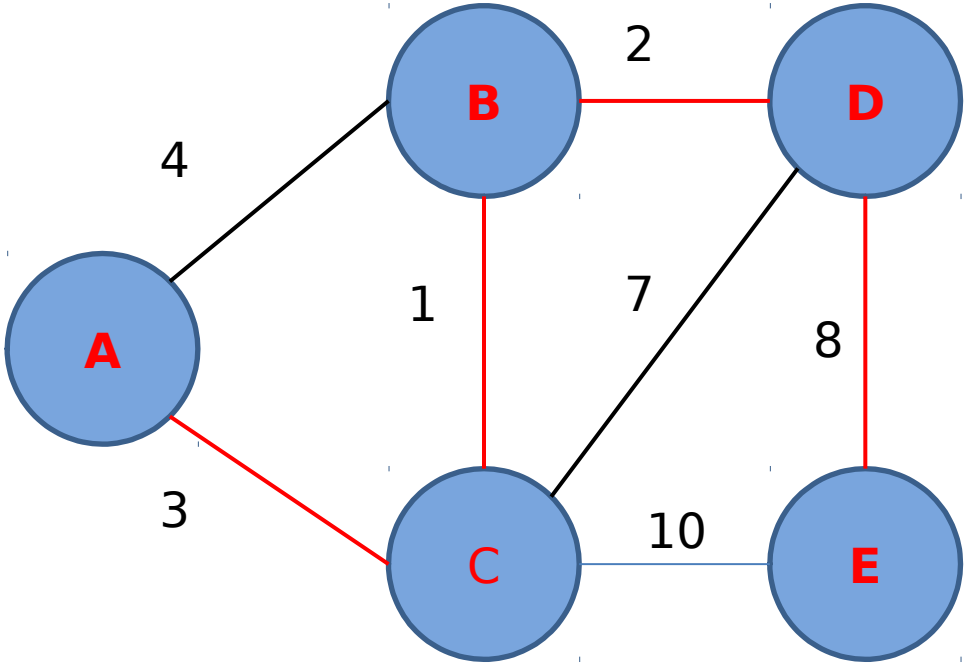
Cycle formed



Edge List:
DE 8
CE 10

Prim’s Algorithm

Edge List:
CE 10



We are done

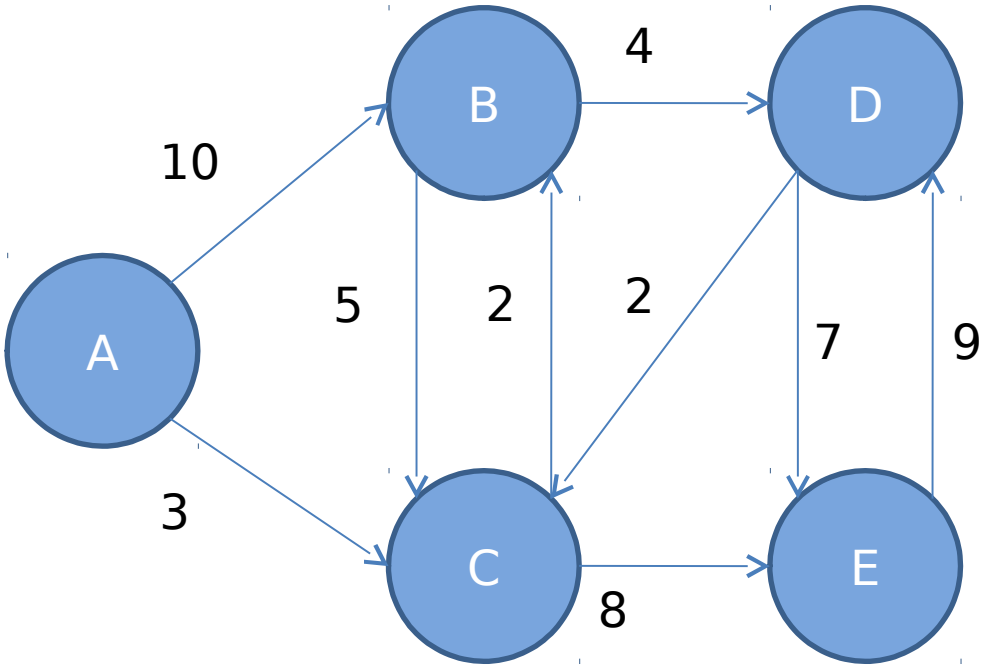
Directed Minimum Spanning Tree

- **We have just dealt with undirected MST.**
- **How about directed?**
- **Using Kruskal's or Prim's cannot solve directed MST problem**
- **How?**
- **Chu-Liu/Edmonds Algorithm**

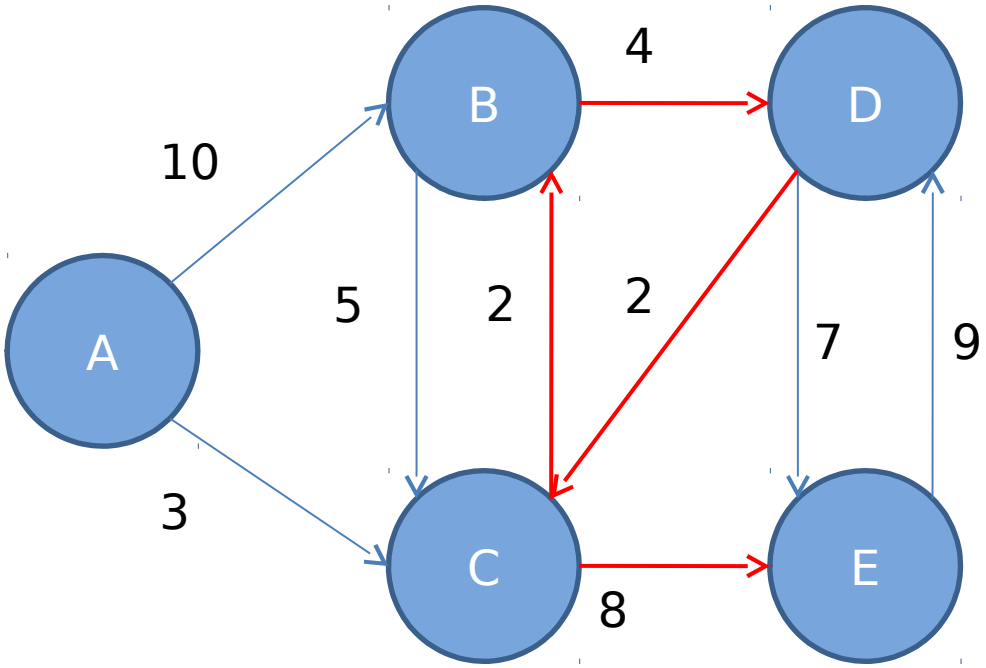
Chu-Liu/Edmonds Algorithm

- **For each vertex, find the smallest incoming edge**
- **While there is cycle**
  - **Find an edge entering the cycle with smallest increase**
  - **Remove the edge pointing to the same node and replace by new edge**
- **Print out the tree**

Chu-Liu/Edmonds Algorithm



Chu-Liu/Edmonds Algorithm



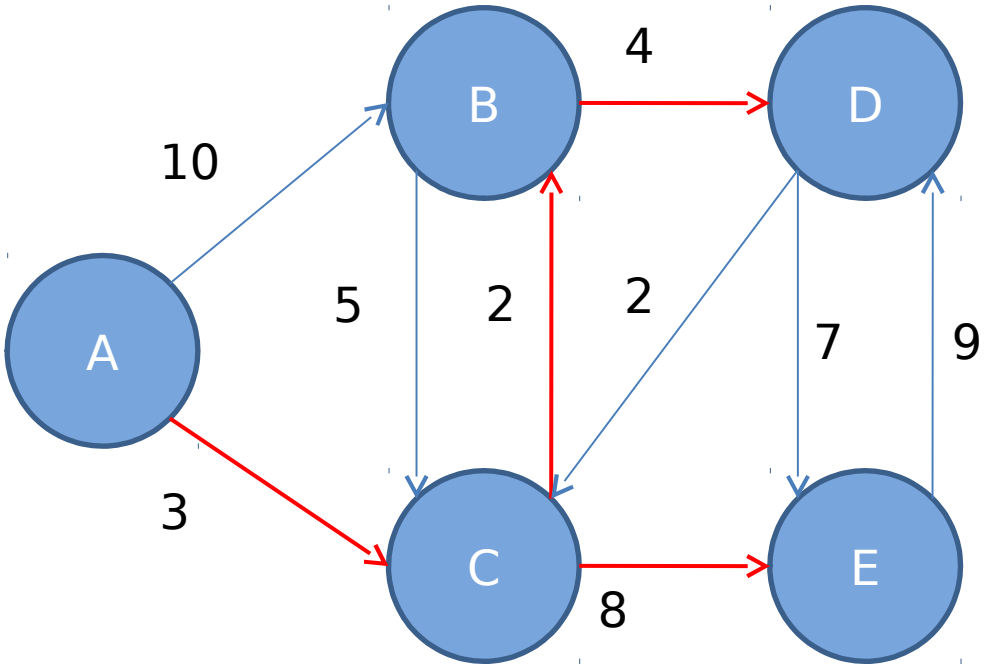
Cycle formed (B, C, D)

A => B, C, D = 10 – 2 = 8

A => B, C, D = 3 – 2 = 1

E => B, C, D = 9 – 4 = 5

Chu-Liu/Edmonds Algorithm



No cycle formed, we terminate

Chu-Liu/Edmonds Algorithm

- **Using this algorithm may unloop a cycle and create another cycle…**
- **… But since the length of tree is increasing, there will have an end eventually**
- **Worst Case is doing n-1 times**
- **Time Complexity is O(EV)**

Flow

- **A graph with weight (capacity)**
- **Mission: Find out the maximum flow from source to destination**
- **How?**

Edmond-Karp

- **Do {**
  - **Do BFS on the graph to find maximum flow in the graph**
  - **Add it to the total flow**
  - **For each edge in the maximum flow**
  - **Deduct the flow just passed**
- **} While the flow is not zero;**

The End