CS202 – Data Structures

LECTURE-20

Graphs – III

Graph Problems

Dr. Maryam Abdul Ghafoor
Assistant Professor
Department of Computer Science, SBASSE

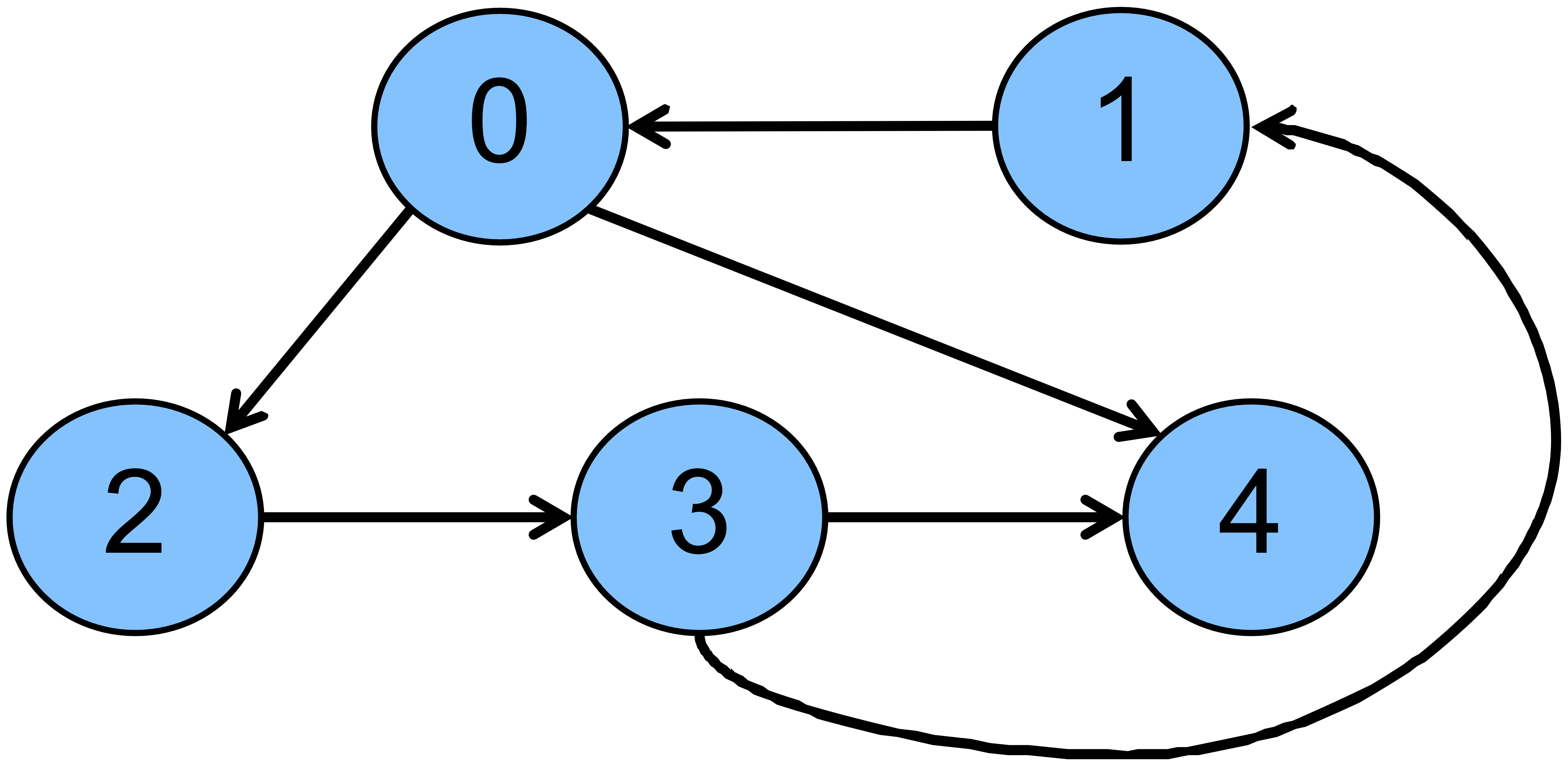
Agenda

- Problem Solving using Graphs
  - Cycle detection
  - Path between two vertices
  - Shortest path algorithms

Problem: Cycle detection

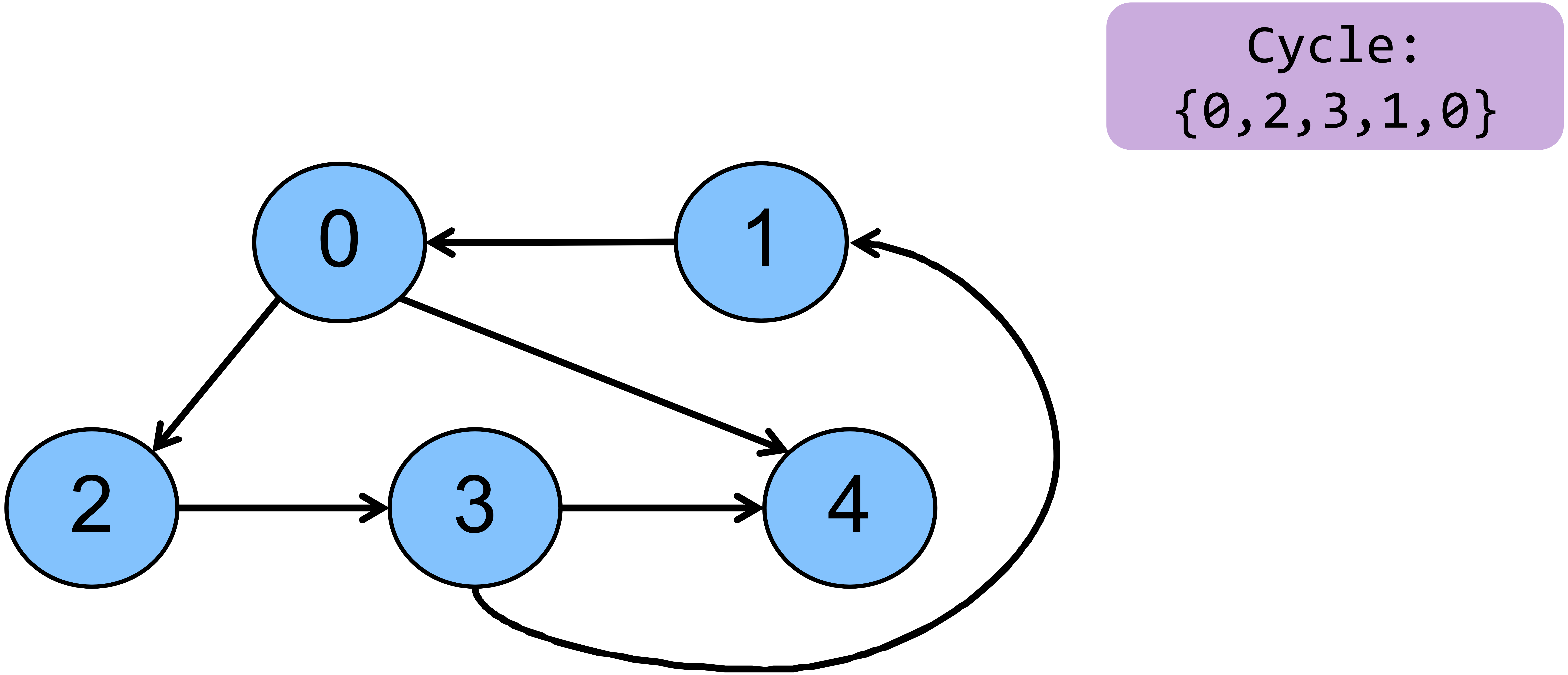
"Does the graph G contain a cycle?"

Cycle Detection



Cycle:
{0,2,3,1,0}

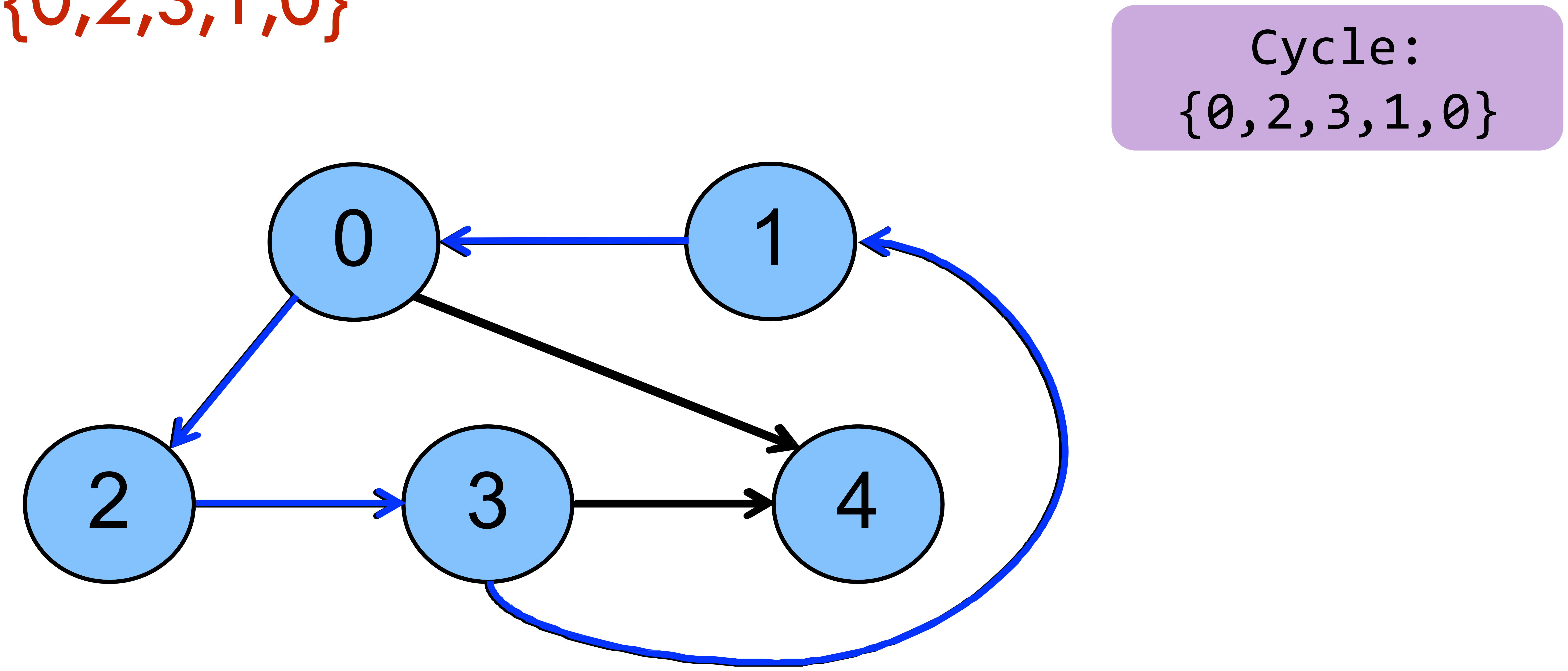
Cycle Detection



Approach 1: Run DFS, if you encounter a vertex that is already visited then return "there is a cycle"

Cycle Detection

- dfs(0) = {0,2,3,1,0}

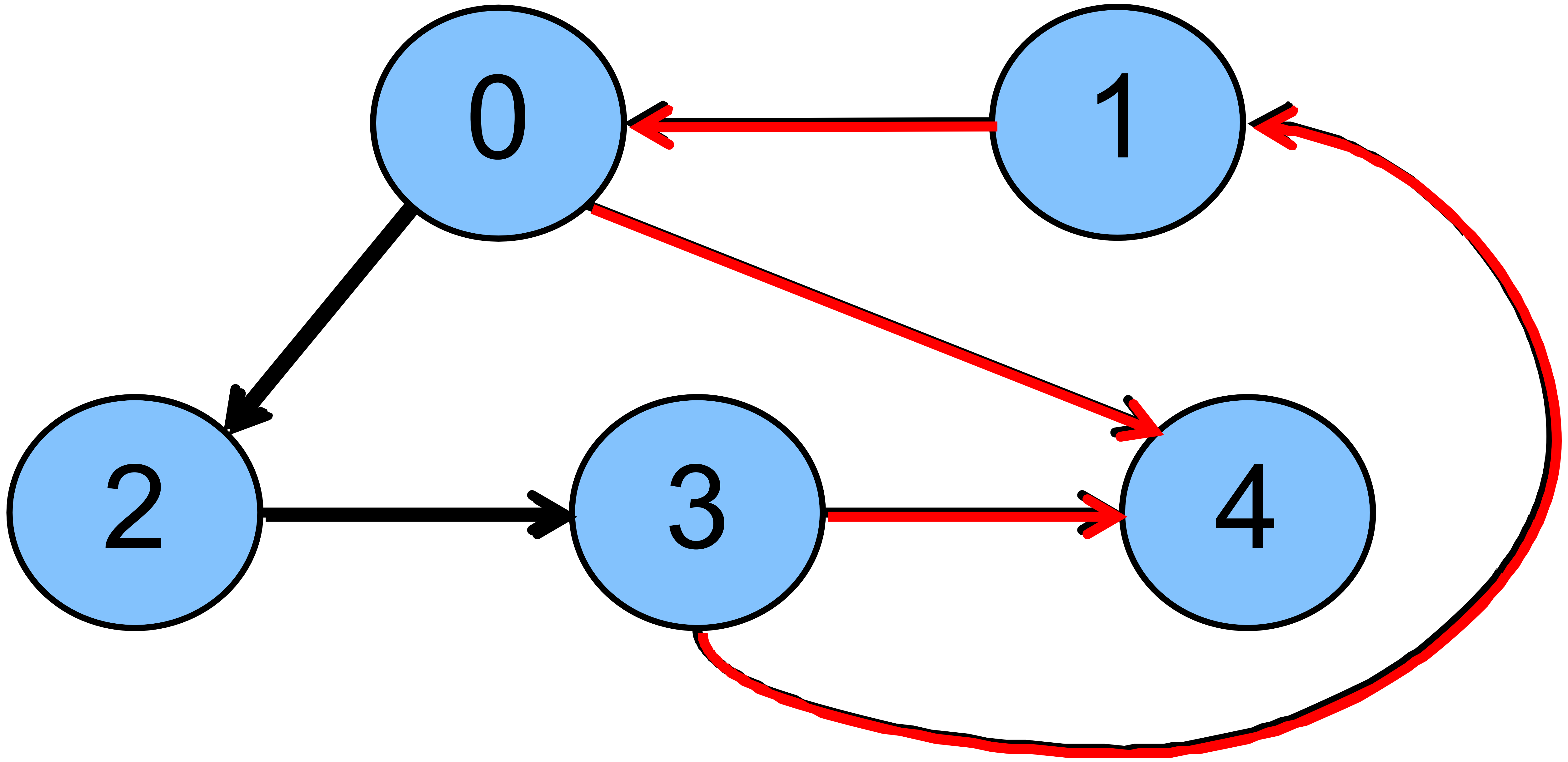


Approach 1: Run DFS, if you encounter a vertex that is already visited then return “there is a cycle”

Cycle Detection

- dfs(0) = {0,2,3,1,0}
- dfs(3) ={3,4,1,0,4*} NOT a cycle!

Cycle:
{0,2,3,1,0}

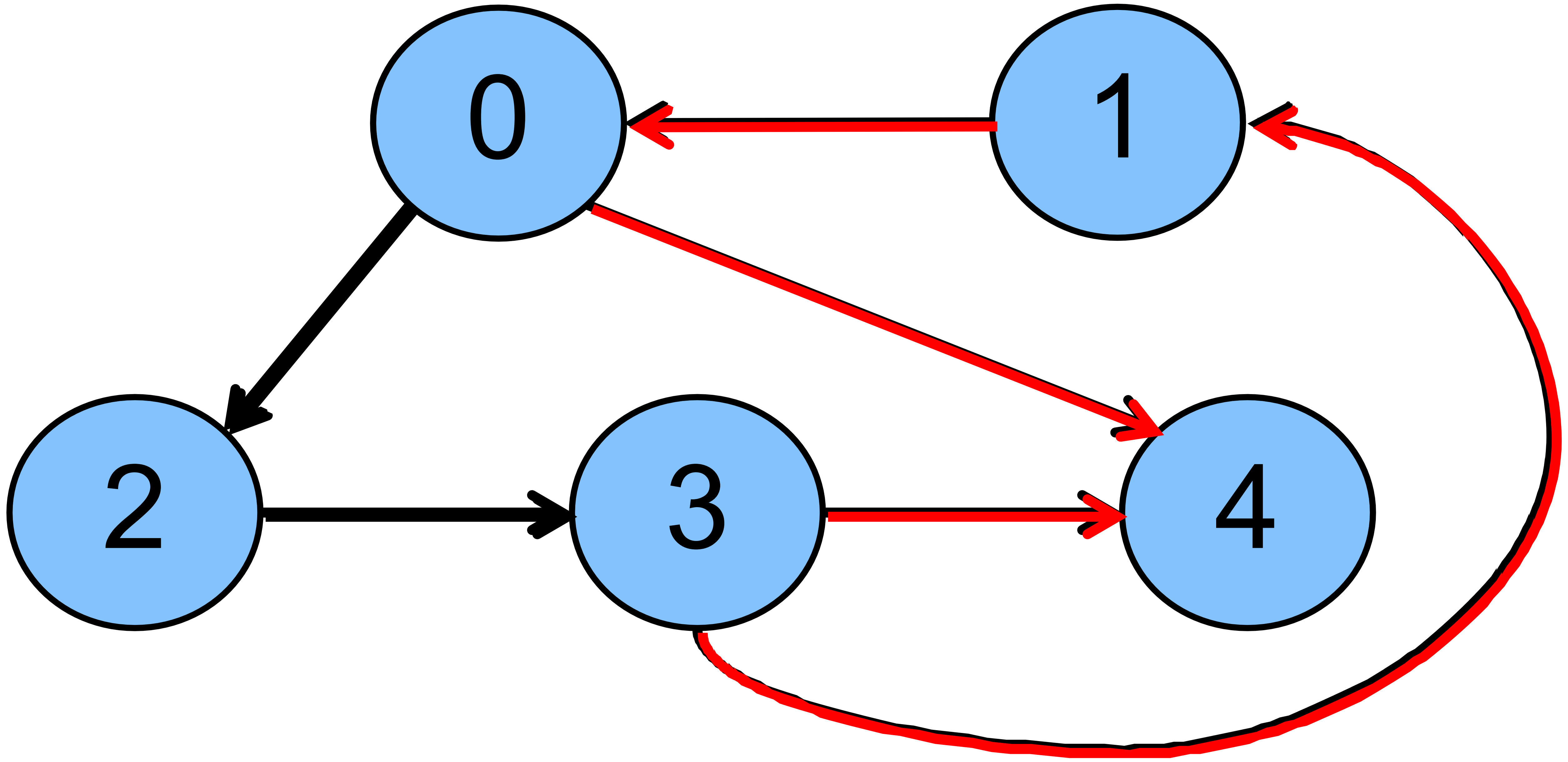


Approach 1: Run DFS, if you encounter a vertex that is already visited then return “there is a cycle”

Cycle Detection

- dfs(0) = {0,2,3,1,0}
- dfs(3) ={3,4,1,0,4*} NOT a cycle!

```
Cycle:
{0,2,3,1,0}
```

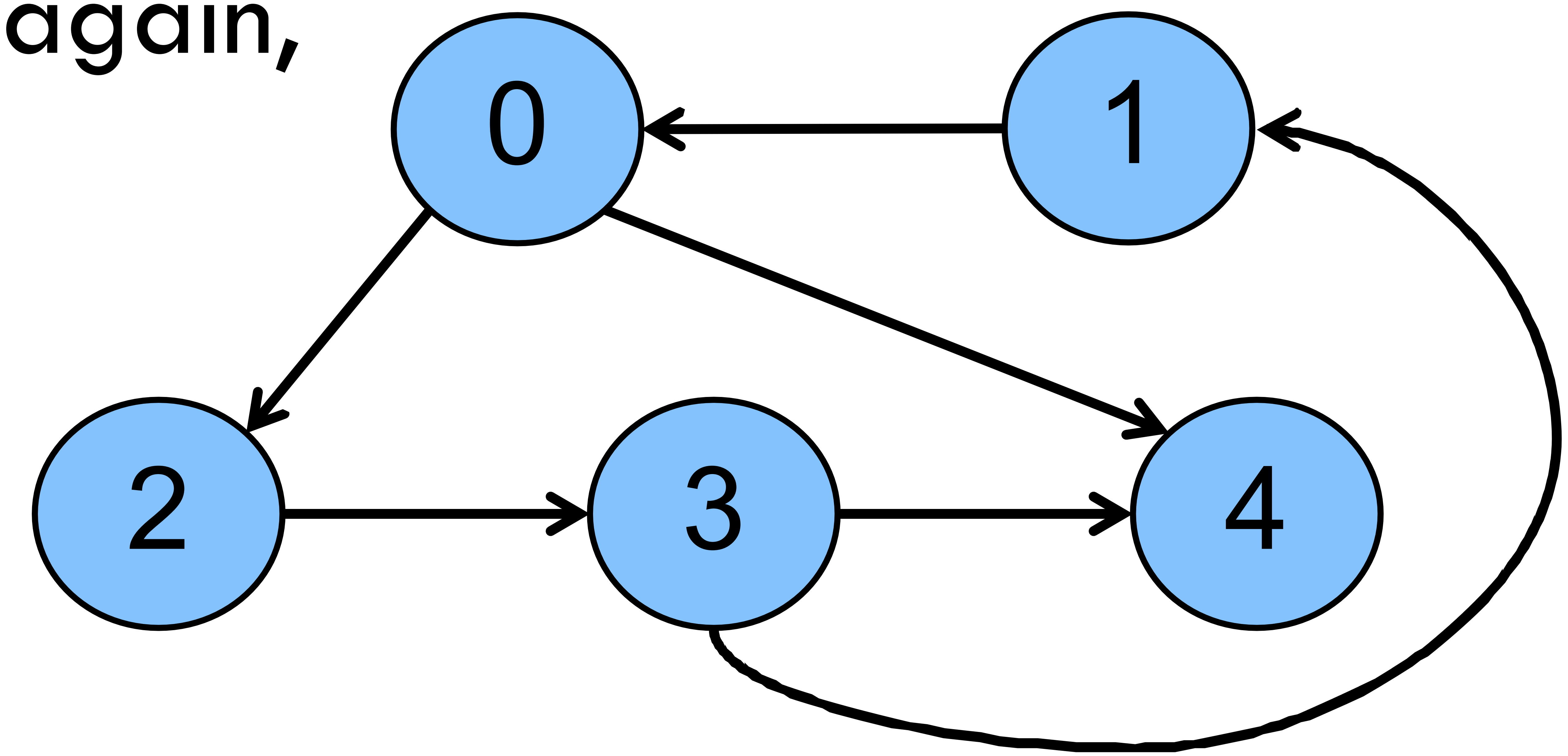


Approach 1: Run DFS, if you encounter a vertex that is already visited then return “there is a cycle”

Cycle Detection – Observations

- Case-1: when we visited vertex 0 again, dfs(0) was still active!
- Case-2: when we visited 4 again, dfs(4) was already done

```
Cycle:
{0,2,3,1,0}
```



Approach 2: Keep track of when a vertex is "inprogress"
Use a 3-state field to mark progress: (unvisited, inprogress, done)

Cycle Detection – Observations

- **Approach 2:** Keep track of when a vertex is "inprogress"
- Use a 3-state field to mark progress: (unvisited, inprogress, done)
    1. Initially, all nodes are unvisited
    2. When a node is first visited, we mark it as "inprogress"
    3. Once all successor nodes are visited, we mark it as done
    4. There is a cyclic path reachable from vertex i iff some node's successor is found to be marked "inProgress" during dfs(i)

Time Complexity

AL: $O(|V| + |E|)$

AM: $O(|V|^2)$

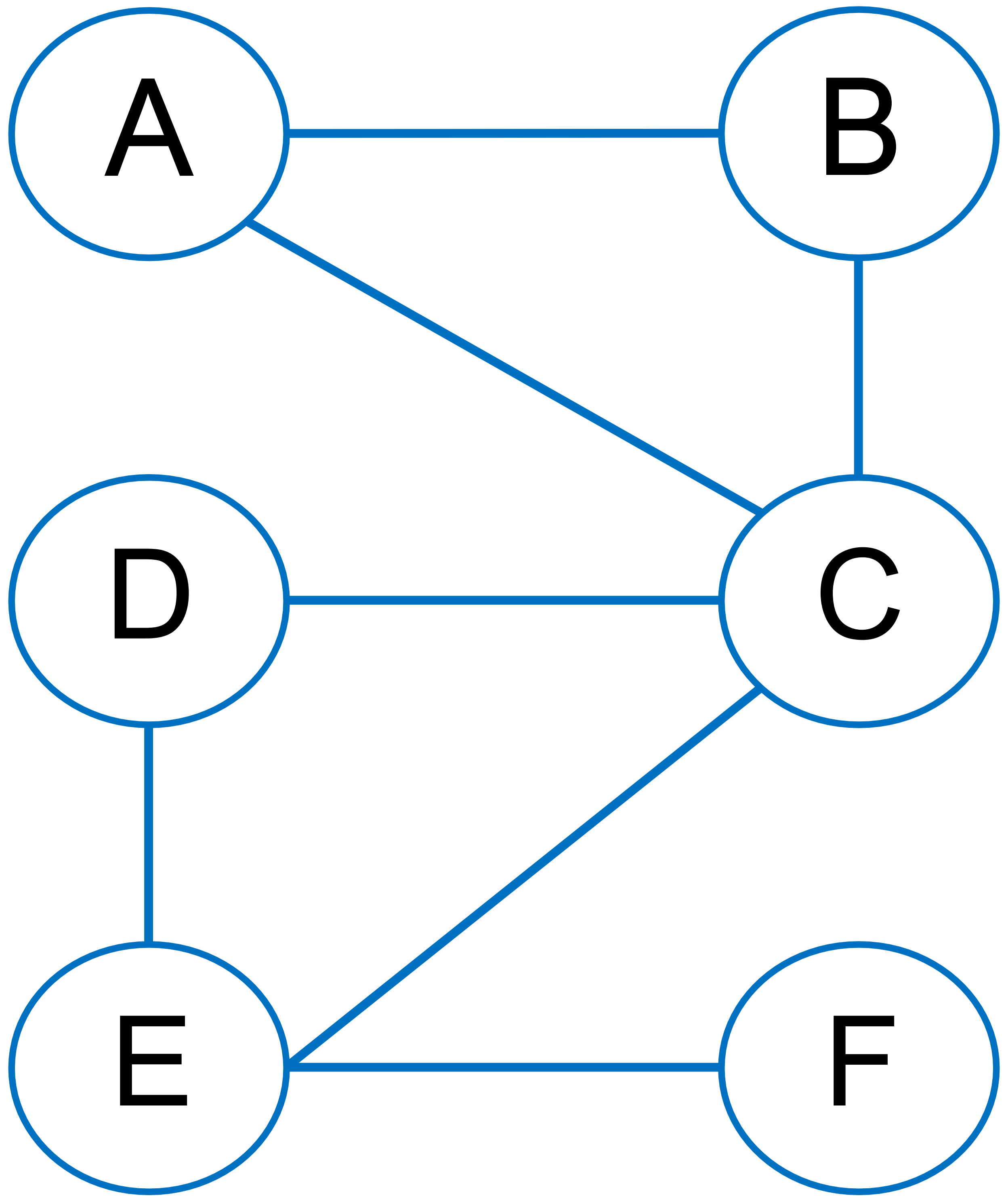
Let's try to solve some problems

- Problem (Path detection): "Is there a path from vertex s to vertex t?"

- Solution:
  - Run DFS start from vertex s
  - If vertex k is visited → there is a path from j to k
  - Time complexity?

Problem: Shortest Path

- How can we find the shortest path between two vertices in a graph



Problem: Shortest Path

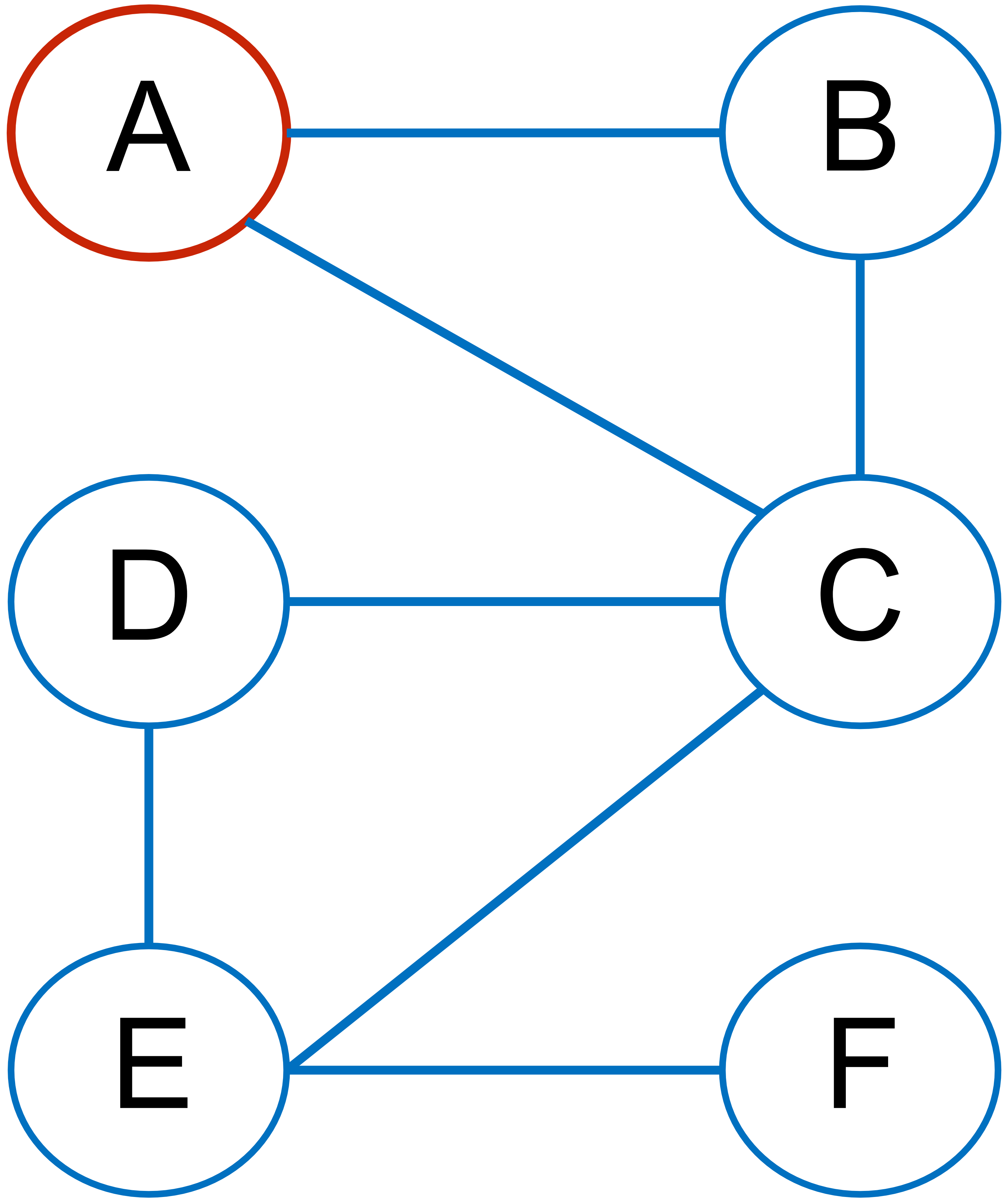
- How can we find the shortest path between two vertices using BFS in an unweighted graph

- Solution:

(a) Add a distance field to each node
(b) When bfs is called on a vertex v, set v's distance to zero
(c) When a vertex w is to be enqueued, set its distance to the distance of the current vertex + 1

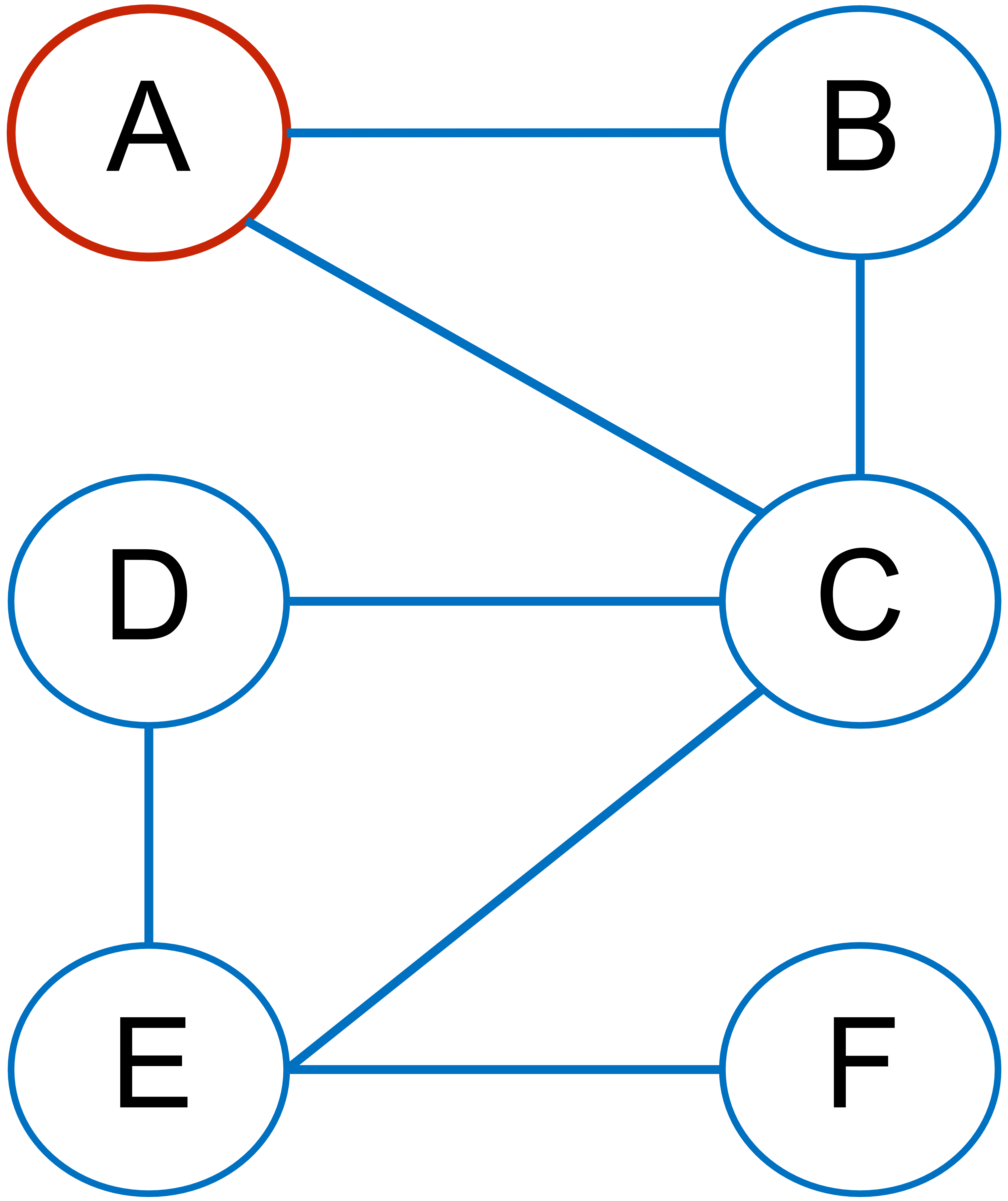
BFS

dist=0, parent=∅

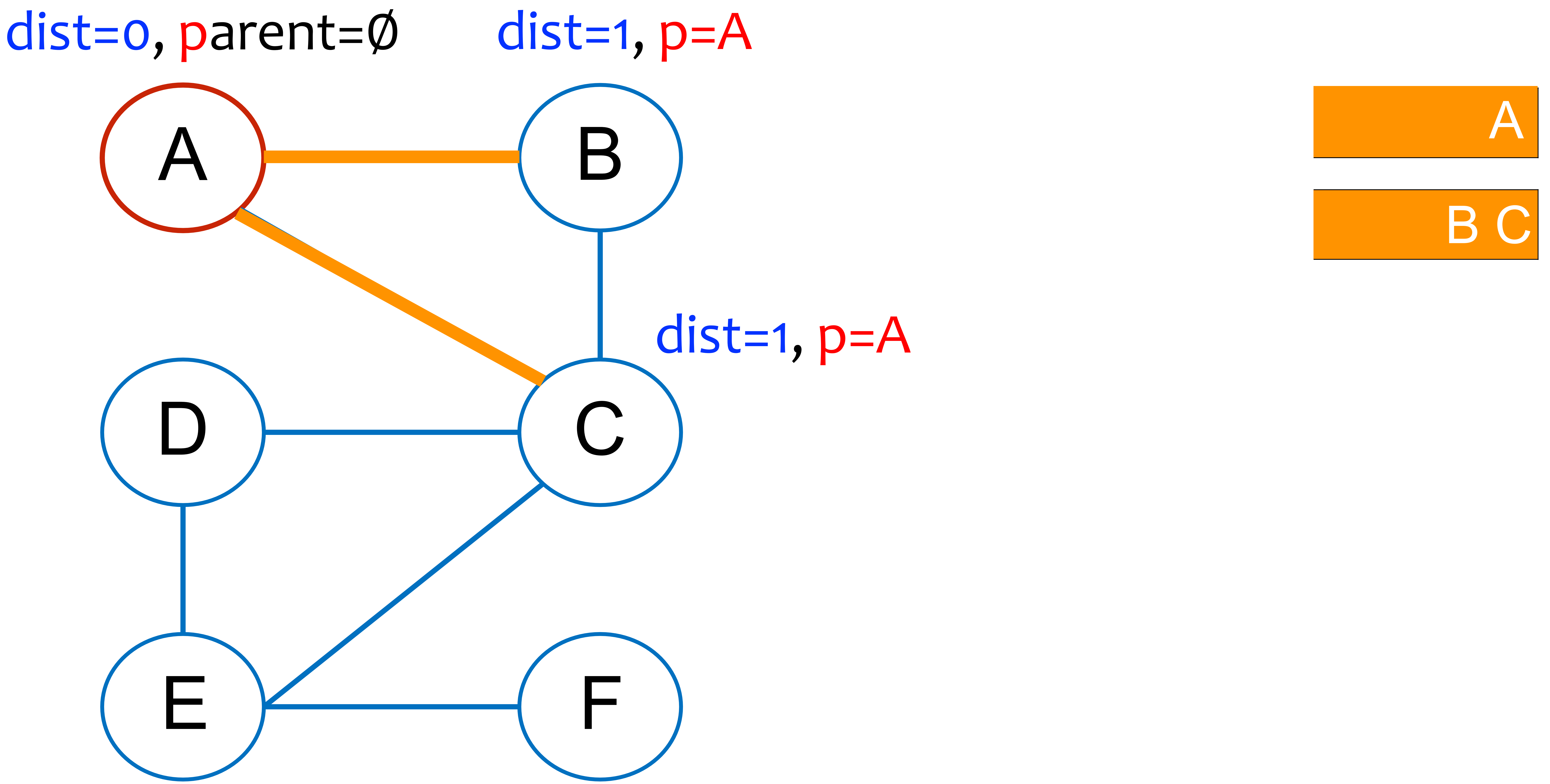


BFS

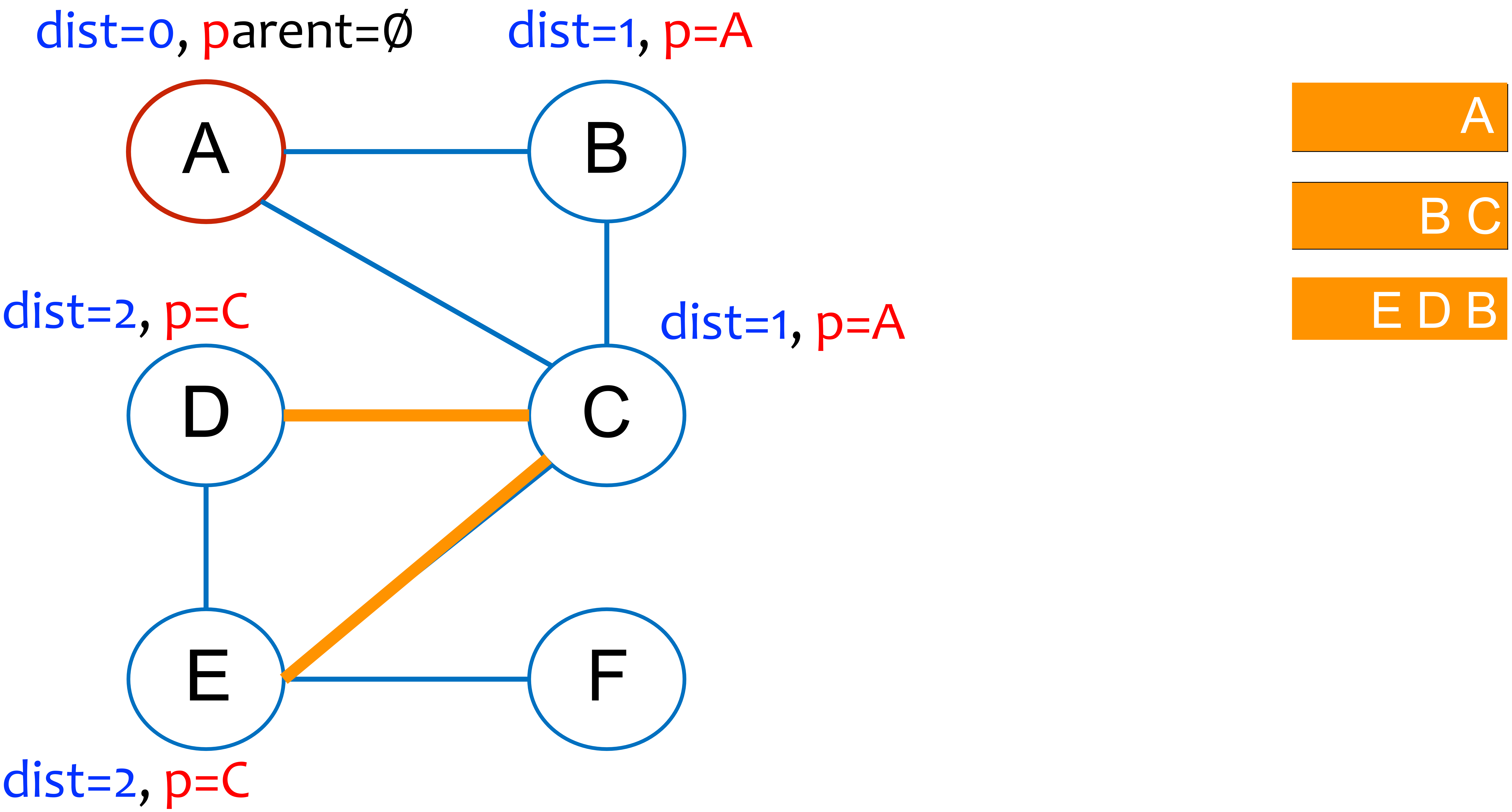
dist=0, parent=∅



BFS

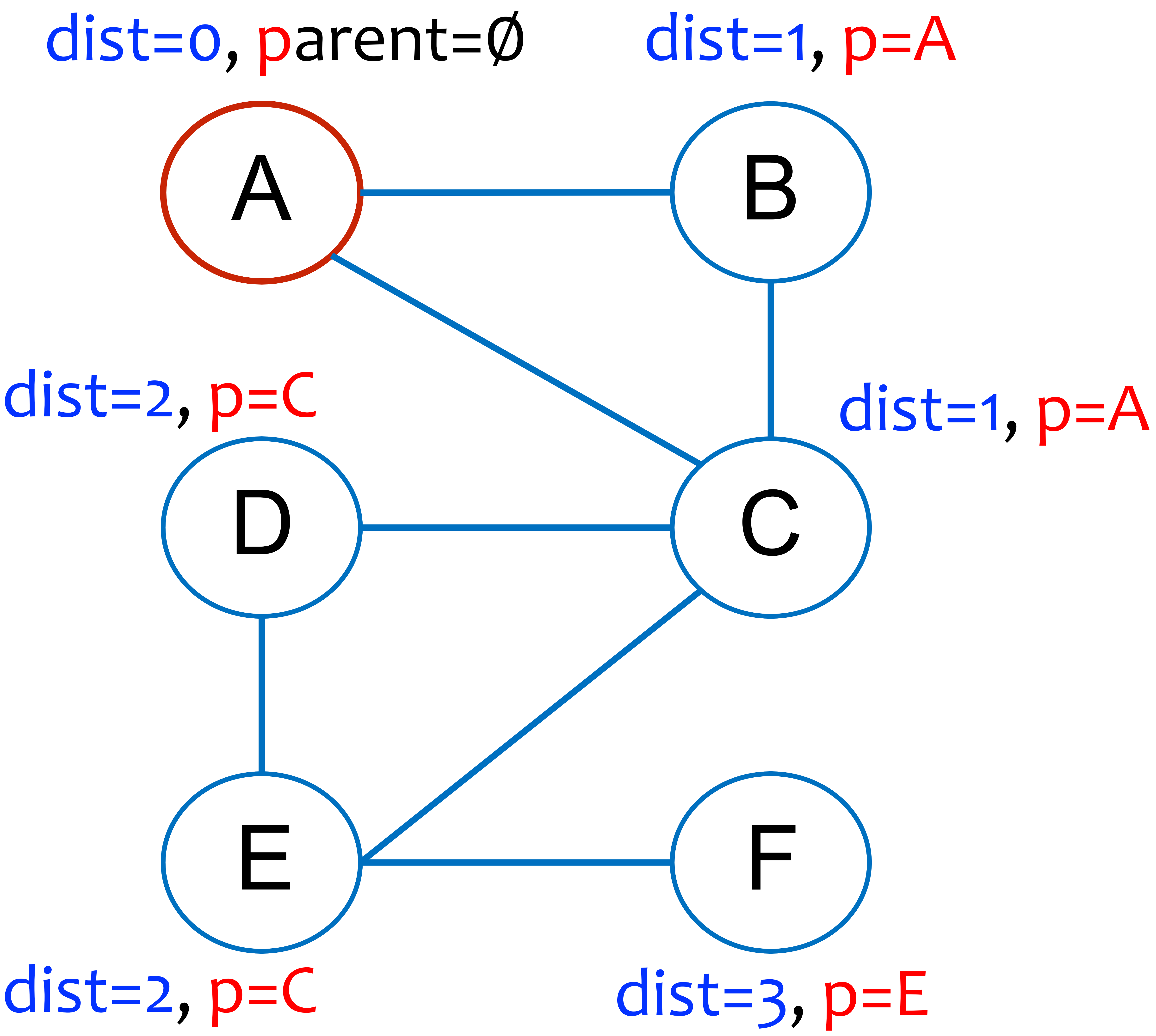


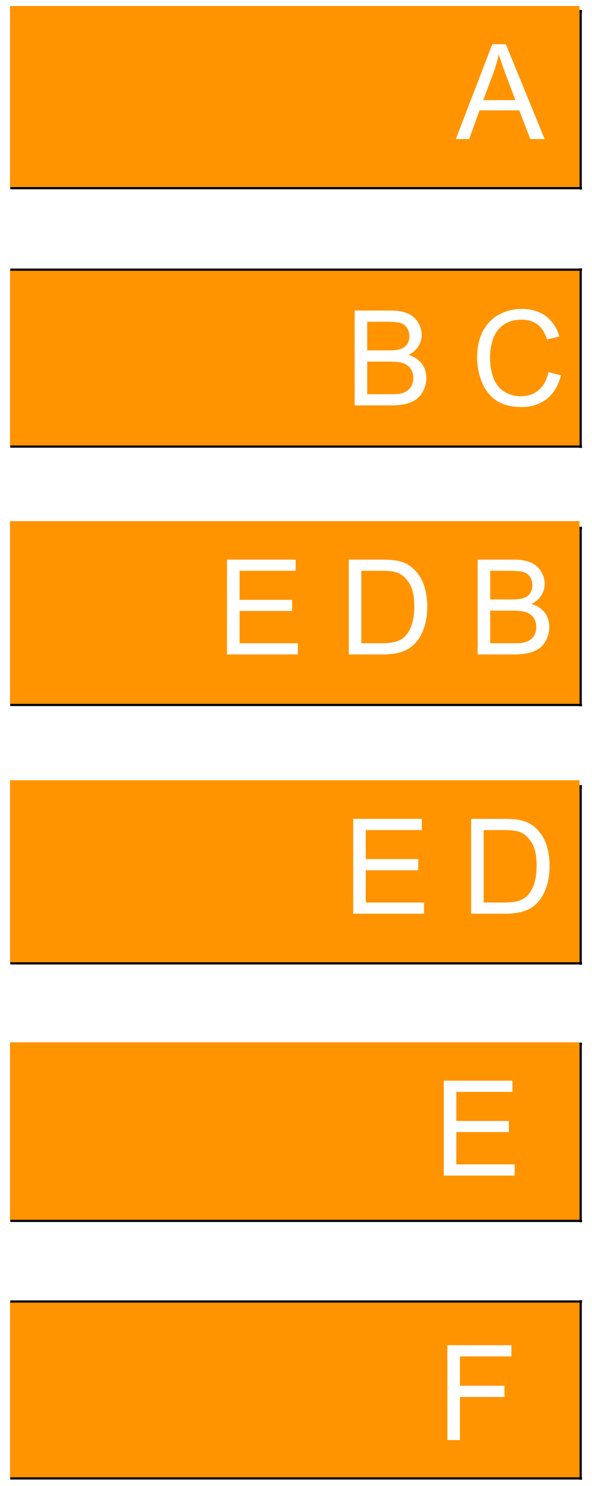
BFS



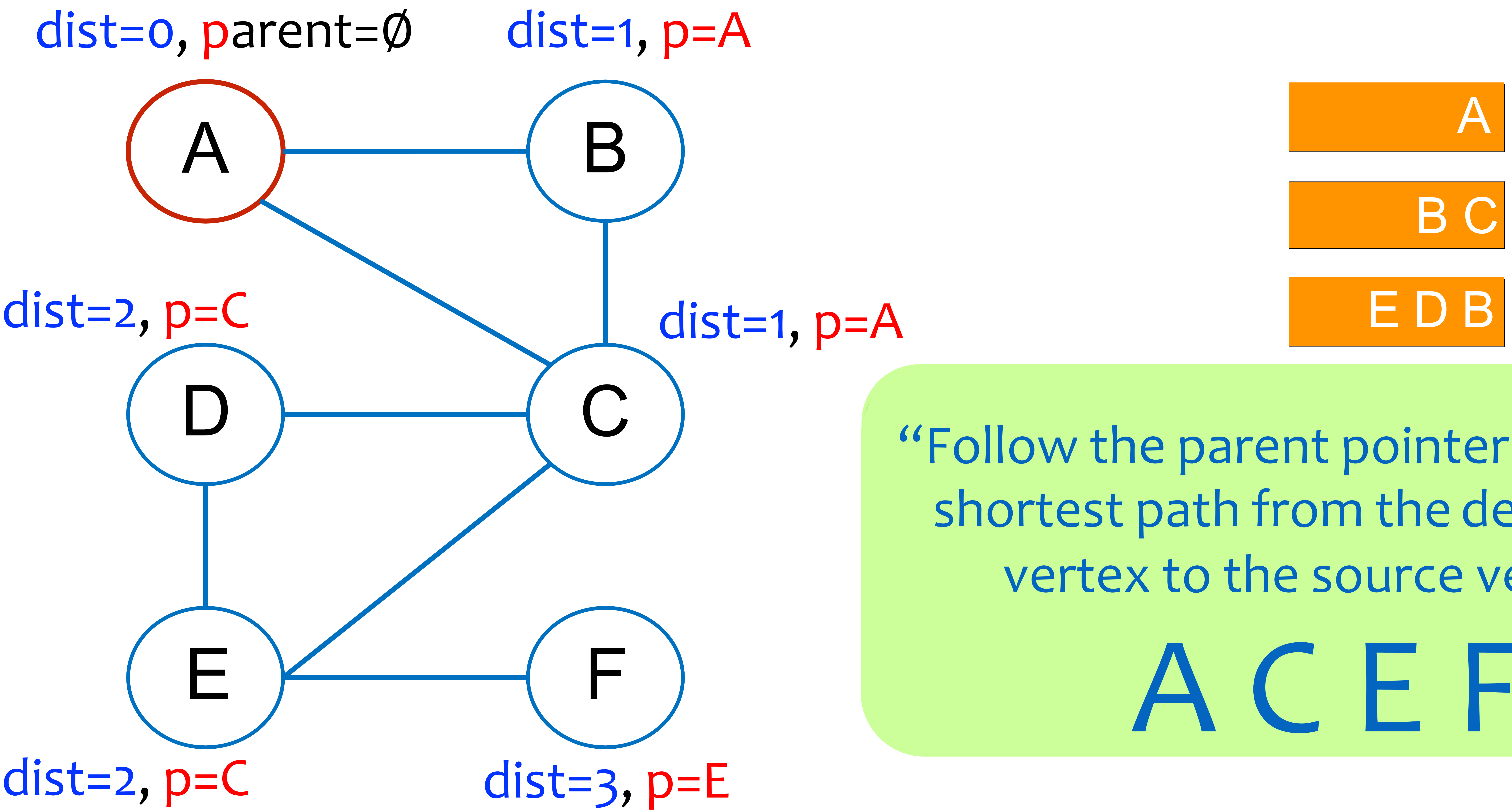
dist=0, parent=∅
dist=1, p=A
A
B
dist=2, p=C
dist=1, p=A
D
C
E
F
dist=2, p=C
A
B C
E D B

BFS


dist=0, parent=∅
dist=1, p=A
A
B
dist=2, p=C
dist=1, p=A
D
C
E
F
dist=2, p=C
dist=3, p=E


A
B C
E D B
E D
E
F

BFS



“Follow the parent pointer to find the shortest path from the destination vertex to the source vertex”

A C E F

Weighted Graphs

- Each edge is labelled with a numerical weight
  - Distances between nodes
  - Cost of moving from one node to another
  - Signal strength between two wireless nodes

How do we represent weighted graphs?

Weighted Graphs - Representation

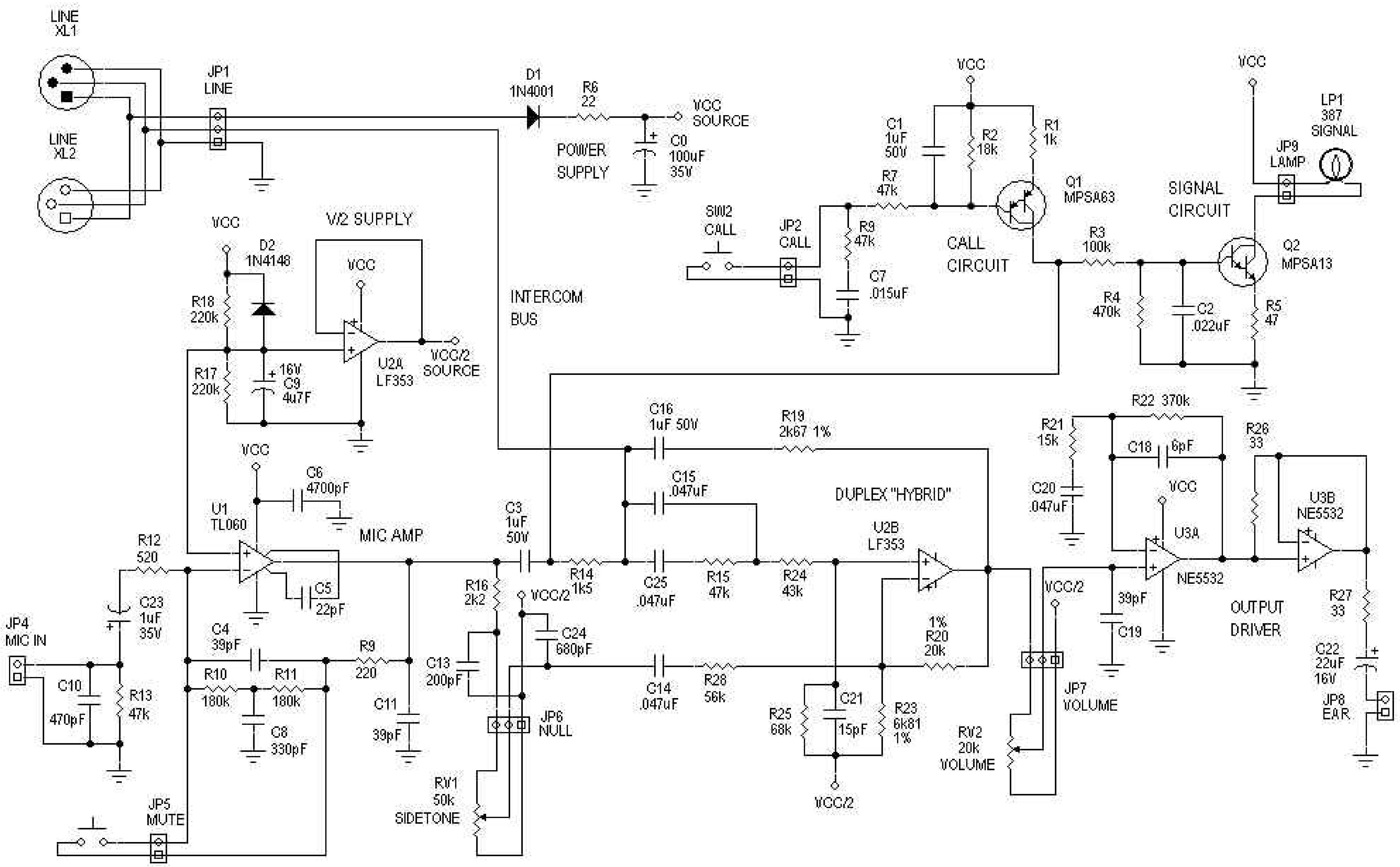
- Adjacency Matrix
  - 2D array of ints/double/…
  - But… how do we represent the absence of an edge?
    - 2D array of structs having one field for edge connectivity and another for edge weights

	0	1	2	3
0	1	3	9	4
1	0	4	7	4
2	2	-5	4	-5
3	1	4	2	-9

- Adjacency List
  - Each list node contains a weight variable

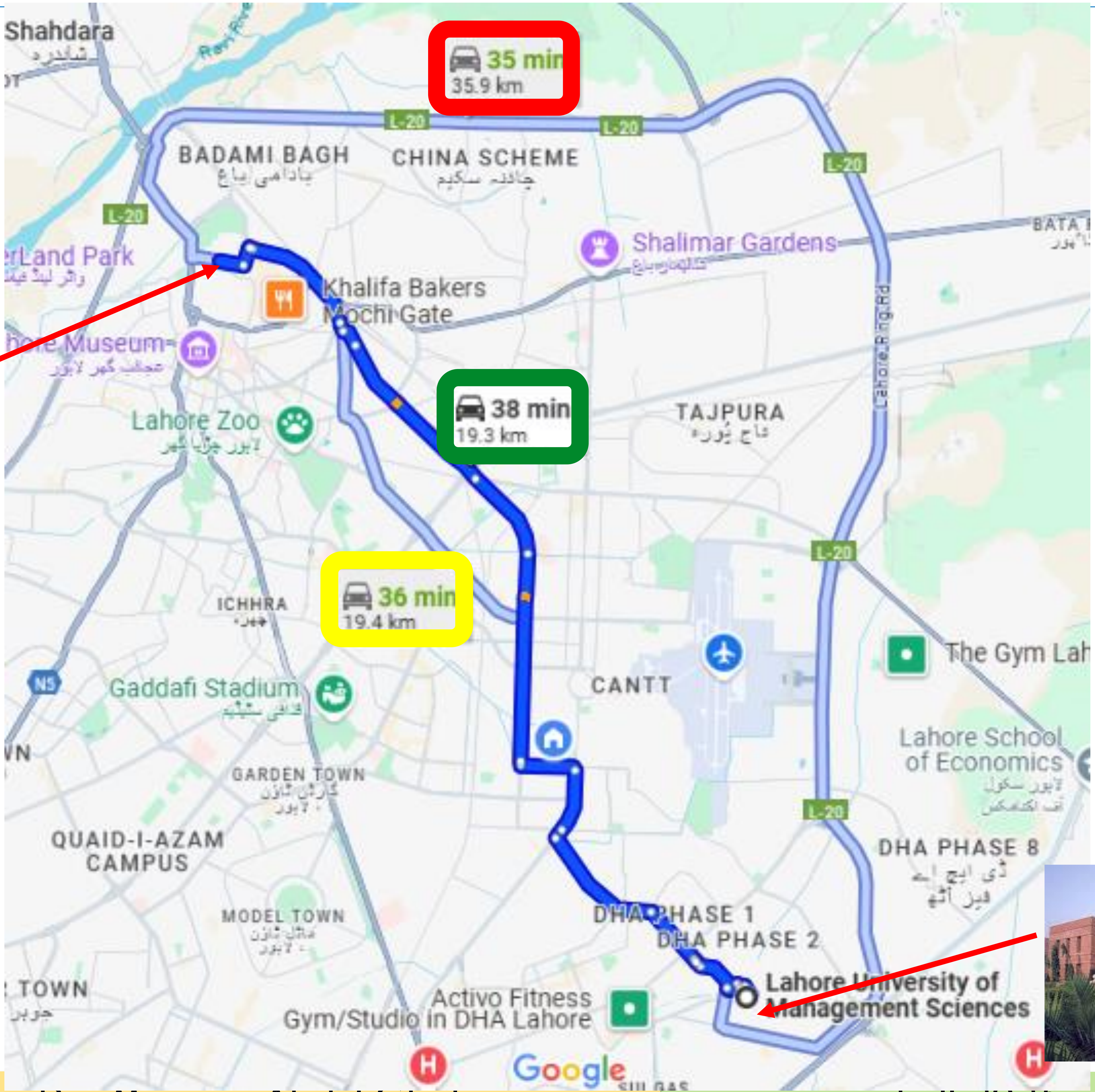
Circuit Design

- Signal propagation delay depends on path length between components
- Shortest path ensures faster response times in critical circuits
- Used in timing analysis to optimize logic gate placement



Driving Directions

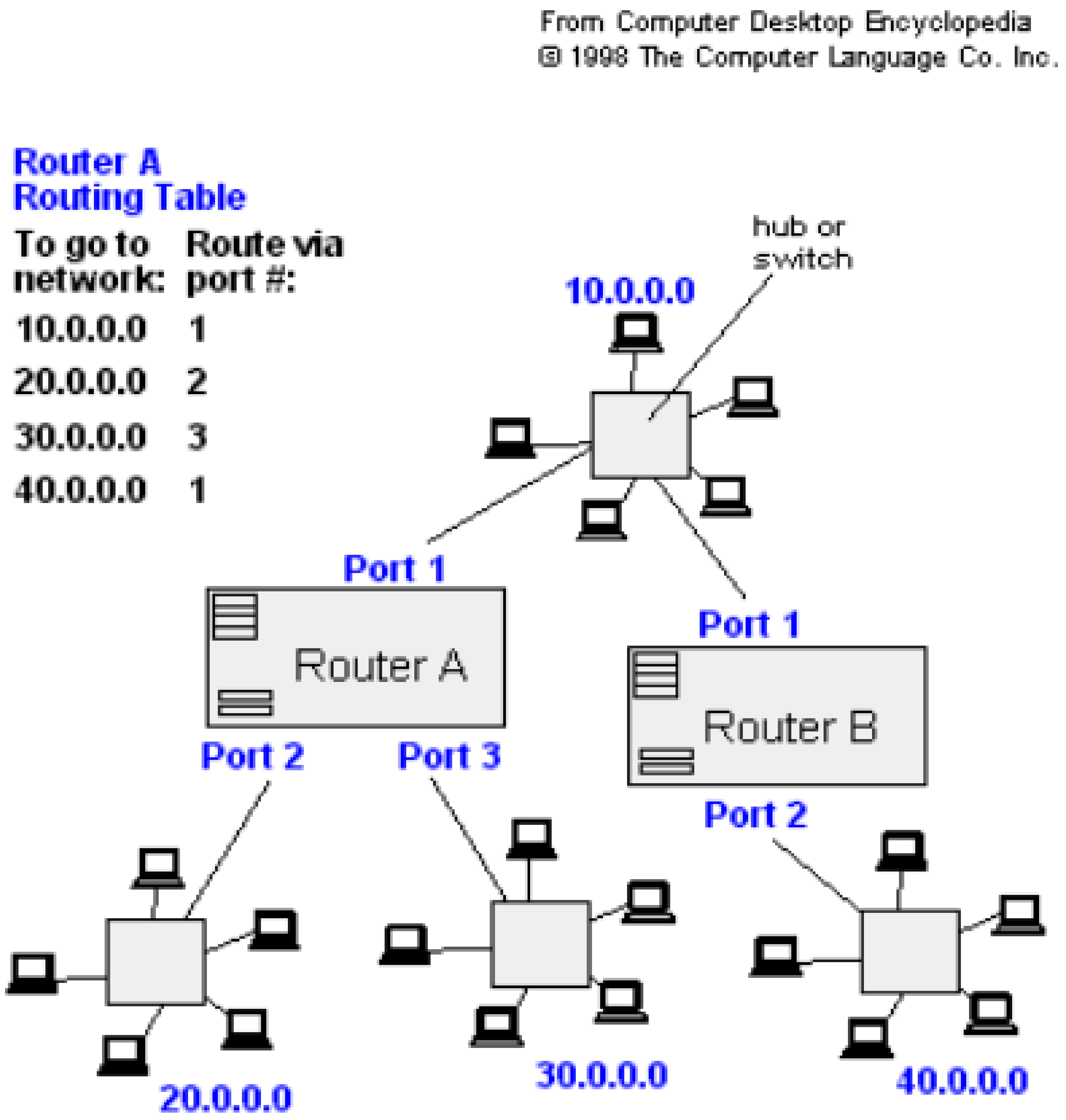
- Google Maps



Routing Data

Routers use graph algorithms to find the fastest path for data packets

Edges represent links with weights like latency or congestion

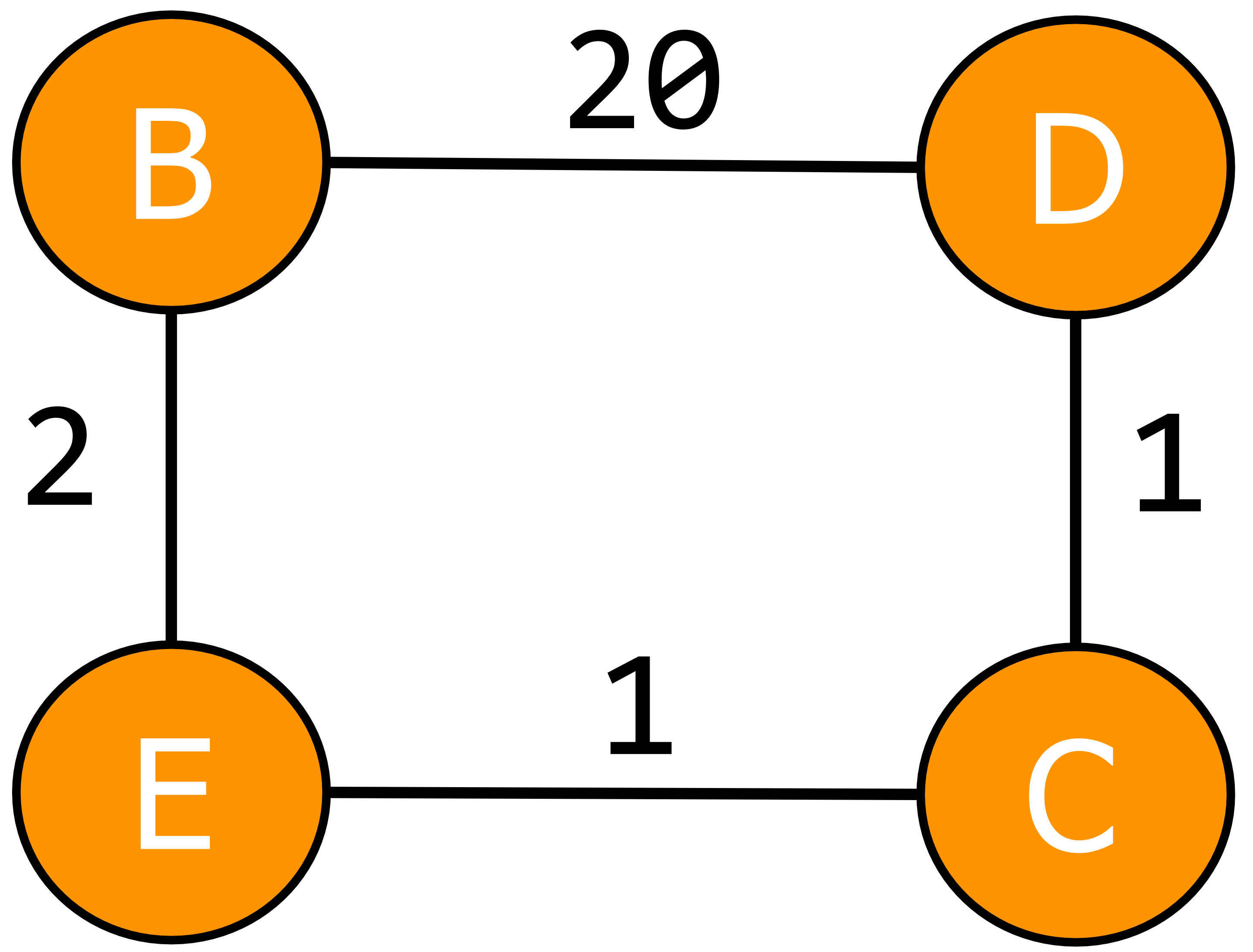


Shortest Path Problem

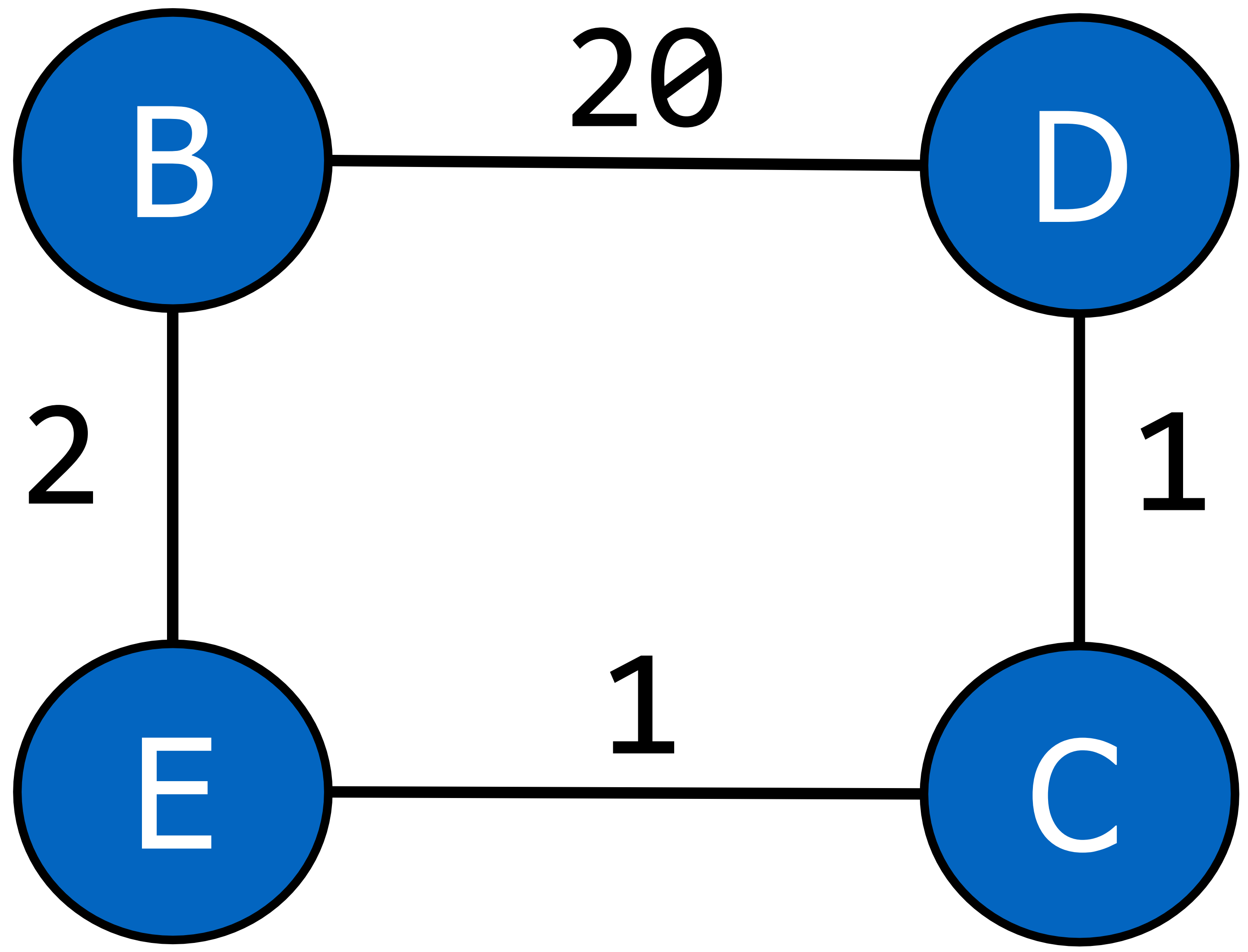
Problem: Given a graph $G = \{V, E\}$ representing different locations, where each edge has an associated cost (weight)

Assumption: The costs are non-negative

Goal: Find the least-cost path from node B to a node D

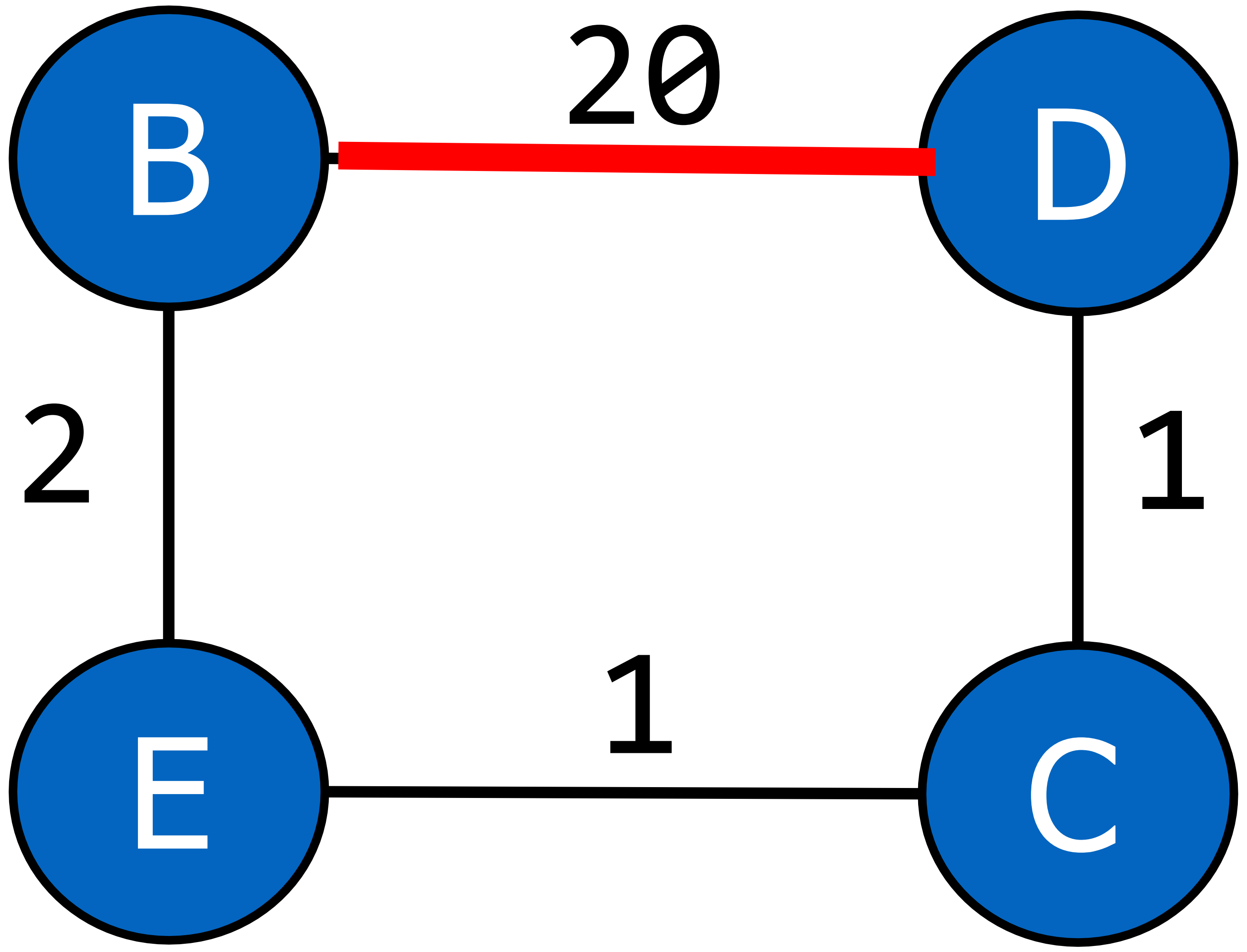


BFS for Shortest Path



Which Path BFS will find?

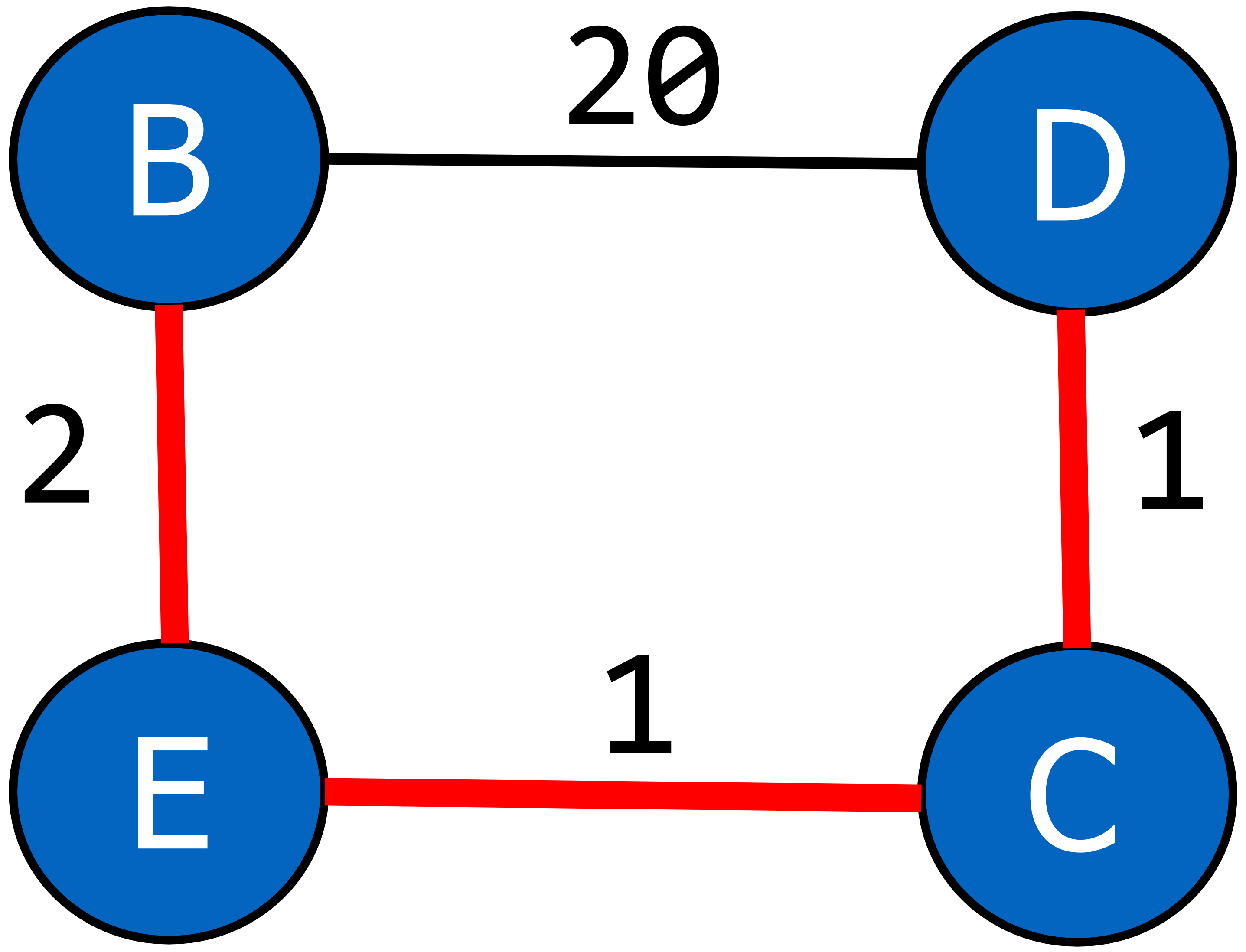
BFS for Shortest Path



BFS finds the shortest path based on the "least number of edges" but that path may have greater cost!

BFS for Shortest Path on Weighted Graph

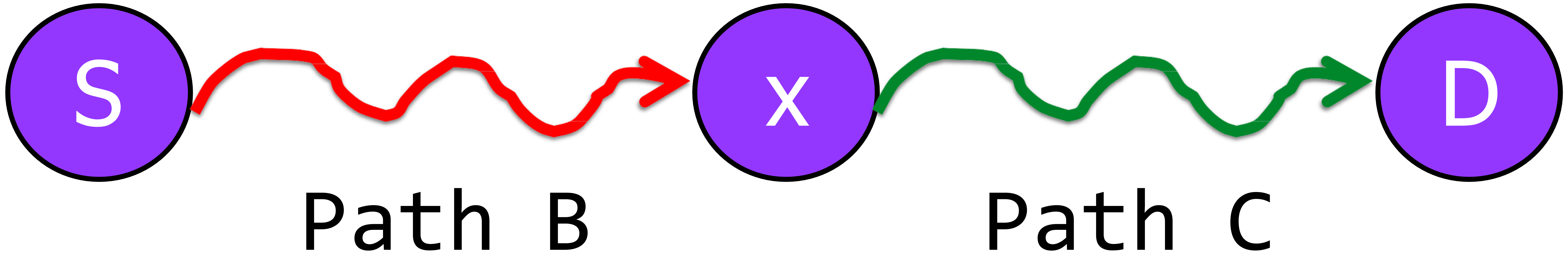
Always pick the next vertex with the least cost!



The greedy choice may not be optimal, need to keep track of other possible paths as they may turn out to be short!

Optimal Substructure Property

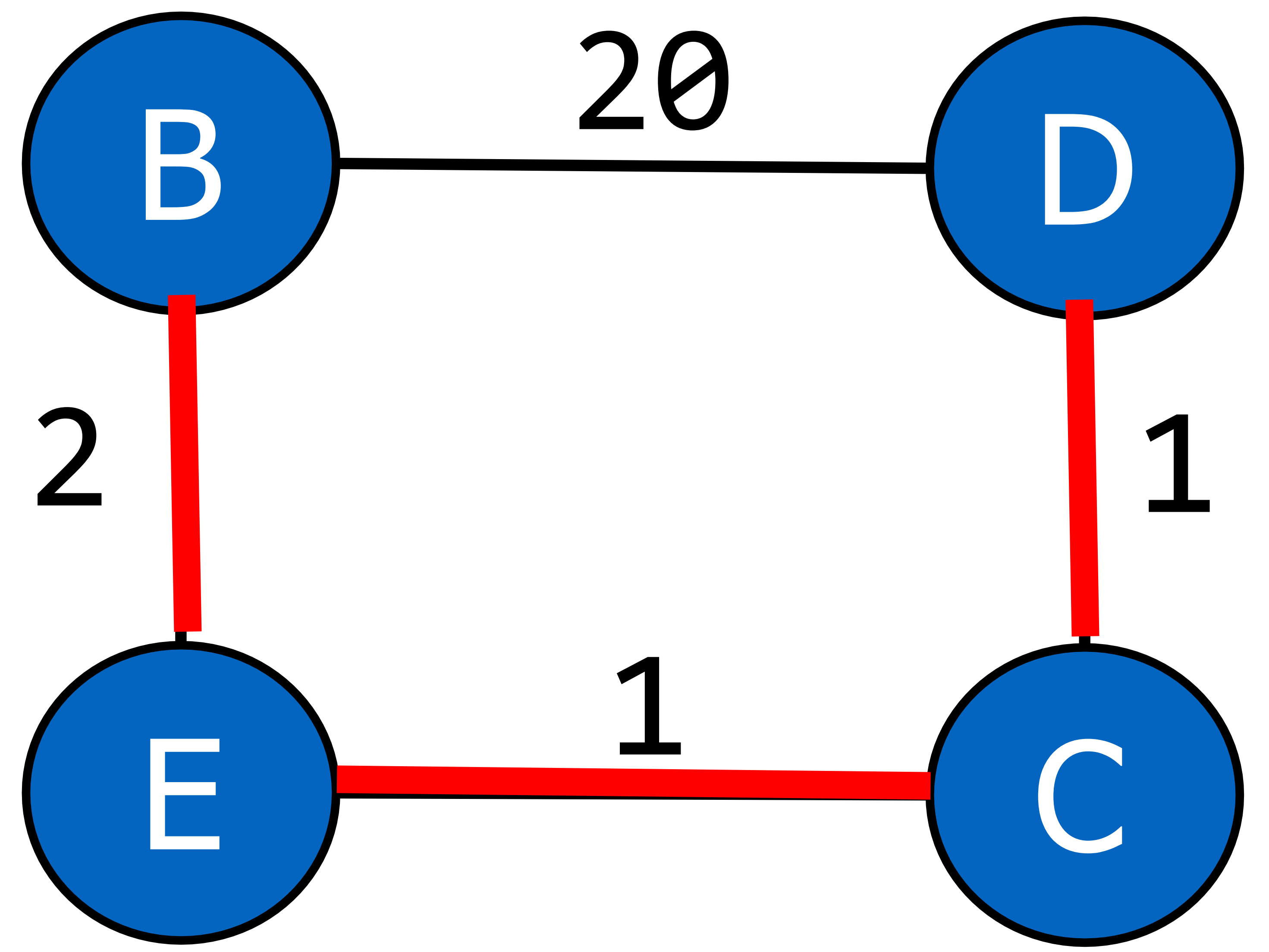
Any sub-path of an optimal path is also optimal



If the path (S,…,x,…,D) is the shortest path from S to D then Path B (S,…,x) is the shortest path from S to x

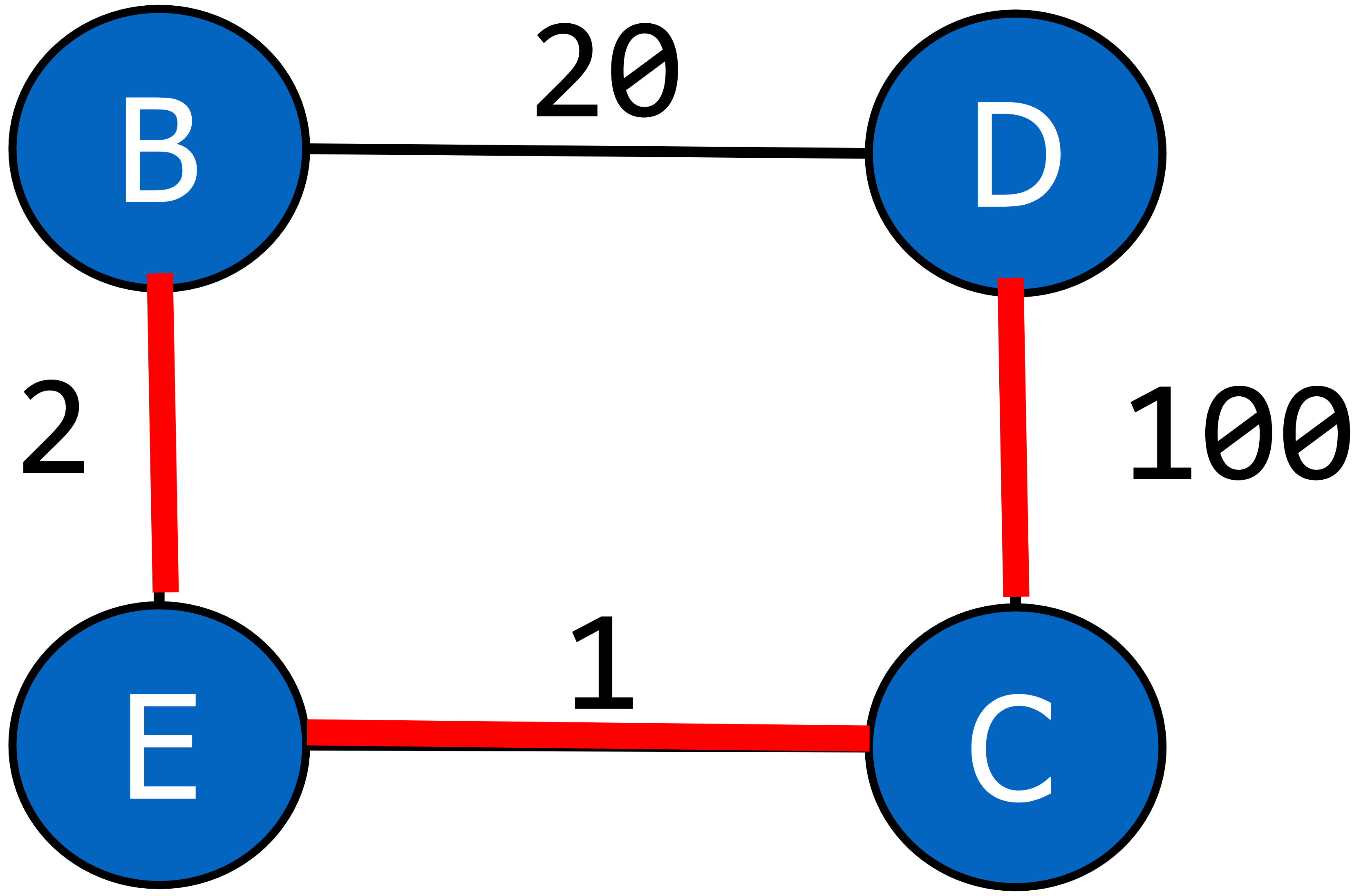
Example – 1

- Examine the shortest path from B to D and all its sub-paths



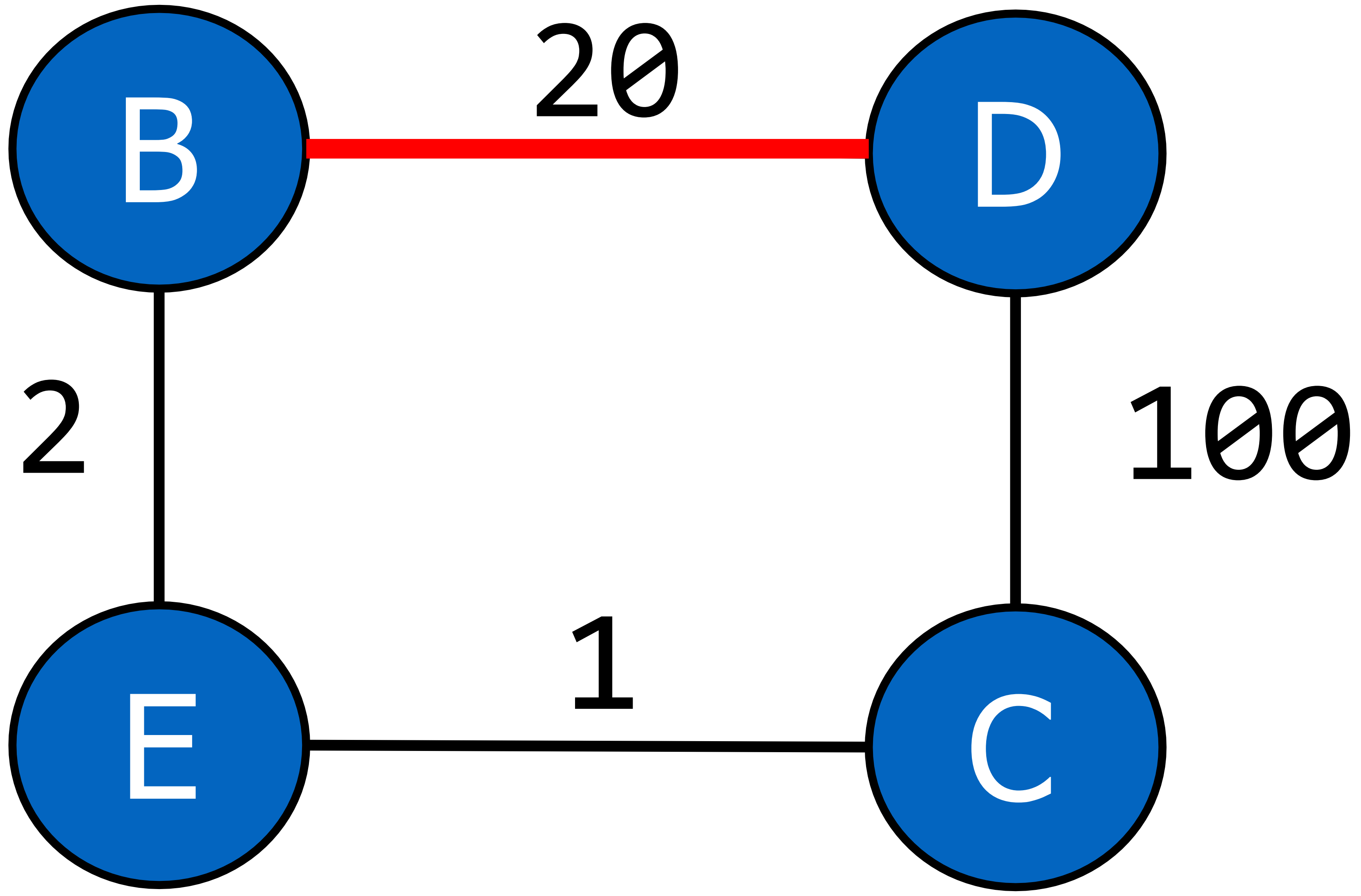
Example – 2

- Examine the shortest path from B to D and all its sub-paths



Example – 2

- Examine the shortest path from B to D and all its sub-paths

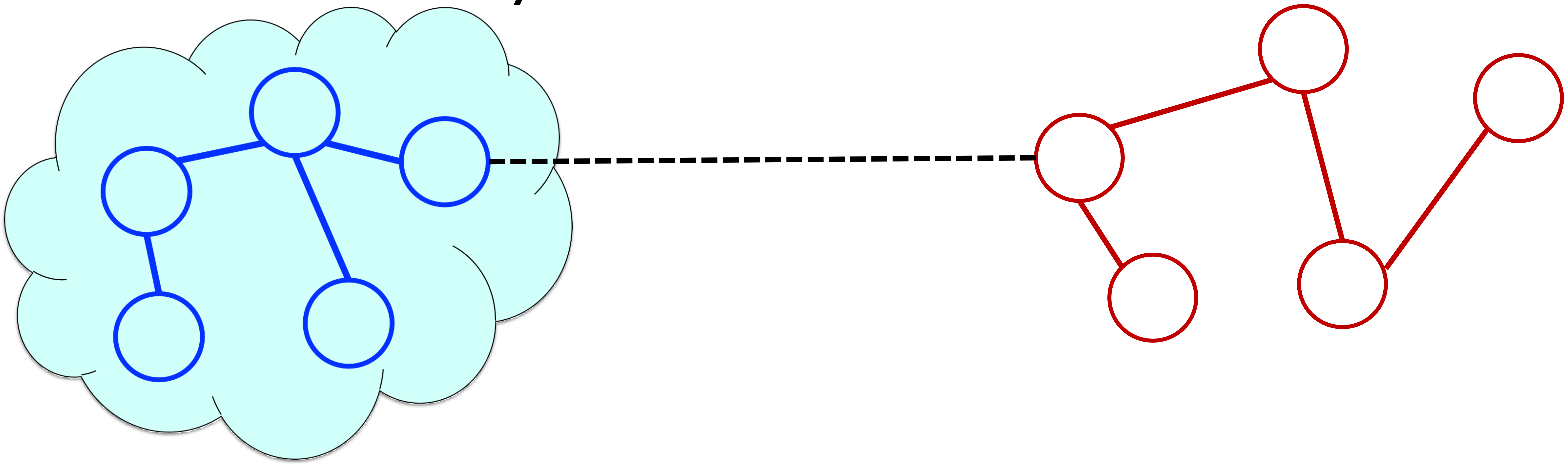


Key Insight

- Adapt BFS to handle weighted graphs (with no negative edge weights) and keep track of learnt paths

Dijkstra’s Algorithm: Big Picture

- Two Kinds of Vertices
  - Vertices inside the cloud
    - Vertices for whom the shortest path is known
- Vertices outside the cloud
  - Vertices for whom only tentative distances are known



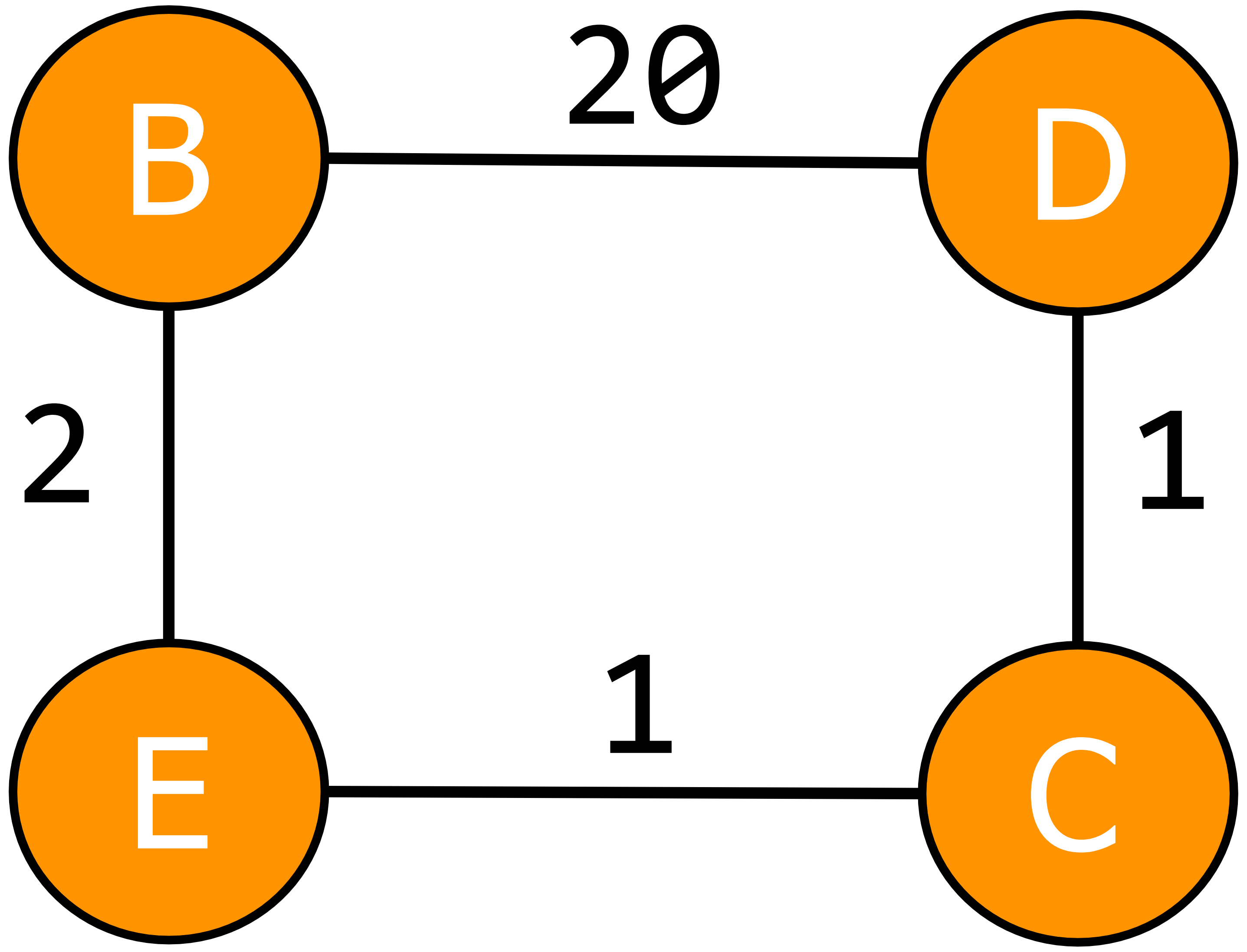
At Each Step/Iteration

- Pick the closest vertex outside the cloud
- Add it to the cloud
- Update distances

- Greedy algorithm: makes choices that currently seem best

Let's see how Dijkstra would work here

- Optimal substructure property + non-negative edge weights + keeping track



Goal: Find the Shortest Path (SP) from B to D

To find the SP from B to D, we need to find the SP from B to C

To find the SP from B to C, we need to find the SP from B to E

So, let's start by find the SP from B to any of the adjacent vertices!

Initialization and Update

- The shortest path from the starting vertex v to v, D[v]=0
- If all edge weights are positive, then the least cost edge incident to v, say (v, u), defines D[u]
- When we establish the shortest path from v to a new node u, we go through each of its incident edges to see if there is a better way from v to other nodes through u

More Details

D[u]: Stores the length of the best path found so far from v to u

Initially $D[v]=0$, $D[u]=\infty$ for all $u \neq v$

Define the set $C=\emptyset$ (called cloud)

At each iteration

1. Select vertex u not in C with the smallest D[u] and pull u into C
2. We update the label D[z] for each vertex z adjacent to u and outside of C. This update is called 'Edge Relaxation'

Building Block – Initialize Single Source

Initially D[v]=0, D[u]=∞ for all u≠v

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$

Building Block – Relax Edge

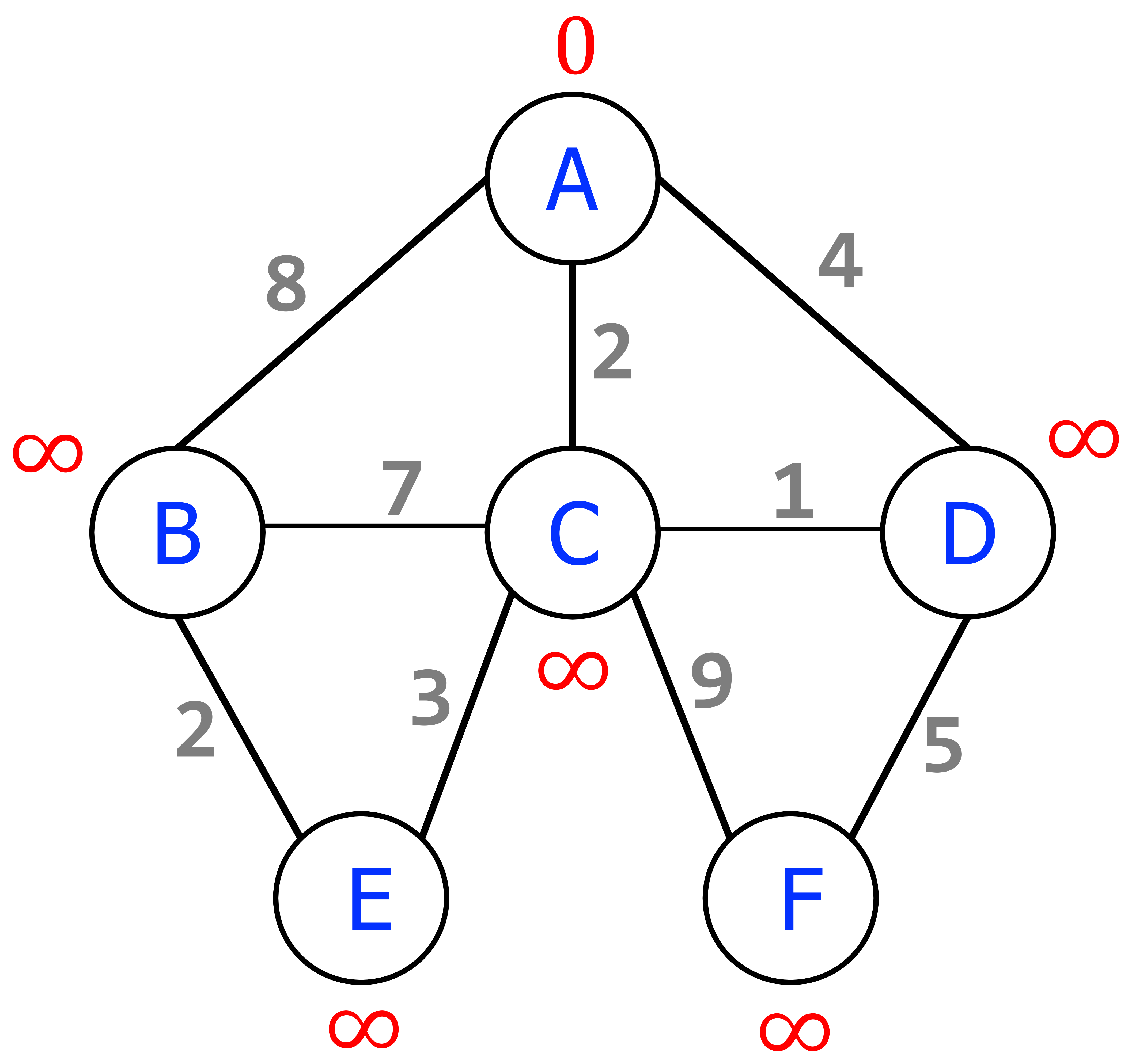
At each iteration

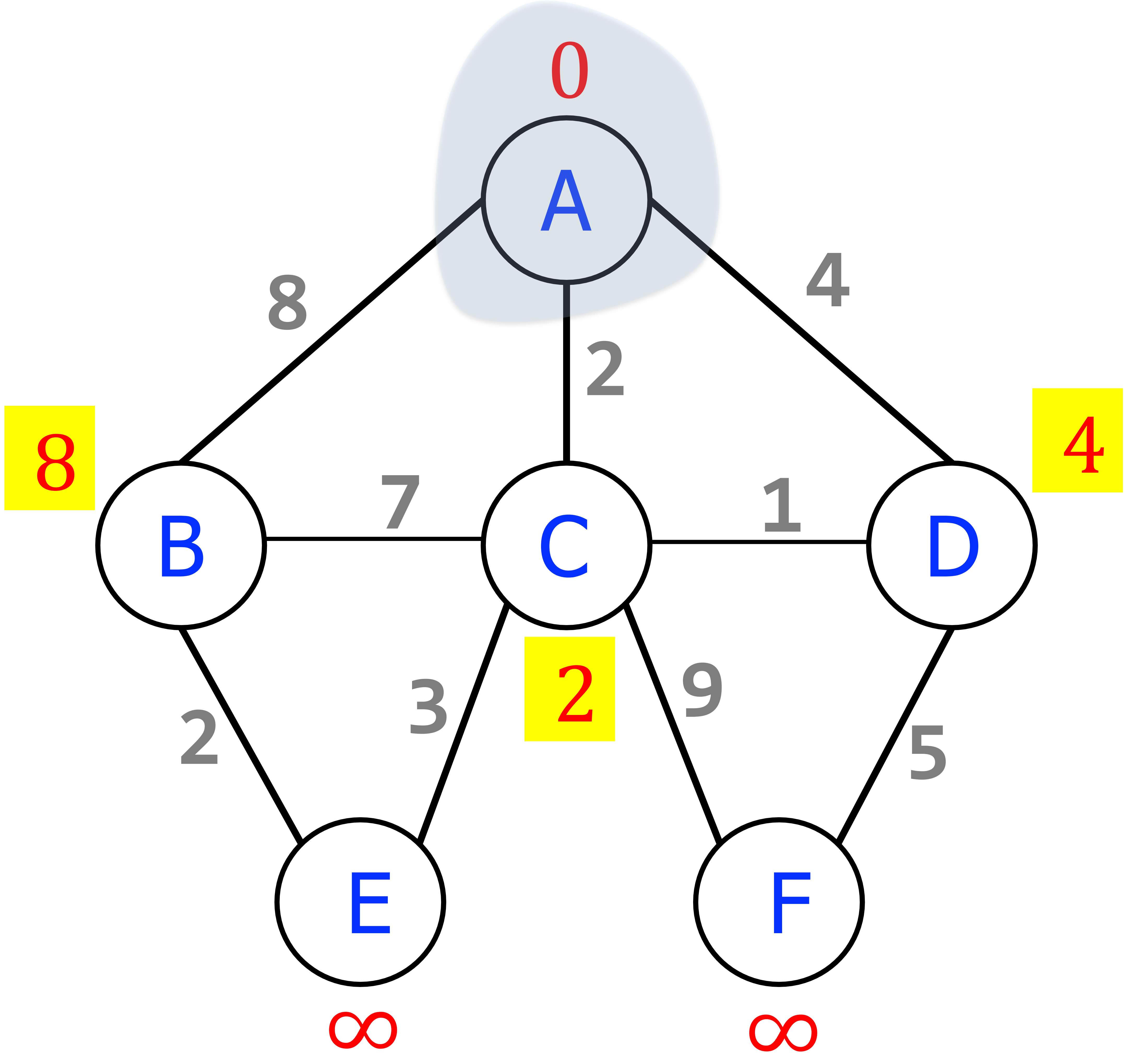
1. Select vertex u not in C with the smallest D[u] and pull u into C
2. We update the label D[z] for each vertex z adjacent to u and outside of C. This update is called 'Edge Relaxation'

RELAX(u, v, w)

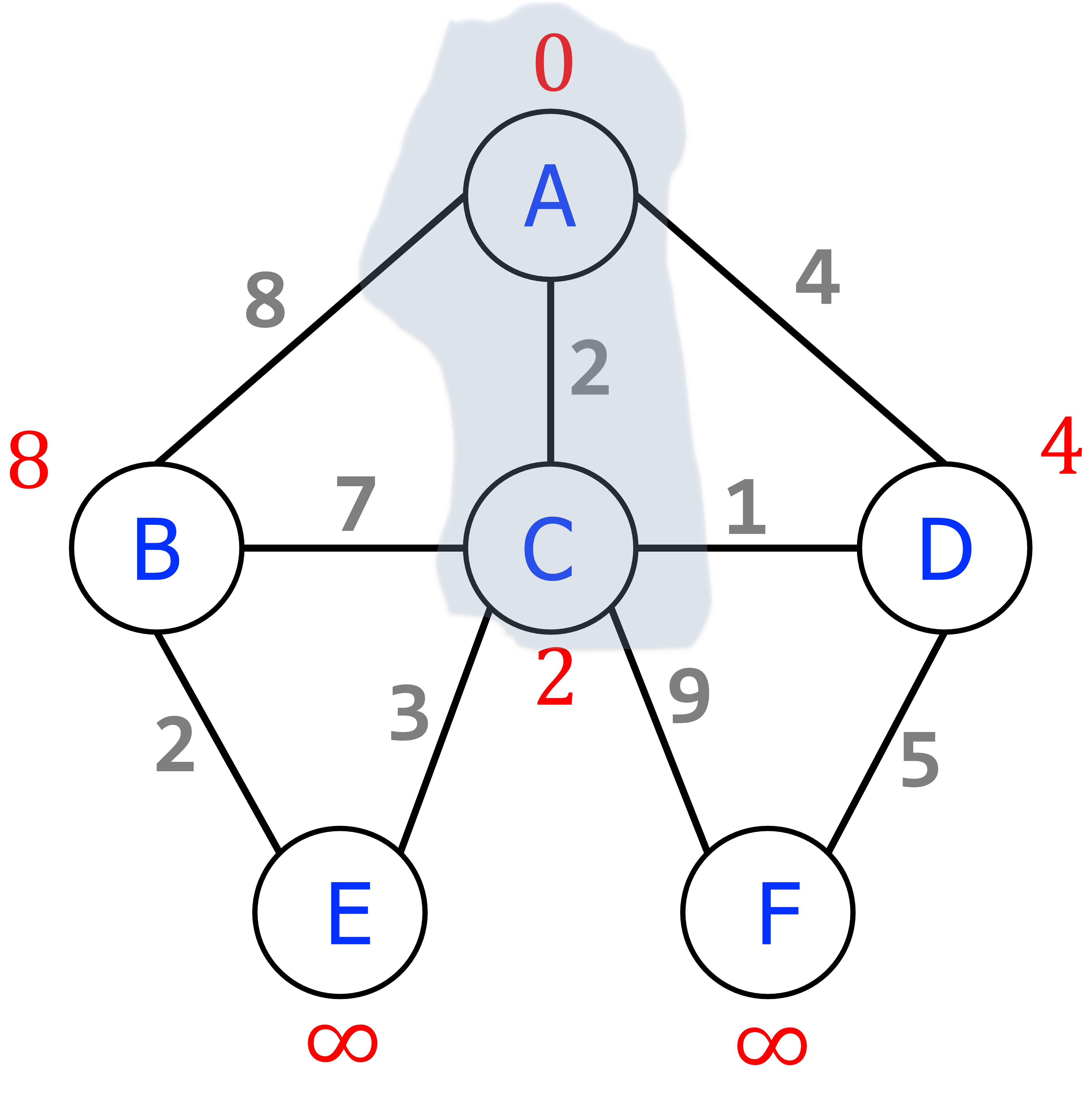
```
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

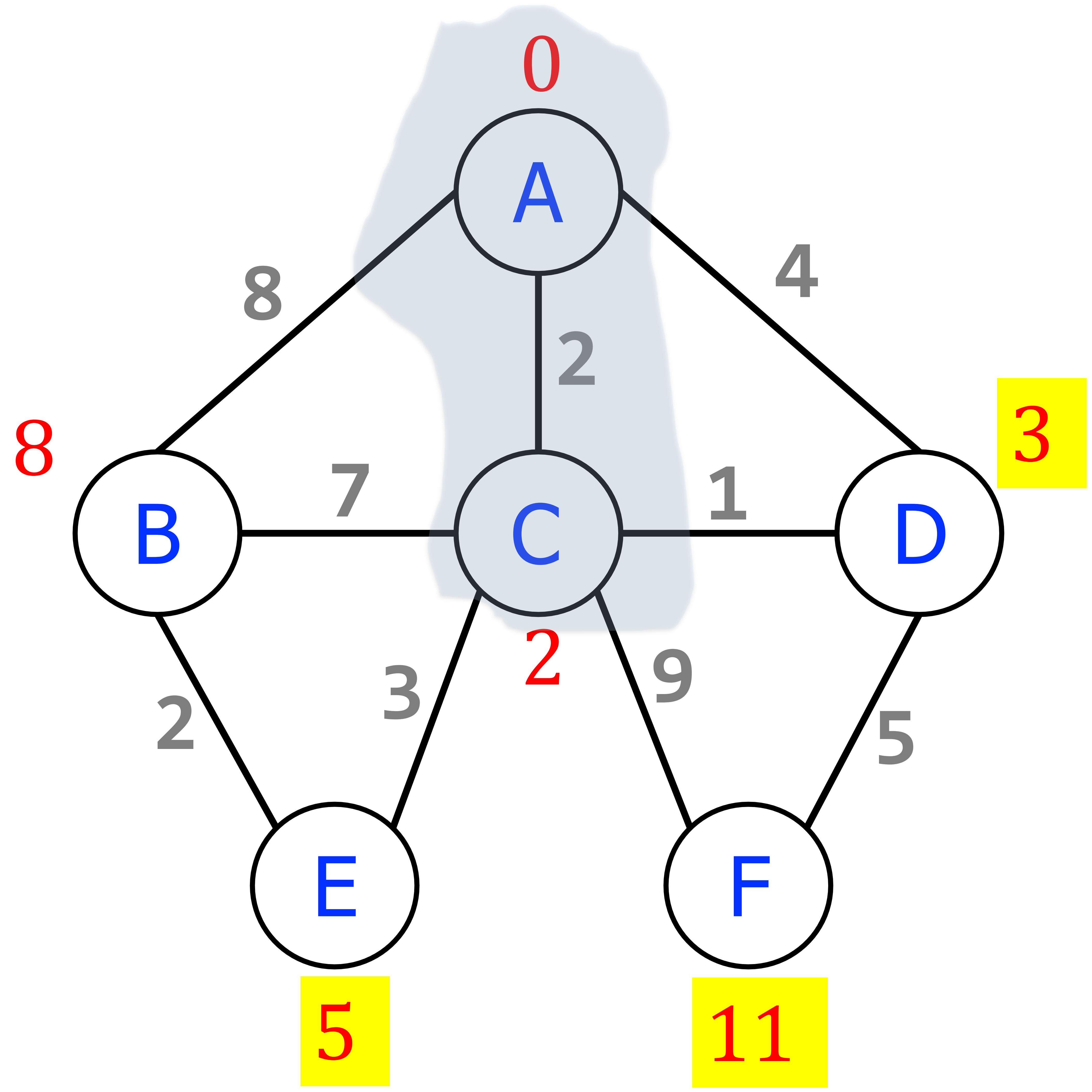
Dijkstra's Algorithm Example



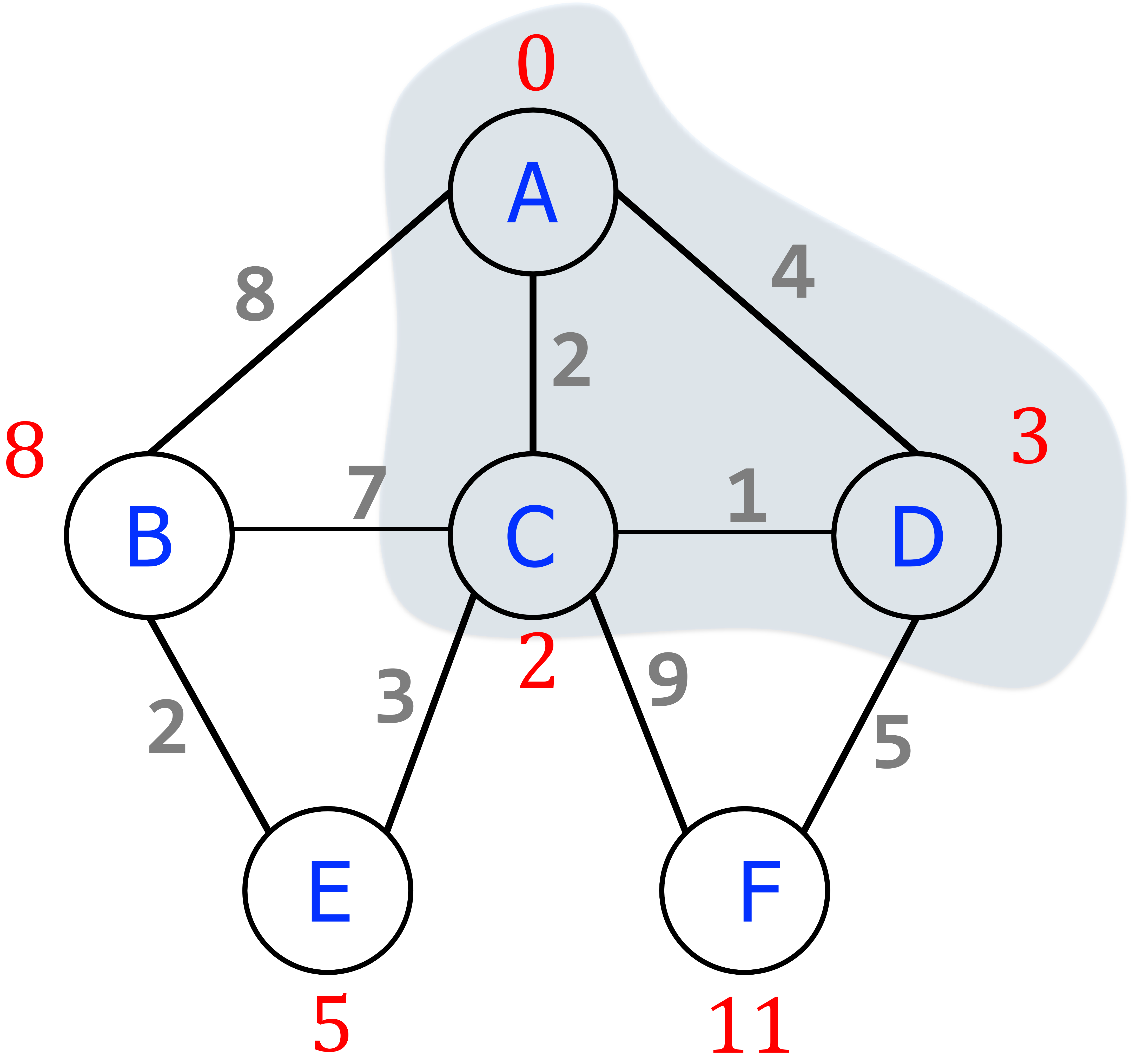


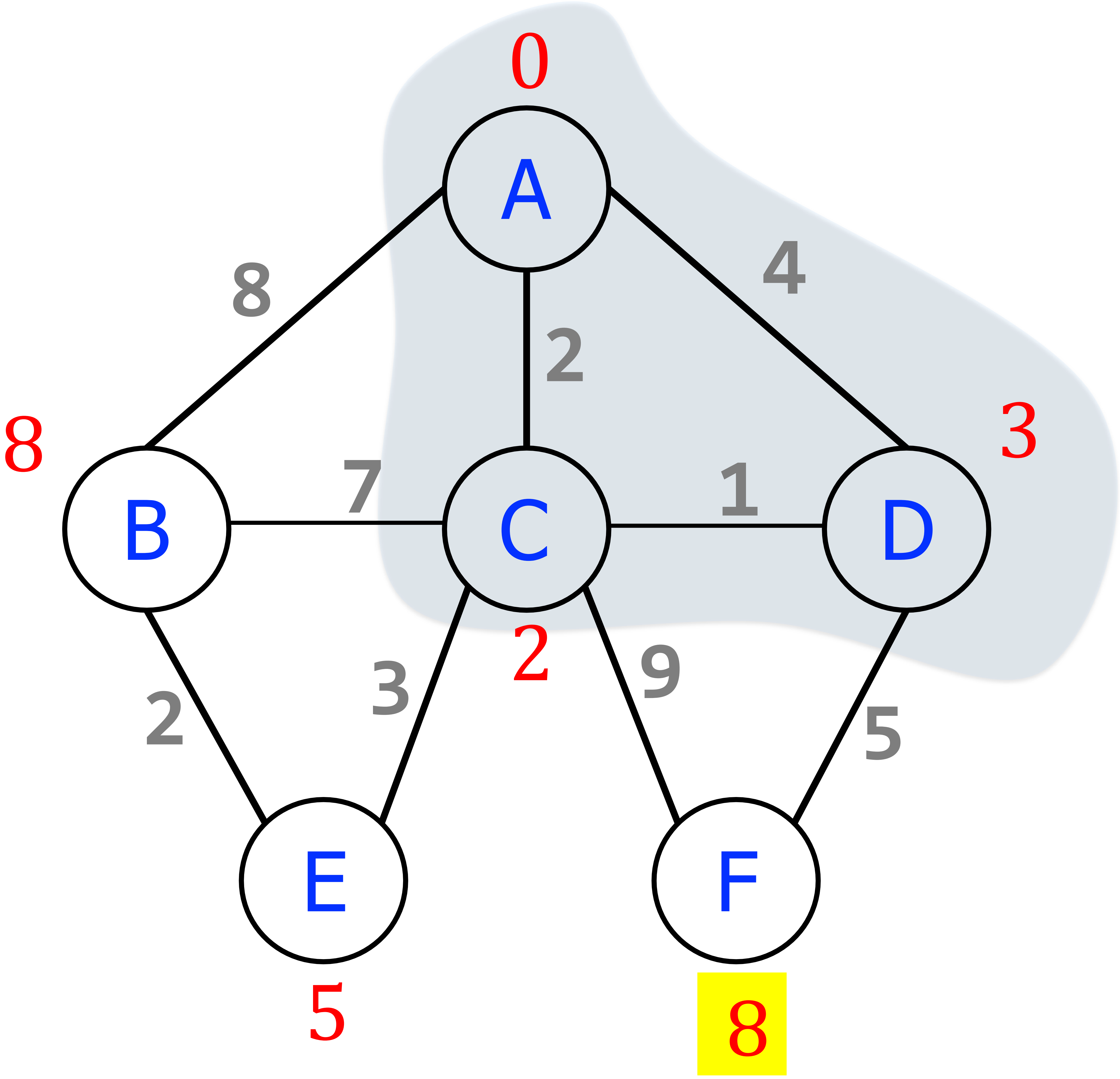
Dijkstra's Algorithm Example



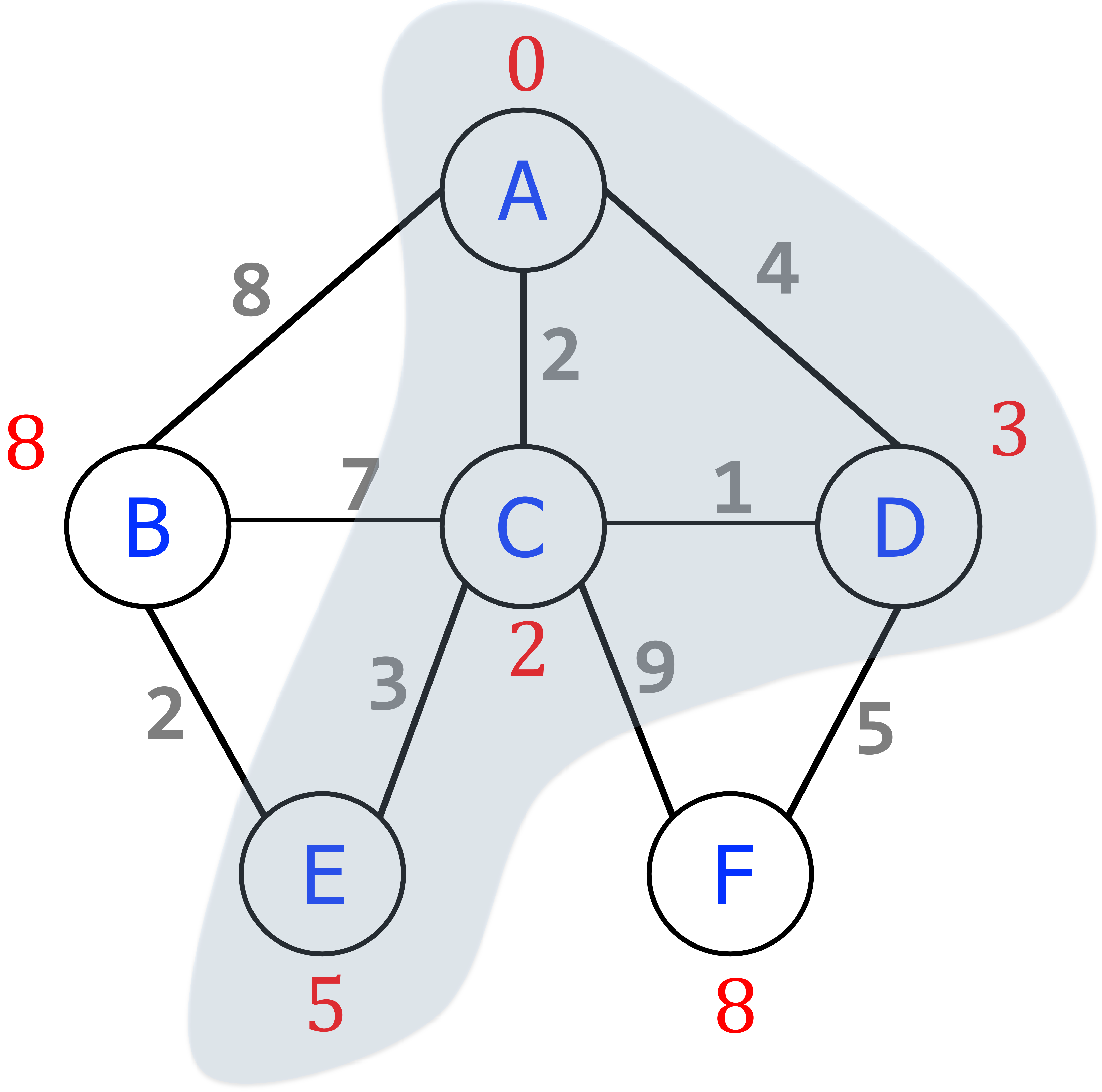


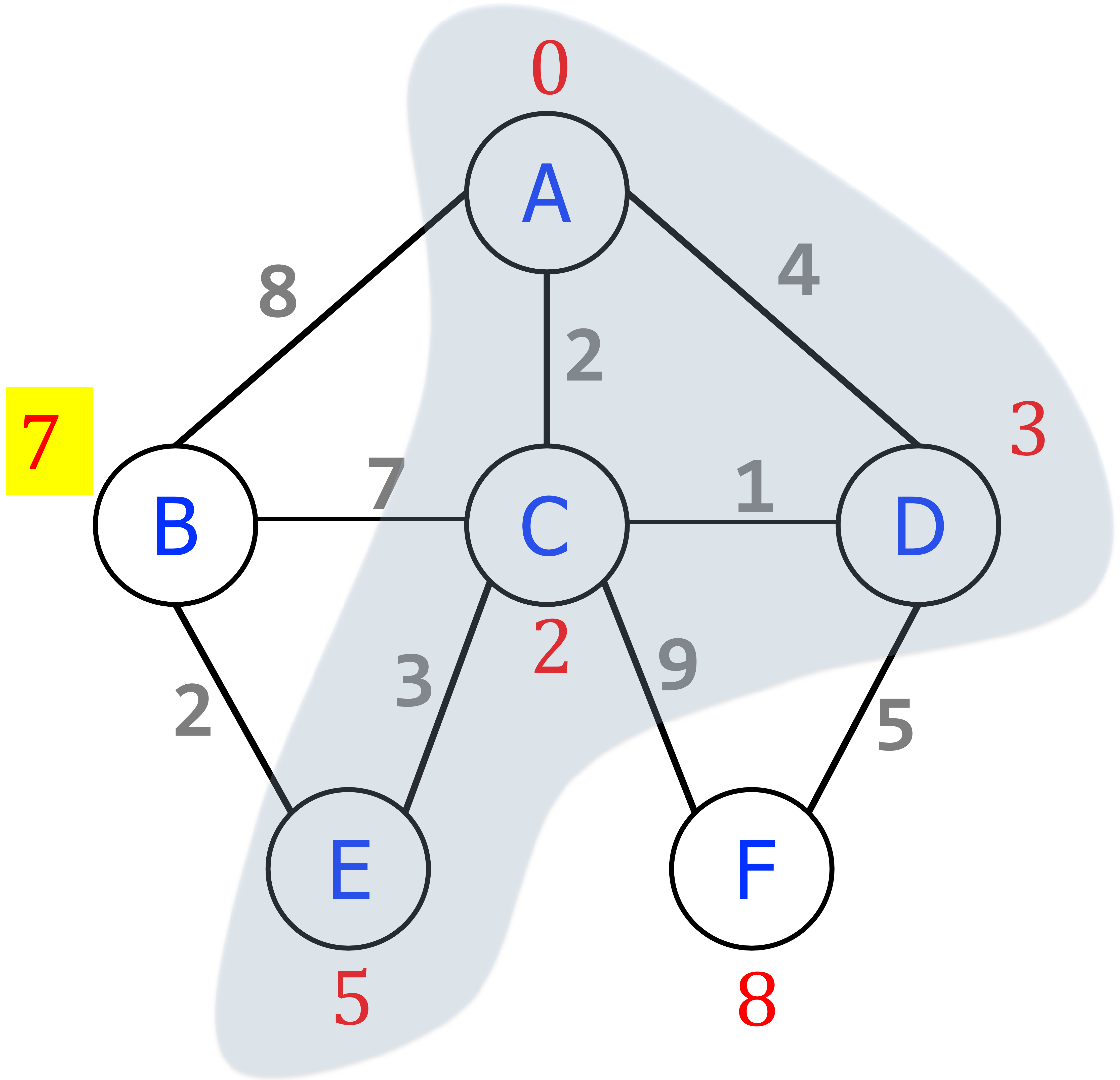
Dijkstra's Algorithm Example



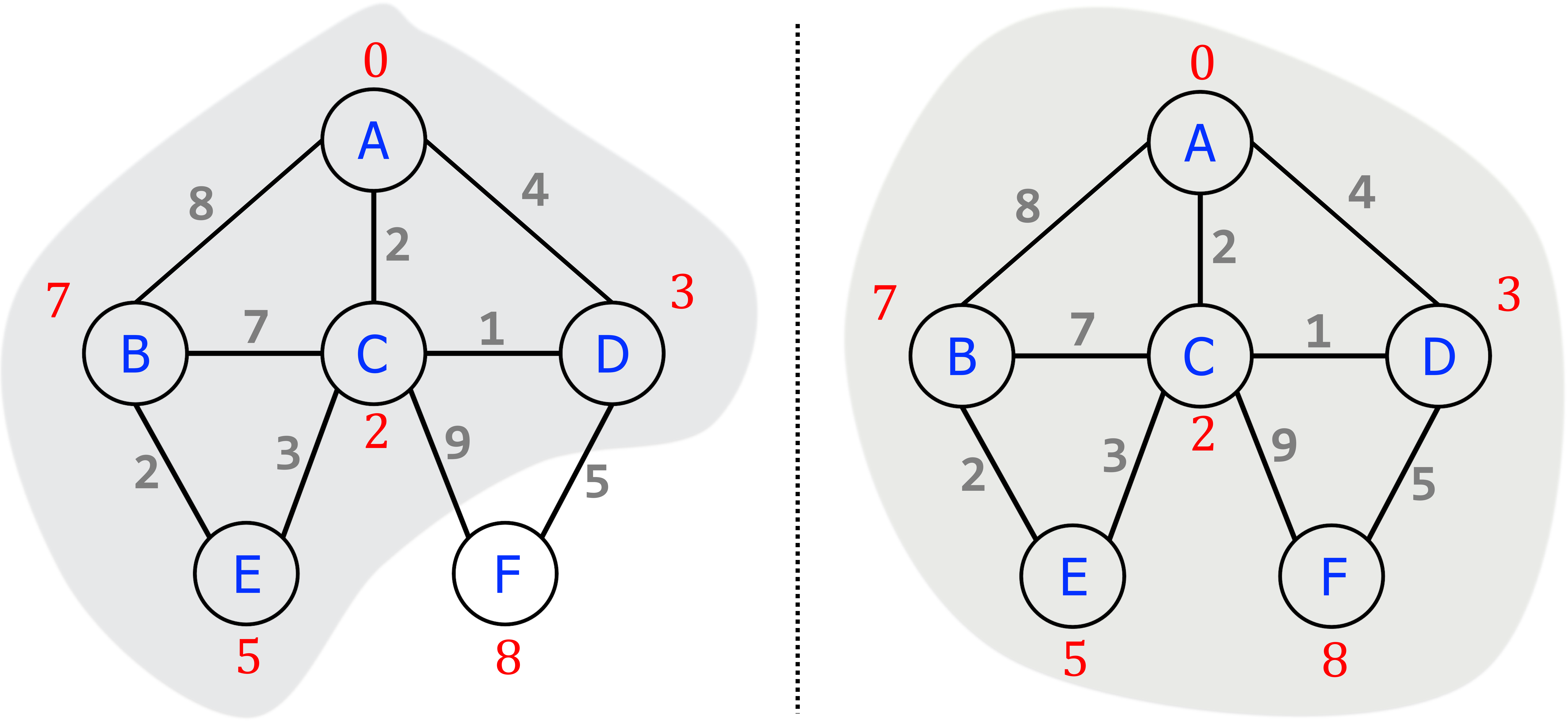


Dijkstra's Algorithm Example

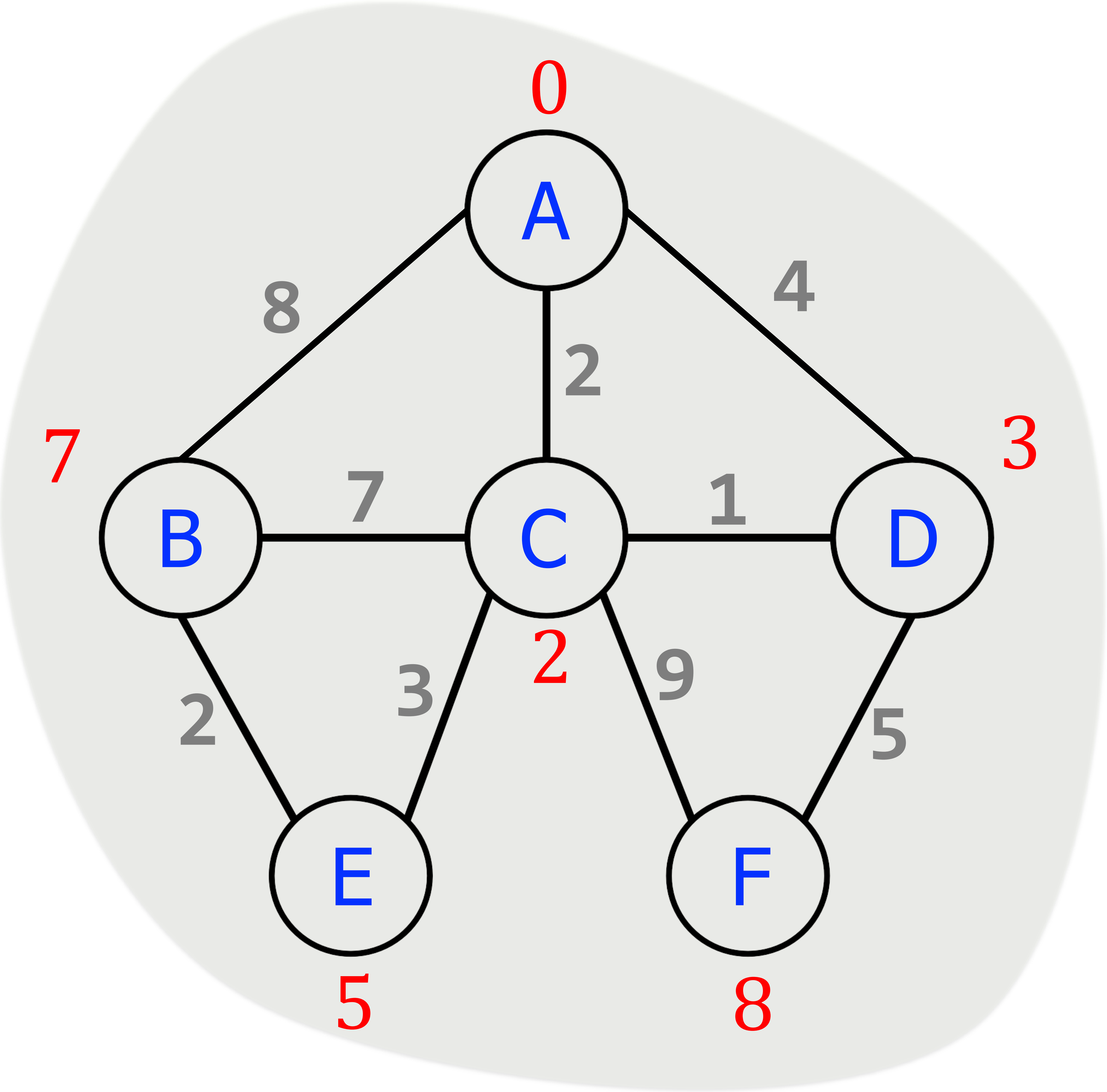




Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



Path	Shortest Path & Cost
A → B	A → C → E → B | 7
A → C	A → C | 2
A → D	A → C → D | 3
A → E	A → C → E | 5
A → F	A → C → D → F | 8

Pseudocode & Efficiency

```
dijkstra(Graph G, Node v) {
  for each vertex:
     D[u]= ∞,
     known[u]=false
  D[v]=0 //source vertex
  build-heap with all vertices //using D[u] as key
  while(heap is not empty) {
    b = removeMin()
    Known[b] = true //moves inside the cloud
    for each edge (b,a) in G {
     if(!a.known) //it is outside the cloud
      if(D[b] + weight((b,a)) < D[a]){
        D[a] = D[b] + weight(b,a)//decreaseKey in heap
        path[a] = b //for computing actual paths
      }
   }
}
```

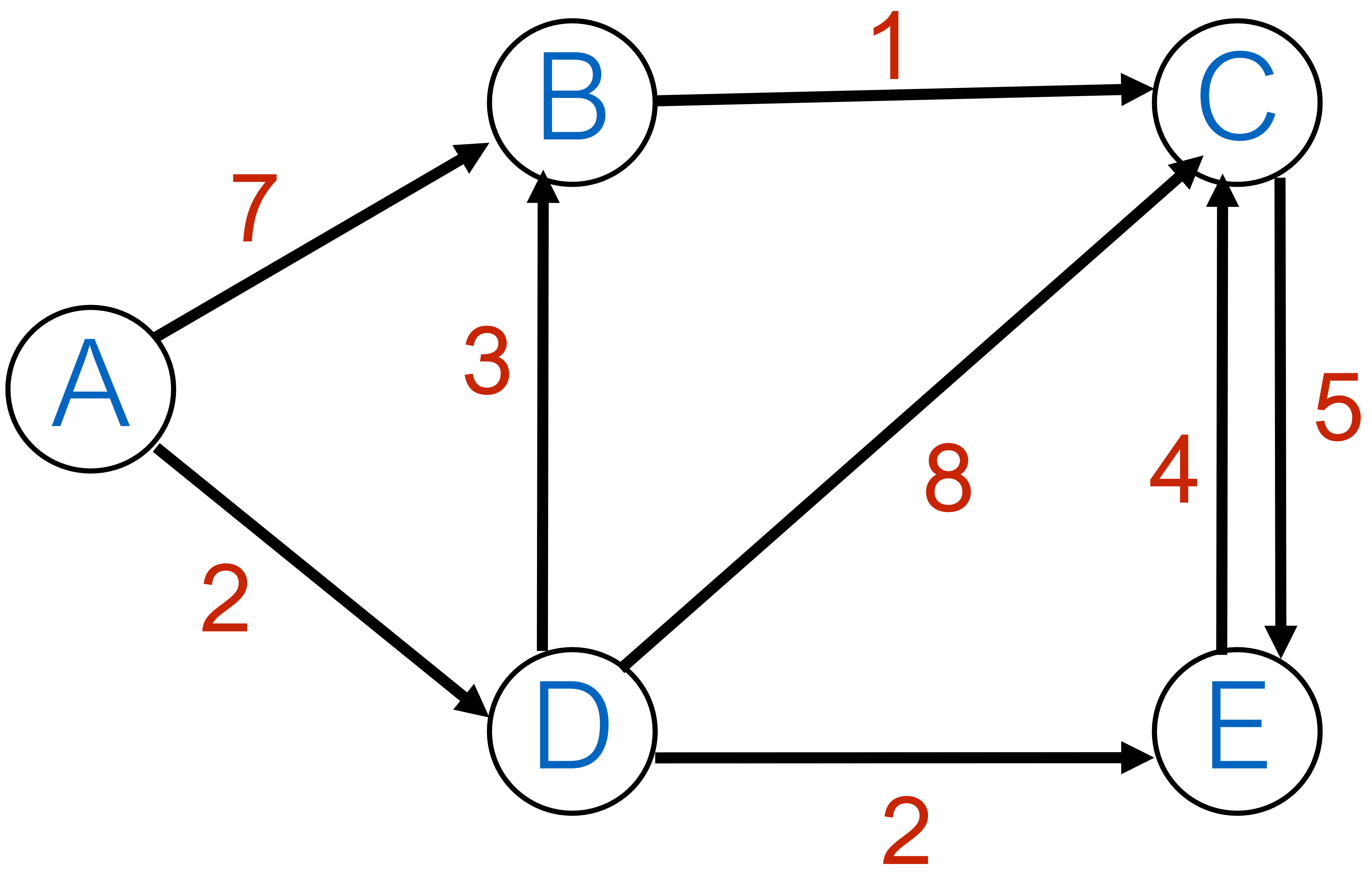
$O(|V|)$

$O(|V|\log|V|)$

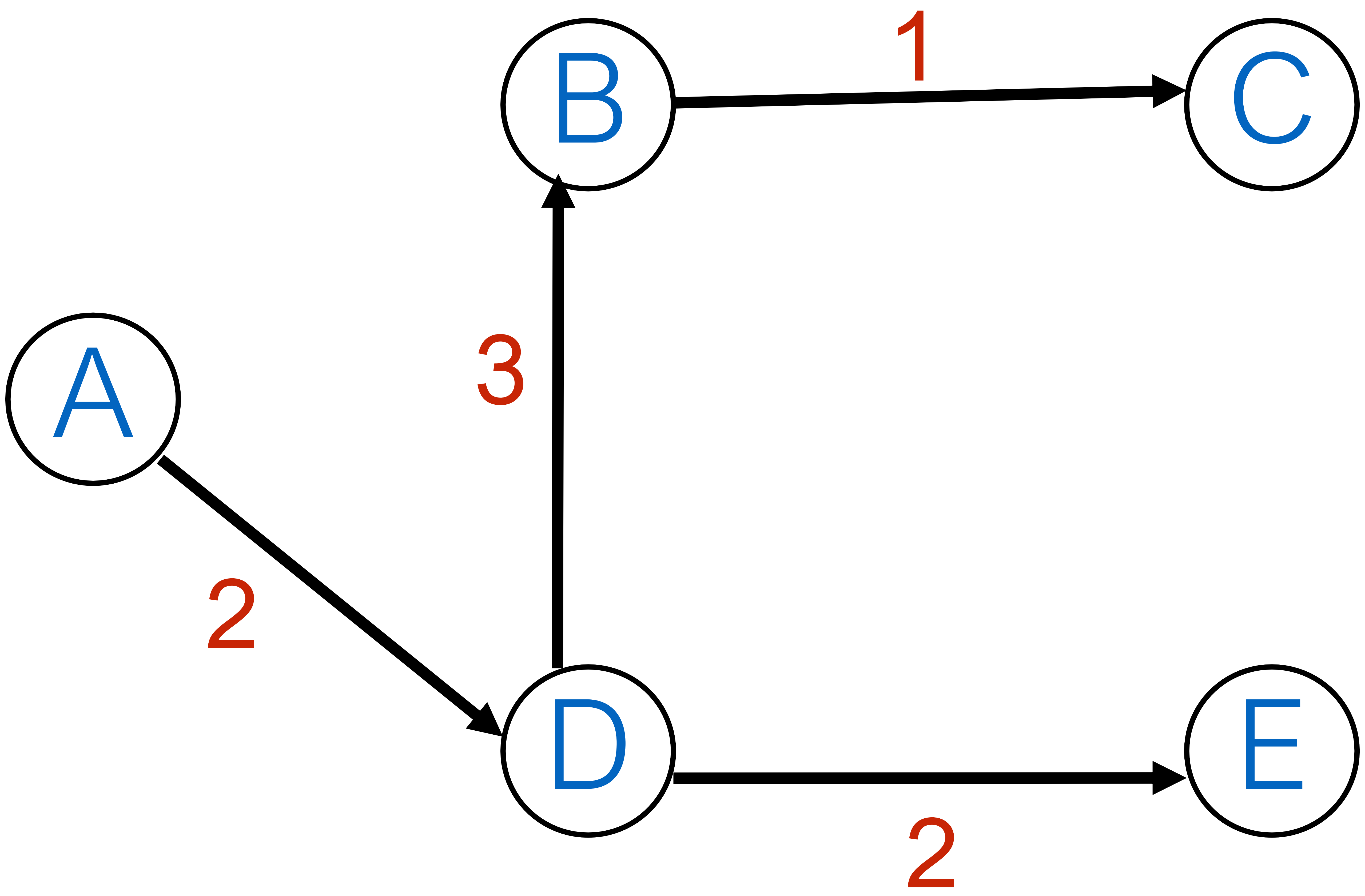
$O(|E|\log|V|)$

$O(|V|\log|V|+|E|\log|V|)$

Let's Try another Example



Example



Questions