Last time: Kosaraju's algorithm

# Shortest Paths [BFS, Djikstra]

## Sides based on material by Kani, Chekuri, Erickson et. al.

**All mistakes are my own! - Ivan Abraham (Fall 2024)**

Image by ChatGPT (probably collaborated with DALL-E)

# Breadth first search (BFS)

## Overview

- Breadth-first search (BFS) is an algorithm for traversing or searching a Tree or Graph data structure which returns the nodes of the graph level by level.
- BFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time (obtained from BasicSearch by processing edges using a queue data structure).
- It processes the vertices in the graph in the order of their shortest distance from the vertex $s$ (the start vertex)
- **DFS** good for exploring graph structure | **BFS** good for exploring distances

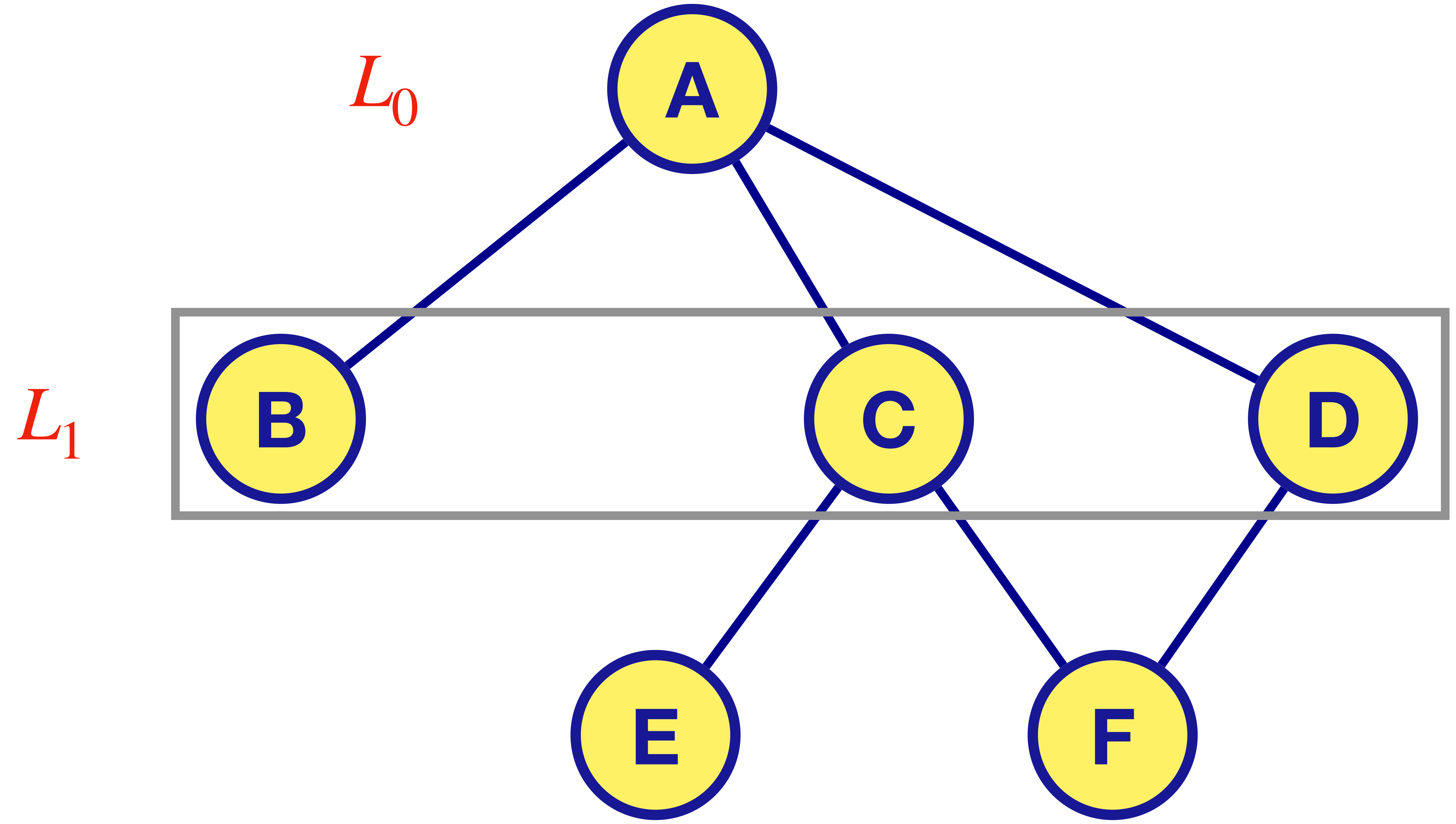
# Breadth first search (BFS)

BFS traversal of a graph returns the nodes of the graph level by level.

*The Idea of the BFS:*

Visit the vertices as follows:

- Visit all vertices at distance 1



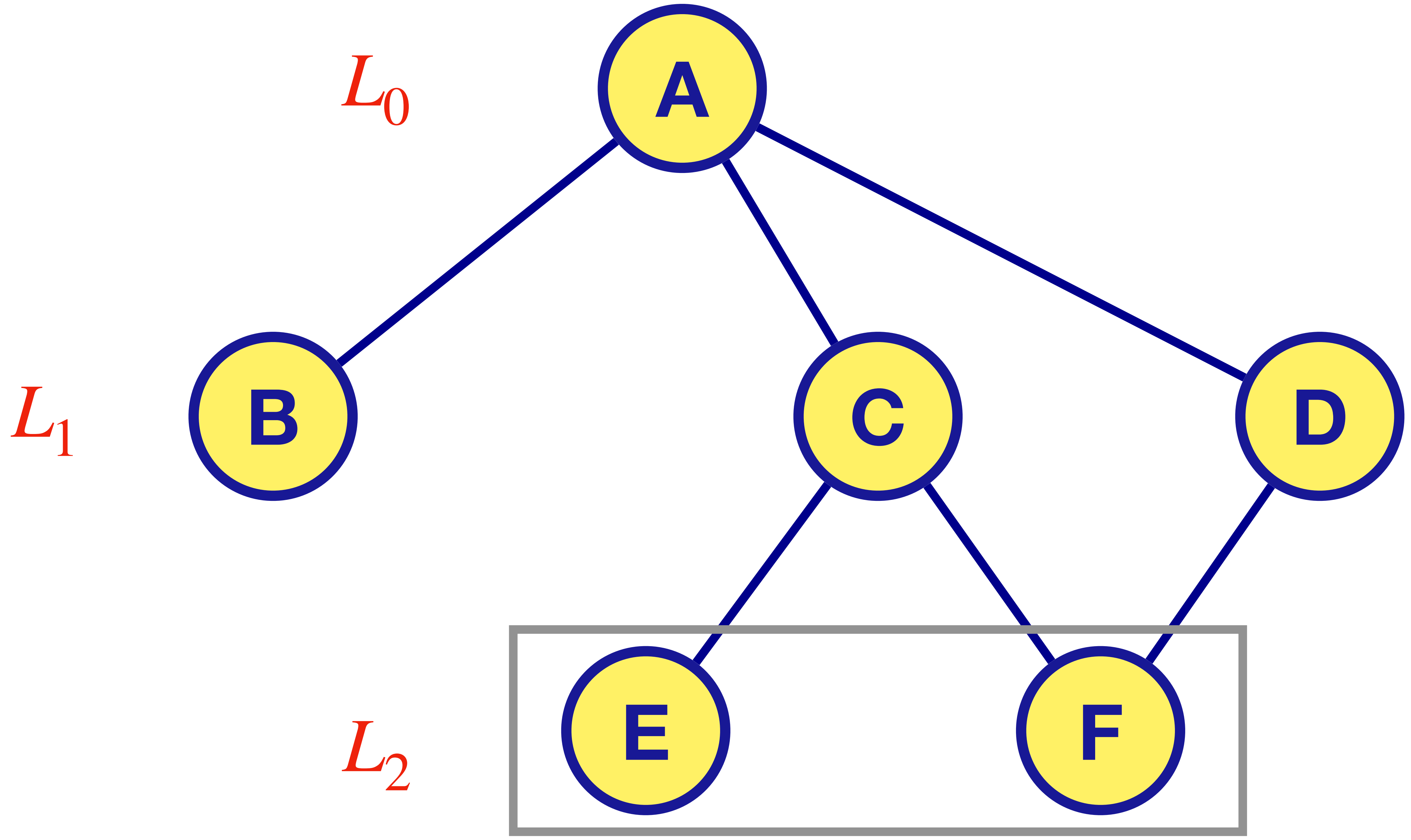
# Breadth first search (BFS)

BFS traversal of a graph returns the nodes of the graph level by level.

*The Idea of the BFS:*

Visit the vertices as follows:

- Visit all vertices at distance 1
- Visit all vertices at distance 2



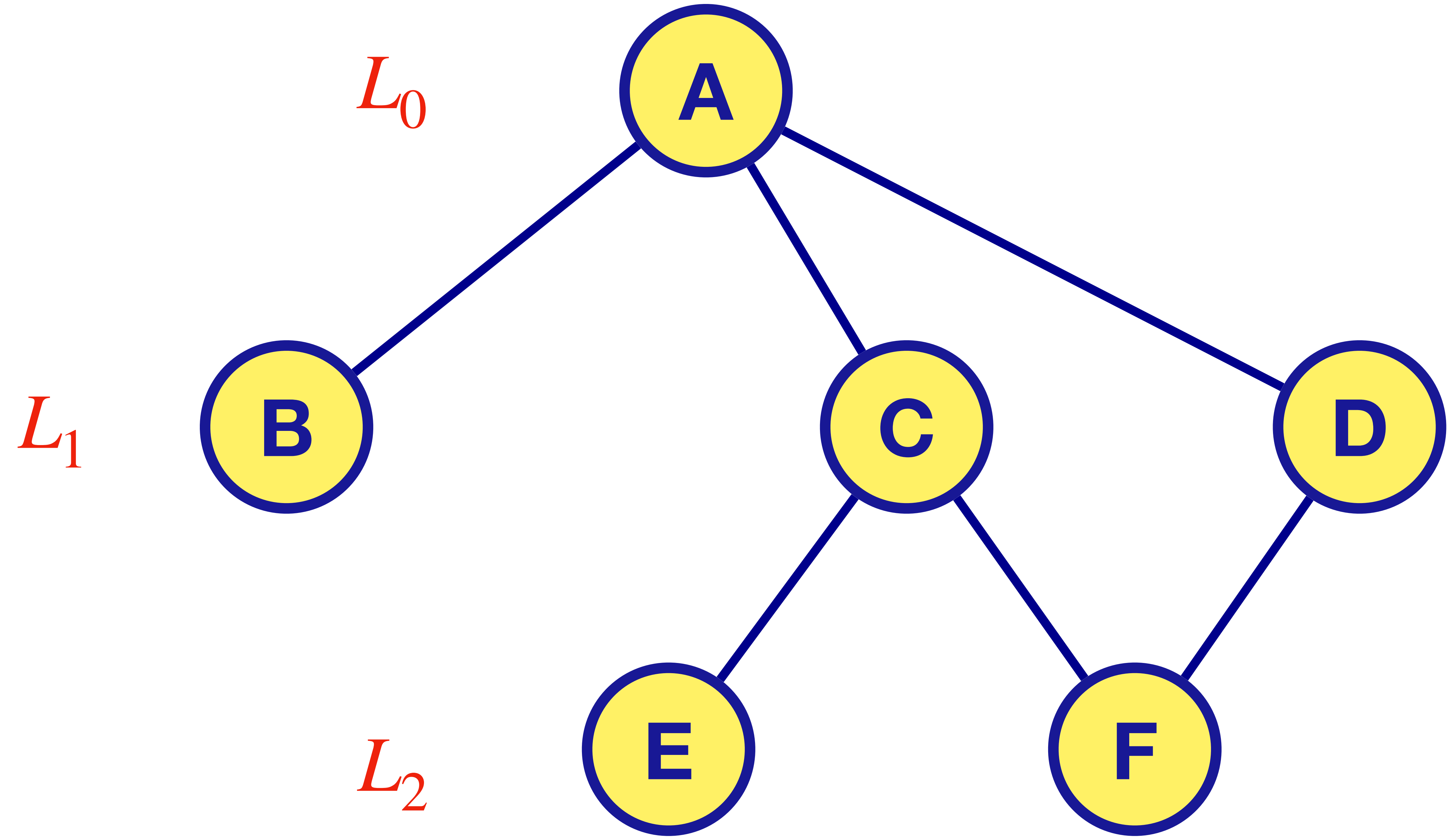
# Breadth first search (BFS)

BFS traversal of a graph returns the nodes of the graph level by level.

*The Idea of the BFS:*

Visit the vertices as follows:

- Visit all vertices at distance 1
- Visit all vertices at distance 2
- Visit all vertices at distance 3 etc.



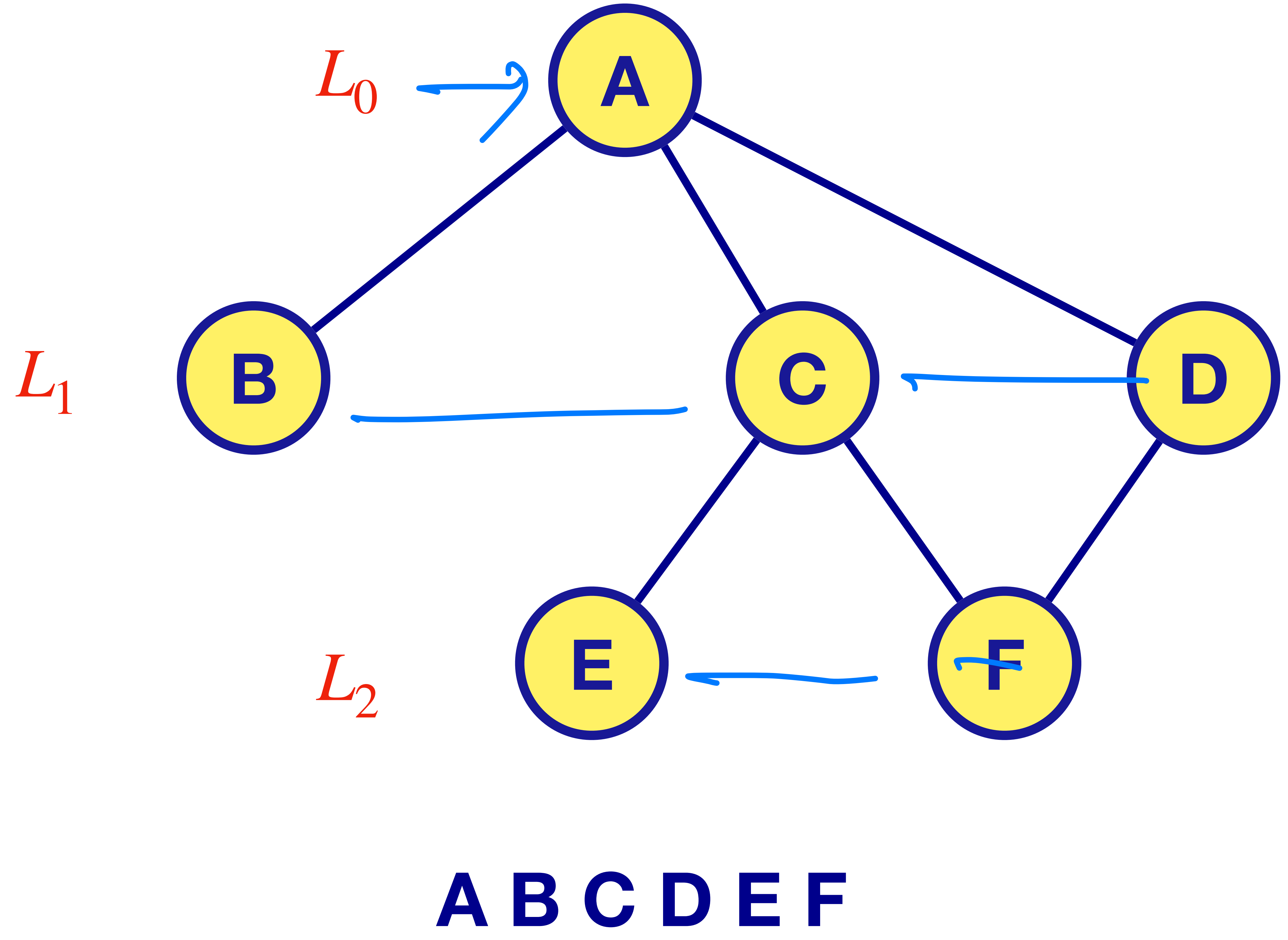
# Breadth first search (BFS)

BFS traversal of a graph returns the nodes of the graph level by level.

*The Idea of the BFS:*

Visit the vertices as follows:

- Visit all vertices at distance 1
- Visit all vertices at distance 2
- Visit all vertices at distance 3 etc.



# Queue data structure

## Queues

A queue is a list of elements which supports the operations:

- **Enqueue**: Adds an element to the end of the list
- **Dequeue**: Removes an element from the front of the list
- Elements are extracted in first-in first-out (FIFO) order, i.e., elements are picked in the order in which they were inserted.
    - Contrast with LIFO (stacks)

# BFS algorithm
## Pseudocode

Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

```
BFS(s):
    Mark all vertices as unvisited;
    Initialize search tree T to be empty
    Mark vertex s as visited
    set Q to be the empty queue
    enqueue(Q,s)
    while Q is non-empty do
          u = dequeue(Q)
          for each vertex v ∈ Adj(u)
              if v is not visited then
                add edge (u,v) to T
                Mark v as visited and enqueue(v)
```

# BFS algorithm

## Pseudocode

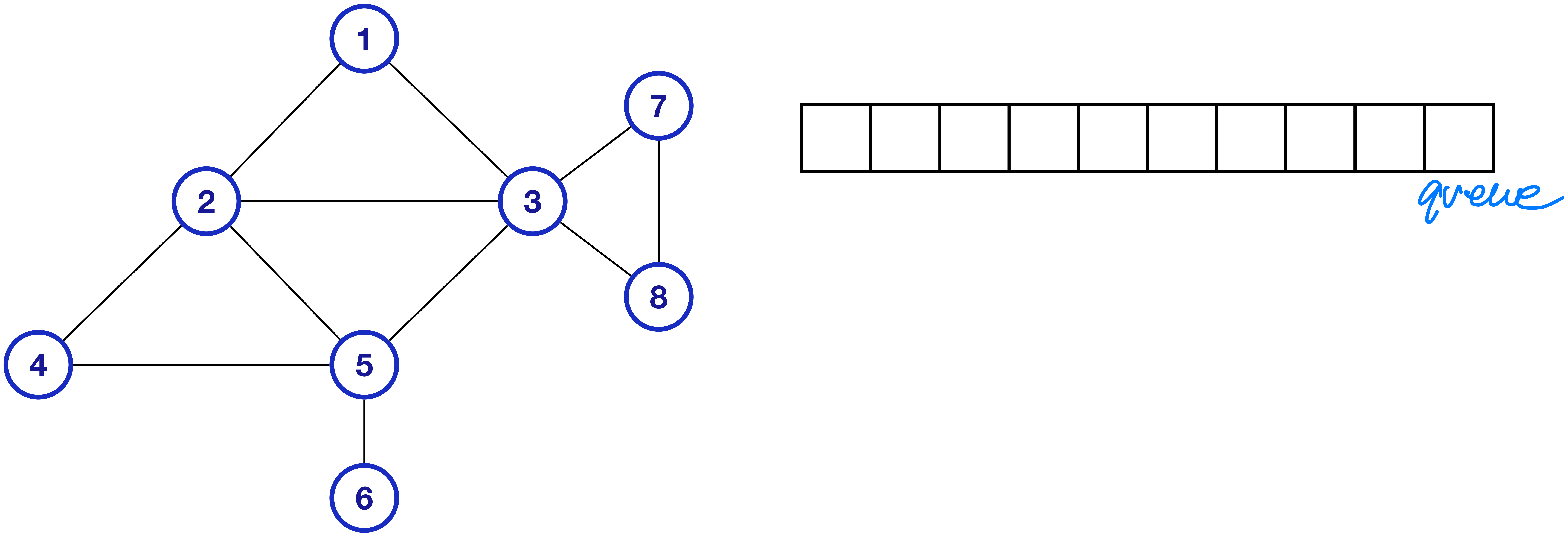
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

```
BFS(s):
    Mark all vertices as unvisited;
    Initialize search tree T to be empty
    Mark vertex s as visited
    set Q to be the empty queue
    enqueue(Q,s)
    while Q is non-empty do
          u = dequeue(Q)
          for each vertex v ∈ Adj(u)
              if v is not visited then
                add edge (u,v) to T
                Mark v as visited and enqueue(v)
```

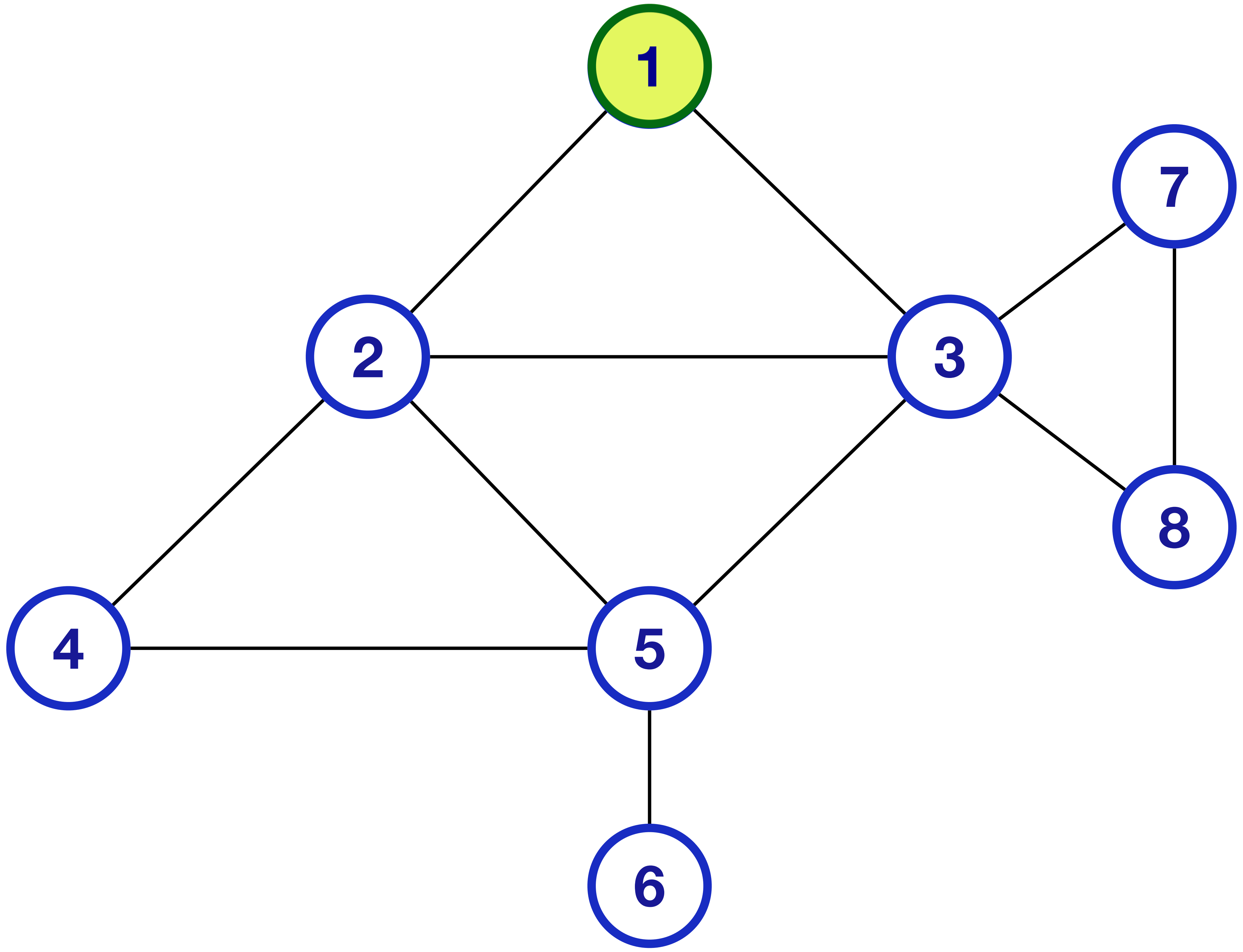
**Proposition**

BFS(s) runs in $O(n+m)$ time

# BFS: An example in undirected graphs



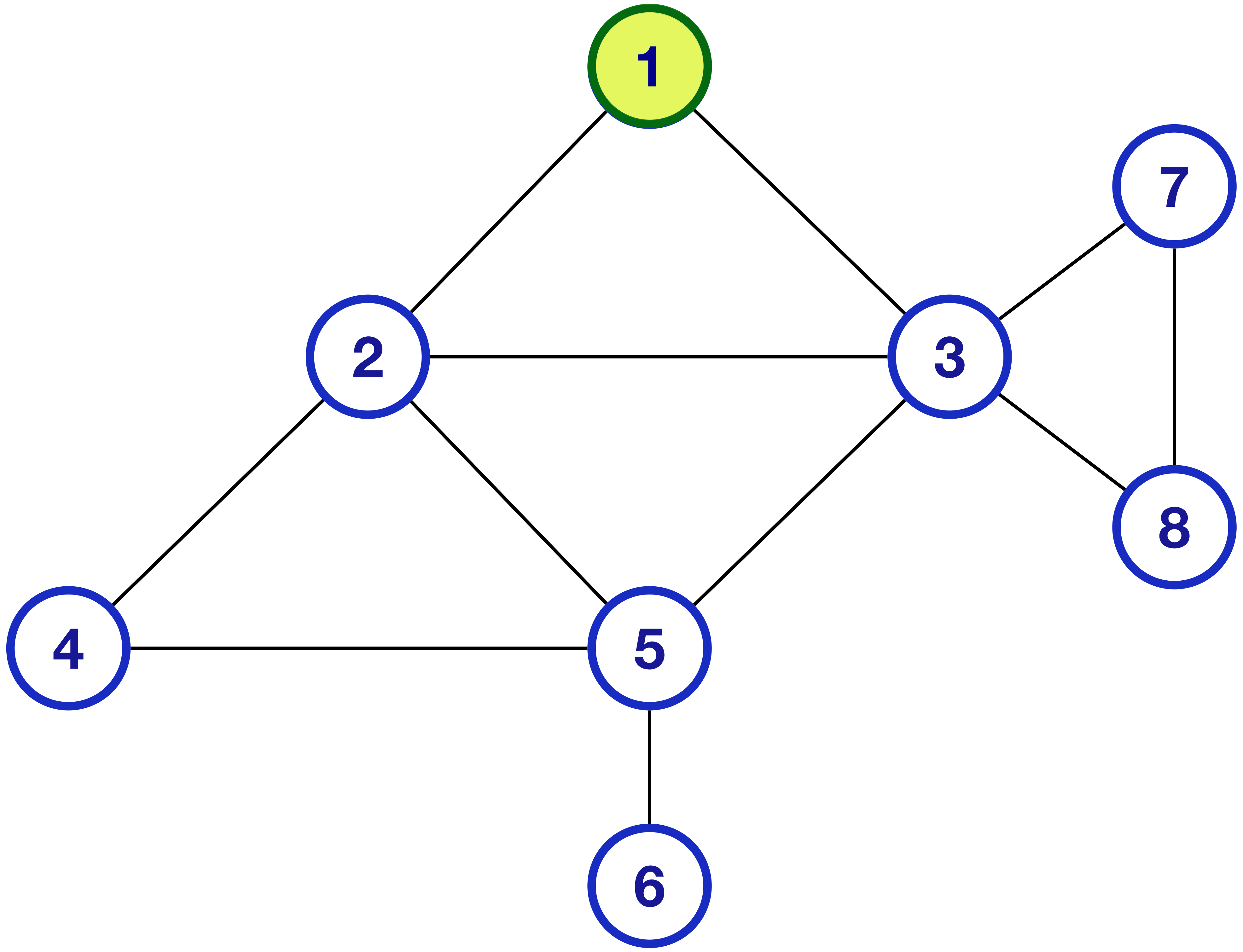
# BFS: An example in undirected graphs

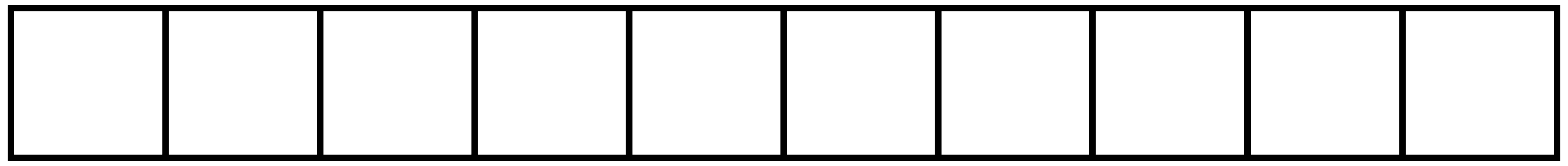



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | |

Mark and enqueue **1**

# BFS: An example in undirected graphs

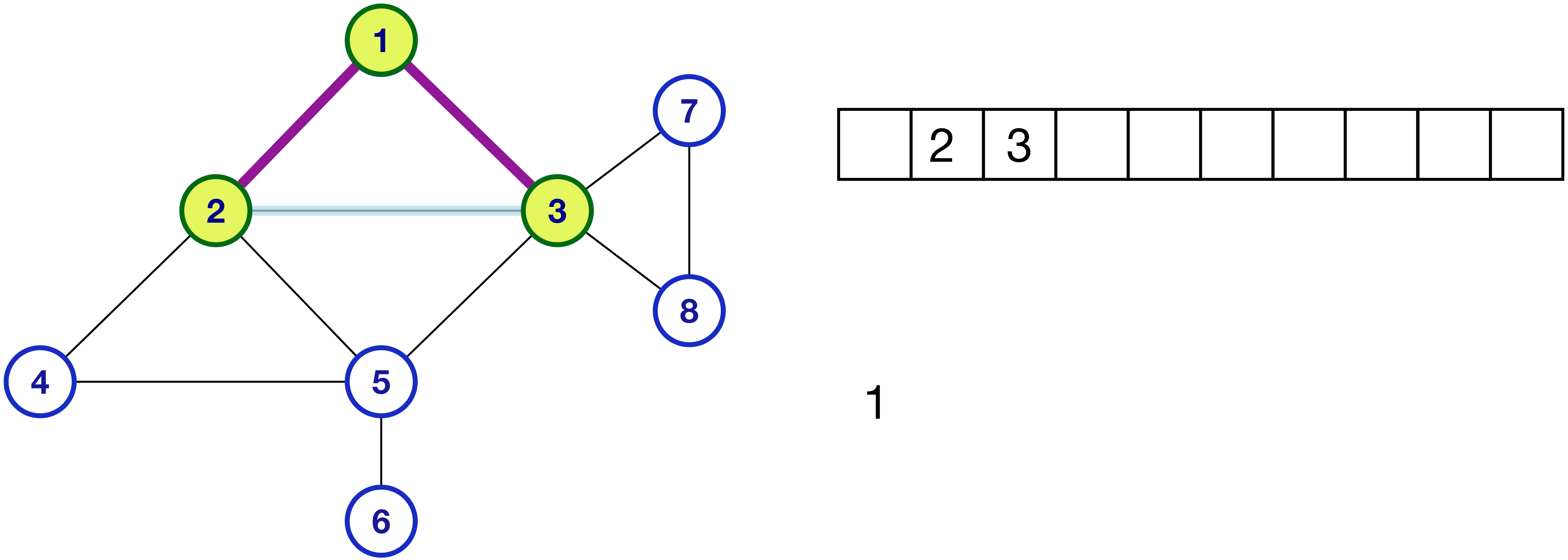


1

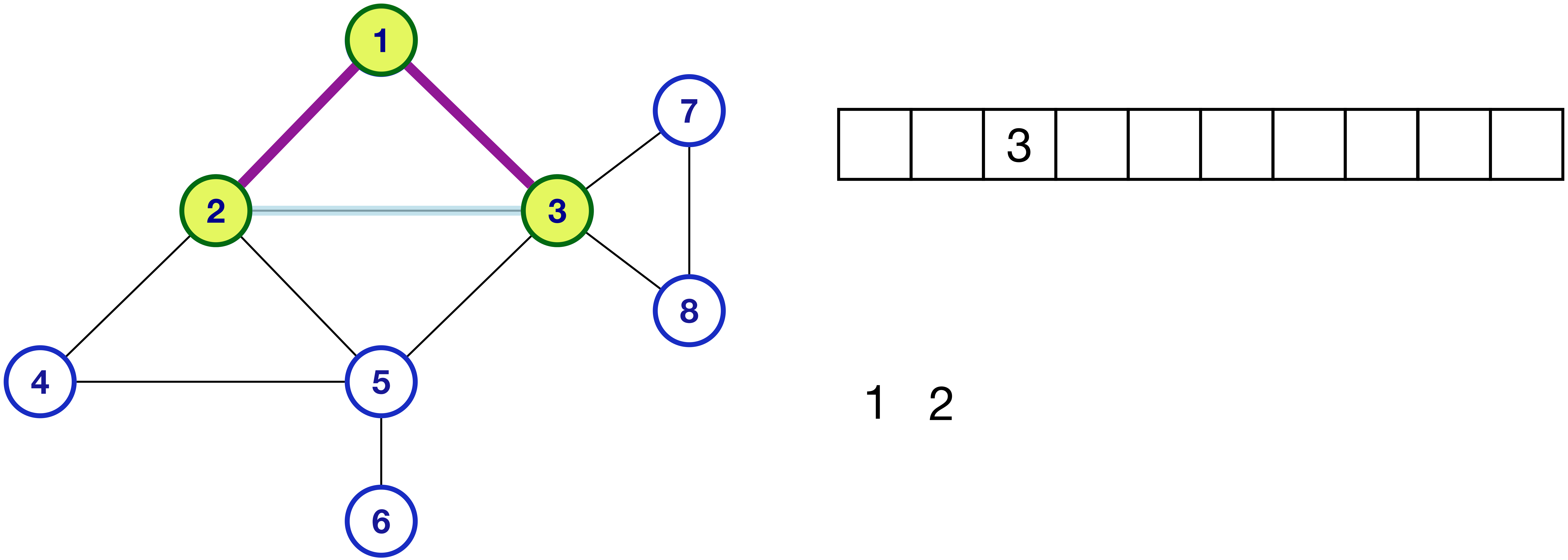
Dequeue **1**

# BFS: An example in undirected graphs



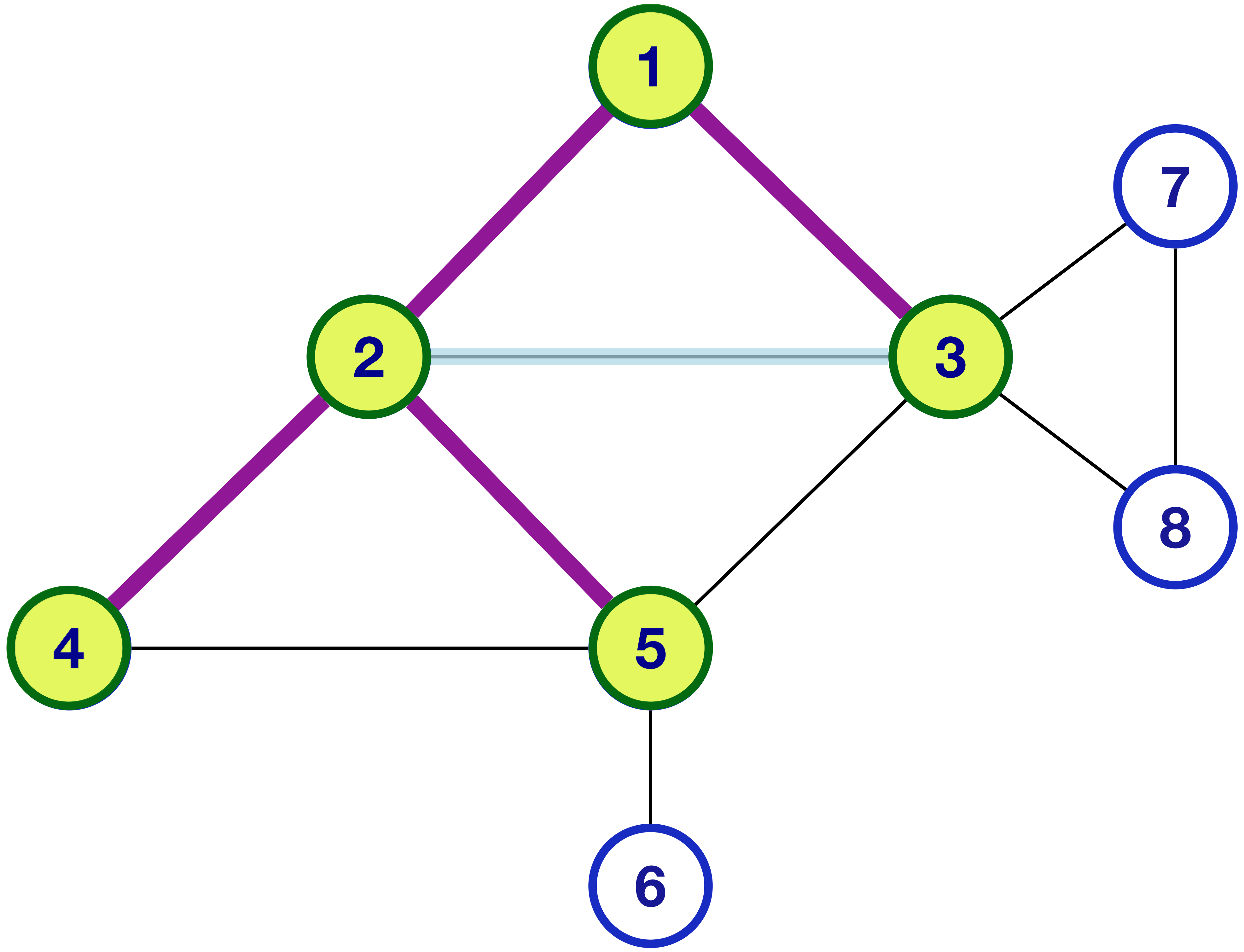

Mark and enqueue **2** and **3**

# BFS: An example in undirected graphs




Dequeue **2**

# BFS: An example in undirected graphs

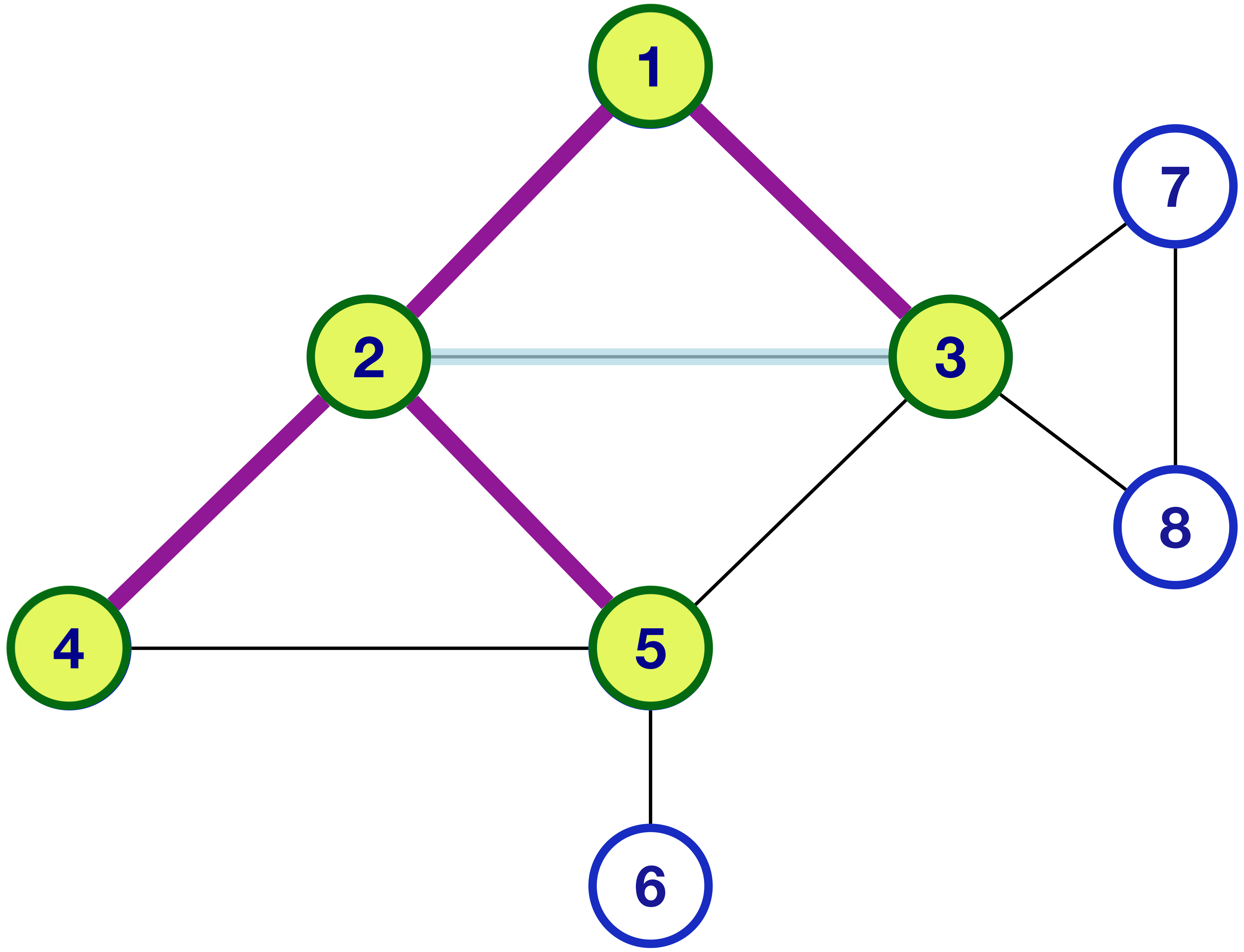



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 3 | 4 | 5 | | | | | |

1 2

Mark and enqueue **4** and **5**

# BFS: An example in undirected graphs

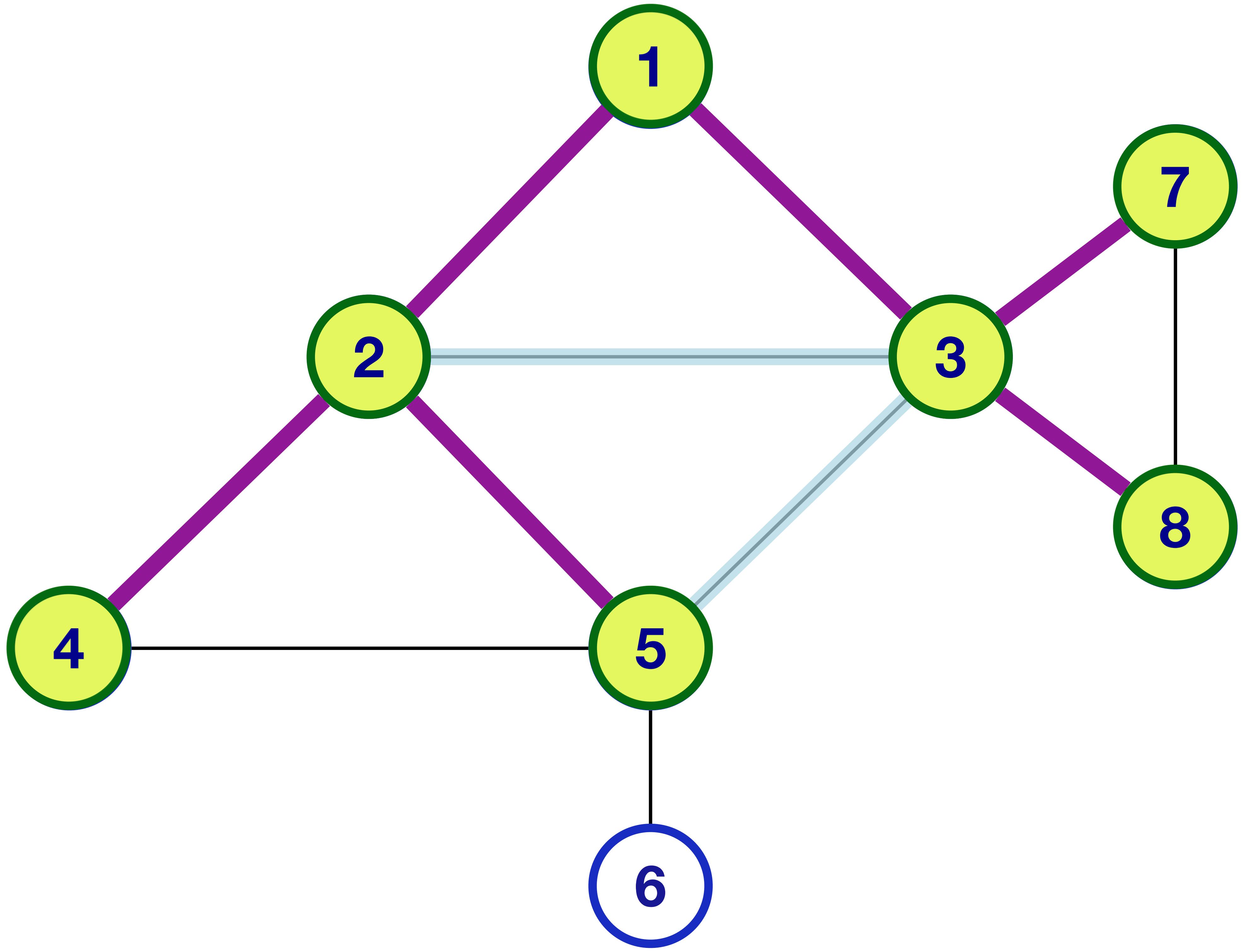


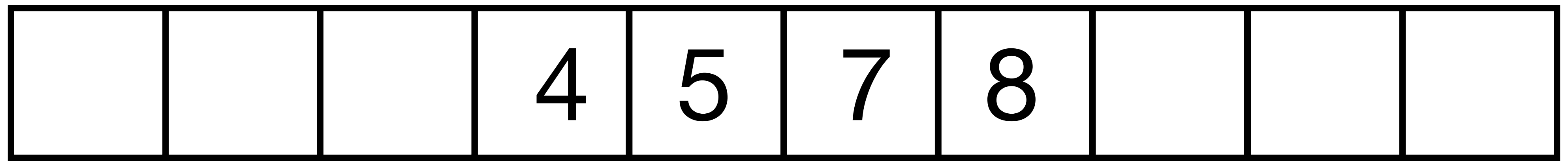

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 4 | 5 | | | | | |

1 2 3

Dequeue **3**

# BFS: An example in undirected graphs

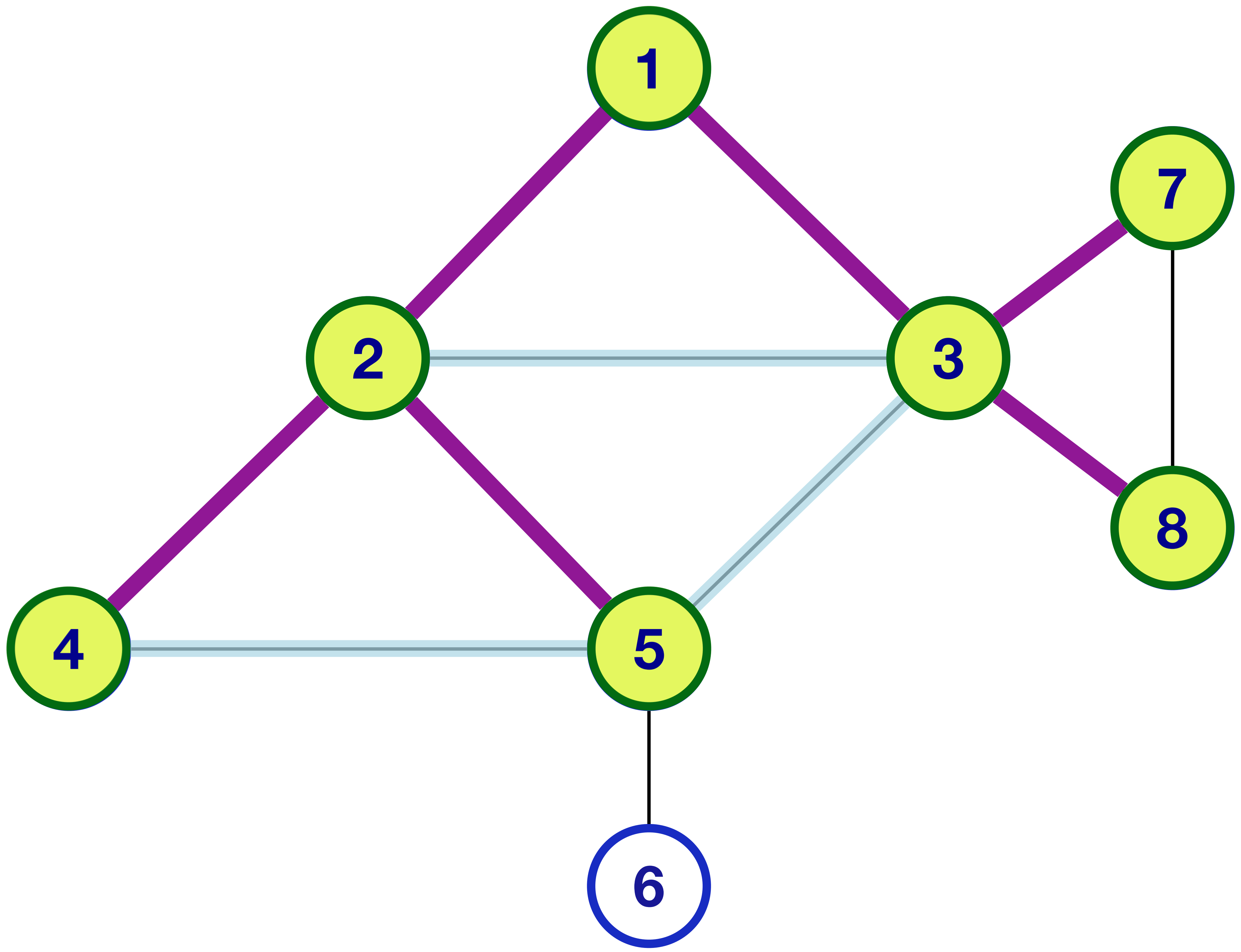



1 2 3

Mark and enqueue **7** and **8**

# BFS: An example in undirected graphs

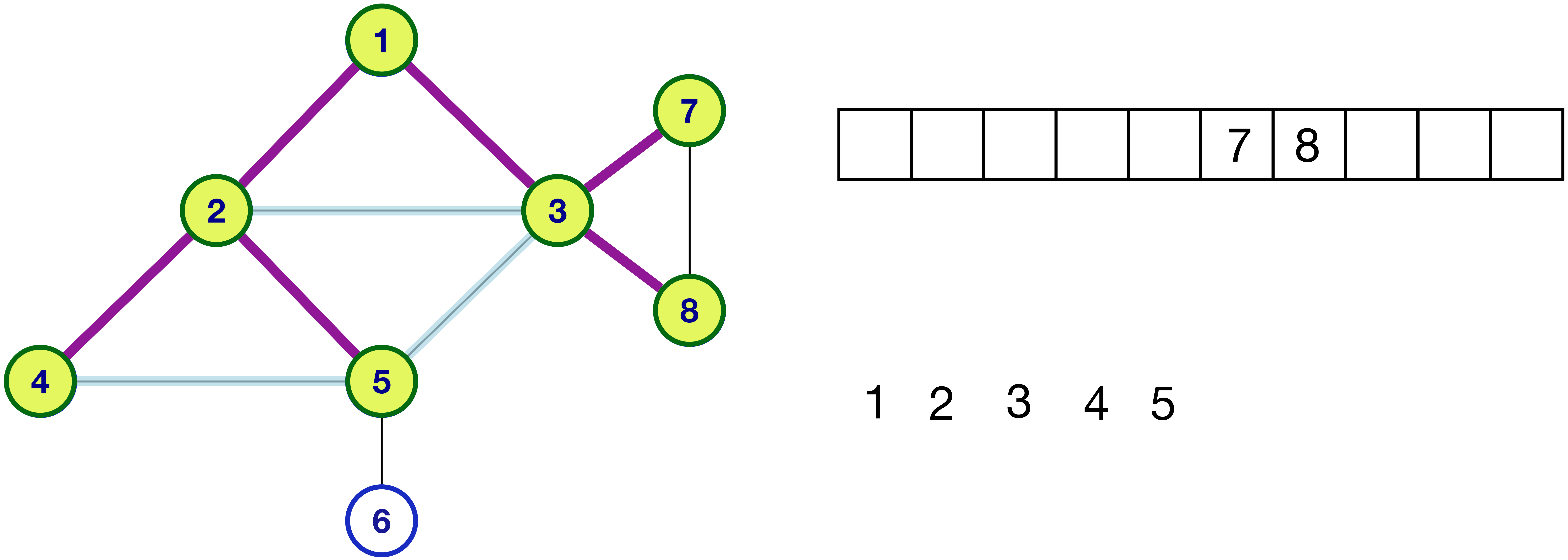



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 5 | 7 | 8 | | | |

1 2 3 4

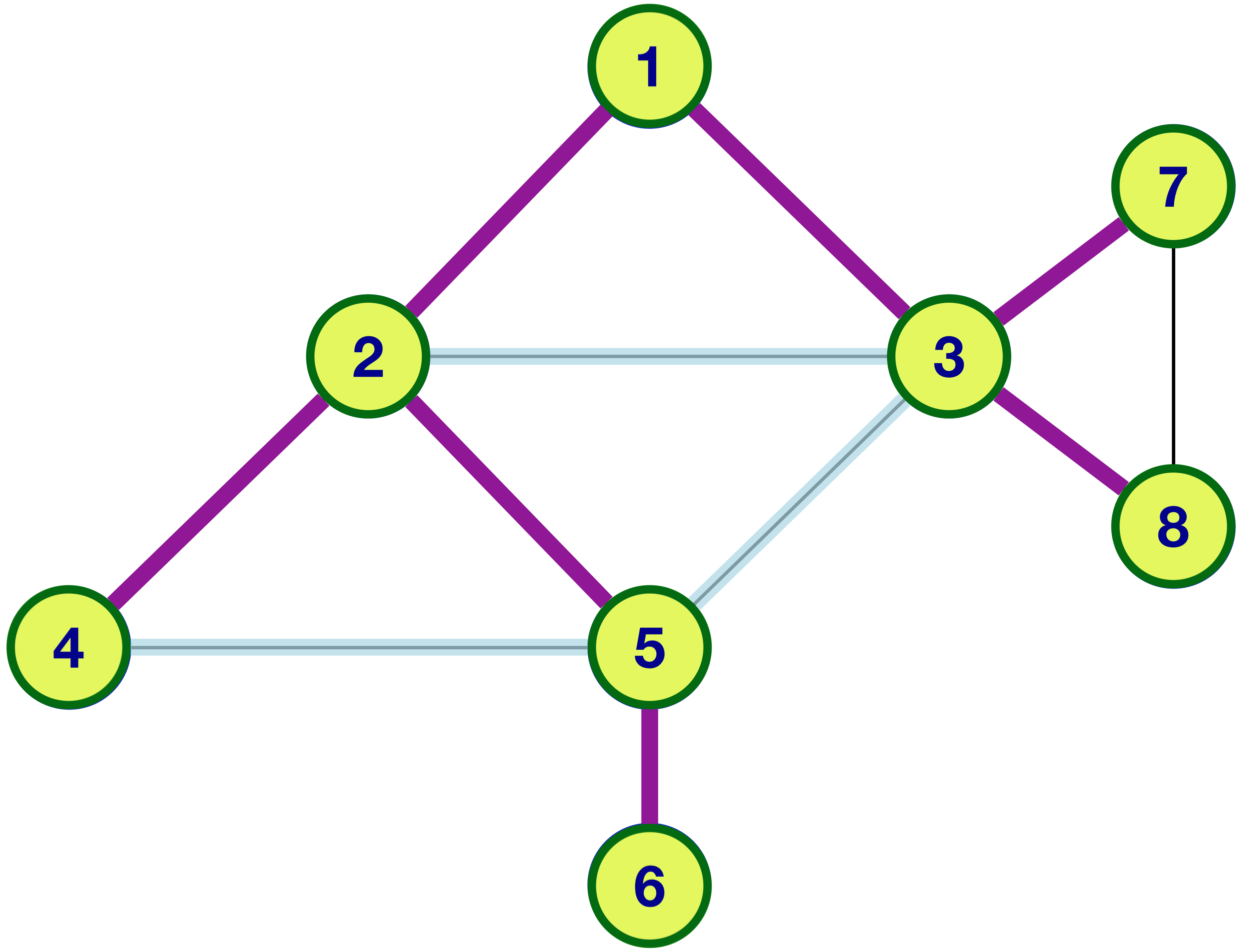
Dequeue **4**

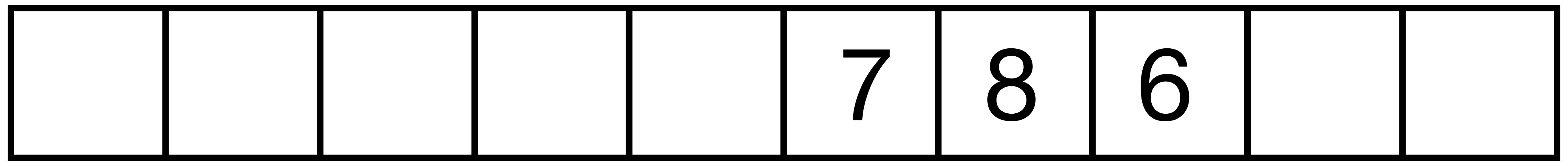
# BFS: An example in undirected graphs




Dequeue **5**

# BFS: An example in undirected graphs

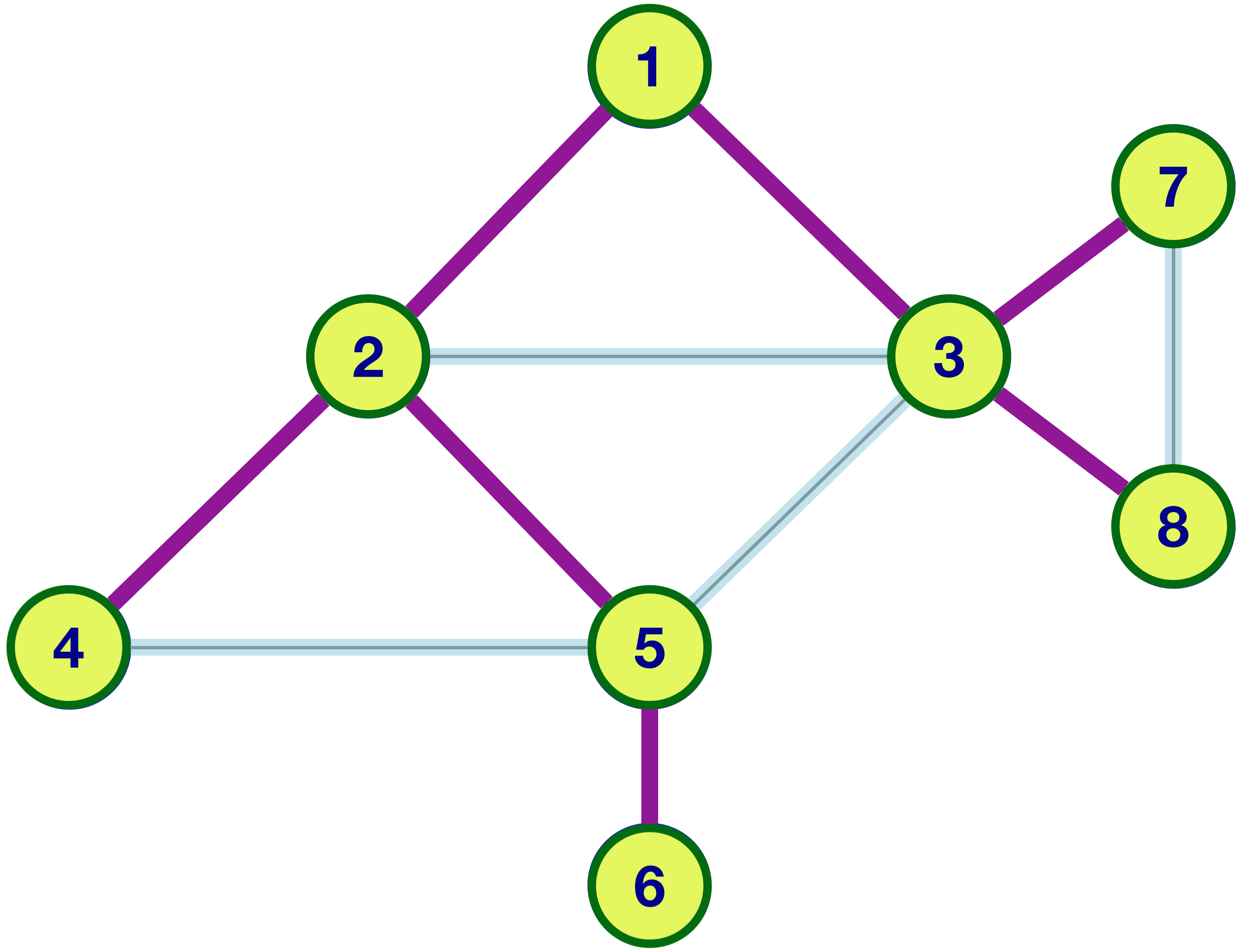



1 2 3 4 5

Mark and enqueue **6**

# BFS: An example in undirected graphs

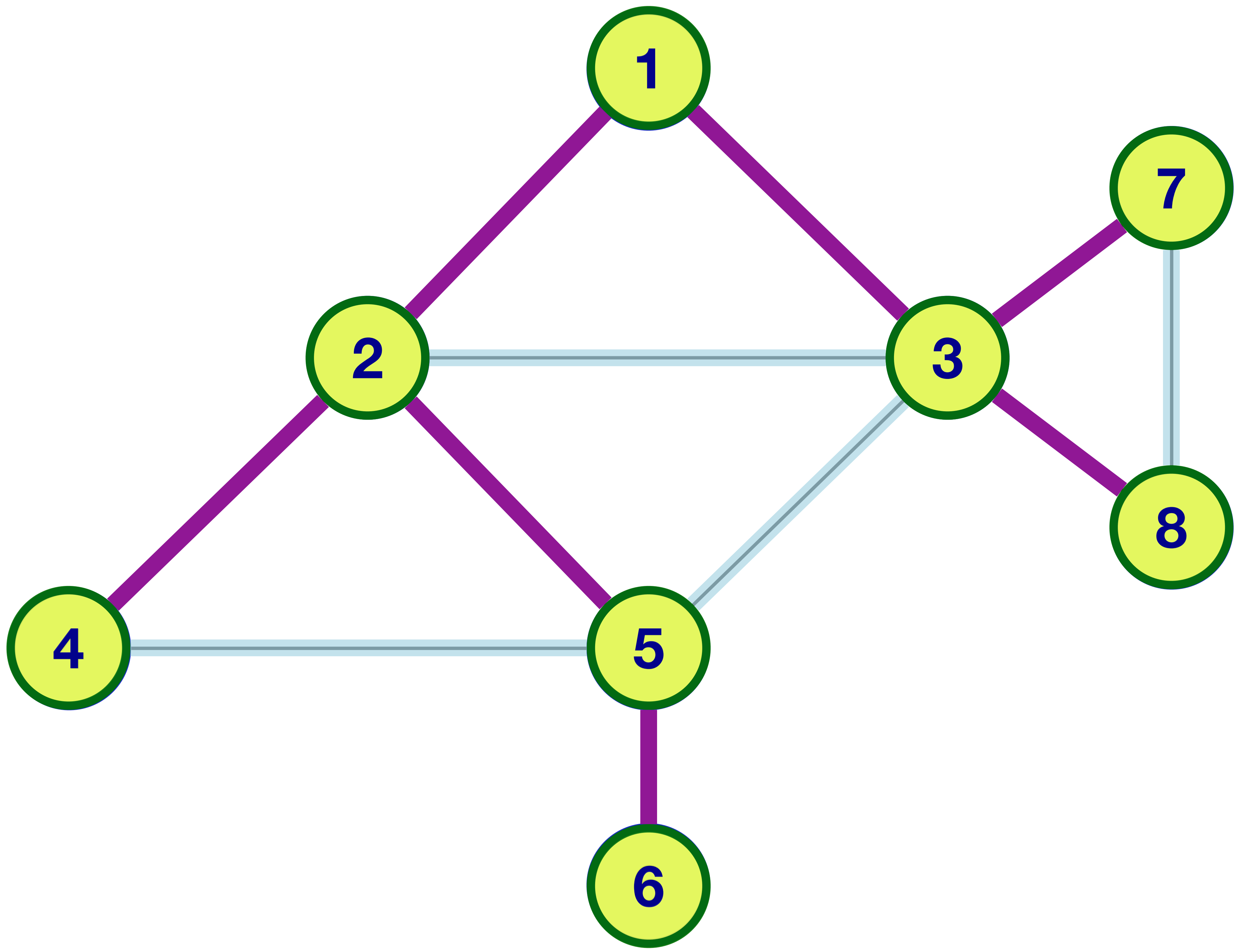



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | 8 | 6 | | |

1 2 3 4 5 7

Dequeue **7**

# BFS: An example in undirected graphs

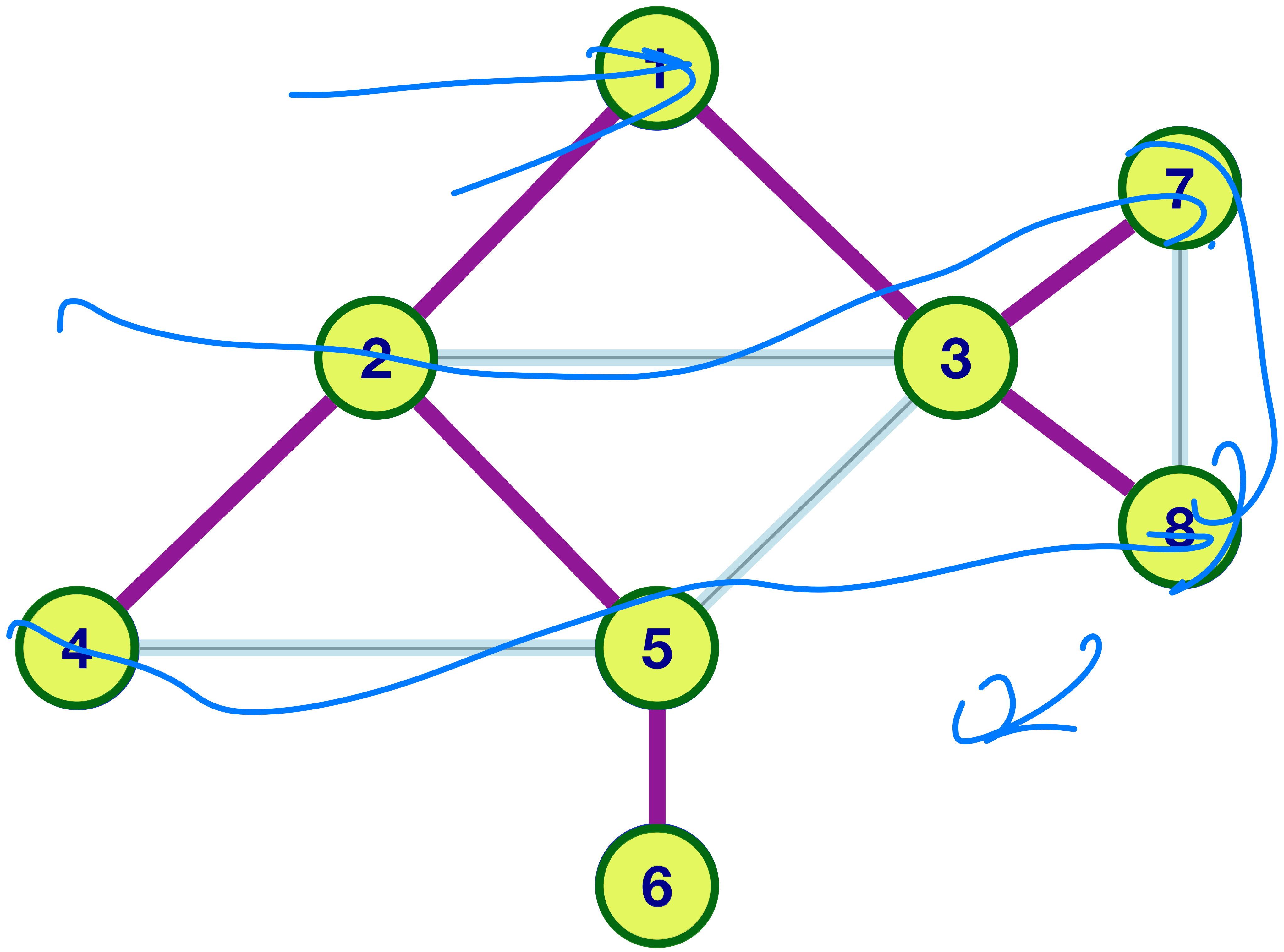


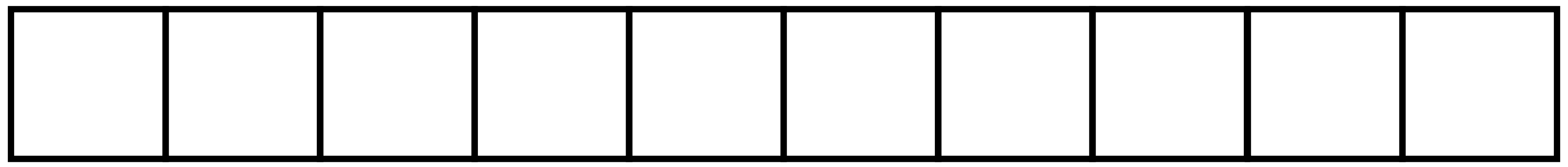

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | 6 | | |

1 2 3 4 5 7 8

Dequeue **8**

# BFS: An example in undirected graphs

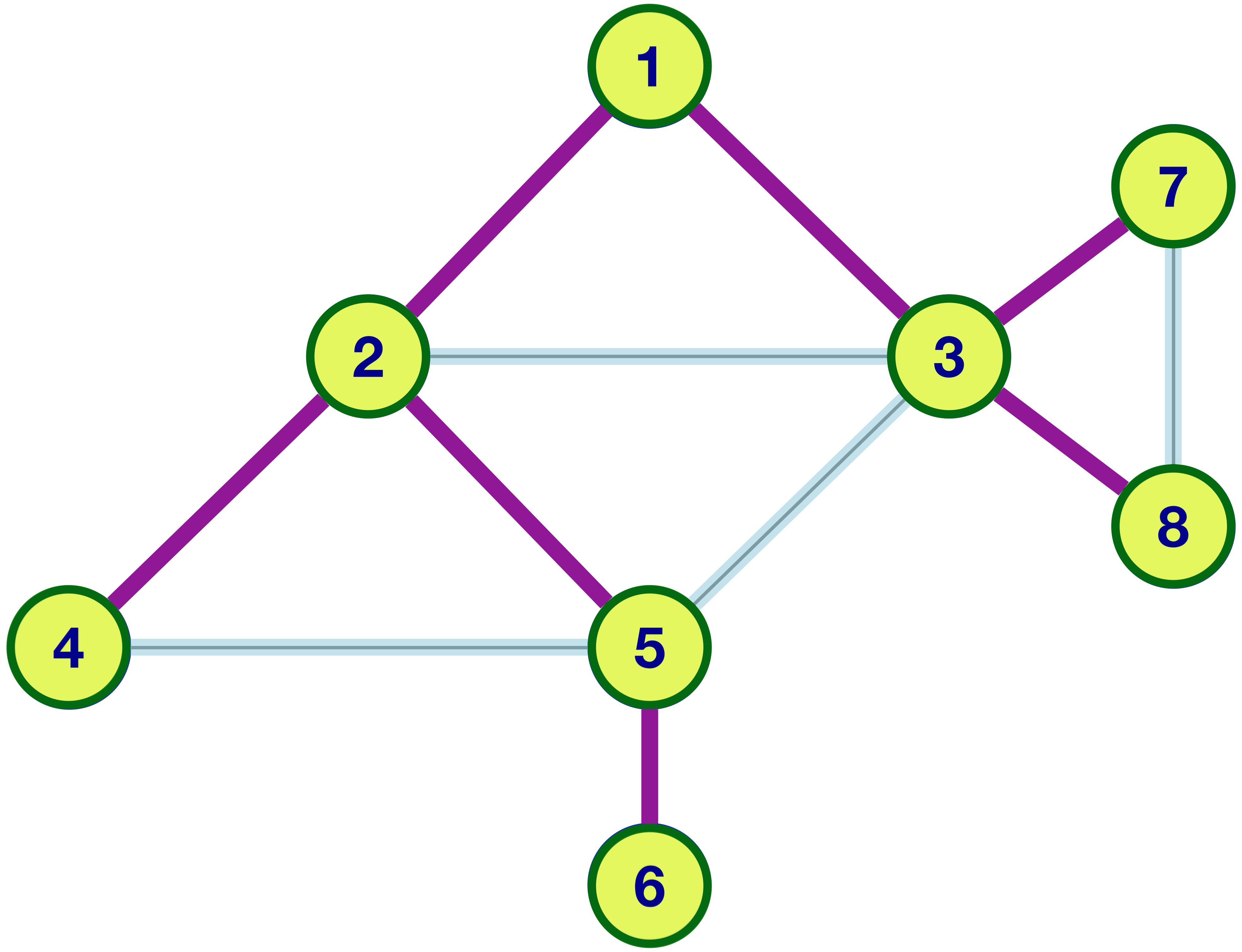


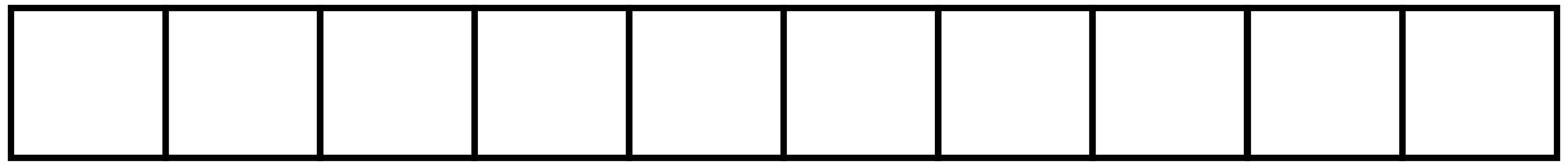



1 2 3 4 5 7 8 6

Dequeue **6**

# BFS: An example in undirected graphs

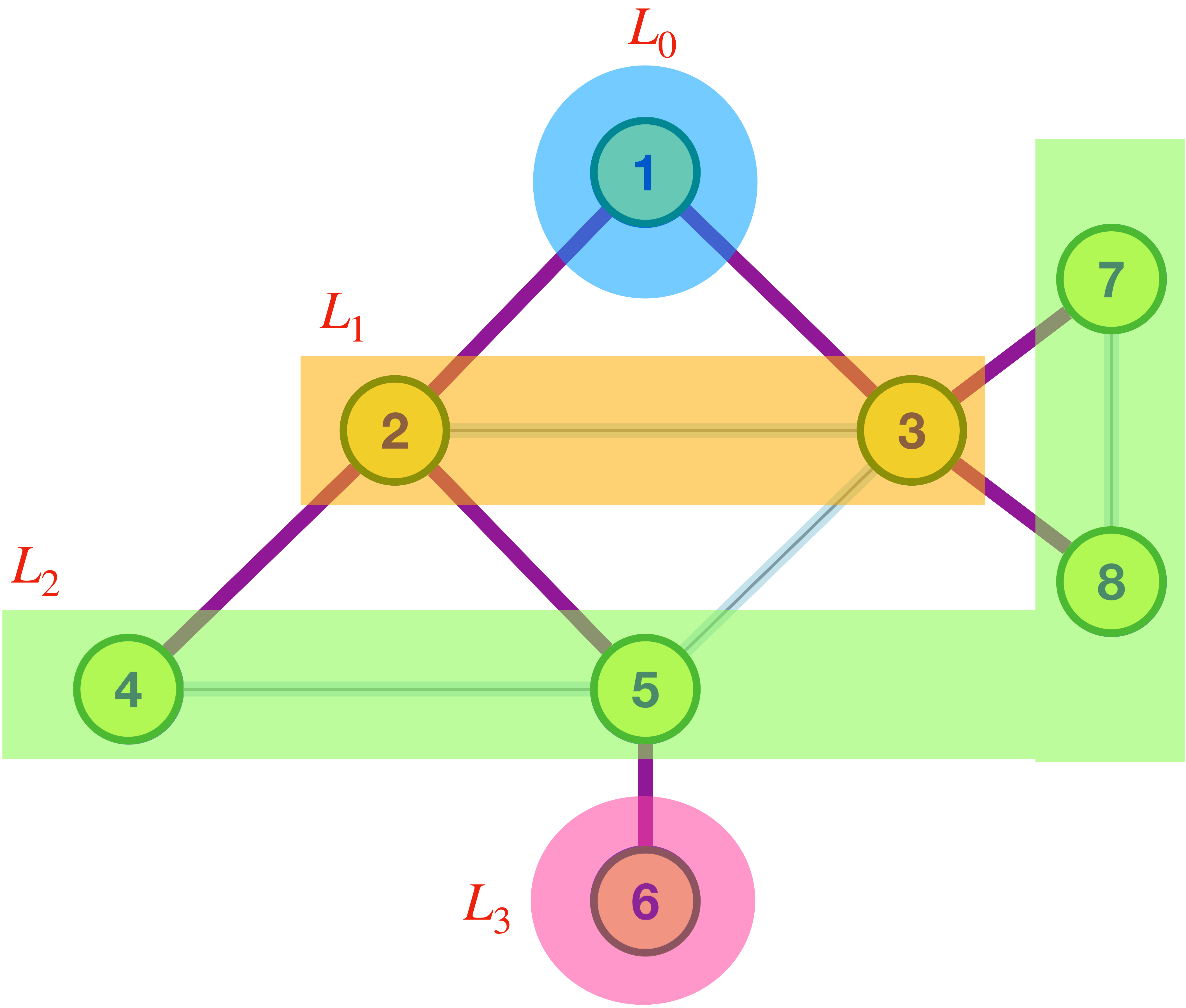


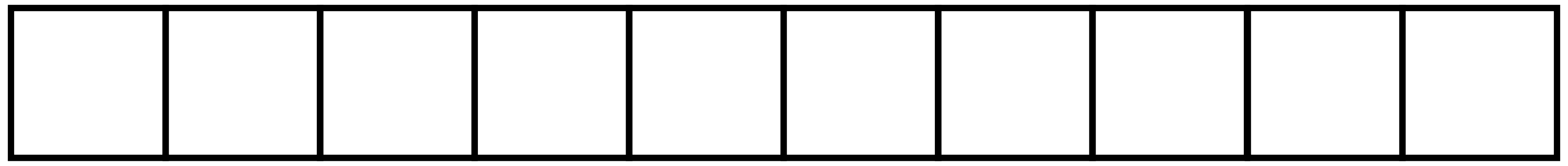
1 2 3 4 5 7 8 6

**BFS** tree is the set of **purple edges**

# BFS: An example in undirected graphs

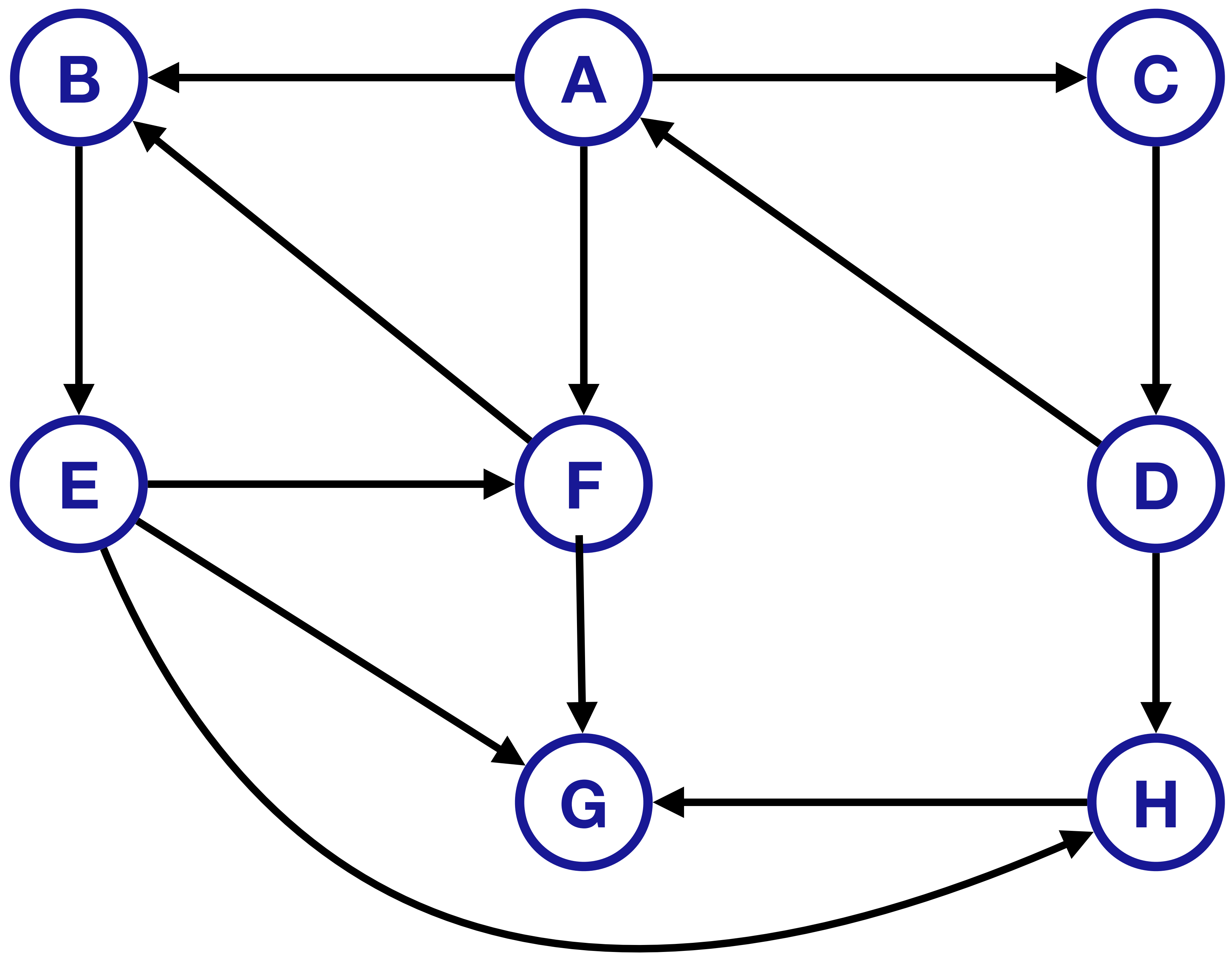



**BFS** tree is the set of **purple edges**

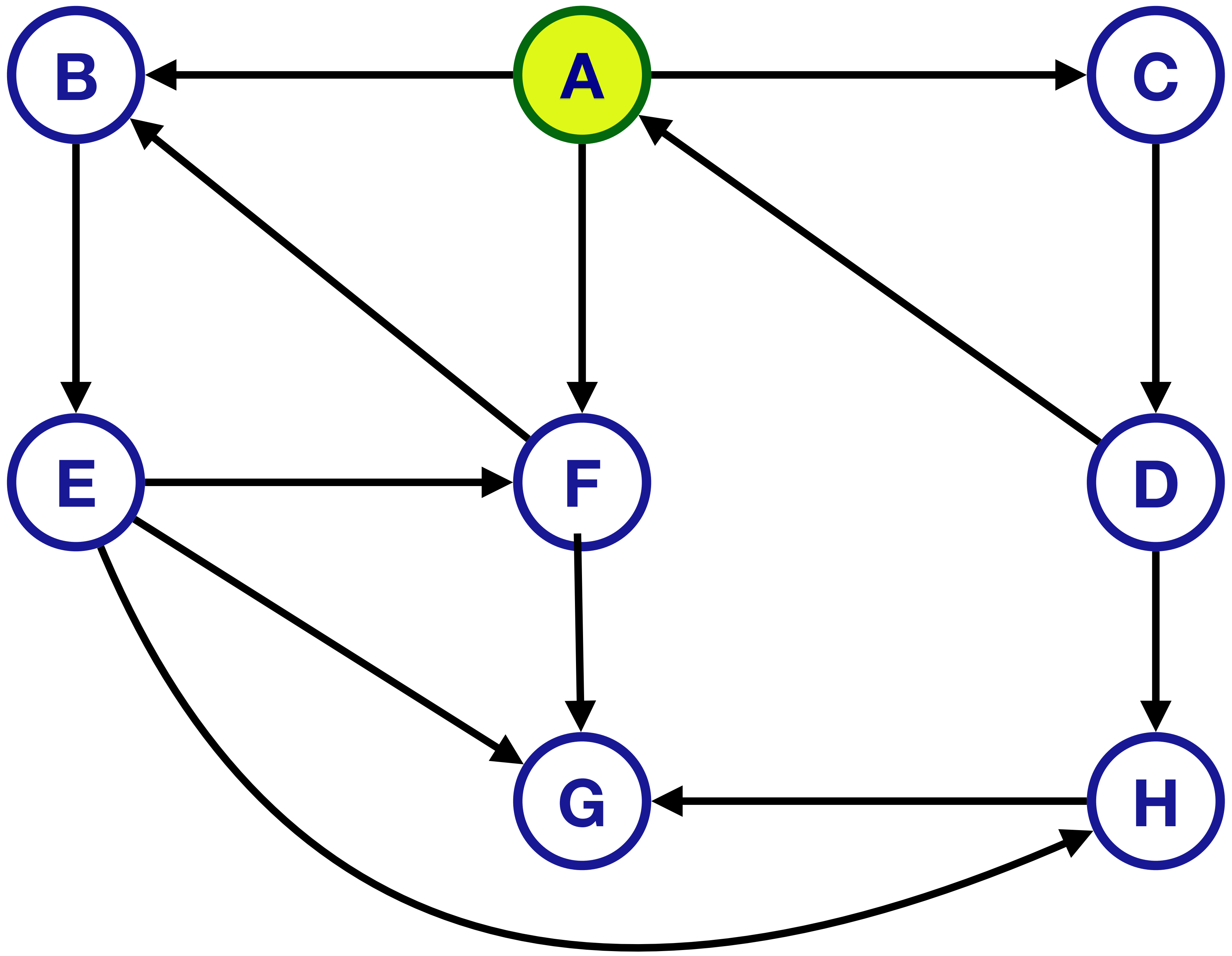


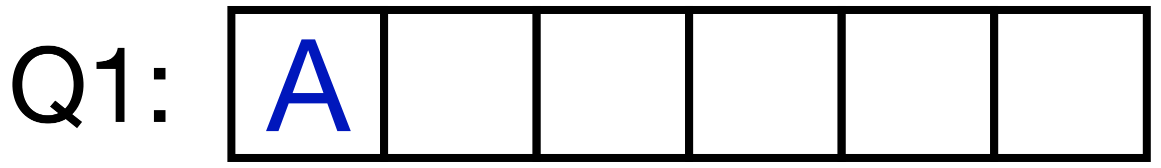
1 2 3 4 5 7 8 6

# BFS: An example in directed graphs

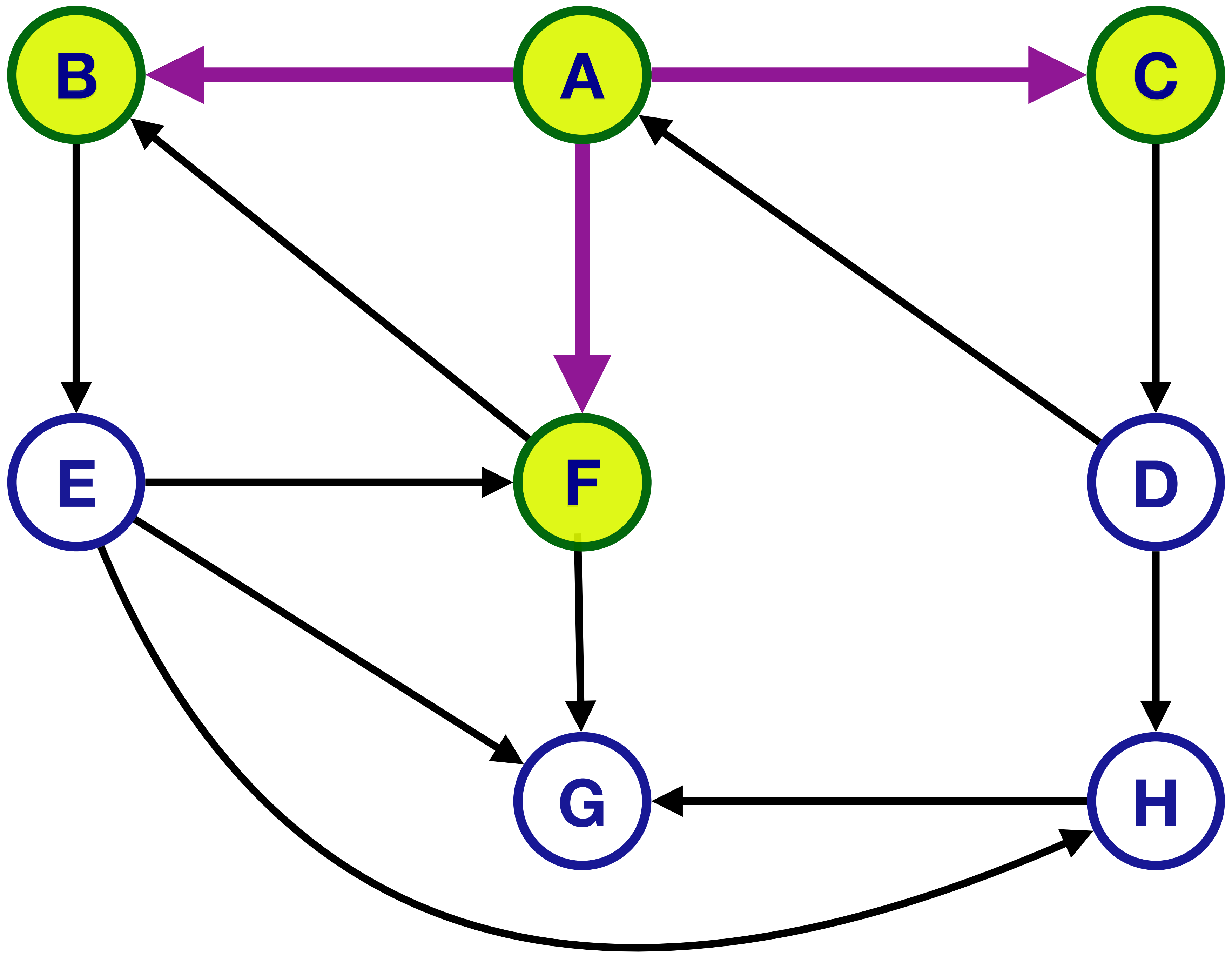


# BFS: An example in directed graphs



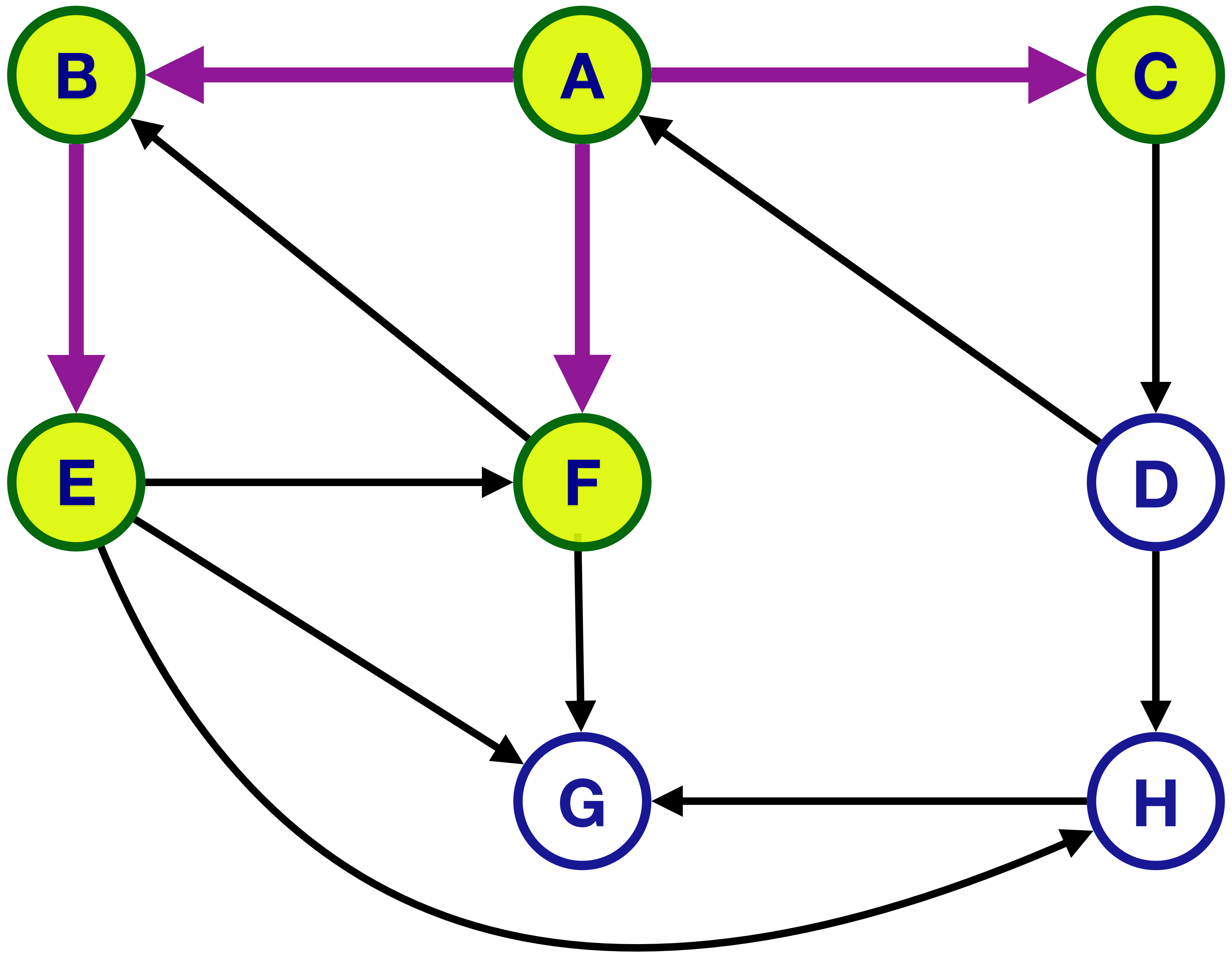



# BFS: An example in directed graphs



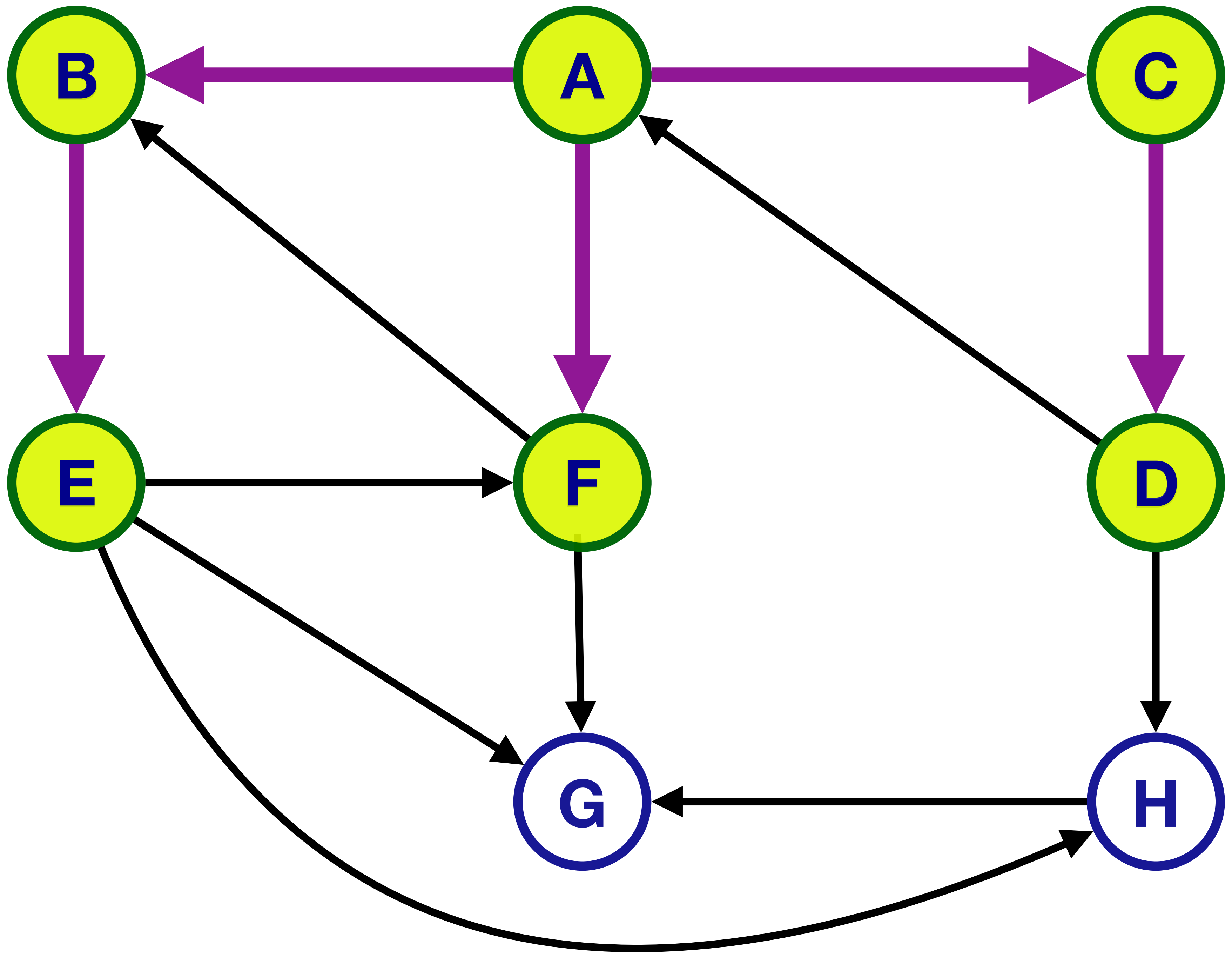

| | | | | | | |
|---|---|---|---|---|---|---|
| Q1: | A | | | | | |
| Q2: | B | C | F | | | |

# BFS: An example in directed graphs



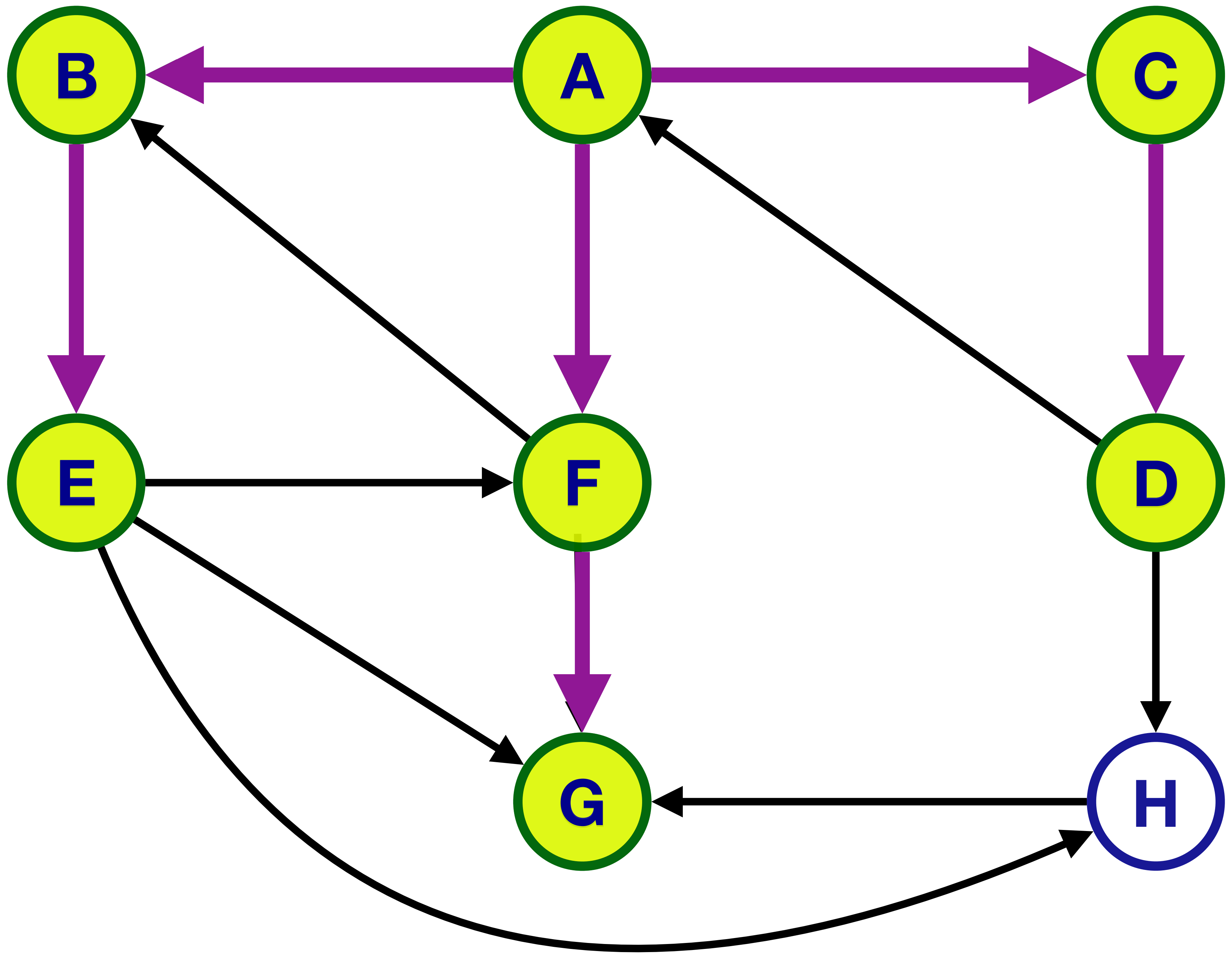

| | | | | | | |
|---|---|---|---|---|---|---|
| Q1: | A | | | | | |
| Q2: | B | C | F | | | |
| Q3: | C | F | E | | | |

# BFS: An example in directed graphs




| | | | | | | |
|---|---|---|---|---|---|---|
| Q1: | A | | | | | |
| Q2: | B | C | F | | | |
| Q3: | C | F | E | | | |
| Q4: | F | E | D | | | |

# BFS: An example in directed graphs




Q1: A

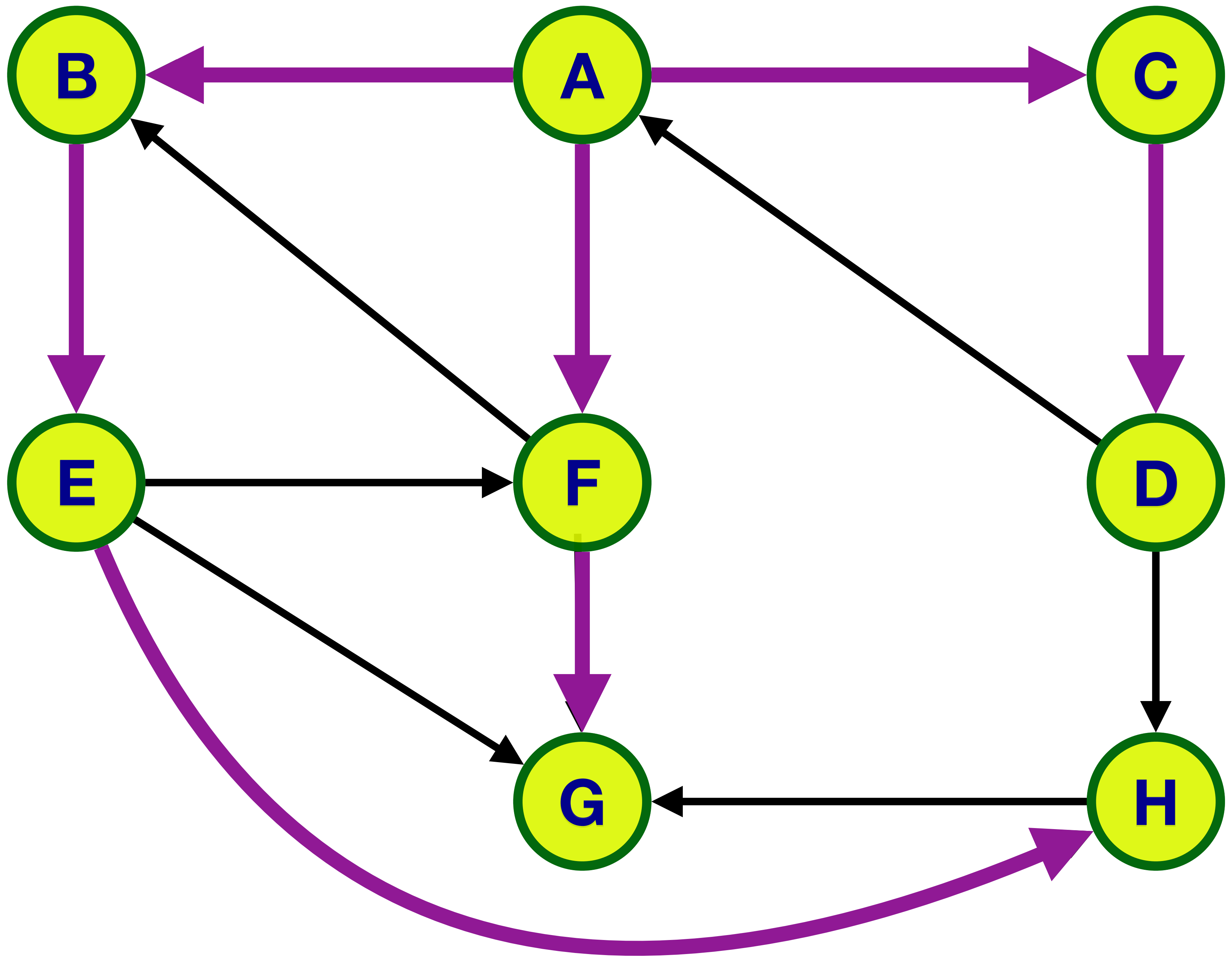
Q2: B C F

Q3: C F E

Q4: F E D

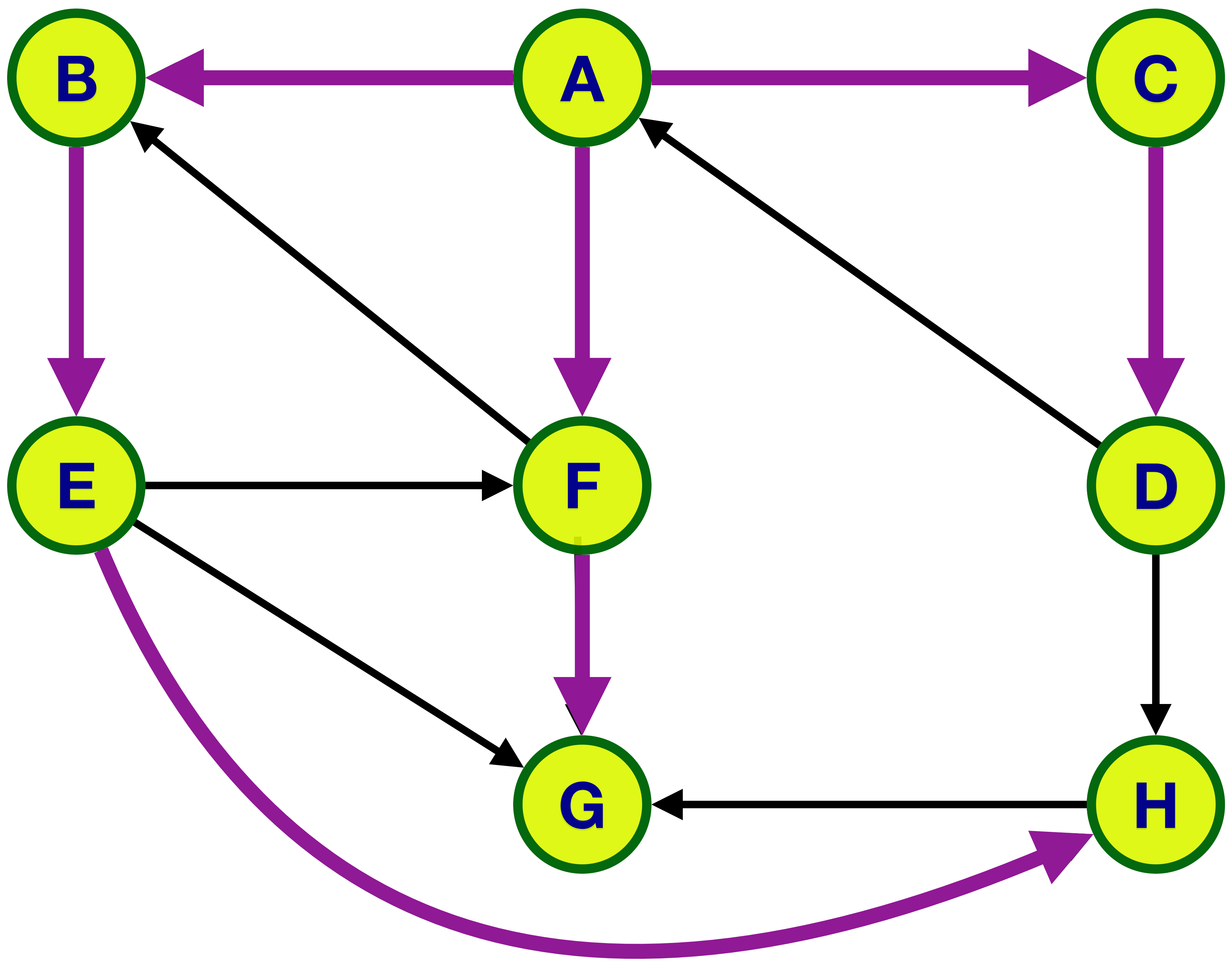
Q5: E D G

# BFS: An example in directed graphs



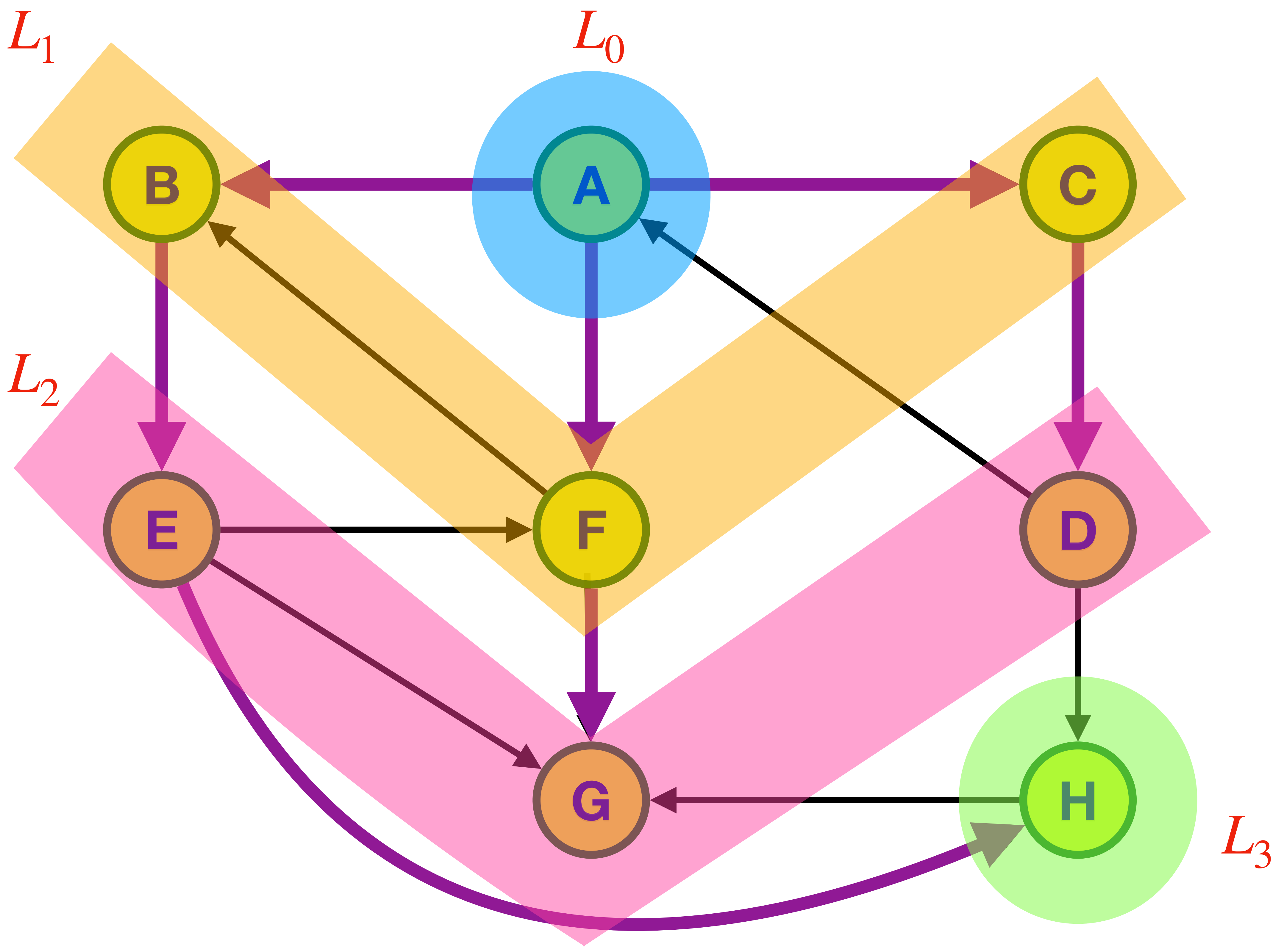

| | | | | | | |
|---|---|---|---|---|---|---|
| Q1: | A | | | | | |
| Q2: | B | C | F | | | |
| Q3: | C | F | E | | | |
| Q4: | F | E | D | | | |
| Q5: | E | D | G | | | |
| Q6: | D | G | H | | | |

# BFS: An example in directed graphs




| | | | | | | |
|---|---|---|---|---|---|---|
| Q1: | A | | | | | |
| Q2: | B | C | F | | | |
| Q3: | C | F | E | | | |
| Q4: | F | E | D | | | |
| Q5: | E | D | G | | | |
| Q6: | D | G | H | | | |
| Q7: | G | H | | | | |
| Q8: | H | | | | | |
| Q9: | | | | | | |

# BFS: An example in directed graphs




| Queue | | | | | | |
|---|---|---|---|---|---|---|
| Q1: | A | | | | | |
| Q2: | B | C | F | | | |
| Q3: | C | F | E | | | |
| Q4: | F | E | D | | | |
| Q5: | E | D | G | | | |
| Q6: | D | G | H | | | |
| Q7: | G | H | | | | |
| Q8: | H | | | | | |
| Q9: | | | | | | |

# BFS with distances

```
BFS(s):
    Mark all vertices as unvisited; for each v set dist(v) = ∞
    Initialize search tree T to be empty
    Mark vertex s as visited and set dist(s) = 0
    set Q to be the empty queue
    enqueue(s)
    while Q is non-empty do
          u = dequeue(Q)
          for each vertex v ∈ Adj(u) do
              if v is not visited do
                add edge (u,v) to T
                Mark v as visited, enqueue(v)
                and set dist(v) = dist(u) + 1
```

# Properties of BFS

## Undirected graphs

**Theorem:** *The following properties hold upon termination of BFS(s)*

- Search tree is the set of vertices in the connected component of $s$.

# Properties of BFS

## Undirected graphs

**Theorem:** *The following properties hold upon termination of BFS(s)*

- Search tree is the set of vertices in the connected component of $s$.
- If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$.

# Properties of BFS

## Undirected graphs

**Theorem:** *The following properties hold upon termination of BFS(s)*

- Search tree is the set of vertices in the connected component of $s$.
- If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$.
- For every vertex $u$, $\text{dist}(u)$ is the length of a shortest path (in terms of number of edges) from $s$ to $u$.

# Properties of BFS

## Undirected graphs

**Theorem:** *The following properties hold upon termination of BFS(s)*

- Search tree is the set of vertices in the connected component of $s$.
- If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$.
- For every vertex $u$, $\text{dist}(u)$ is the length of a shortest path (in terms of number of edges) from $s$ to $u$.
- If $u, v$ are in connected component of $s$ and $e = \{u, v\}$ is an edge of $G$, then $|\text{dist}(u) - \text{dist}(v)| \leq 1$.

# Properties of BFS

## Directed graphs

**Theorem:** *The following properties hold upon termination of BFS(s)*

- Search tree contains exactly the set of vertices reachable from $s$.

# Properties of BFS

## Directed graphs

**Theorem:** *The following properties hold upon termination of BFS(s)*

- Search tree contains exactly the set of vertices reachable from $s$.
- If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$.

# Properties of BFS

## Directed graphs

**Theorem:** *The following properties hold upon termination of BFS(s)*

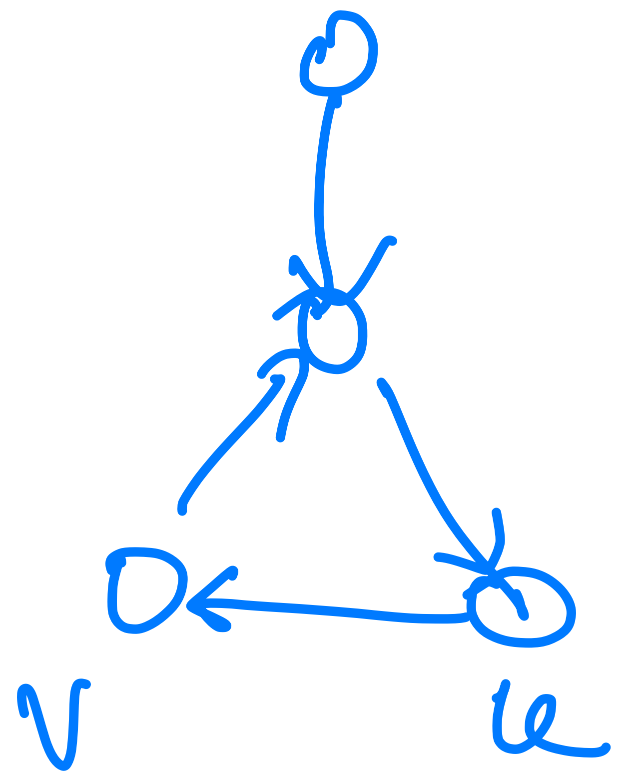
- Search tree contains exactly the set of vertices reachable from $s$.
- If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$.
- For every vertex $u$, $\text{dist}(u)$ is indeed the length of shortest path from $s$ to $u$.

# Properties of BFS

## Directed graphs

**Theorem:** *The following properties hold upon termination of BFS(s)*

- Search tree contains exactly the set of vertices reachable from $s$.
- If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$.
- For every vertex $u$, $\text{dist}(u)$ is indeed the length of shortest path from $s$ to $u$.
- If $u$ is reachable from $s$ and $e = (u, v)$ is an edge of $G$, then $\text{dist}(v) \leq 1 + \text{dist}(u)$.



# BFS with layers

- BFS is a simple algorithm but proving its properties formally is not straight forward

- Since BFS explores graph in increasing order of distance from source s, there is a simpler variant that makes BFS exploration transparent and easier to understand.

  - Given $G$ and $s \in V$, define $L_i = \{v \mid \text{dist}(s, v) = i\}$.

  - Then $L_0 = \{s\}$

# BFS with layers

- BFS is a simple algorithm but proving its properties formally is not straight forward
- Since BFS explores graph in increasing order of distance from source s, there is a simpler variant that makes BFS exploration transparent and easier to understand.
  - Given $G$ and $s \in V$, define $L_i = \{v \mid \text{dist}(s, v) = i\}$.
  - Then $L_0 = \{s\}$
  - And $L_k$ can be found from $L_{k-1}$ for $k \geq 1$ inductively.

# BFS with layers

```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
        initialize L_{i+1} to be an empty list
        for each u in L_i do
            for each edge (u, v) ∈ Adj(u) do
            if v is not visited
                mark v as visited
                add (u, v) to tree T
                add v to L_{i+1}
        i = i + 1
```

# BFS with layers

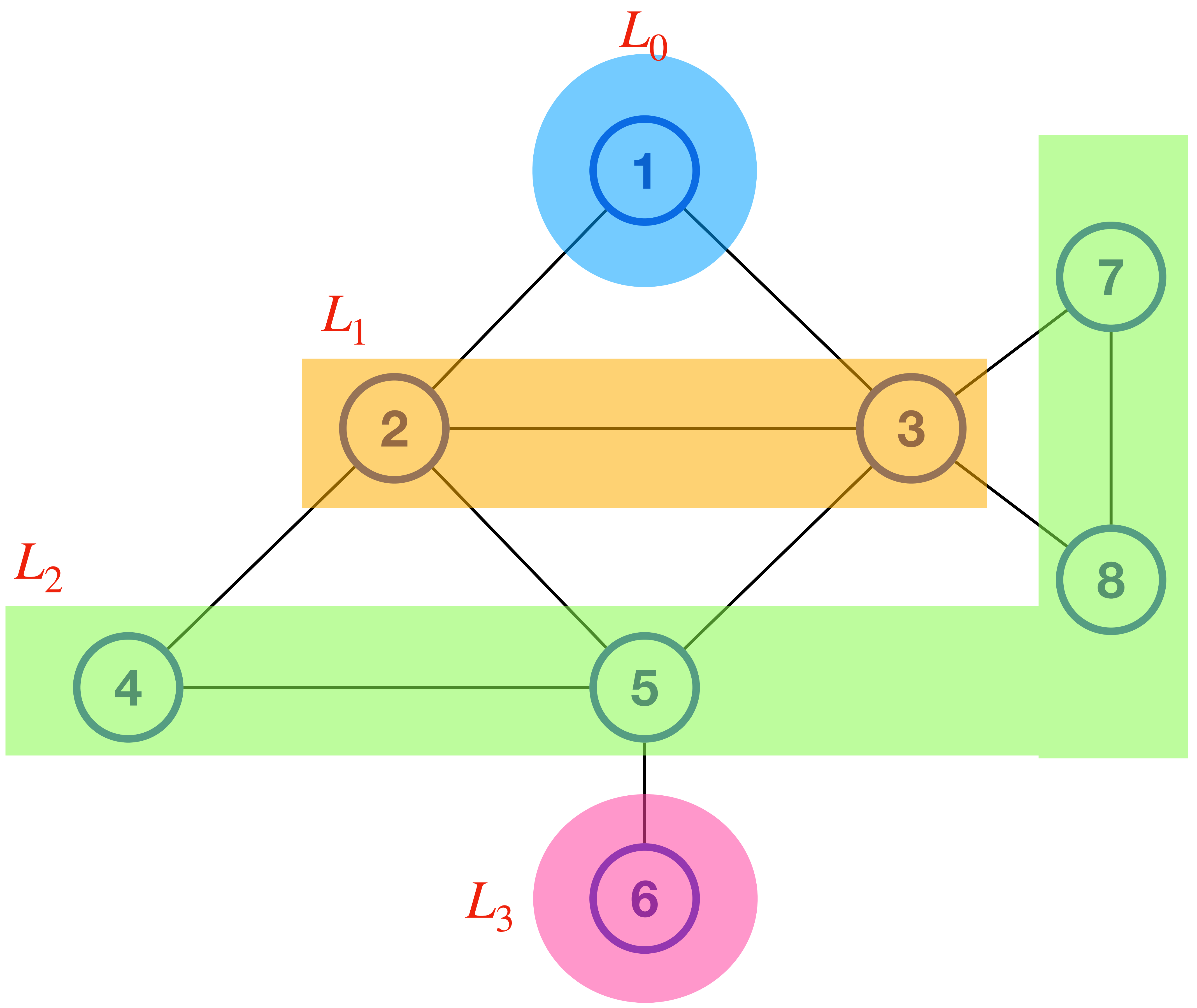
```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
          initialize L_{i+1} to be an empty list
          for each u in L_i do
              for each edge (u, v) ∈ Adj(u) do
              if v is not visited
                 mark v as visited
                 add (u, v) to tree T
                 add v to L_{i+1}
          i = i + 1
```

*Running time:* $O(n+m)$

# BFS with layers

## Example - undirected

- Layer 0: 1
- Layer 1: 2, 3
- Layer 2: 4, 5, 7, 8
- Layer 3: 6



# BFS with layers: undirected graph

## Properties

- BFSLayers(s) outputs a BFS tree
- $L_i$ is the set of vertices at distance exactly $i$ from $s$.
- If $G$ is undirected, each edge $e = \{u, v\}$ is one of three types:

# BFS with layers: undirected graph

## Properties

- BFSLayers(s) outputs a BFS tree
- $L_i$ is the set of vertices at distance exactly $i$ from $s$.
- If $G$ is undirected, each edge $e = \{u, v\}$ is one of three types:
  - _tree_ edge between two _consecutive_ layers

# BFS with layers: undirected graph

## Properties

- BFSLayers(s) outputs a BFS tree
- $L_i$ is the set of vertices at distance exactly $i$ from $s$.
- If $G$ is undirected, each edge $e = \{u, v\}$ is one of three types:
  - *tree* edge between two *consecutive* layers
  - non-tree *forward/backward* edge between two consecutive layers

# BFS with layers: undirected graph

## Properties

- BFSLayers(s) outputs a BFS tree
- $L_i$ is the set of vertices at distance exactly $i$ from $s$.
- If $G$ is undirected, each edge $e = \{u, v\}$ is one of three types:
    - *tree* edge between two *consecutive* layers
- non-tree *forward/backward* edge between two consecutive layers
- non-tree *cross-edge* with both $u, v$ in same layer

non-tree inter-layer edge.

intra-layer

# BFS with layers: undirected graph

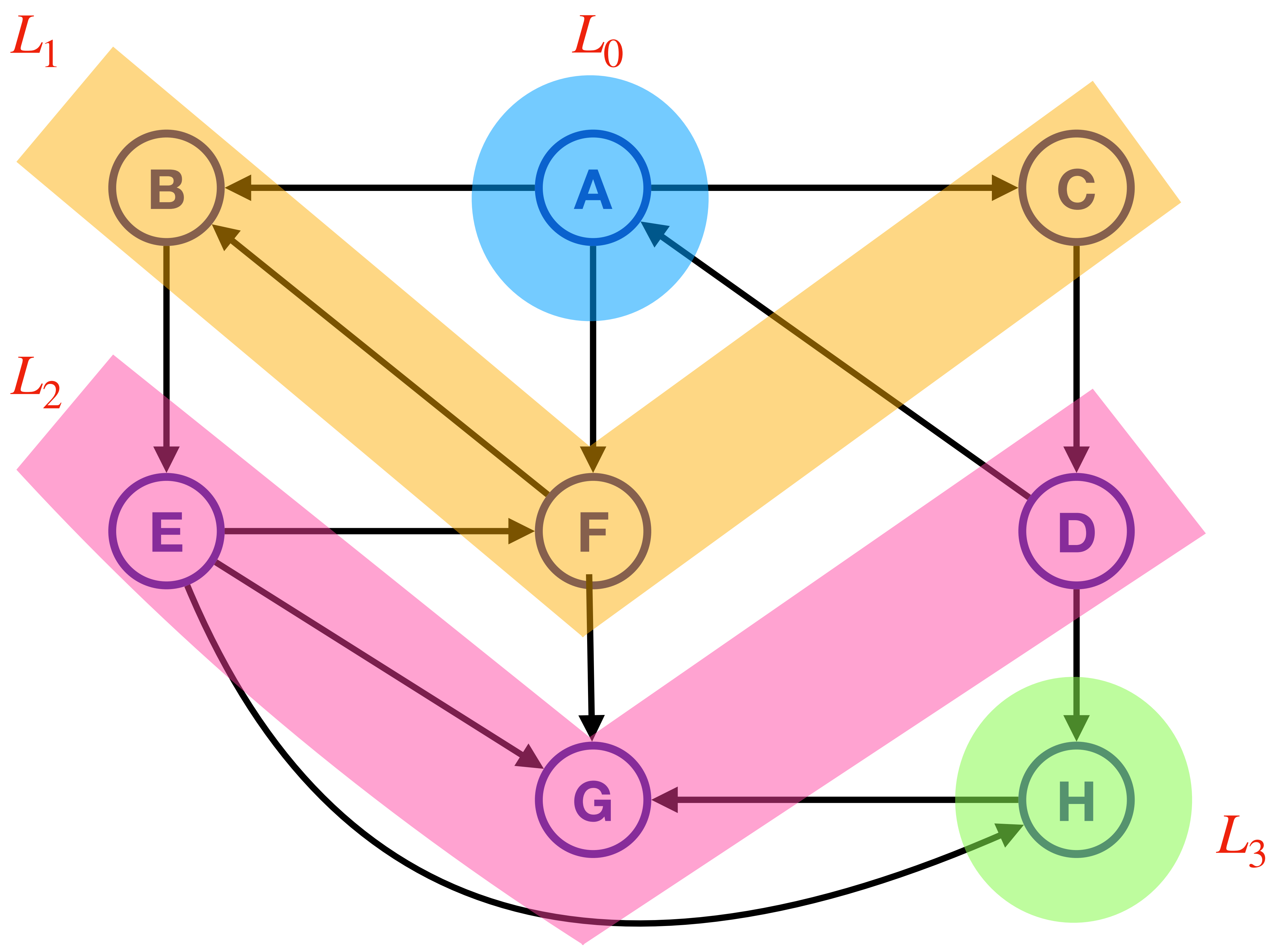
## Properties

- BFSLayers(s) outputs a BFS tree
- $L_i$ is the set of vertices at distance exactly $i$ from $s$.
- If $G$ is undirected, each edge $e = \{u, v\}$ is one of three types:
  - *tree* edge between two *consecutive* layers
  - non-tree *forward/backward* edge between two consecutive layers
  - non-tree *cross-edge* with both $u, v$ in same layer
- Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers!

# BFS with layers

## Example - directed

- Layer 0: A
- Layer 1: B, F, C
- Layer 2: E, G, D
- Layer 3: H



# BFS with layers: directed graph

## Properties

**Proposition:** *The following properties hold on termination of BFS(s) if G is directed.*

- Each edge $e = \{u, v\}$ is one of four types:
  - A *tree* edge between consecutive layers, $u \in L_i,\ v \in L_{i+1}$ for some $i \geq 0$
  - A non-tree *forward* edge between consecutive layers
  - A non-tree *backward* edge

make sense

# BFS with layers: directed graph

## Properties

**Proposition:** *The following properties hold on termination of BFS(s) if G is directed.*

- Each edge $e = \{u, v\}$ is one of four types:
    - A *tree* edge between consecutive layers, $u \in L_i,\ v \in L_{i+1}$ for some $i \geq 0$
    - A non-tree *forward* edge between consecutive layers
    - A non-tree *backward* edge
    - A *cross-edge* with both $u, v$ in same layer

# Shortest path problems

## Description

Given graph $G = (V, E)$ with associated edge lengths (or costs), denote for an edge $e = uv$ the quantity $l(e) = l(uv)$ as its length or cost.

# Shortest path problems

## Description

Given graph $G = (V, E)$ with associated edge lengths (or costs), denote for an edge $e = uv$ the quantity $l(e) = l(uv)$ as its length or cost.

- Given nodes $s, t$ find shortest path (in terms of summed lengths/costs) from $s$ to $t$ . (SSPP)

single source shortest paths

- Given node $s$ find shortest path from $s$ to all other nodes (SSSP)

# Shortest path problems

## Description

Given graph $G = (V, E)$ with associated edge lengths (or costs), denote for an edge $e = uv$ the quantity $l(e) = l(uv)$ as its length or cost.

- Given nodes $s, t$ find shortest path (in terms of summed lengths/costs) from $s$ to $t$ . (SSPP)
- Given node $s$ find shortest path from $s$ to all other nodes (SSSP)
- Find shortest paths between all pairs of nodes (APSP)

# Shortest walks vs. paths

- A path is a sequence of ***distinct*** vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.
- A ~~path~~ walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.

# Shortest walks vs. paths

- A path is a sequence of ***distinct*** vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.
- A ~~path~~ walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.
- Finding walks is often easier than finding paths (concatenating two walks gives a walk, while concatenating two paths may not give a path).

# Shortest walks vs. paths

- A path is a sequence of ***distinct*** vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.

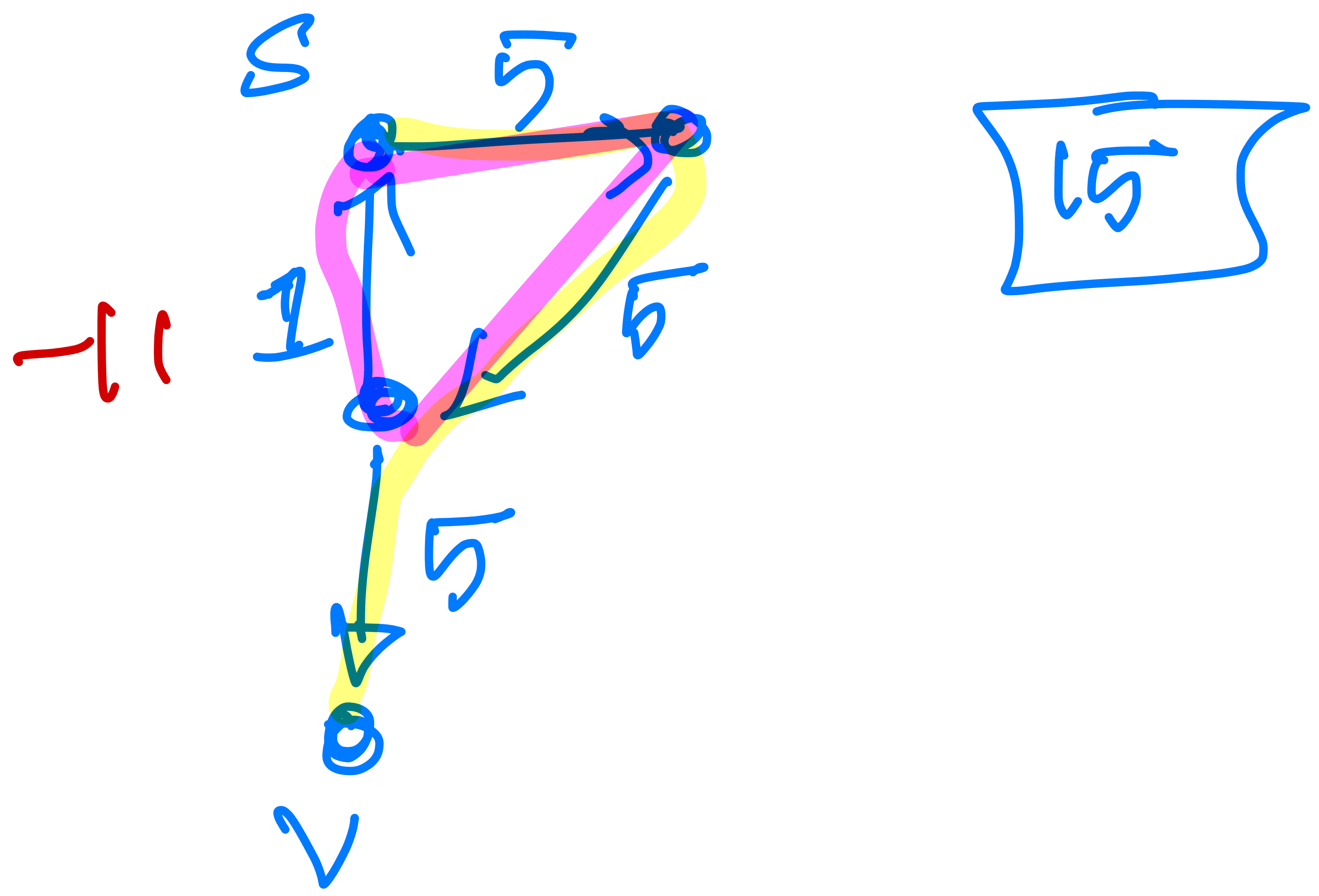
- A ~~path~~ walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.

- Finding walks is often easier than finding paths (concatenating two walks gives a walk, while concatenating two paths may not give a path).

- For edges with non-negative weights/lengths, finding the shortest walk is the same as finding the shortest $s \to t$ path.

# Single-source shortest paths

## Assumption: non-negative edge lengths




Single-source shortest path problems (SSSPs)

- Input: A (undirected or directed) graph $G = (V, E)$ with *non-negative edge lengths.* For edge $e = (u, v)$, $l(e) = l(u, v)$ is its length.

# Single-source shortest paths

## Assumption: non-negative edge lengths

Single-source shortest path problems (SSSPs)

- Input: A (undirected or directed) graph $G = (V, E)$ with *non-negative edge lengths.* For edge $e = (u, v)$, $l(e) = l(u, v)$ is its length.

- Given nodes $s, t$ find shortest path from $s$ to $t$.

# Single-source shortest paths

## Assumption: non-negative edge lengths

Single-source shortest path problems (SSSPs)

- Input: A (undirected or directed) graph $G = (V, E)$ with *non-negative edge lengths.* For edge $e = (u, v)$, $l(e) = l(u, v)$ is its length.

- Given nodes $s, t$ find shortest path from $s$ to $t$.

- Given node $s$ find shortest path from $s$ to all other nodes.

# Single-source shortest paths

## Assumption: non-negative edge lengths

APSP next week
Bellman-Ford
Floyd-Warshall.

Single-source shortest path problems (SSSPs)

- Input: A (undirected or directed) graph $G = (V, E)$ with *non-negative edge lengths.* For edge $e = (u, v)$, $l(e) = l(u, v)$ is its length.

- Given nodes $s, t$ find shortest path from $s$ to $t$. ←

- Given node $s$ find shortest path from $s$ to all other nodes. ← Dijkstra's algorithm.

- Restrict attention to directed graphs

# Single-source shortest paths

## Assumption: non-negative edge lengths

- Undirected graph problem can be reduced to directed graph problem - how?

  - Given undirected graph $G$, create a new directed graph $G'$ by replacing each edge $\{u, v\}$ in $G$ by $(u, v)$ and $(v, u)$ in $G'$ .

# Single-source shortest paths

## Assumption: non-negative edge lengths

- Undirected graph problem can be reduced to directed graph problem - how?
  - Given undirected graph $G$, create a new directed graph $G'$ by replacing each edge $\{u, v\}$ in $G$ by $(u, v)$ and $(v, u)$ in $G'$ .
  - set $l(u, v) = l(v, u) = l(\{u, v\})$

# Single-source shortest paths

## Assumption: non-negative edge lengths

- Undirected graph problem can be reduced to directed graph problem - how?
  - Given undirected graph $G$, create a new directed graph $G'$ by replacing each edge $\{u, v\}$ in $G$ by $(u, v)$ and $(v, u)$ in $G'$.
  - set $l(u, v) = l(v, u) = l(\{u, v\})$
  - Exercise: show reduction works. Relies on non-negativity!

# Shortest path in the weighted case using BFS

# Single-source shortest paths via BFS

- **Special case:** All edge lengths are 1. ← edge weights can be made this
  - Run BFS(s) to get shortest path distances from s to all other nodes.
  - O(m + n) time algorithm.

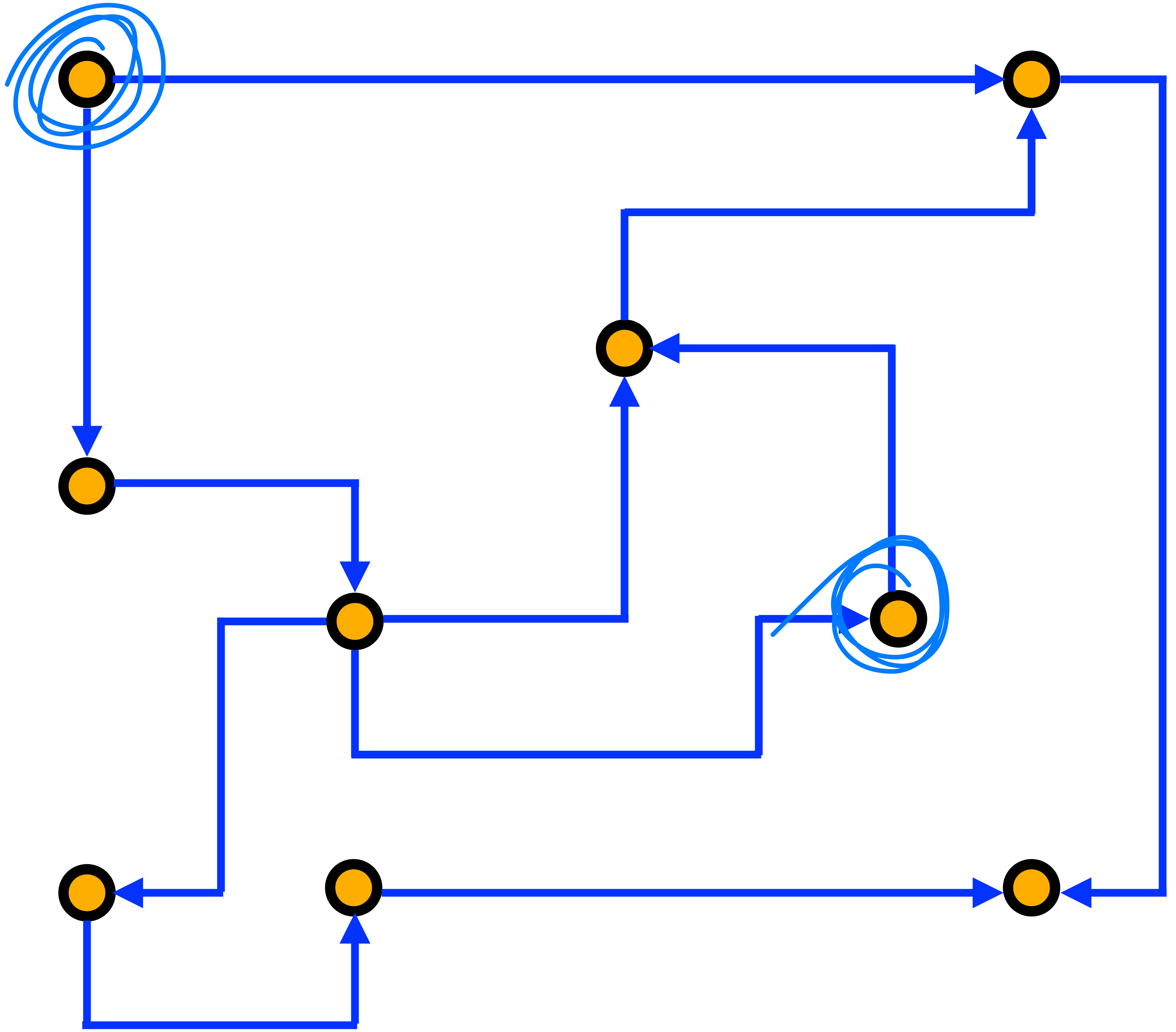
# Single-source shortest paths via BFS

- **Special case:** All edge lengths are 1.
    - Run BFS(s) to get shortest path distances from s to all other nodes.
    - O(m + n) time algorithm.
- **Special case:** Suppose $l(e)$ is an integer for all $e$? Can we use BFS? Reduce to unit edge-length problem by placing $l(e) - 1$ dummy nodes on $e$.

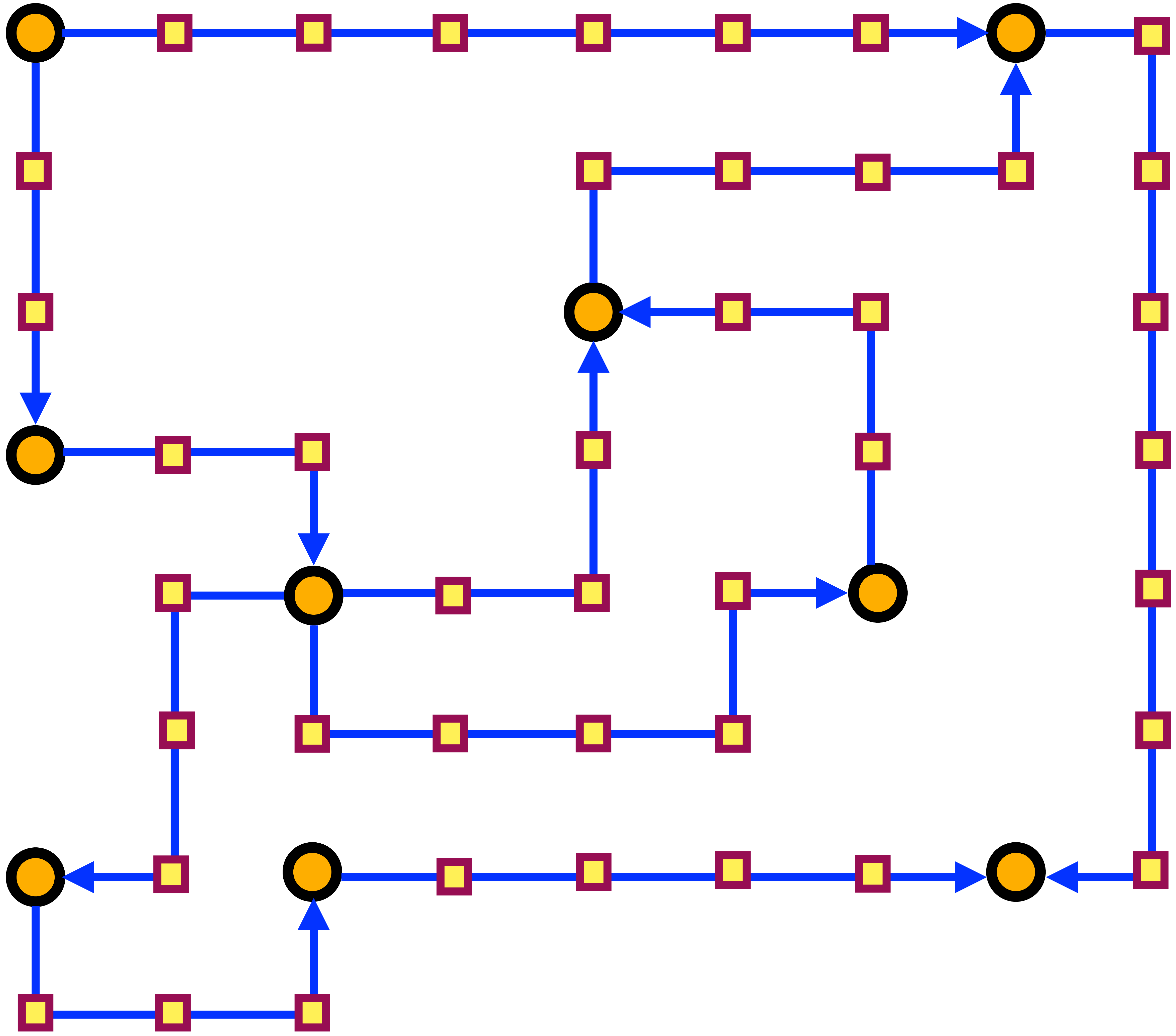
# Single-source shortest paths via BFS

- **Special case:** All edge lengths are 1.
  - Run BFS(s) to get shortest path distances from s to all other nodes.
  - O(m + n) time algorithm.
- **Special case:** Suppose $l(e)$ is an integer for all $e$? Can we use BFS? Reduce to unit edge-length problem by placing $l(e) - 1$ dummy nodes on $e$.
- Let $L = \max_e l(e)$. New graph has $O(mL)$ edges and $O(mL + n)$ nodes. BFS takes $O(mL + n)$ time. Not efficient if $L$ is large.

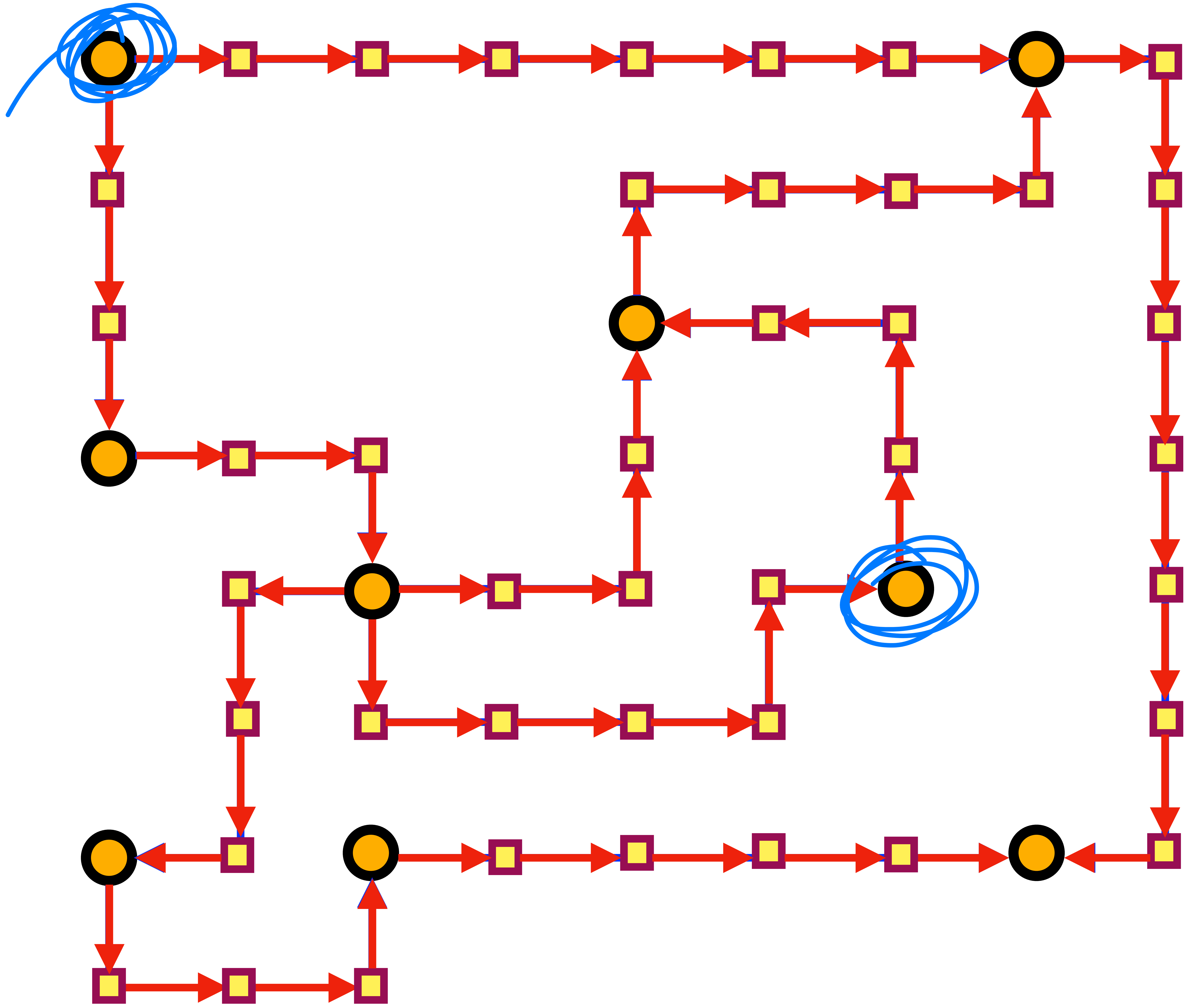
# Example of edge refinement



# Example of edge refinement



# Example of edge refinement



# You can not shortcut a shortest path

## Lemma (… also goes by Bellman's principle of optimality)

Let $G$ be a directed graph with *non-negative* edge lengths. Suppose that

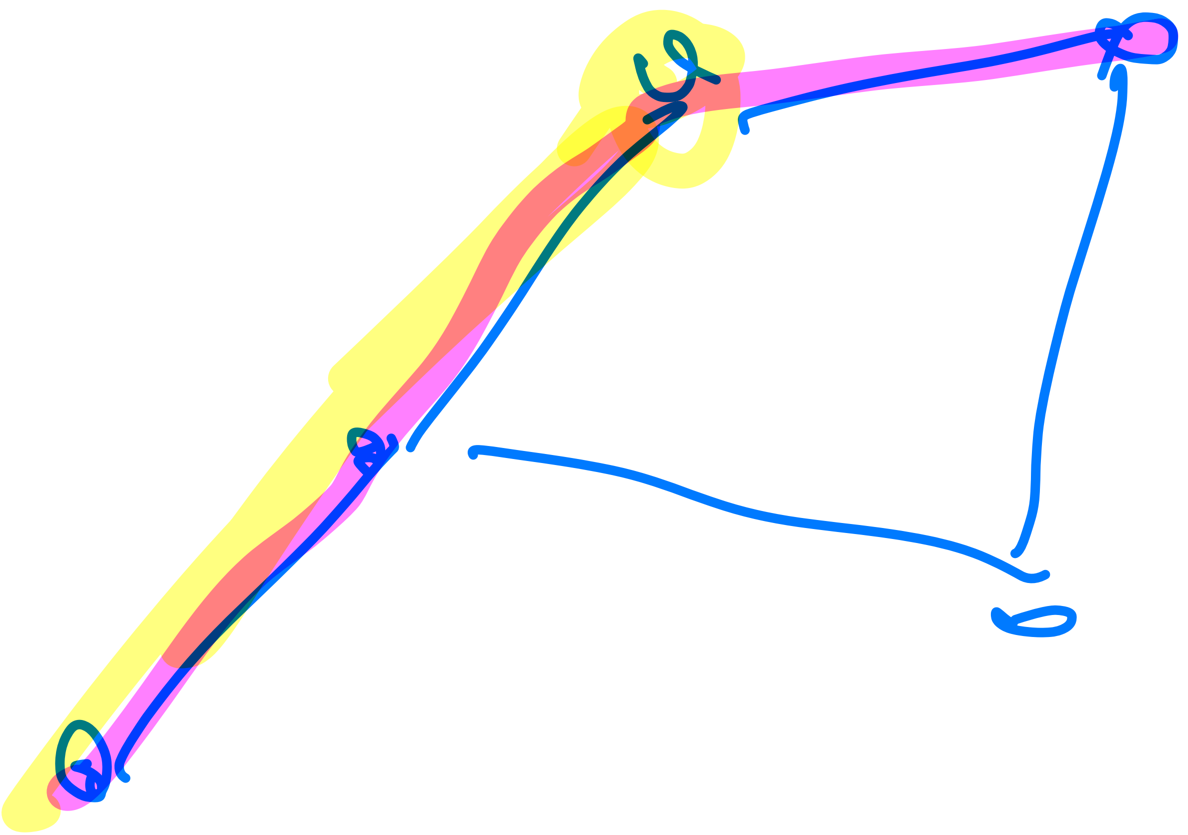
$$p = v_0 \to v_1 \to v_2 \to \ldots \to v_k$$

is the shortest path from $v_0$ to $v_k$.

# You can not shortcut a shortest path

## Lemma (… also goes by Bellman’s principle of optimality)

Let $G$ be a directed graph with *non-negative* edge lengths. Suppose that



$$p = v_0 \to v_1 \to v_2 \to \ldots \to v_k$$

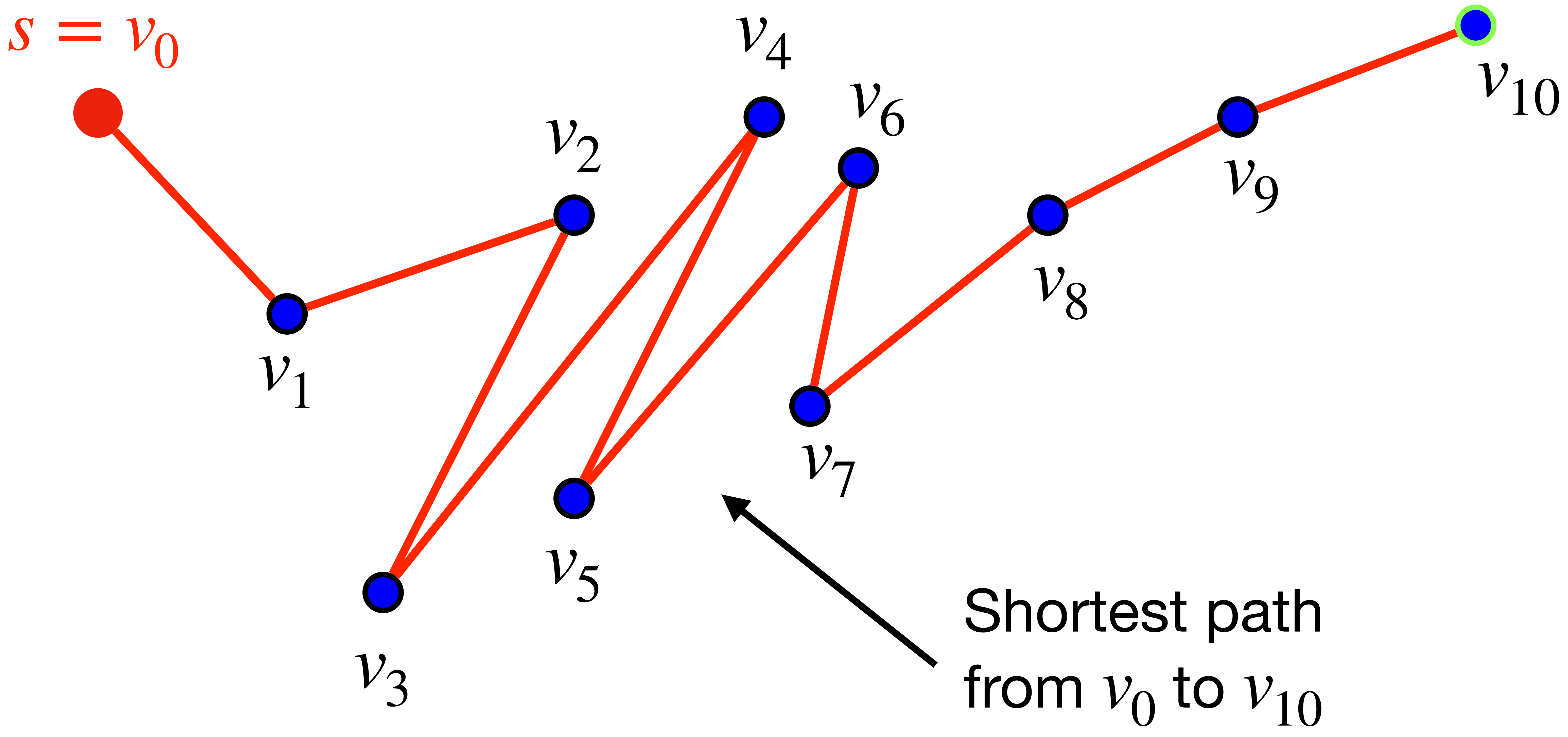
is the shortest path from $v_0$ to $v_k$.

Then for any $0 \leq i < j \leq k$ we have that

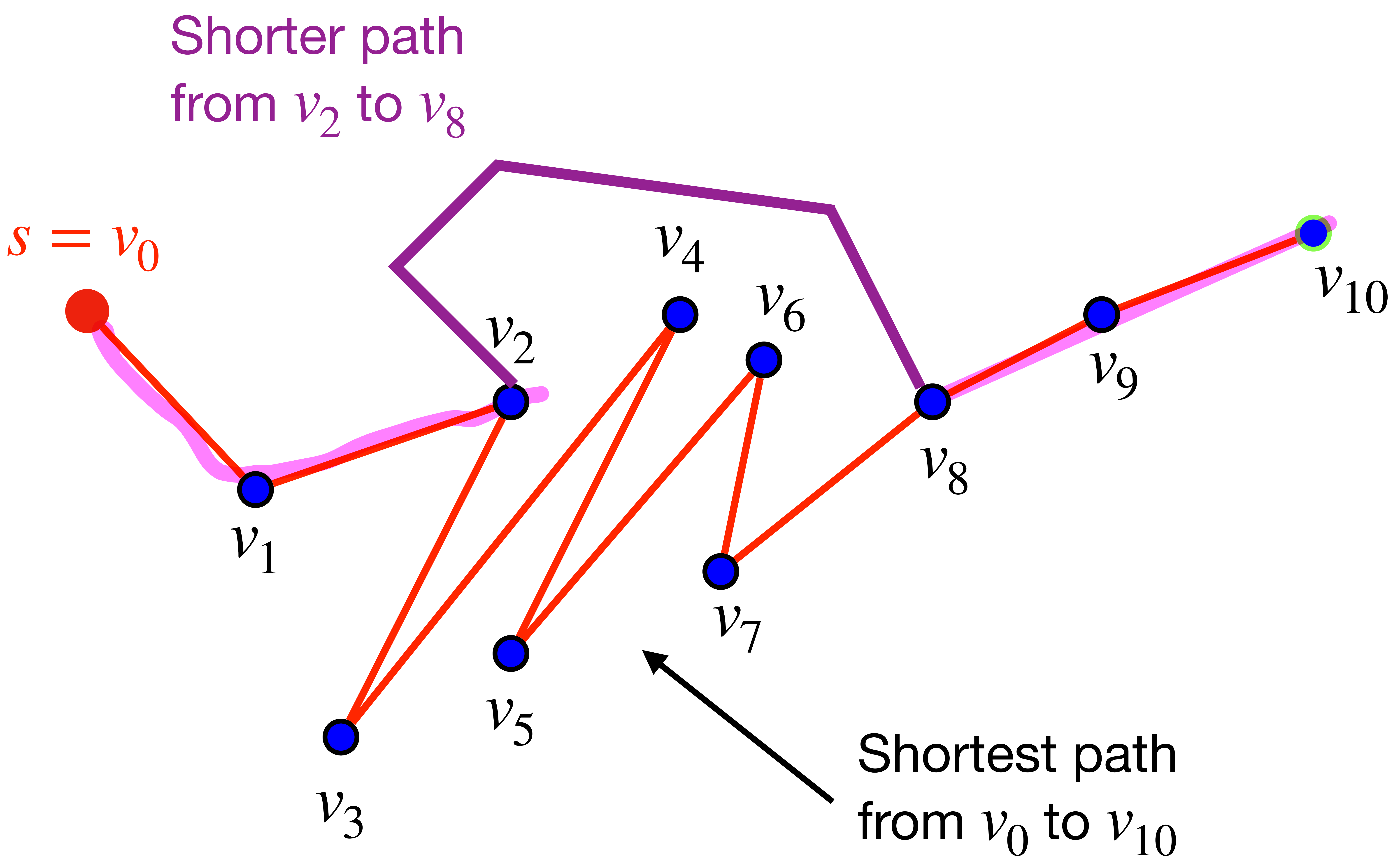
$$v_i \to v_{i+1} \to \ldots \to v_j$$

is the shortest path from $v_i$ to $v_j$.

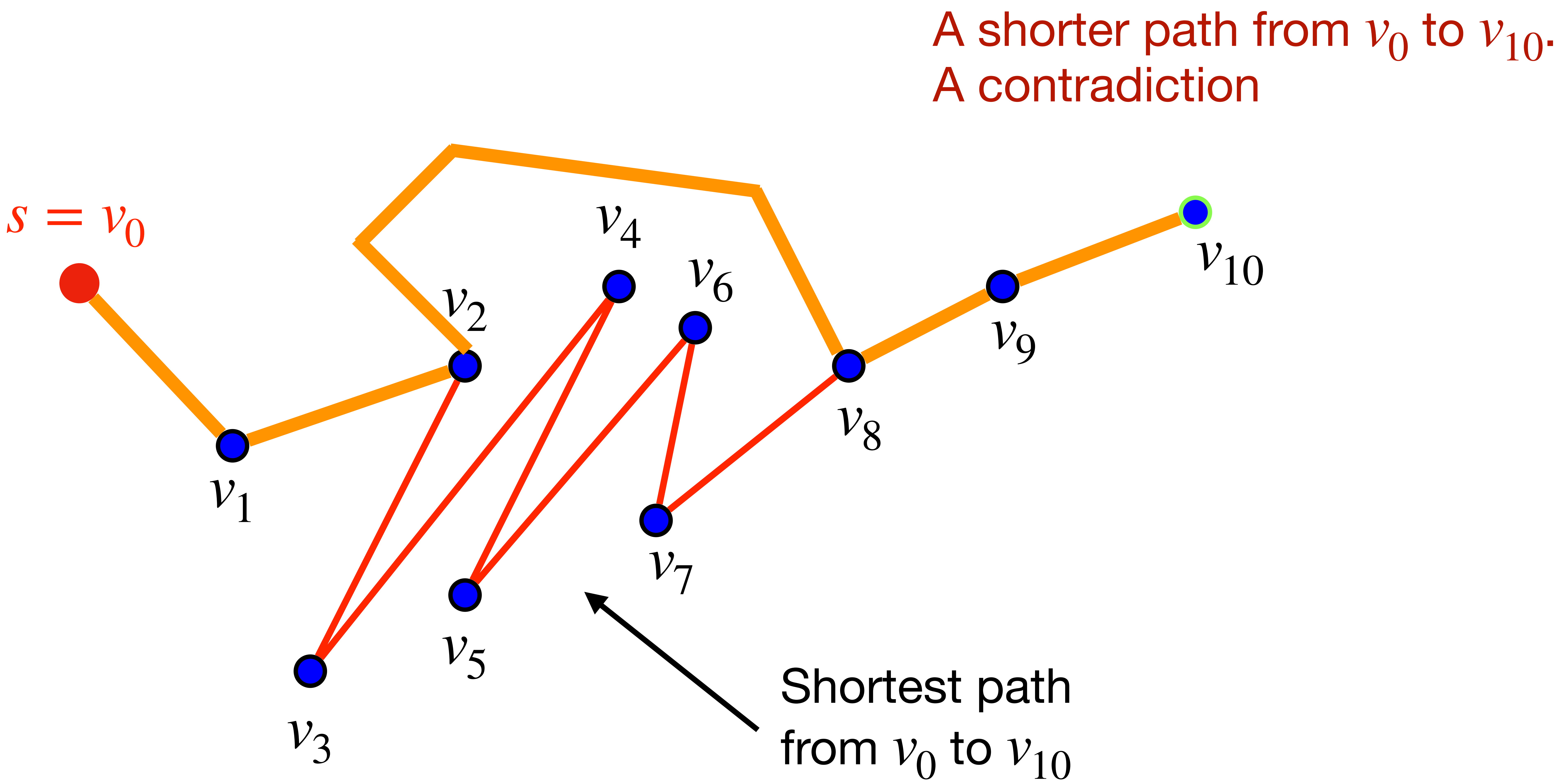
# A proof by picture



# A proof by picture



# A proof by picture



# What we really need…

## Stated in terms of distance

Let $G$ be a directed graph with non-negative edge lengths and let $\text{dist}(s, v)$ denote the length of the shortest path from $s$ to $v$.

# What we really need…

## Stated in terms of distance

Let $G$ be a directed graph with non-negative edge lengths and let $\text{dist}(s, v)$ denote the length of the shortest path from $s$ to $v$.

If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$

is the shortest path from $s = v_0$ to $v_k$ then for any $0 \leq i < j \leq k$ we have that

# What we really need…

## Stated in terms of distance

Let $G$ be a directed graph with non-negative edge lengths and let $\text{dist}(s, v)$ denote the length of the shortest path from $s$ to $v$.

If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$

is the shortest path from $s = v_0$ to $v_k$ then for any $0 \leq i \leq k$ we have that

$s = v_0 \to v_1 \to v_2 \ldots \to v_i$ is shortest path from $s$ to $v_i$ and

$$\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$$

# Find the $i^{th}$ closest vertex

## A basic strategy

Explore vertices in increasing order of distance from $s$: (For simplicity, assume that nodes are at different distances from $s$ and that no edge has zero length)

# Find the $i^{th}$ closest vertex

## A basic strategy

"settled" nodes

Explore vertices in increasing order of distance from $s$: (For simplicity, assume that nodes are at different distances from $s$ and that no edge has zero length)

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = {s},
    for i = 2 to |V| do
        (* Invariant: X contains the i − 1 closest nodes to s *)
        Among nodes in V\X, find the node v that is the
        i^th closest to s
        Update dist(s, v)
        X = X ∪ {v}
```

# Find the $i^{th}$ closest vertex

## A basic strategy

Explore vertices in increasing order of distance from $s$: (For simplicity, assume that nodes are at different distances from $s$ and that no edge has zero length)

Initialize for each node $v$, $dist(s, v) = \infty$
Initialize $X = \{s\}$,
    **for** $i = 2$ to $|V|$ **do**
        (* Invariant: X contains the $i - 1$ closest nodes to s *)
        Among nodes in $V \backslash X$, find the node $v$ that is the $i^{th}$ closest to $s$
        Update $dist(s, v)$
        $X = X \cup \{v\}$

How can we implement the step in the for loop?

# Finding the $i^{th}$ closest node

## What we have …

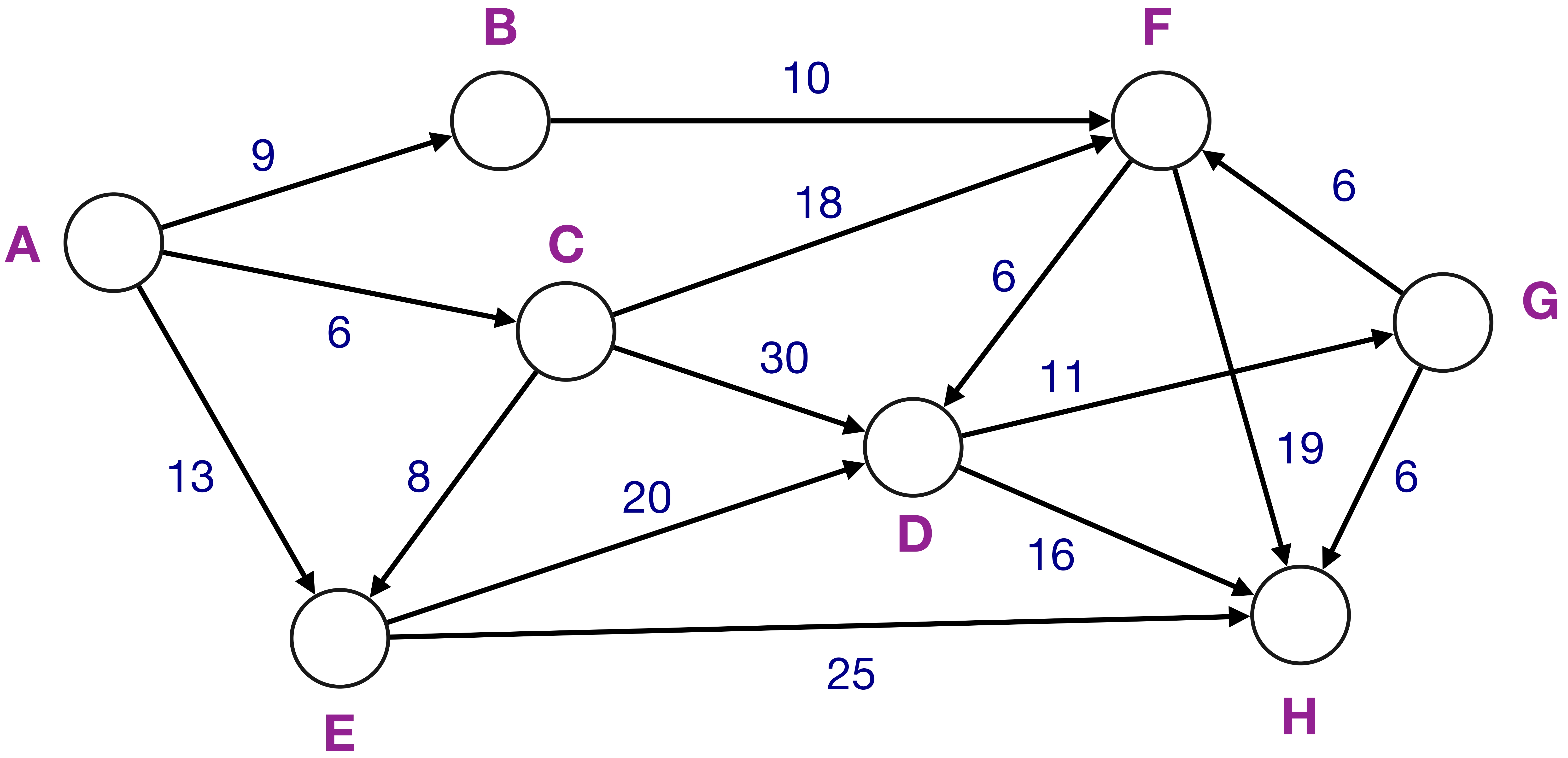
- $X$ contains the $i-1$ closest nodes to $s$
- Want to find the $i^{th}$ closest node from $V \backslash X$.

What do we know about the $i^{th}$ closest node?

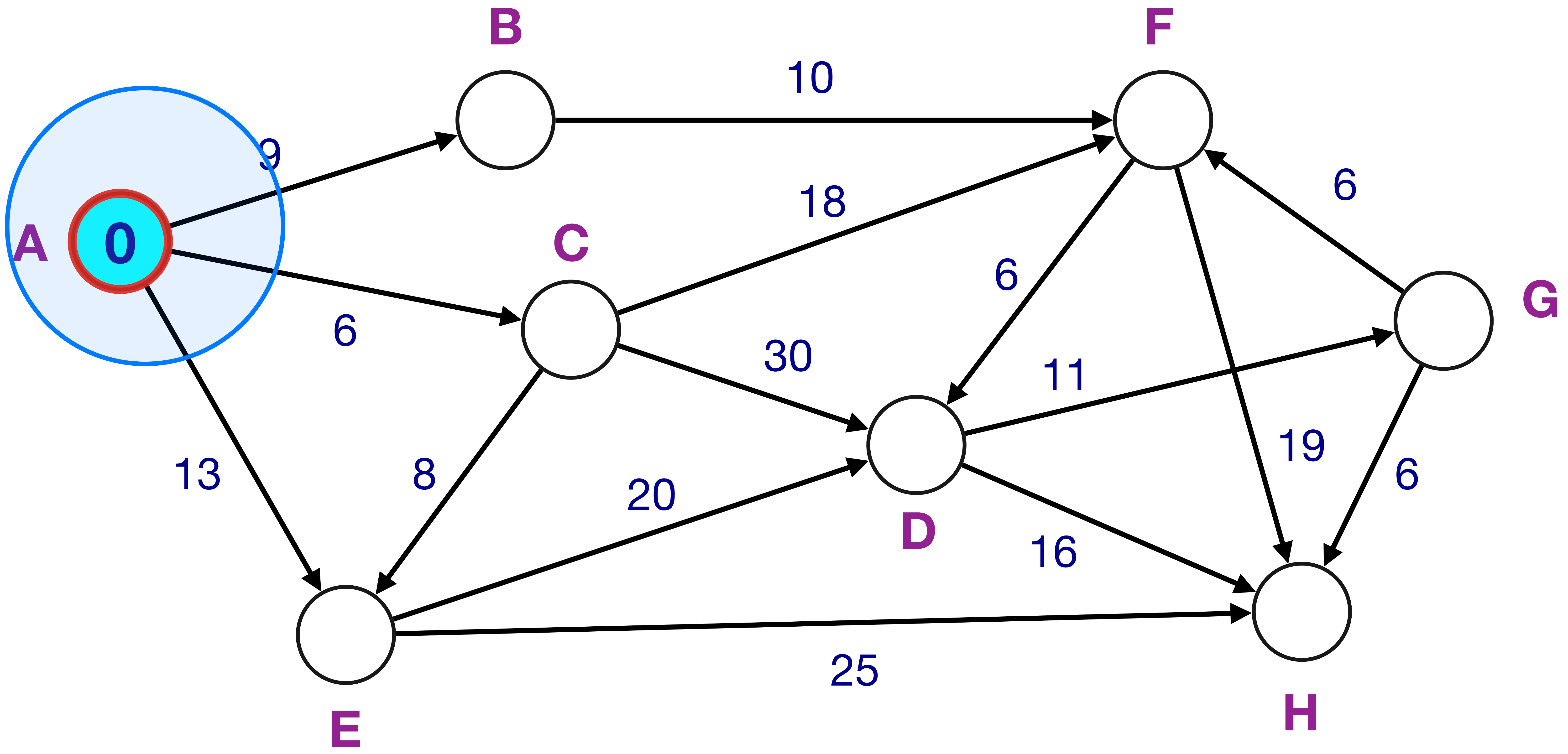
**Claim:** Let $P$ be a shortest path from $s$ to $v$ where $v$ is the $i^{th}$ closest node. Then, all intermediate nodes in $P$ belong to $X$.

**Proof:** If $P$ had an intermediate node $u$ not in $X$ then $u$ will be closer to $s$ than $v$. Implies $v$ is **not** the $i^{th}$closest node to $s$ - recall that $X$ already has the $i-1$ closest nodes!

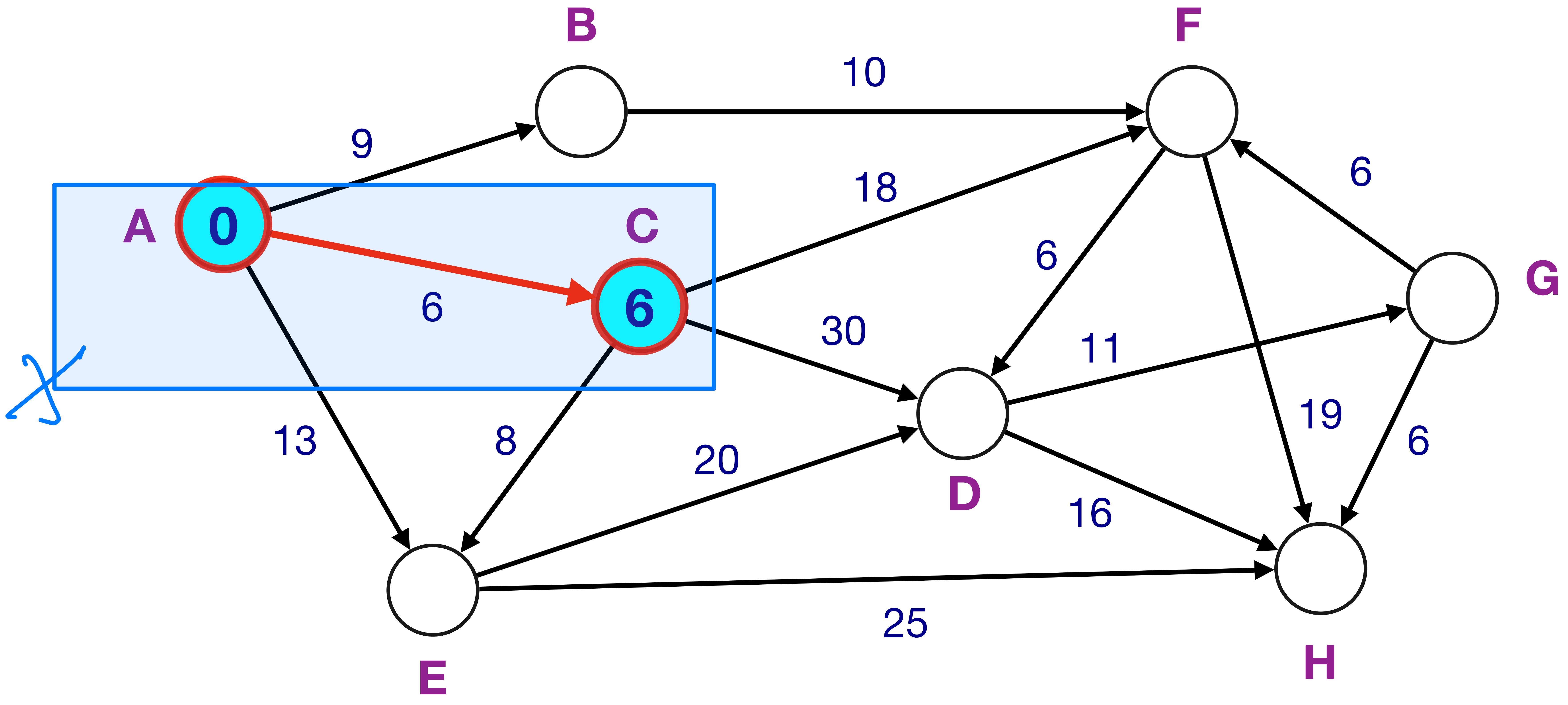
# Finding the $i^{th}$ closest node



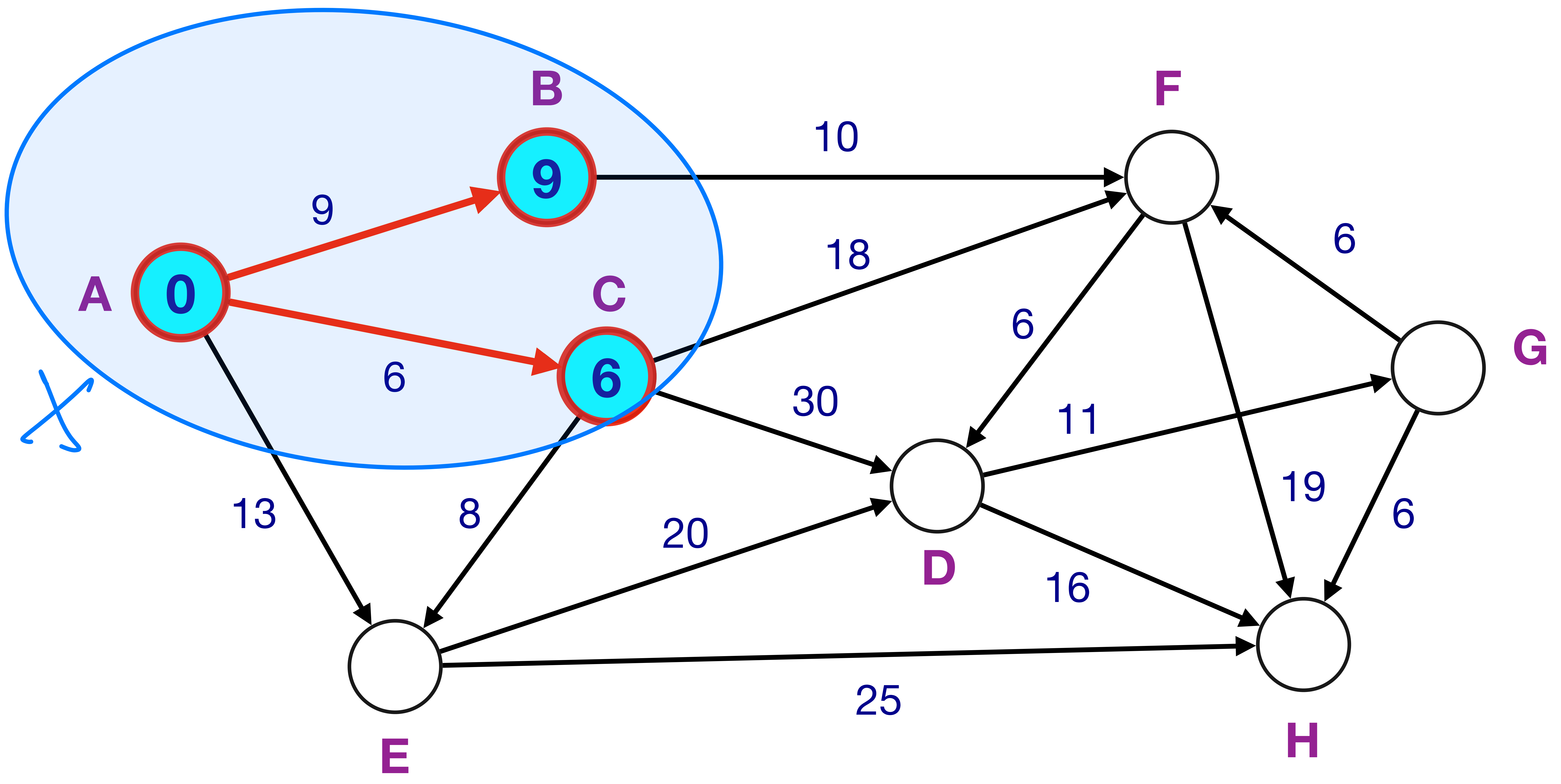
# Finding the $i^{th}$ closest node



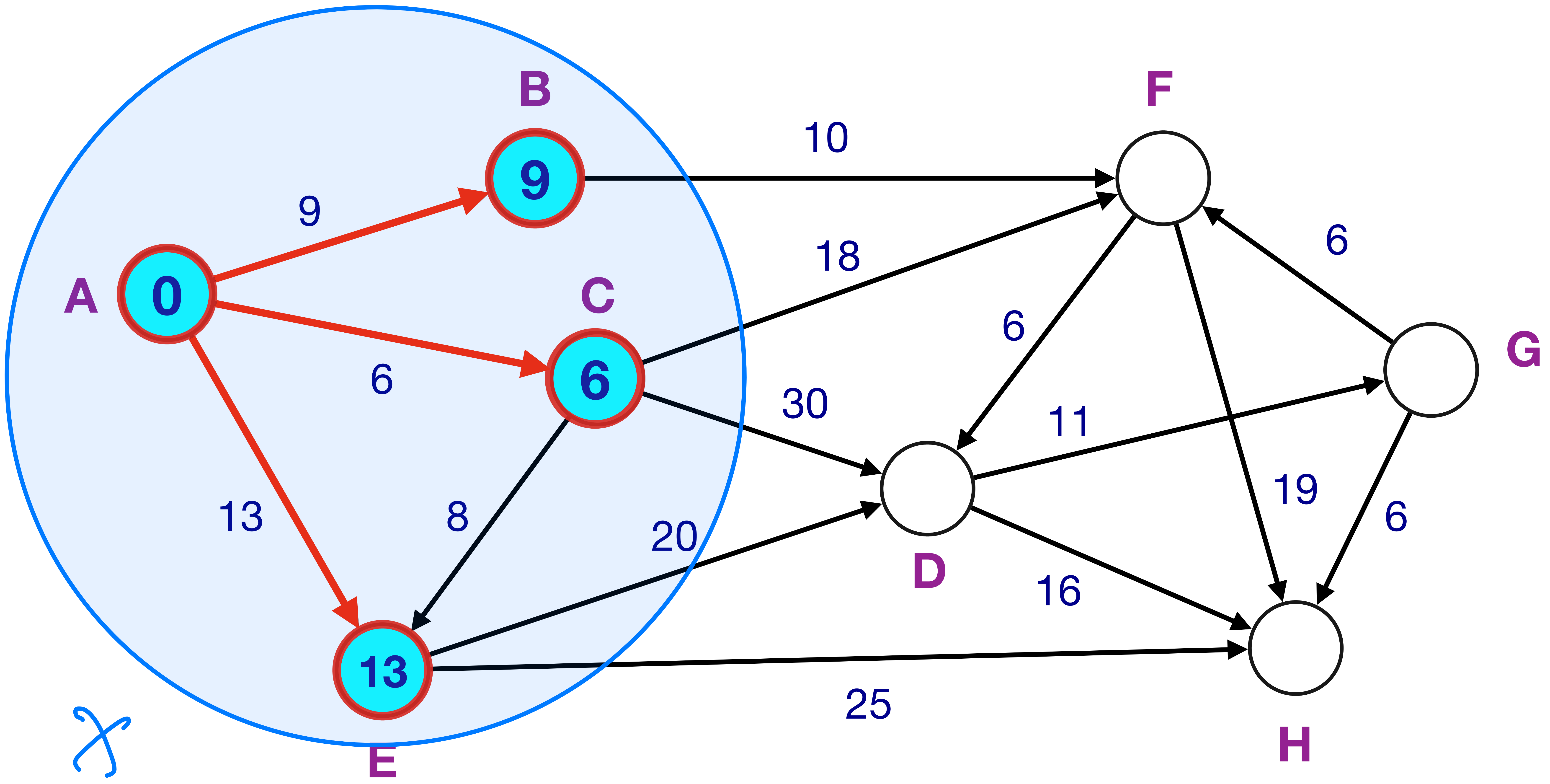
# Finding the $i^{th}$ closest node



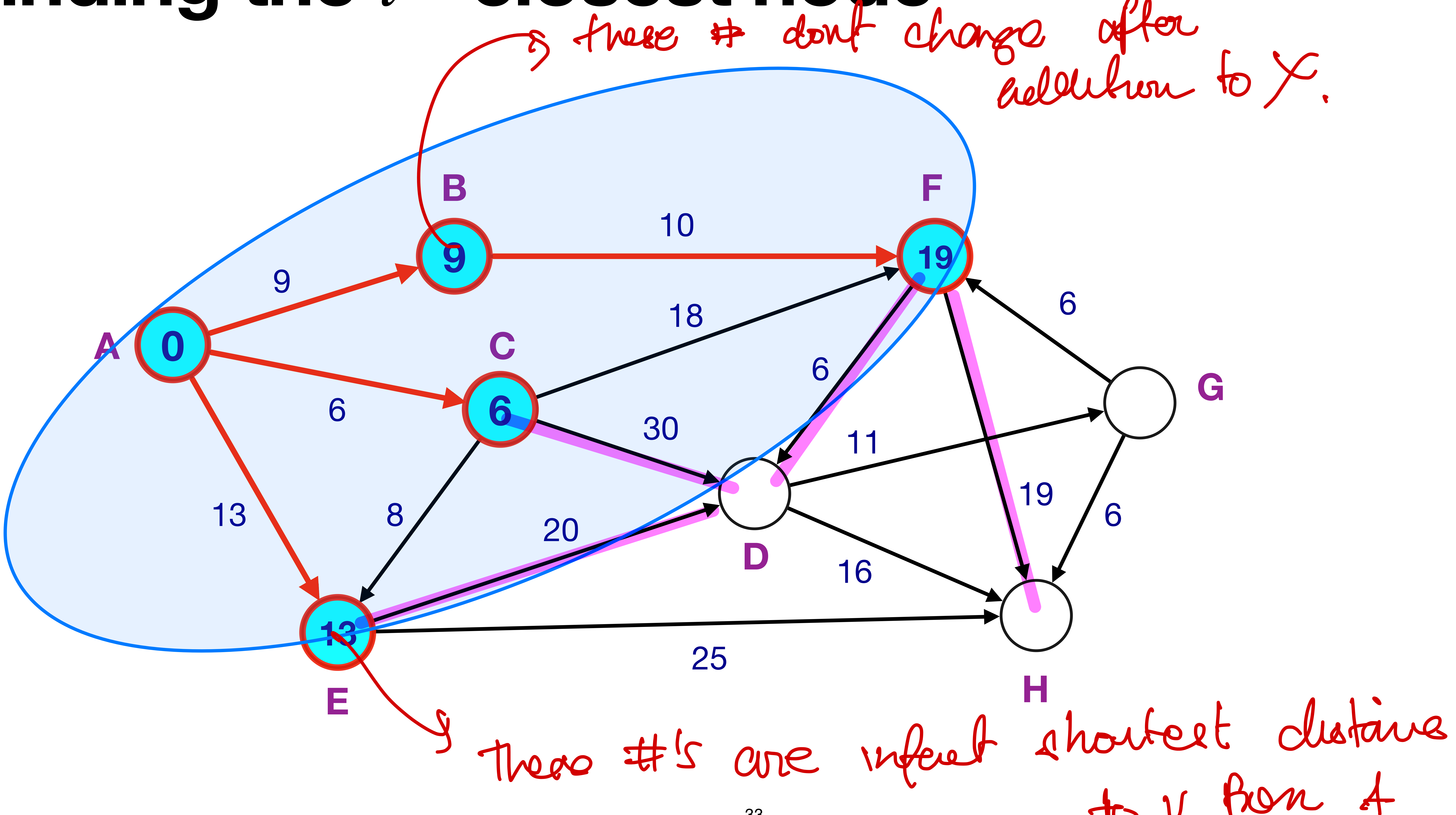
# Finding the $i^{th}$ closest node



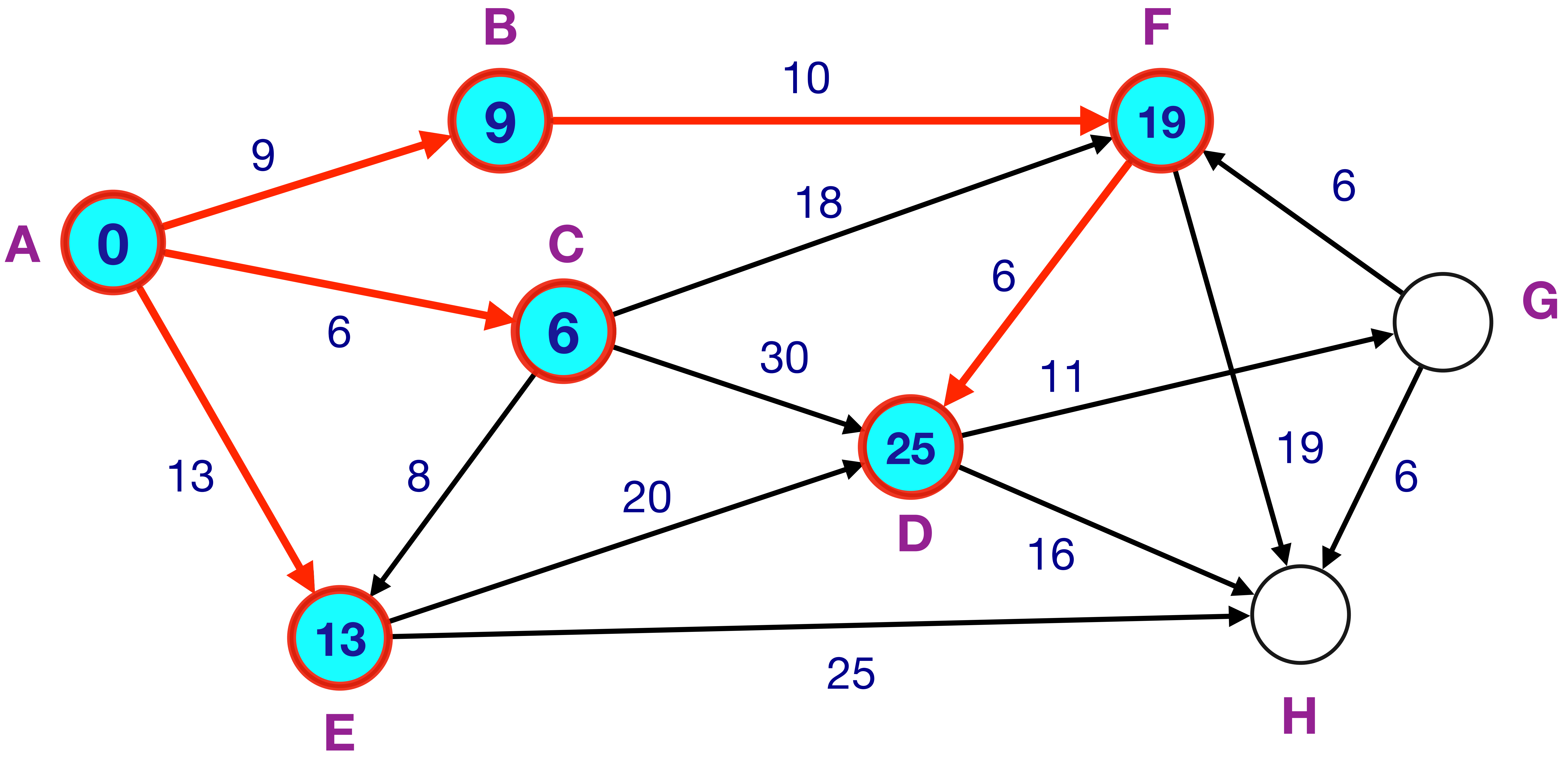
# Finding the $i^{th}$ closest node



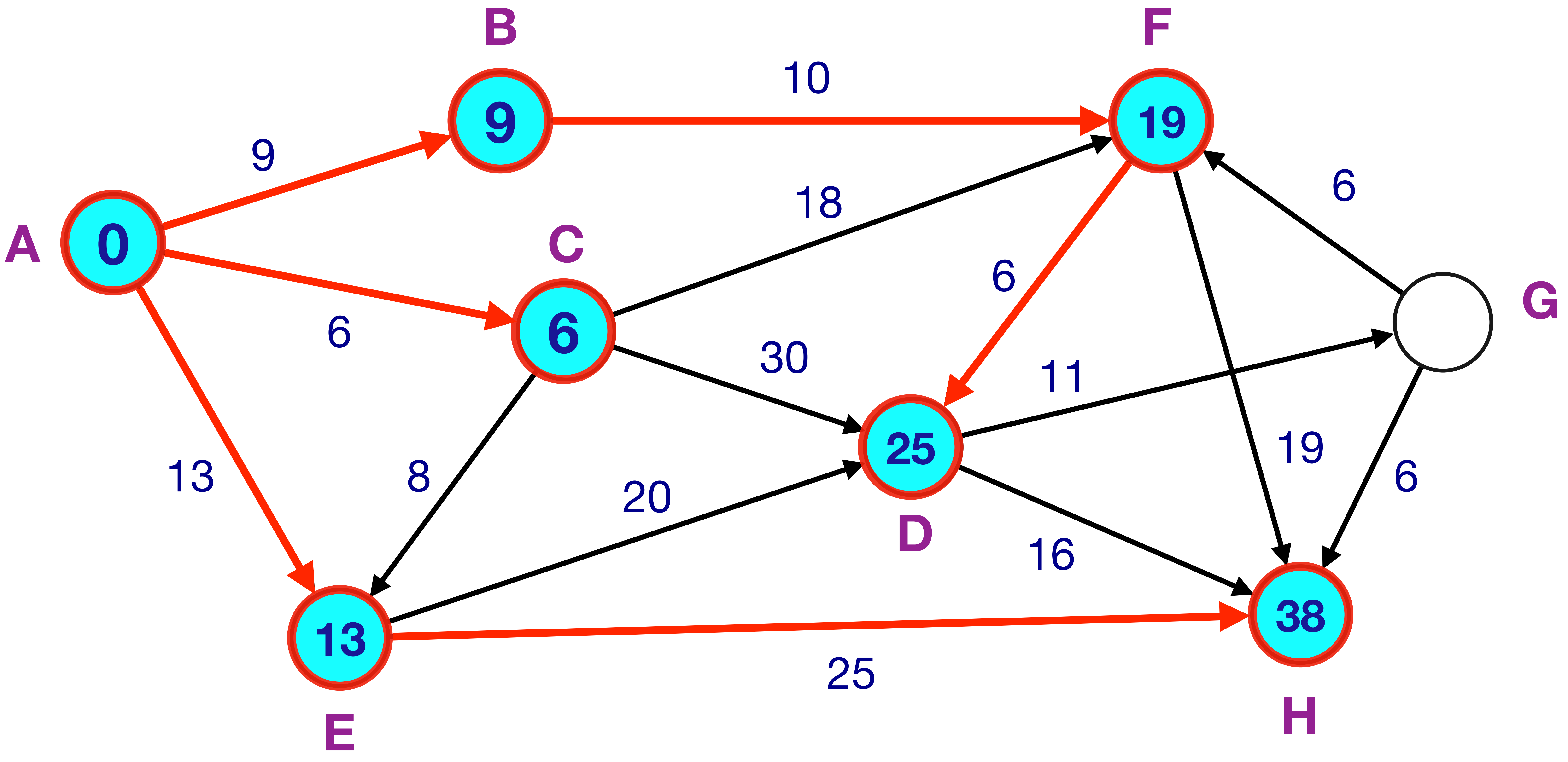
# Finding the $i^{th}$ closest node



# Finding the $i^{th}$ closest node



# Finding the $i^{th}$ closest node



# Algorithm

$V \setminus X$ → elements in $V$ that are not in $X$

\setminus

$V - X$

Initialize for each node $v$: $\text{dist}(s, v) = \infty$

Initialize $X = \varnothing$, $d'(s, s) = 0$

**for** $i = 1$ to $|V|$ **do**

(* Invariant: X contains the $i - 1$ closest nodes to s *)

(* Invariant: $d'(s, u)$ is shortest path distance from $u$ to $s$ using only X as intermediate nodes*)

Let $v$ be such that $d'(s, v) = \min_{u \in V \setminus X} d'(s, u)$

$\text{dist}(s, v) = d'(s, v)$

$X = X \cup \{v\}$

**for** each node $u$ in $V - X$ **do**

$d'(s, u) = \min_{t \in X}(\text{dist}(s, t) + l(t, u))$

# Algorithm

Initialize for each node $v$: $\text{dist}(s, v) = \infty$

Initialize $X = \emptyset$, $d'(s, s) = 0$

**for** $i = 1$ to $|V|$ **do**

(* Invariant: X contains the $i - 1$ closest nodes to s *)

(* Invariant: $d'(s, u)$ is shortest path distance from $u$ to $s$ using only X as intermediate nodes*)

Let $v$ be such that $d'(s, v) = \min_{u \in V \setminus X} d'(s, u)$

$\text{dist}(s, v) = d'(s, v)$

$X = X \cup \{v\}$

**for** each node $u$ in $V - X$ **do**

$d'(s, u) = \min_{t \in X}(\text{dist}(s, t) + l(t, u))$

Running time: $O(n \,.\, (n + m))$ time

# Algorithm

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d′(s, s) = 0
for i = 1 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    (* Invariant: d′(s, u) is shortest path distance from u to s
    using only X as intermediate nodes*)
        Let v be such that d′(s, v) = min_{u∈V\X} d′(s, u)
        dist(s, v) = d′(s, v)
        X = X ∪ {v}
        for each node u in V − X do
            d′(s, u) = min_{t∈X}(dist(s, t) + l(t, u))
```

Running time: $O(n\,.\,(n+m))$ time

There are $n$ outer iterations. In each iteration, $d'(s,u)$ for each $u$ by scanning all edges out of nodes in $X$; $O(m+n)$ time/iteration

# Dijkstra’s algorithm

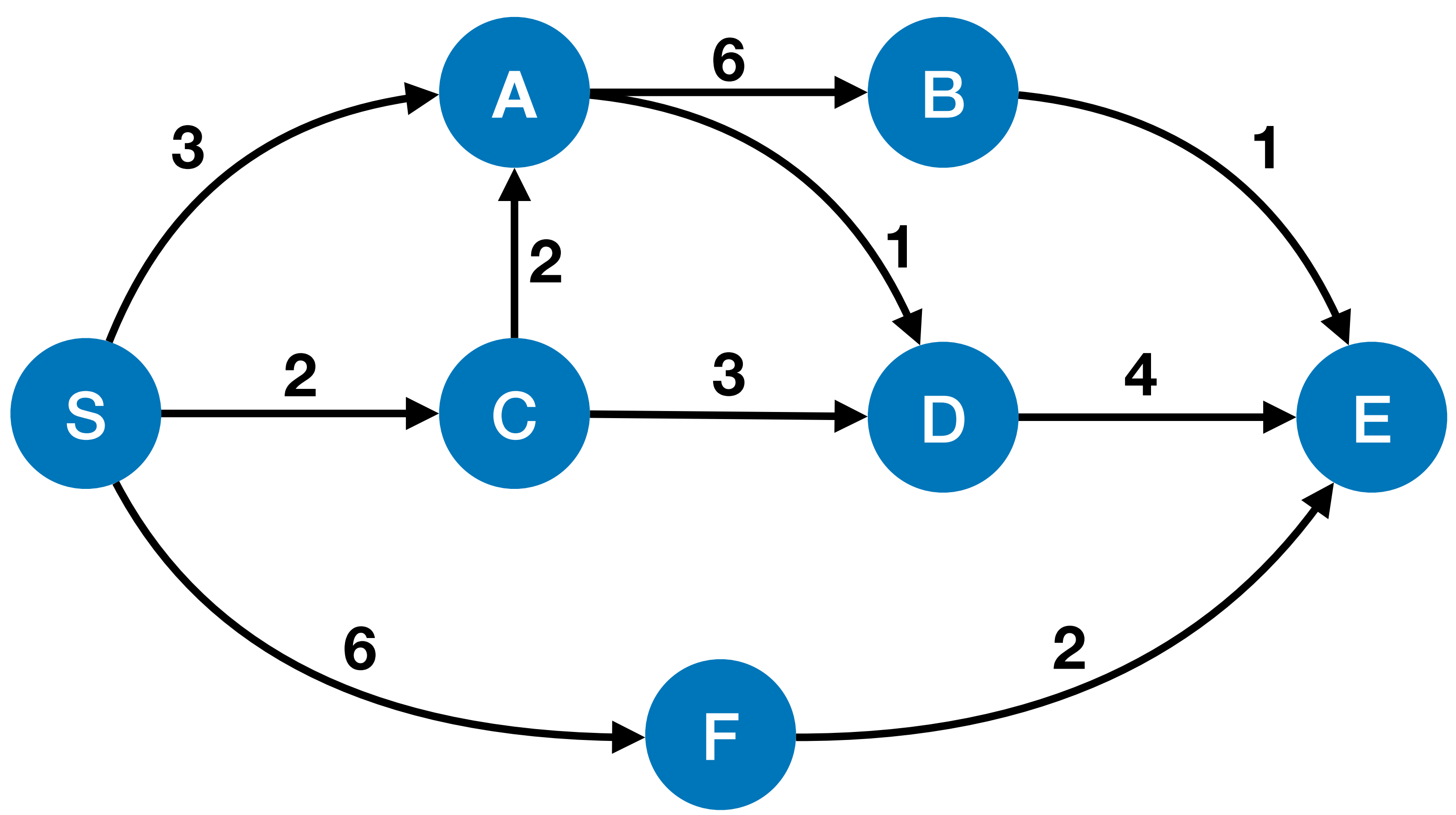
Dijkstra's Algorithm finds the shortest path between a given node (called the *source node*) and ***all*** other nodes in a *non-negatively* edge-weighted graph.

This algorithm was created by **Dr. Edsger W. Dijkstra**, a Dutch computer scientist and software engineer, “in about 20 minutes”.

> *What’s the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. In fact, it was published in 1959, three years later.*

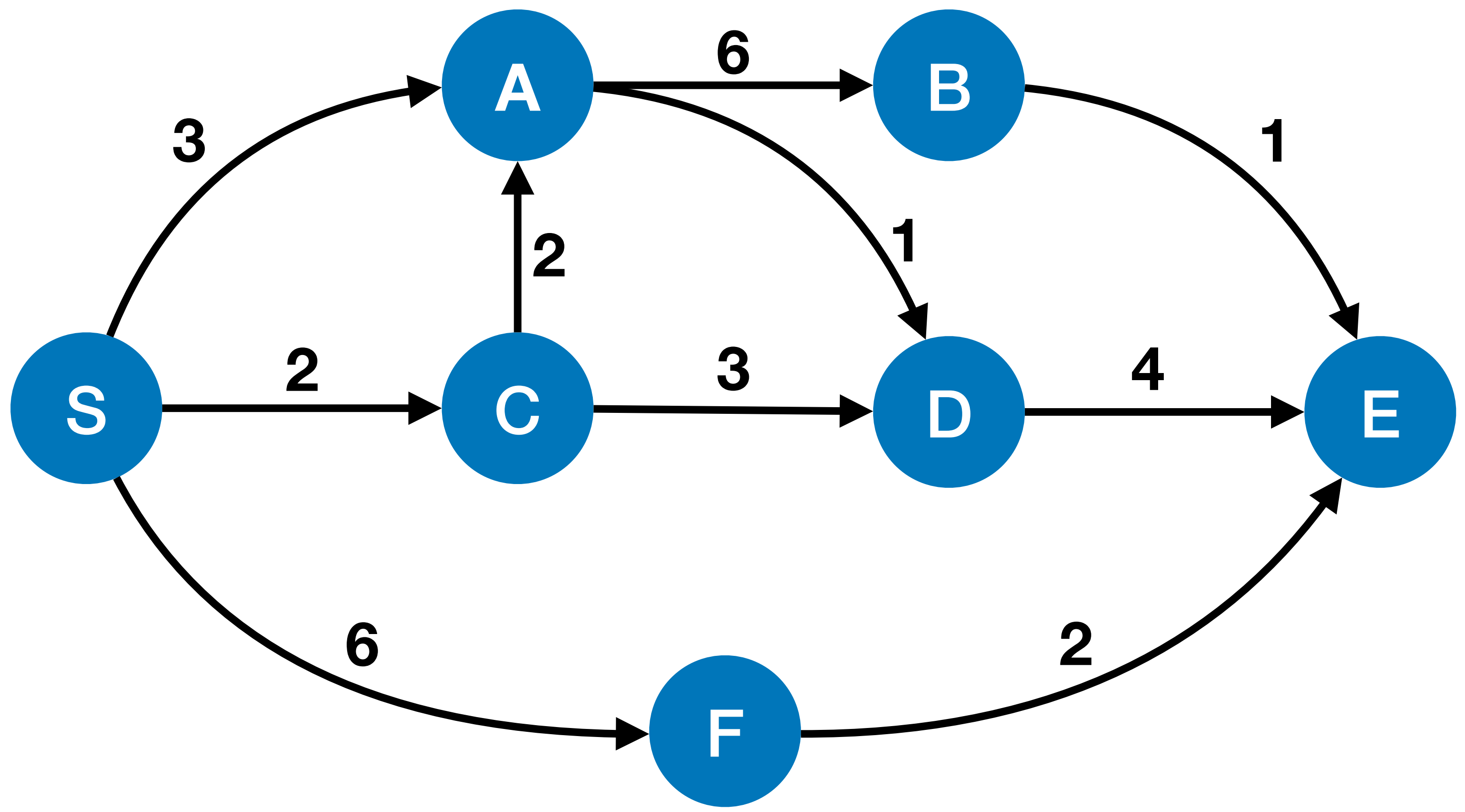
https://doi.org/10.1145/1787234.1787249

# Dijkstra's algorithm



# Dijkstra's algorithm

<u>**Key point:**</u> We keep *distance estimates* from source node to every other node, and keep updating estimates until nodes are "*settled*".

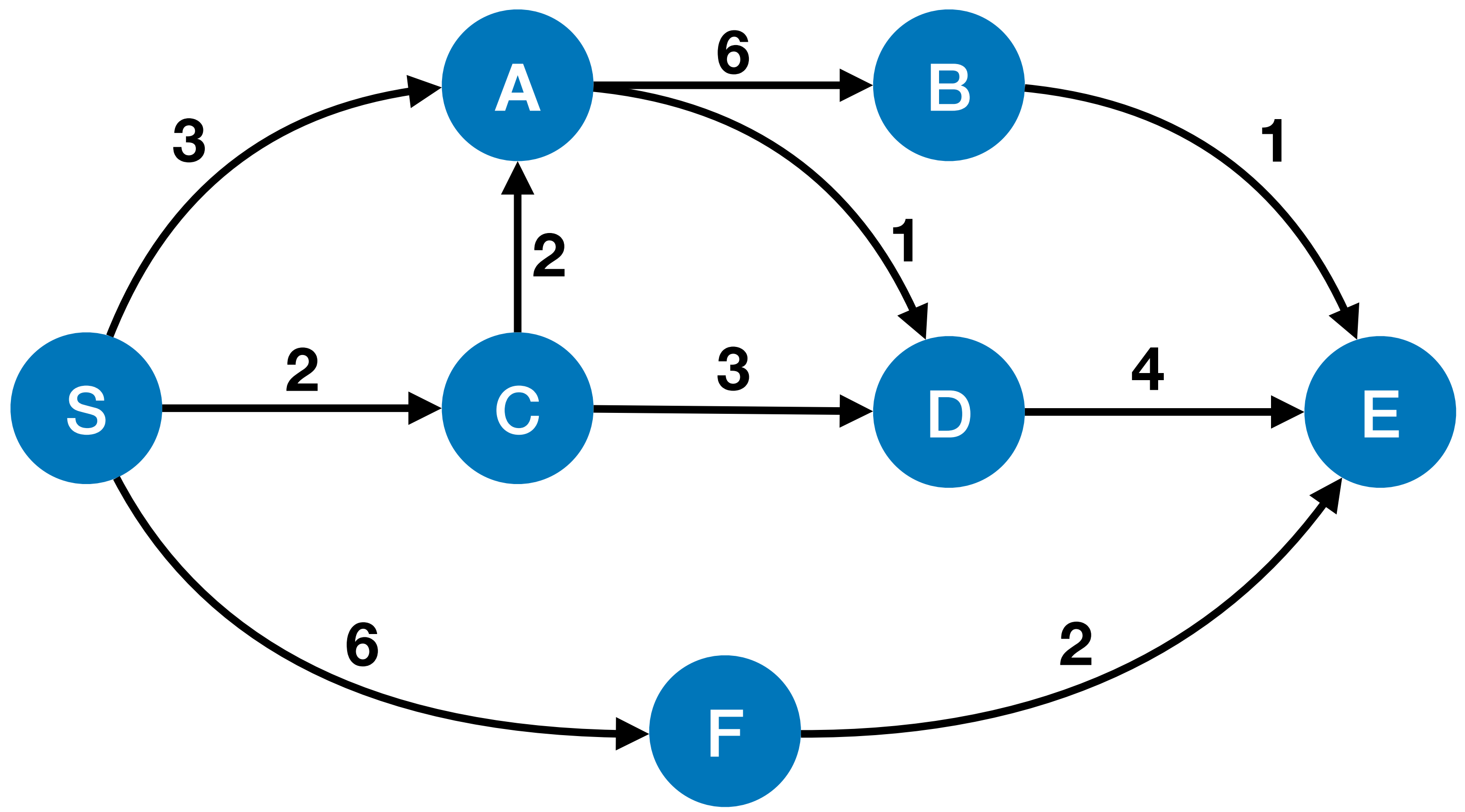





UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

# Dijkstra's algorithm

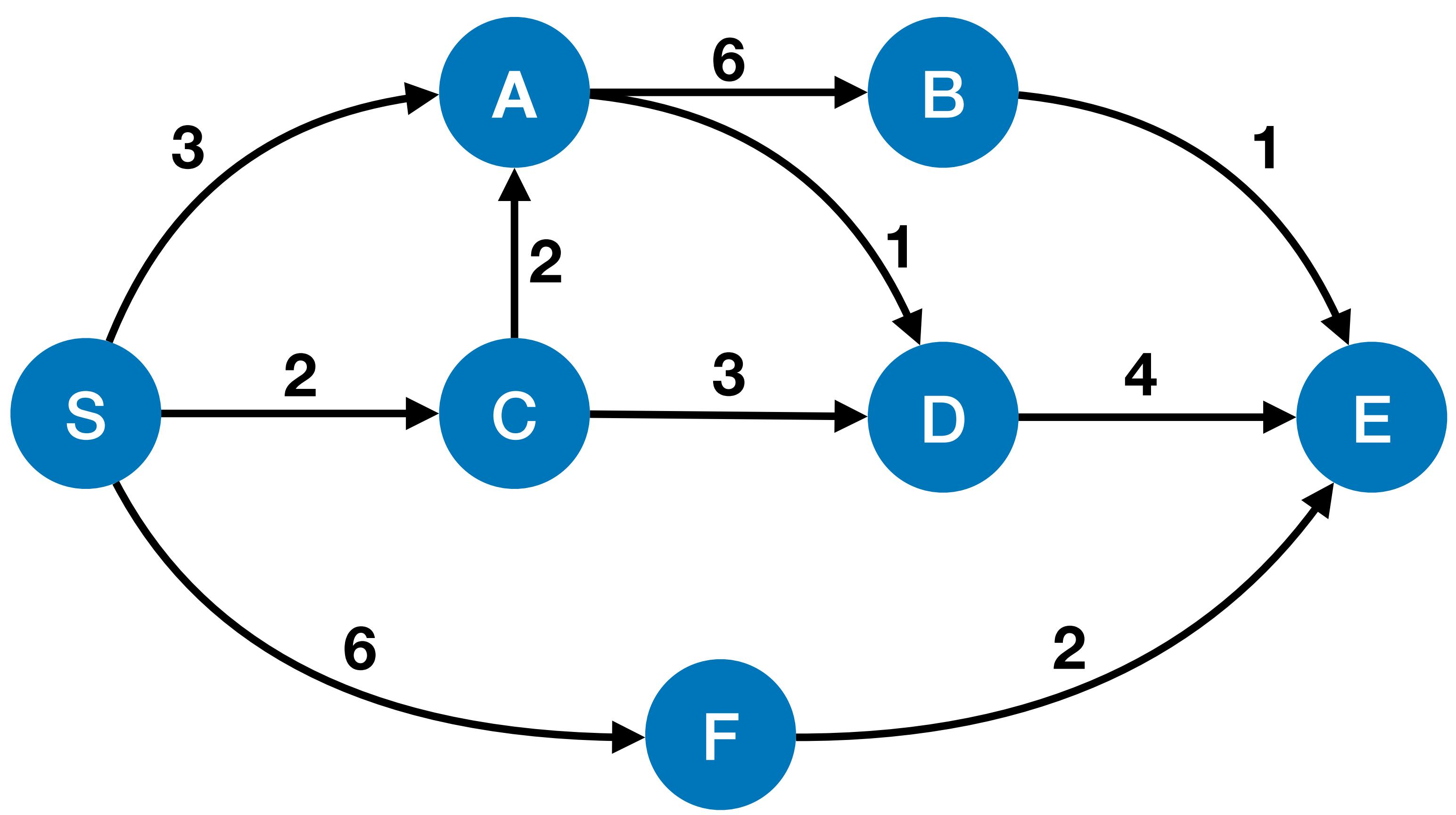
**Key point:** We keep *distance estimates* from source node to every other node, and keep updating estimates until nodes are "*settled*".




| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | | |
| C | | |
| F | | |
| D | | |
| B | | |
| E | | |

# Dijkstra's algorithm

**<u>Key point:</u>** We keep *distance estimates* from source node to every other node, and keep updating estimates until nodes are "*settled*".

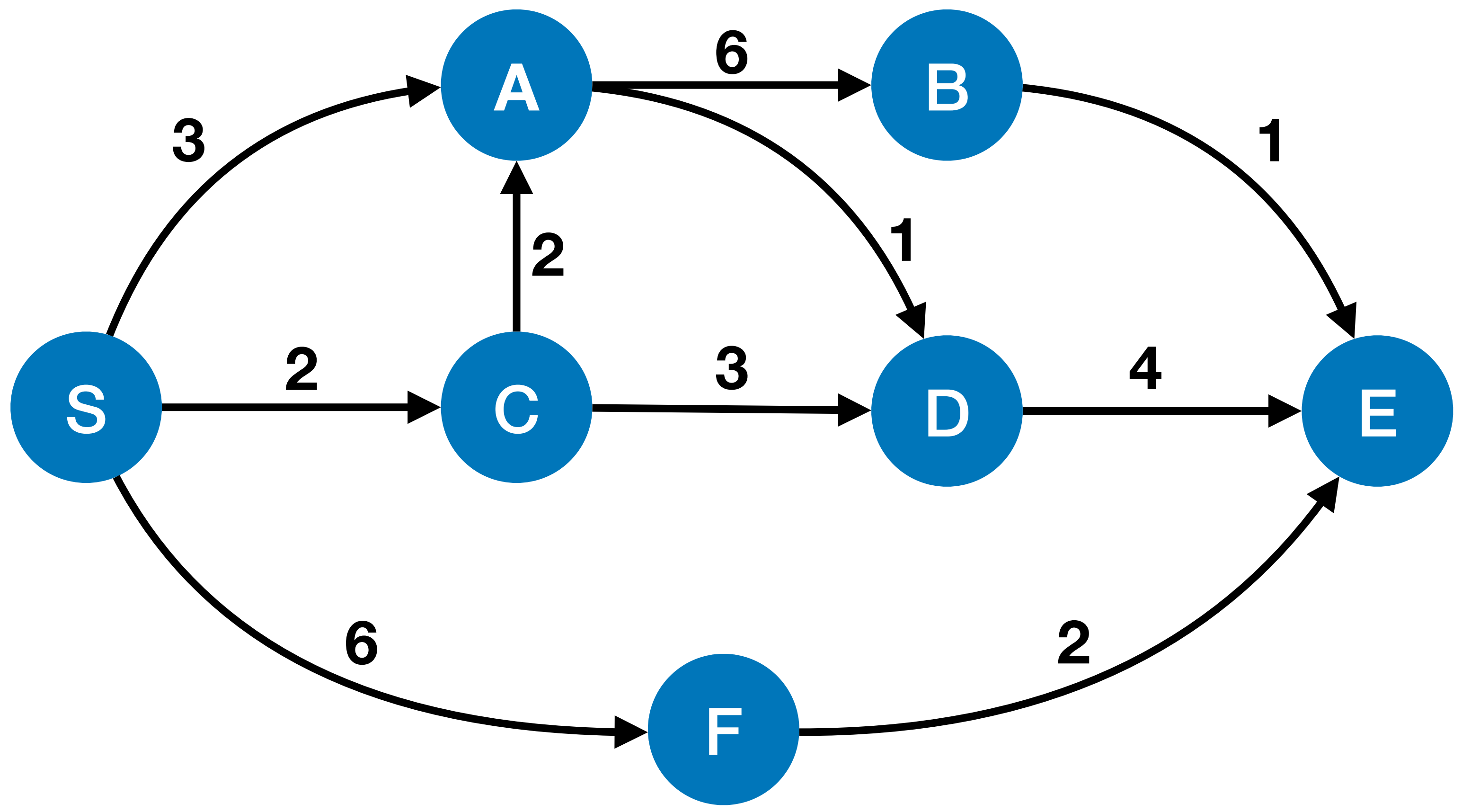



| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | | |
| C | | |
| F | | |
| D | | |
| B | | |
| E | | |

Settled = [ ]

Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm




| Node | Distance estimate | Previous node |
| --- | --- | --- |
| | | |
| | | |
| | | |

Settled = [ ]

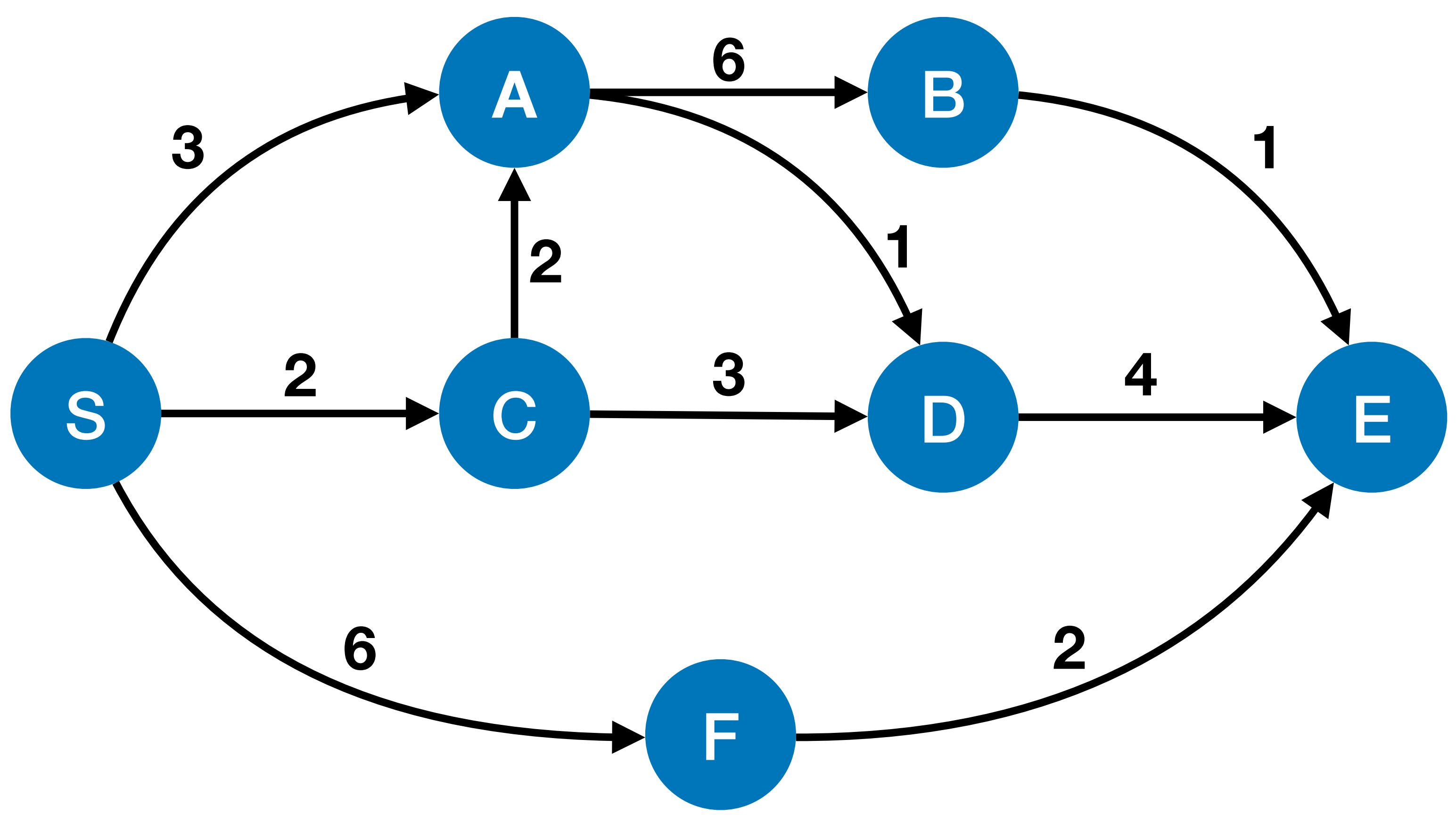
Unexplored = [ S, A, C, F, D, B, E ]



**Initialization step**

# Dijkstra's algorithm

- Set distance to source node = 0.




| Node | Distance estimate | Previous node |
|---|---|---|
| | | |
| | | |
| | | |

Settled = [ ]

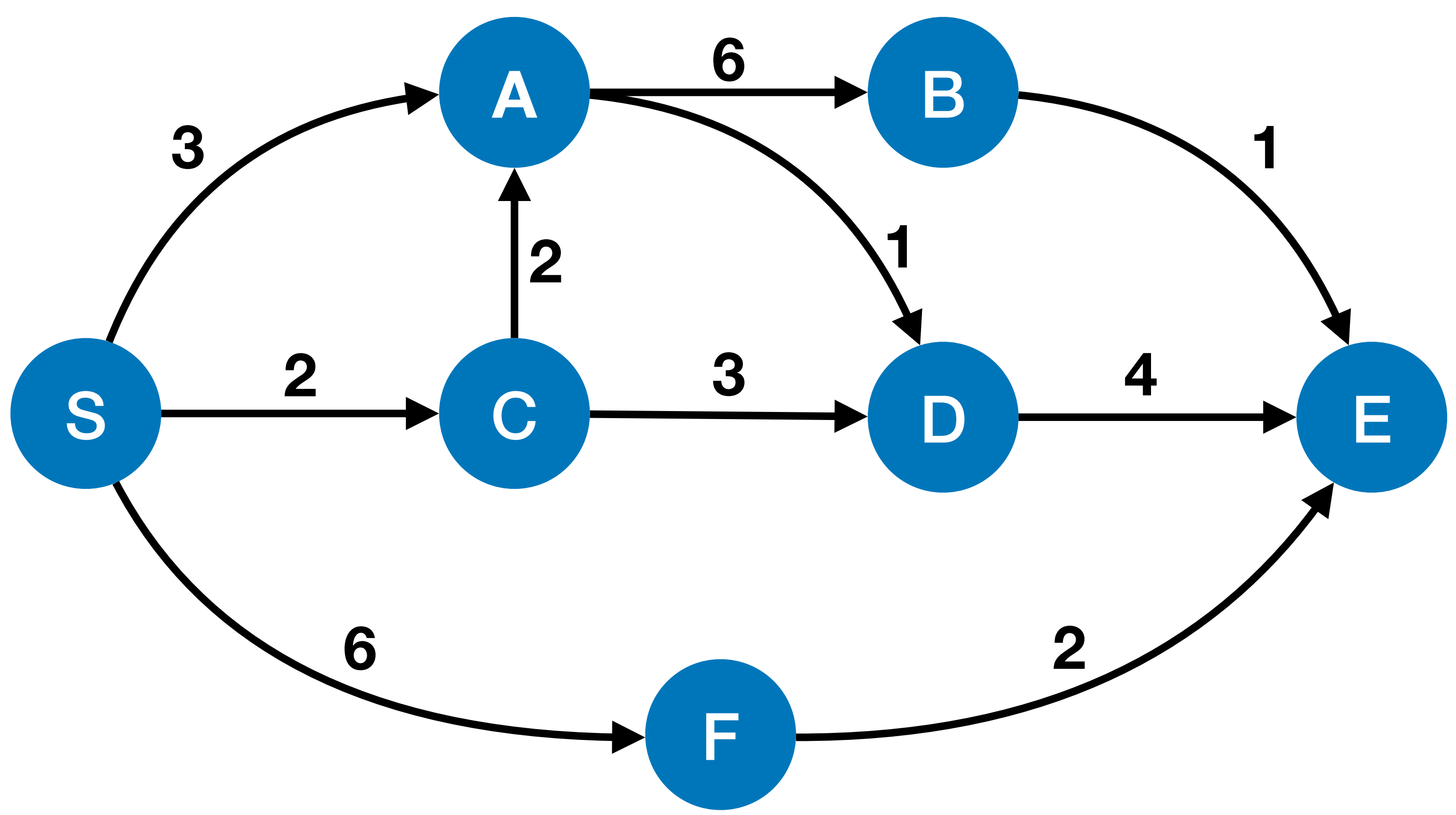
Unexplored = [ S, A, C, F, D, B, E ]



Initialization step

# Dijkstra's algorithm

- Set distance to source node = 0.




| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| | | |
| | | |

Settled = [ ]

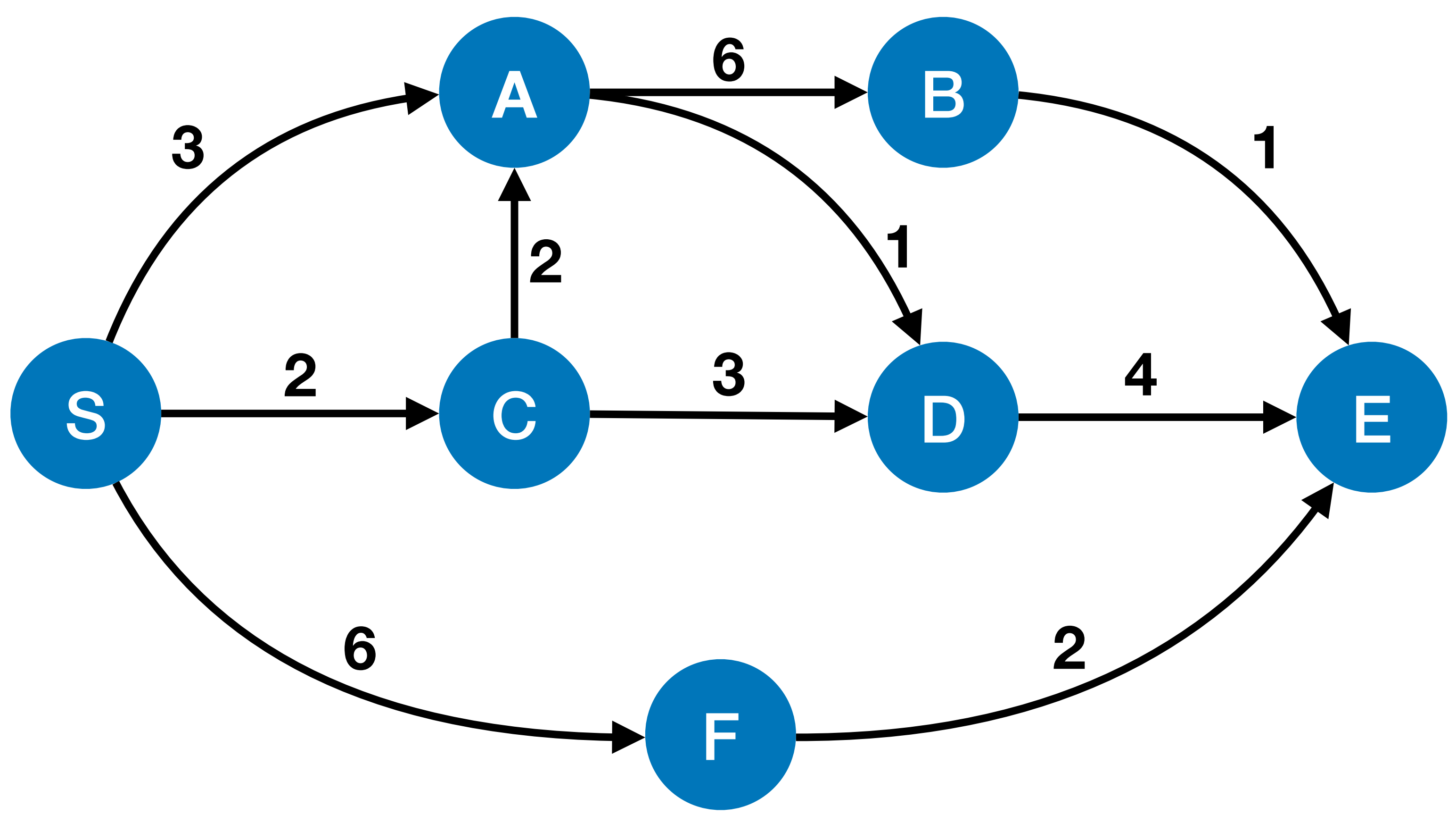
Unexplored = [ S, A, C, F, D, B, E ]



**Initialization step**

# Dijkstra's algorithm

- Set distance to source node = 0.
- Distances to all other nodes from source node are currently unknown, therefore ∞ .



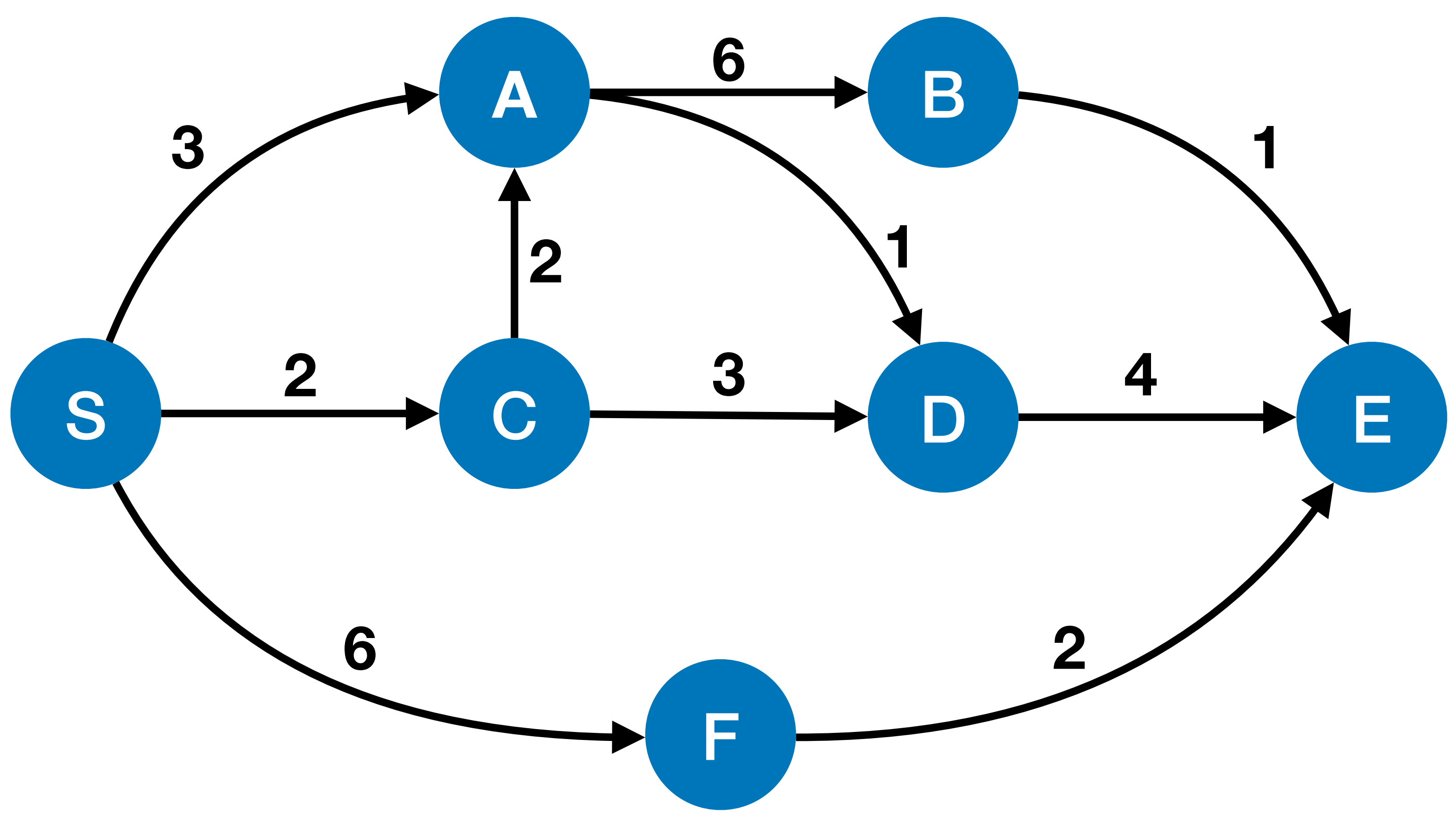

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| | | |
| | | |

Settled = [ ]

Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

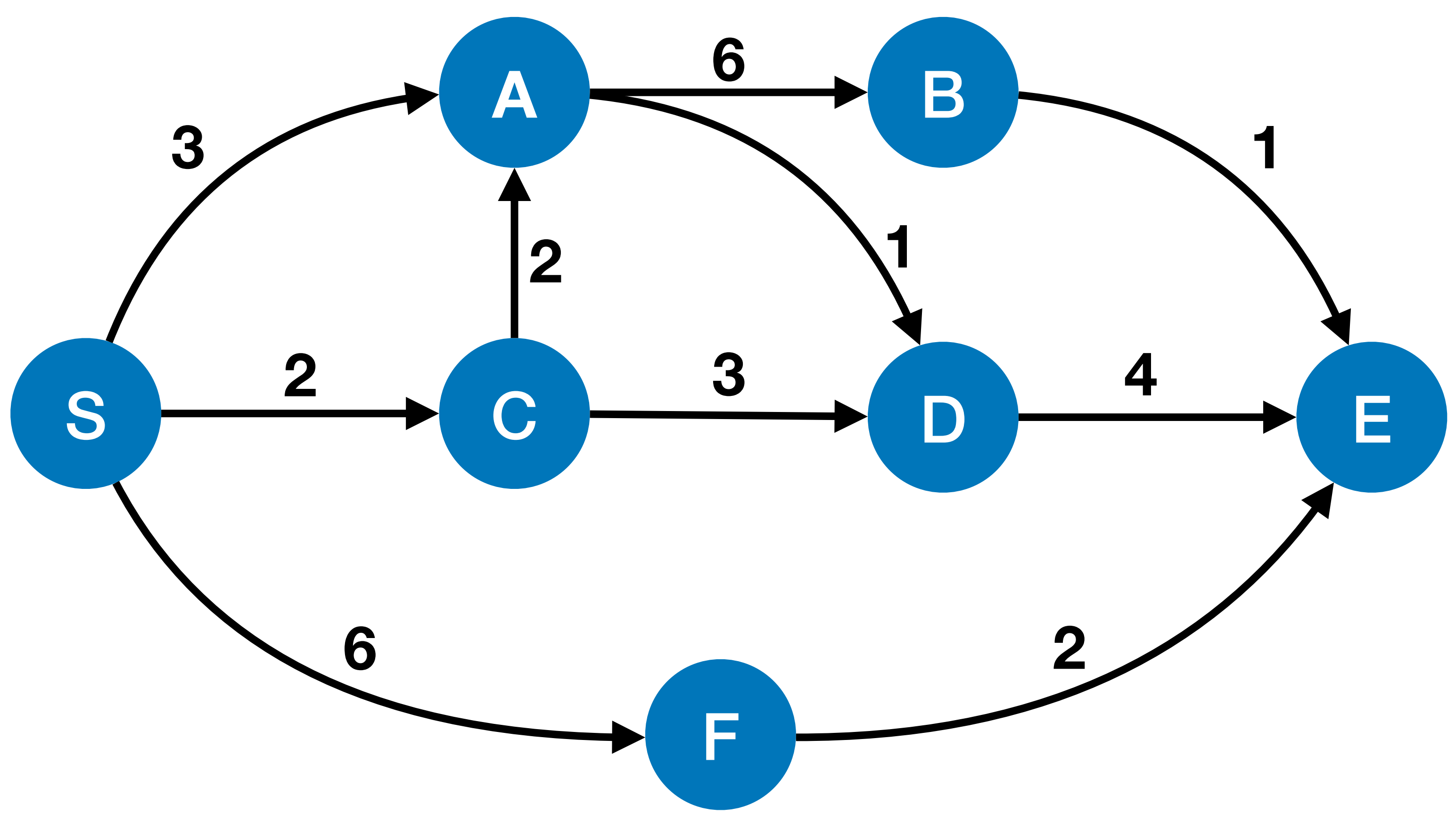
- Set distance to source node = 0.
- Distances to all other nodes from source node are currently unknown, therefore ∞ .




| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]  Unexplored = [ S, A, C, F, D, B, E ]

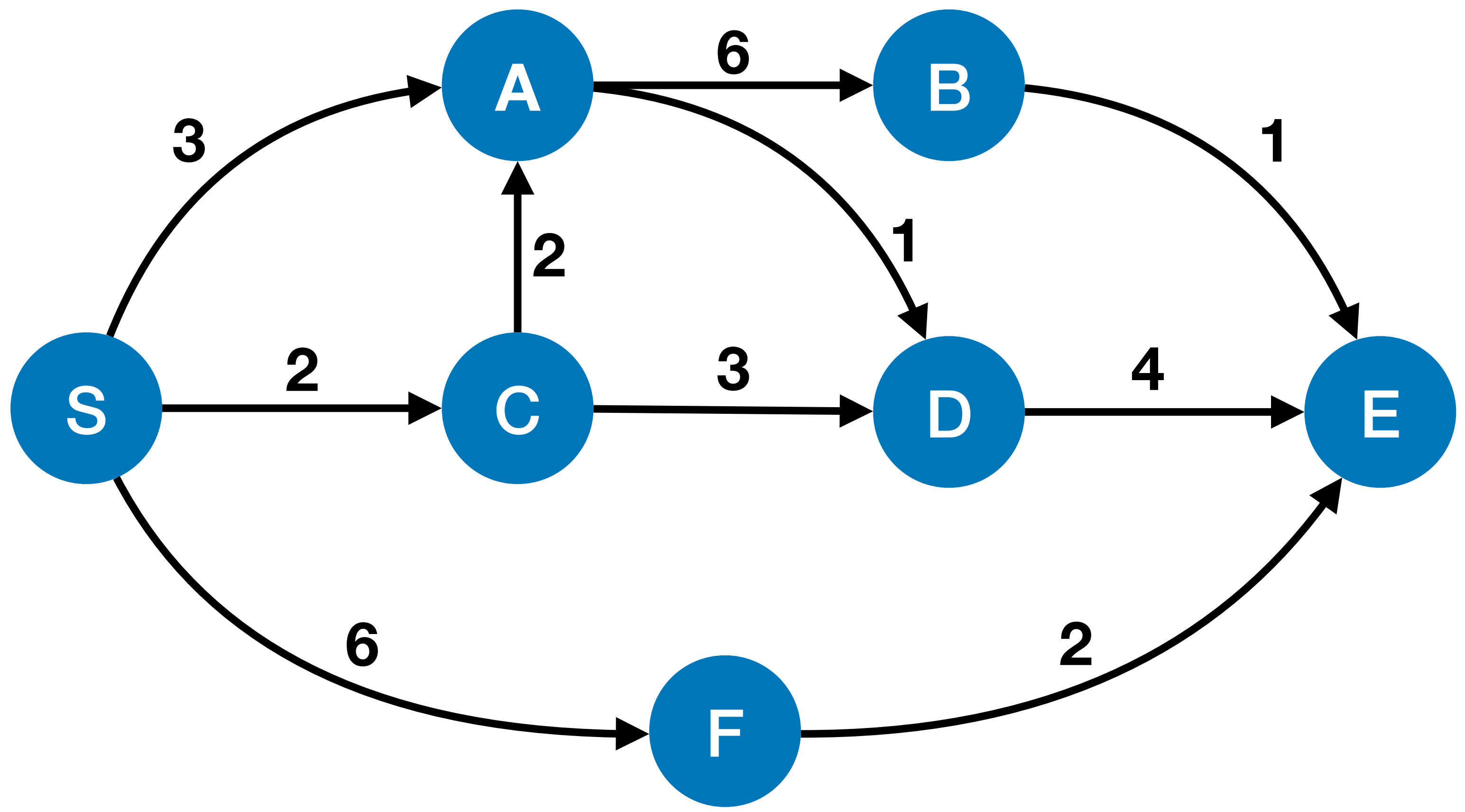
# Dijkstra's algorithm




| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]  Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm



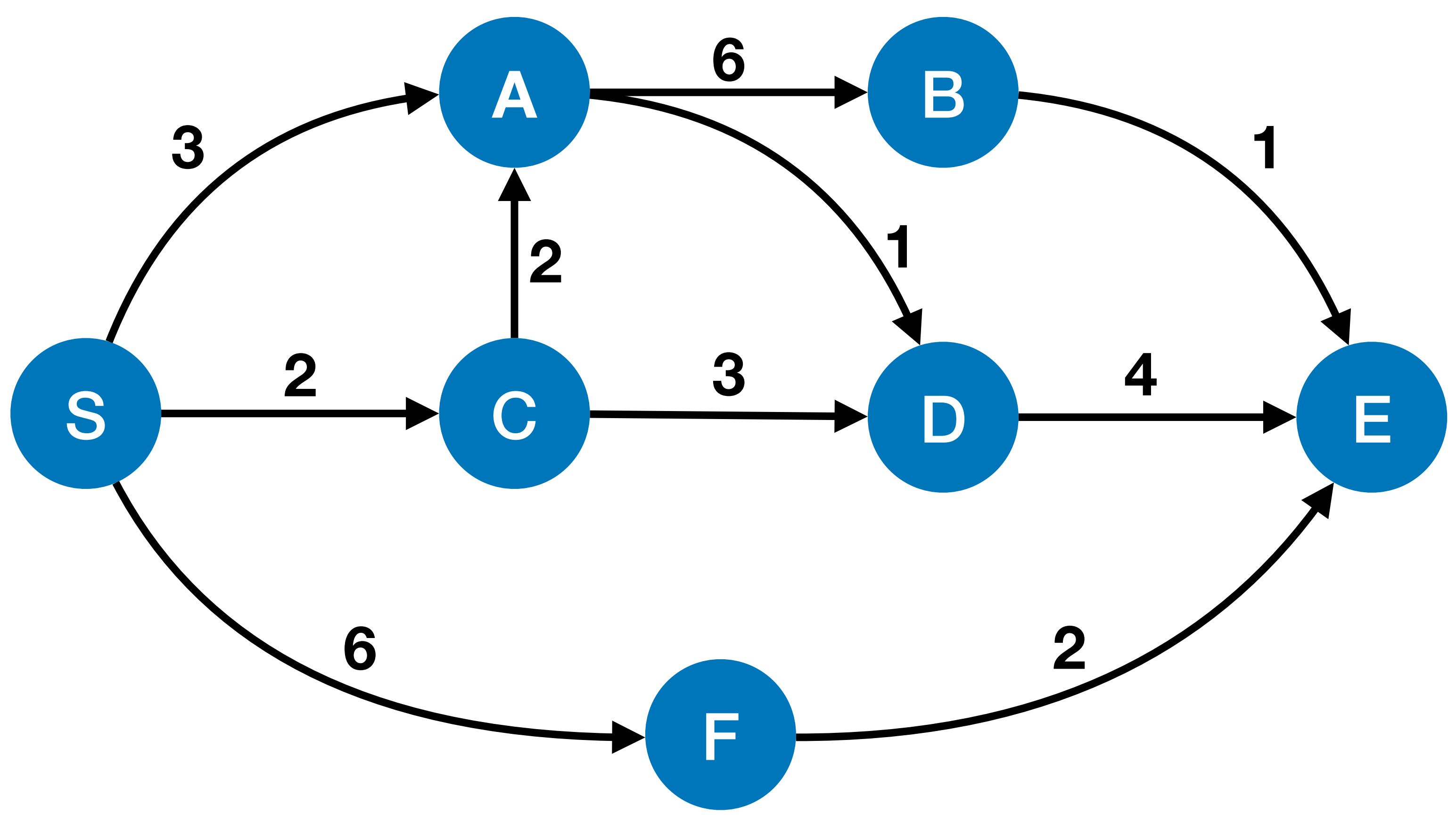

| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]

Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known estimate from the source node

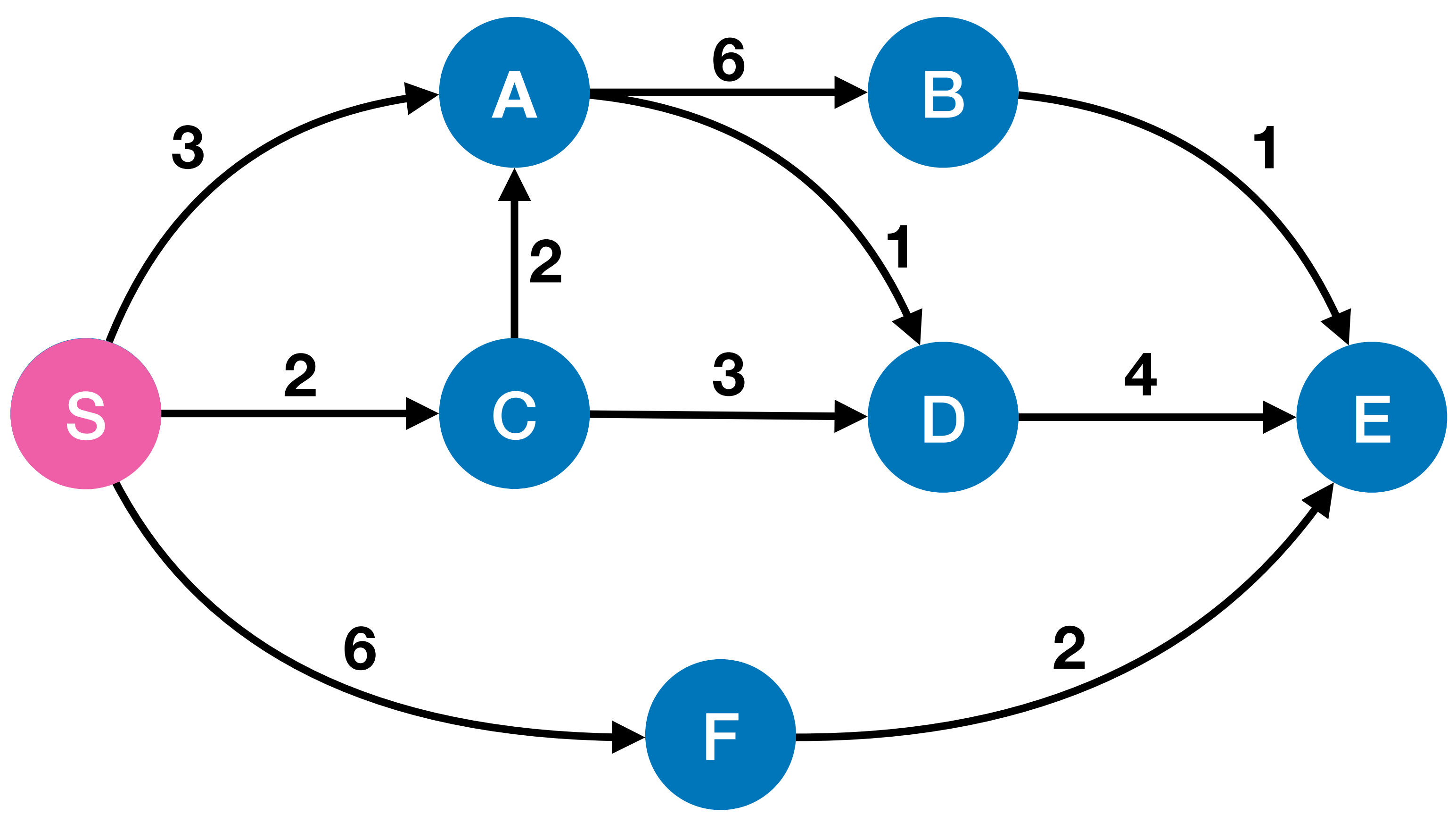



| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]　　　　Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known estimate from the source node
- The first time, it is the source node (S) itself.

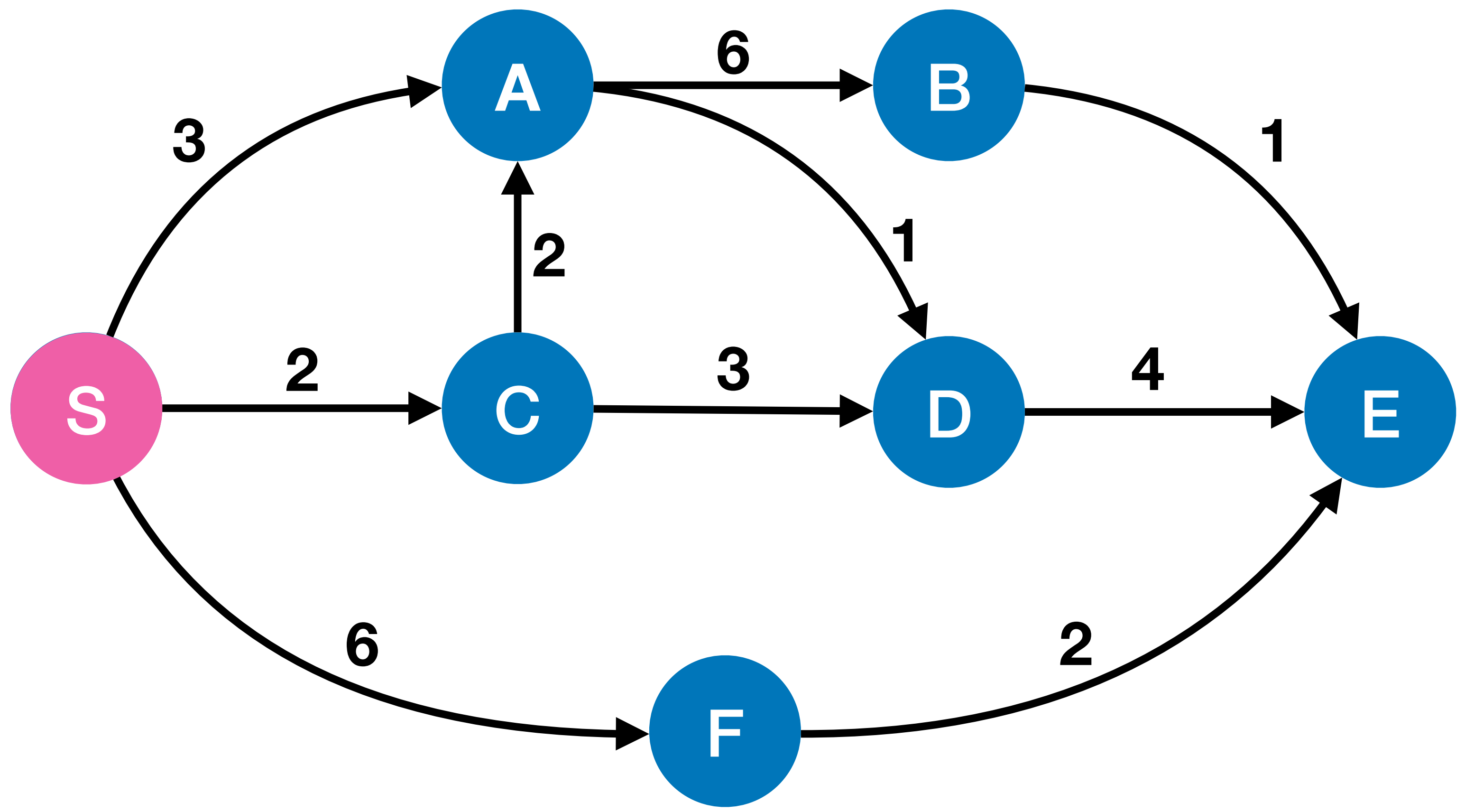



| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]   Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors



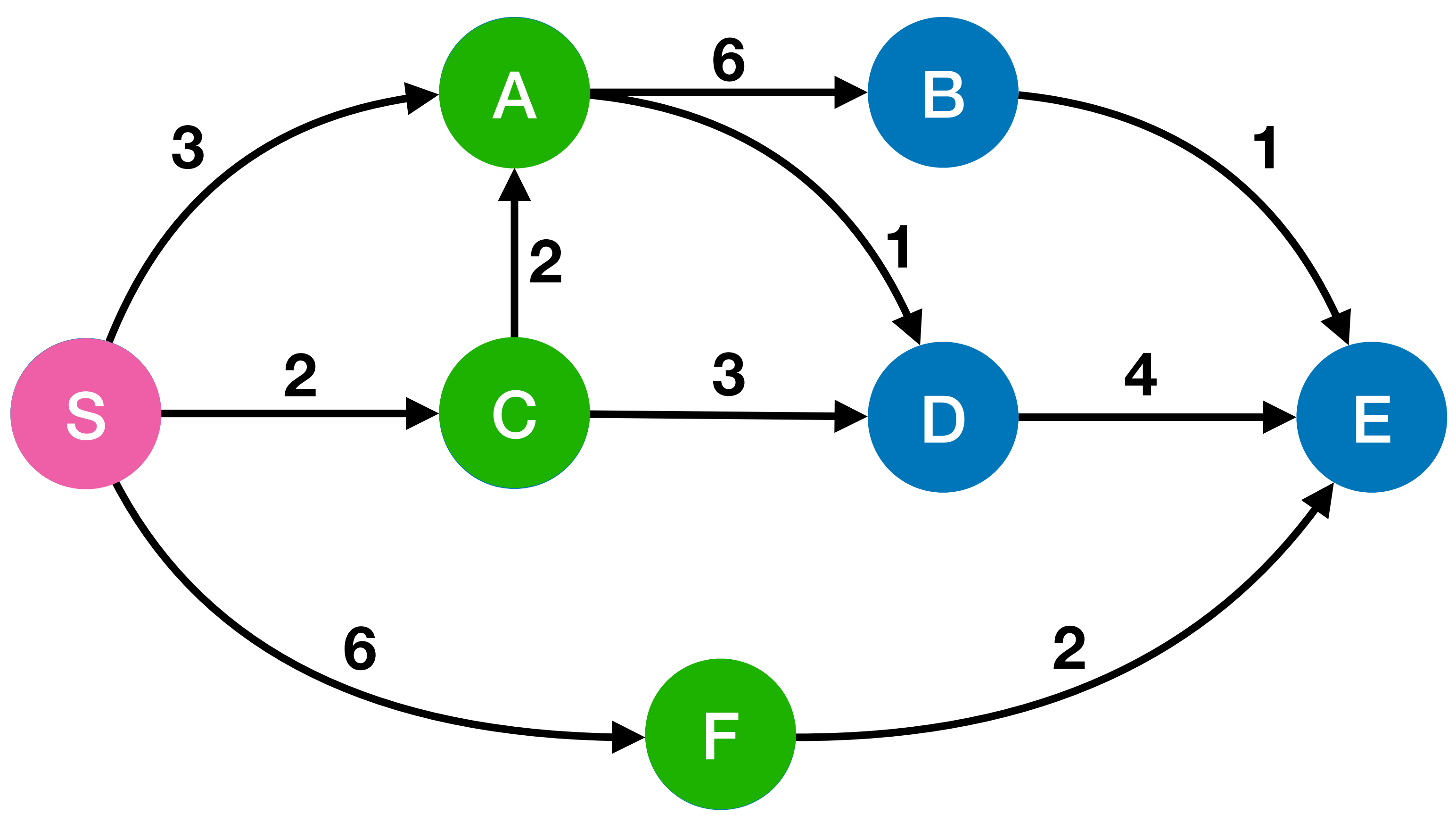

| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]

Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

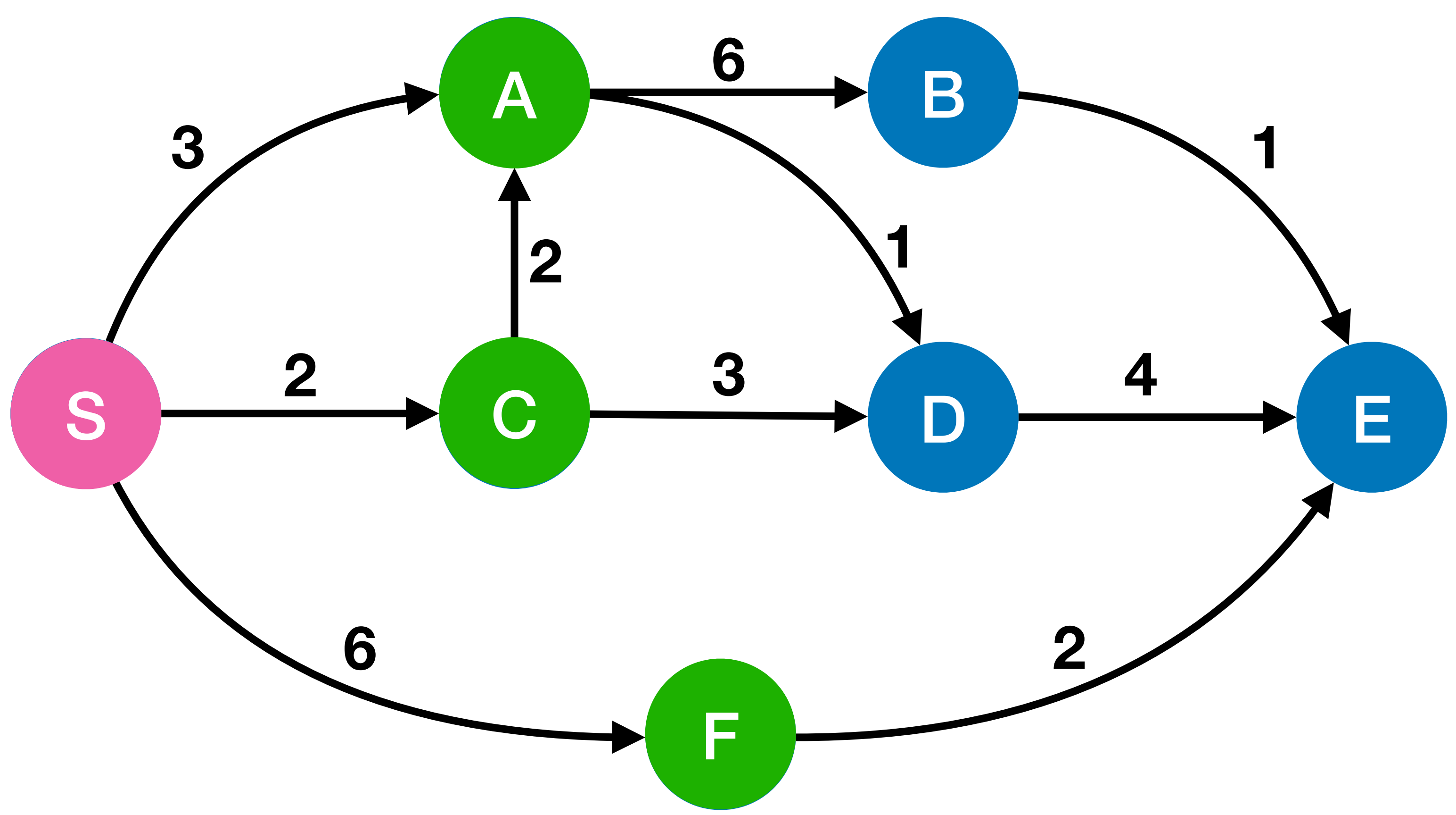
- For the current node, examine its unexplored neighbors
- Current node → S; unexplored neighbors → {A, C & F}




| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ] Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm



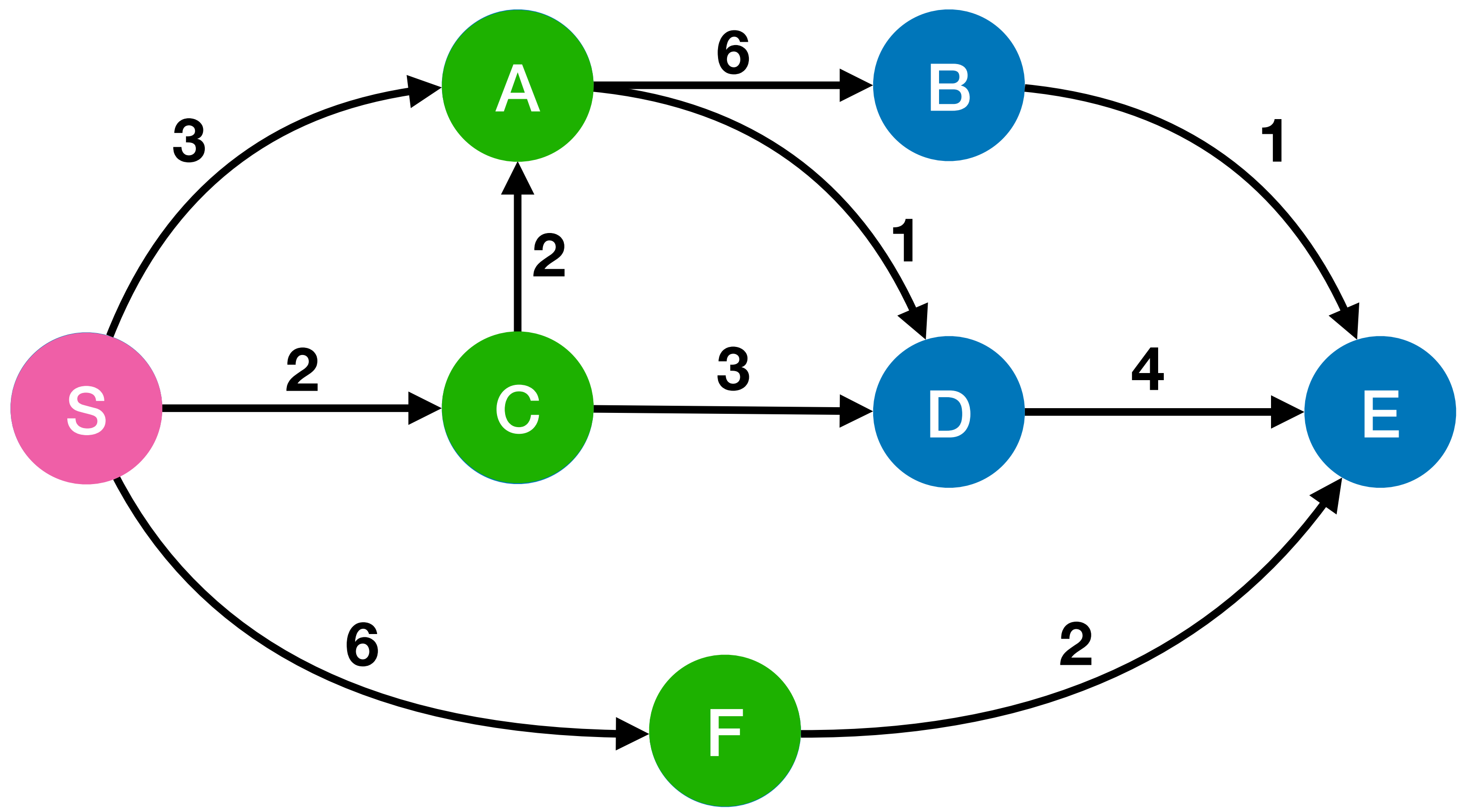

| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]

Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra’s algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.

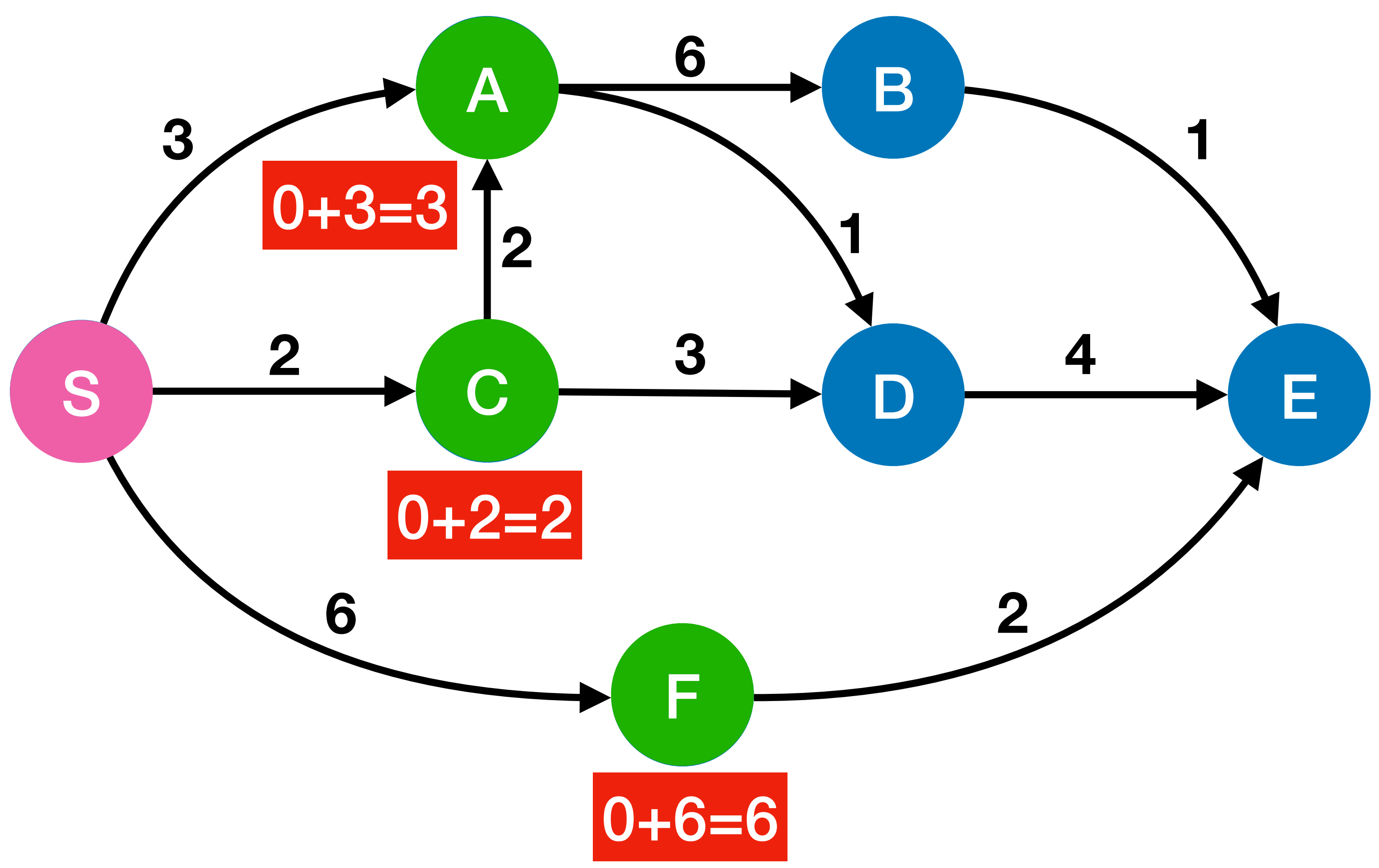



| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]  Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.



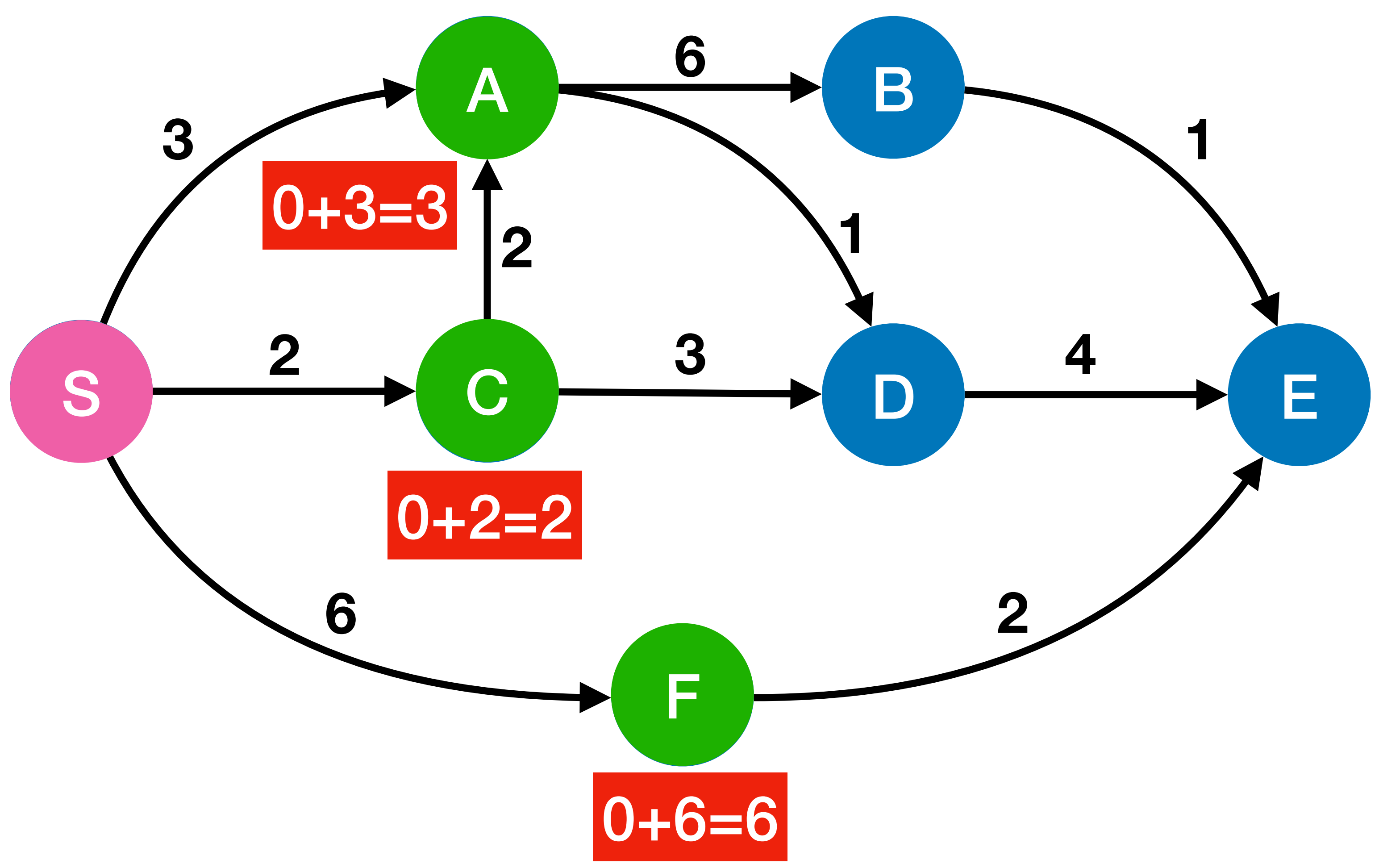

| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]

Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.

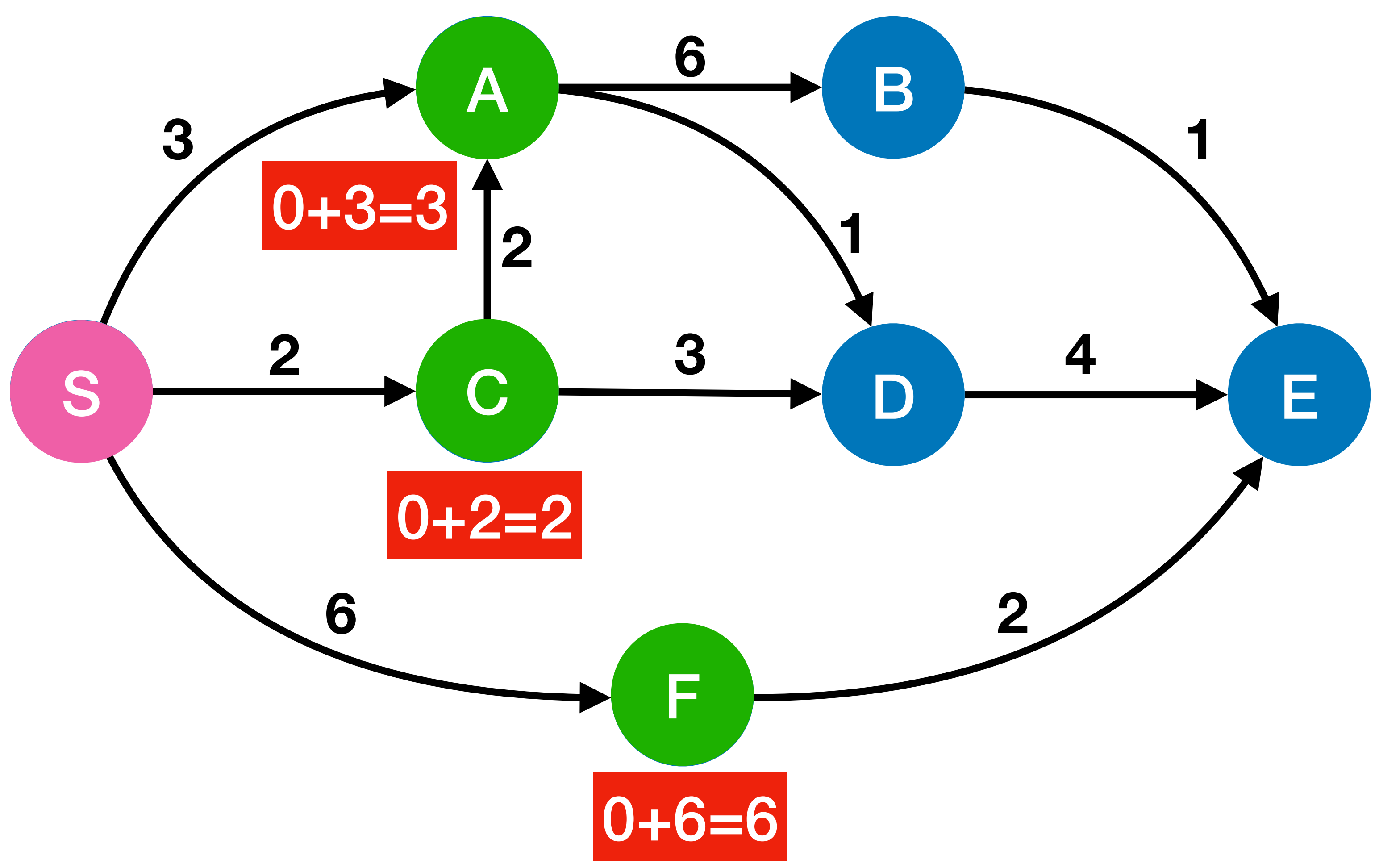



| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | ∞ | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]  Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

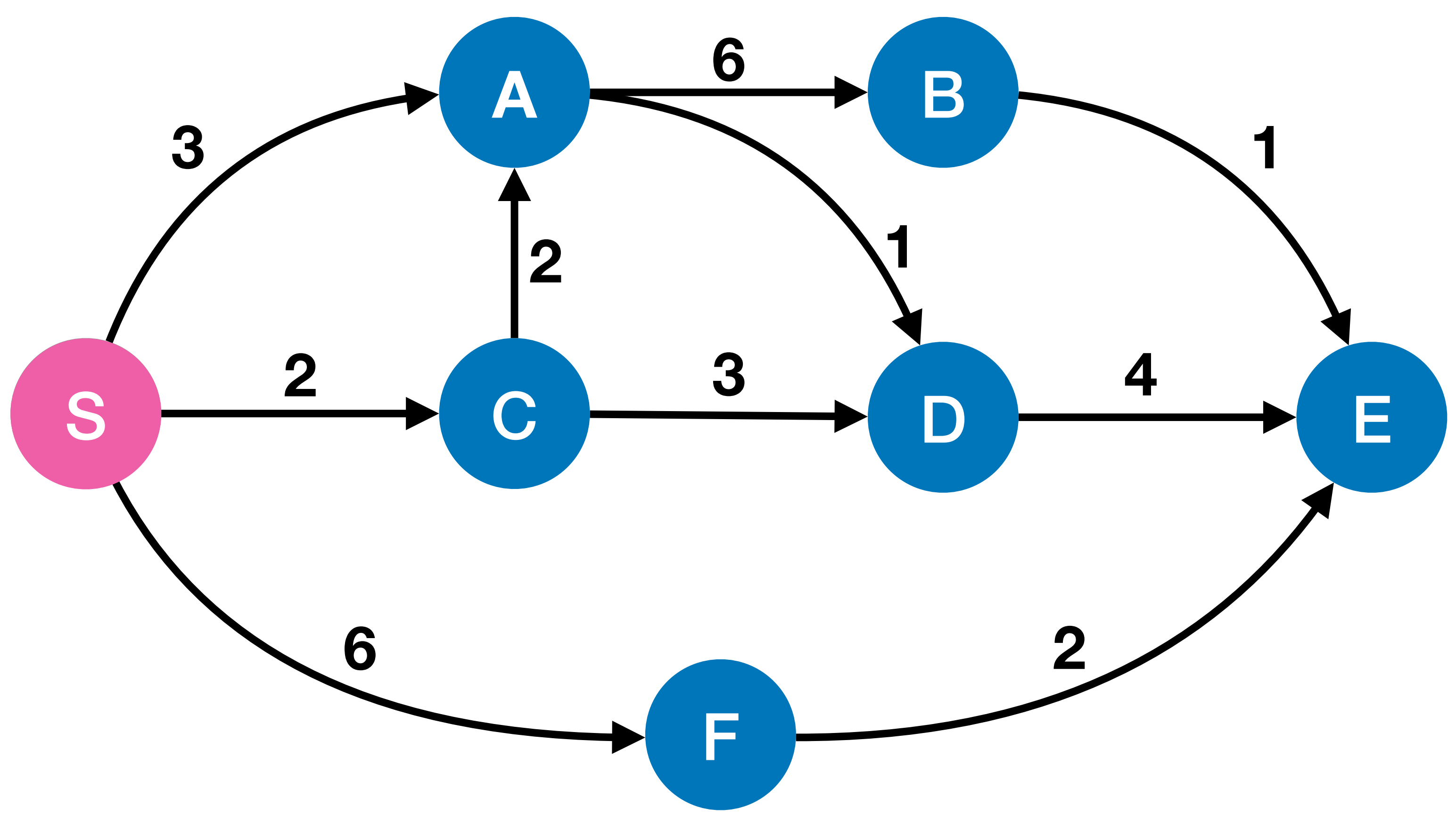
- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.




| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]   Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

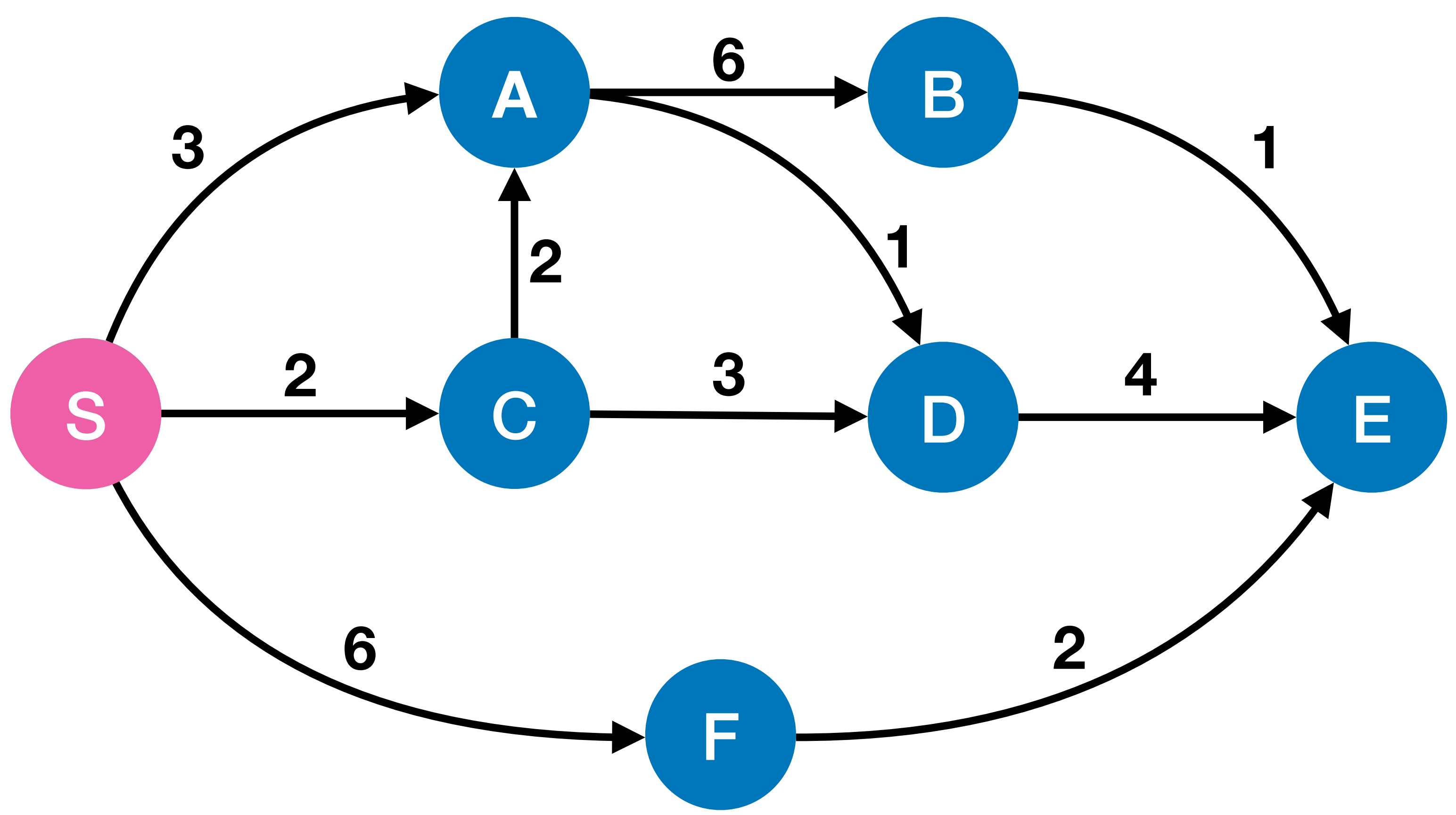



| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]     Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes



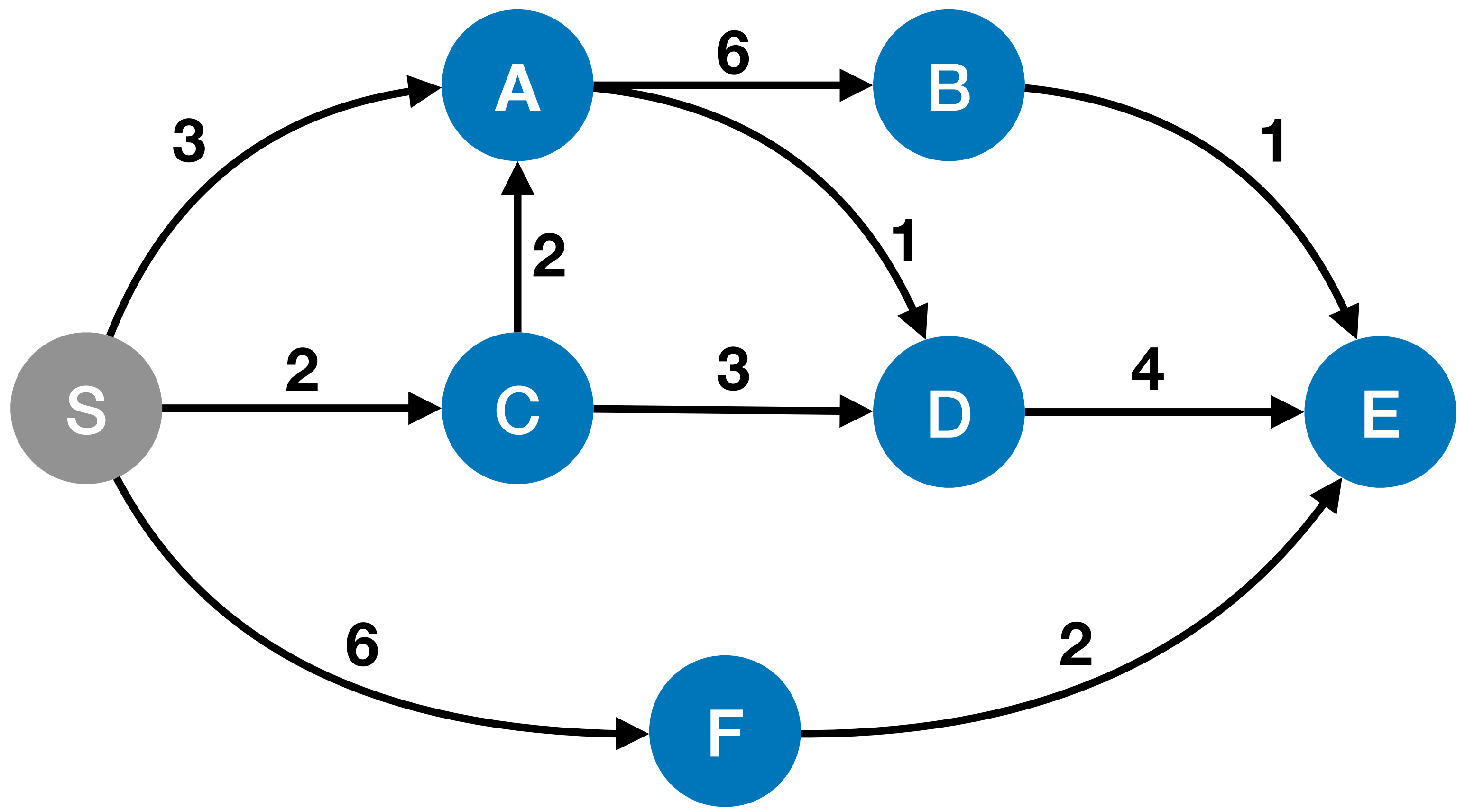

| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]

Unexplored = [ S, A, C, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

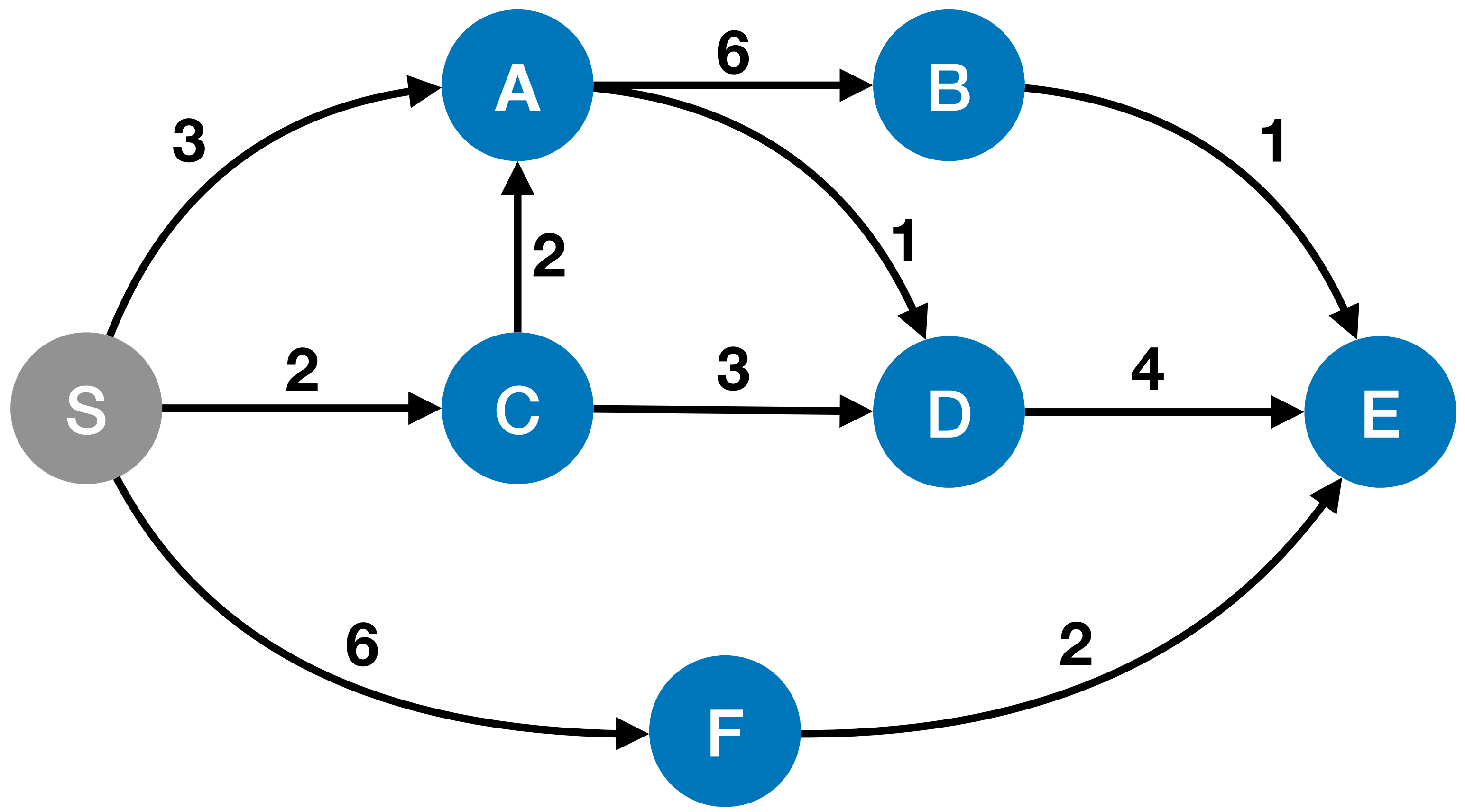



| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ ]   Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

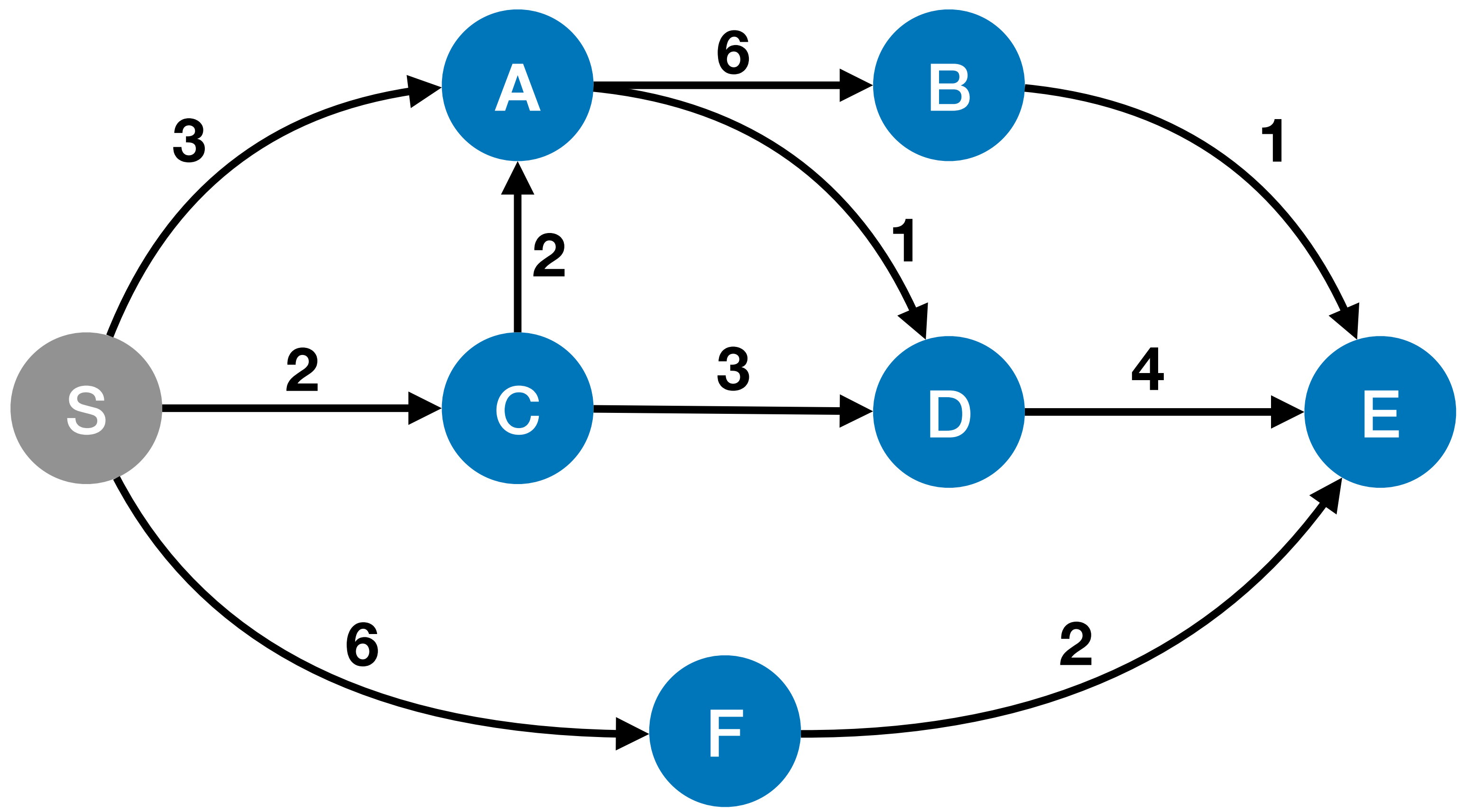



| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]    Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes




| Node | Distance estimate | Previous node |
|---|---|---|
| **S** | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

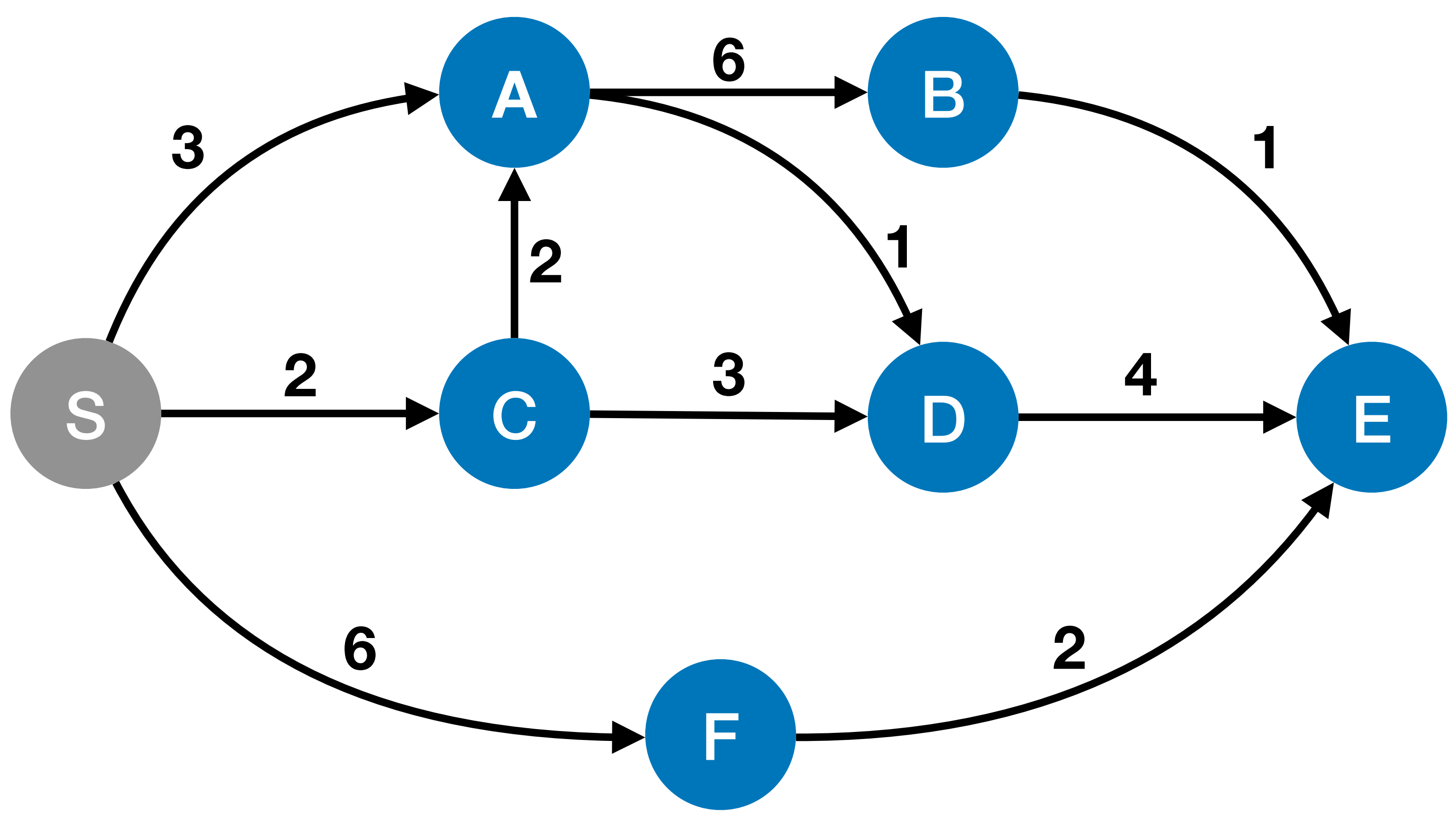
Settled = [ S ]

Unexplored = [A, C, F, D, B, E ]



**Iterative step - End Iter 1**

# Dijkstra's algorithm

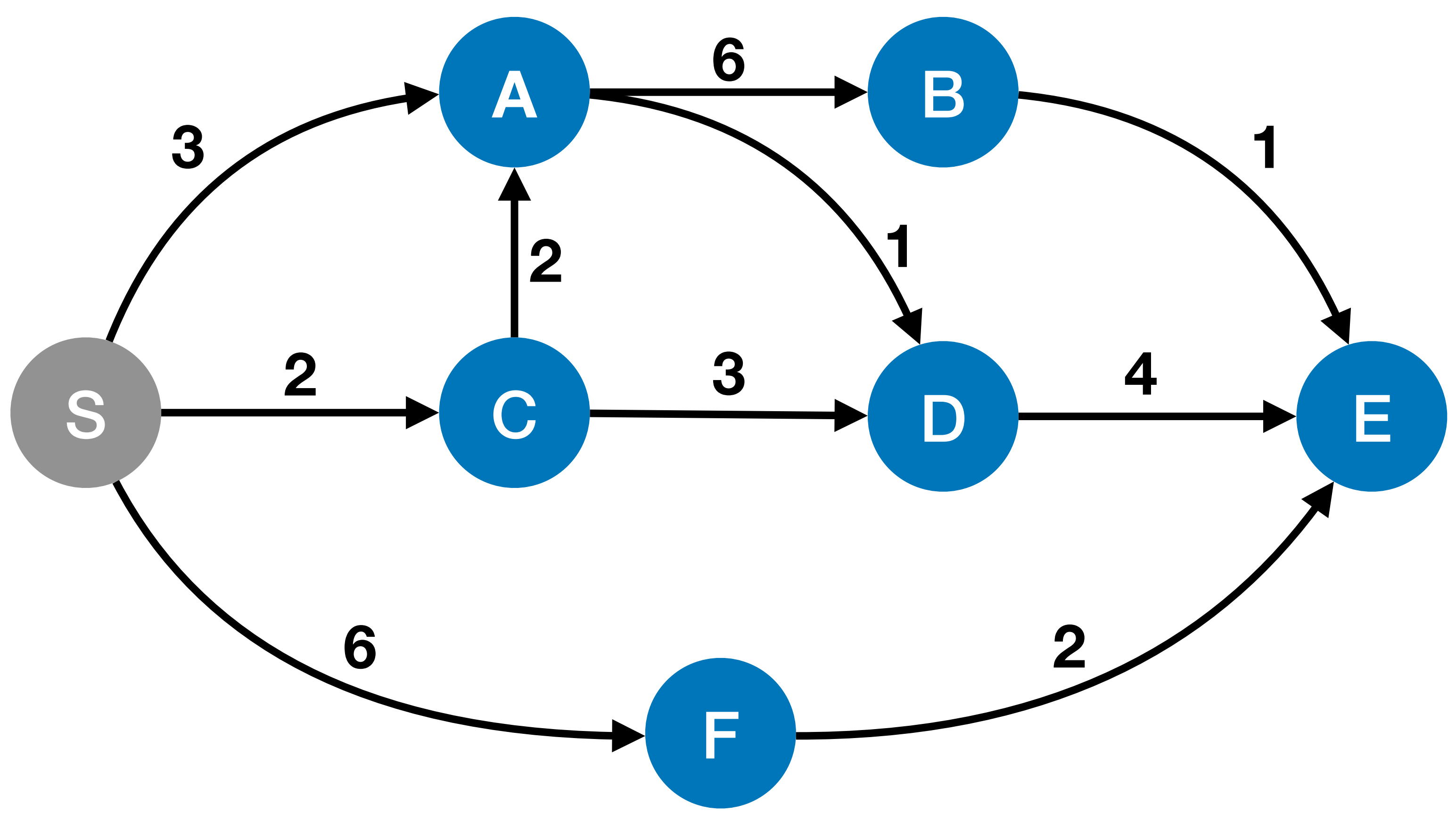



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]

Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

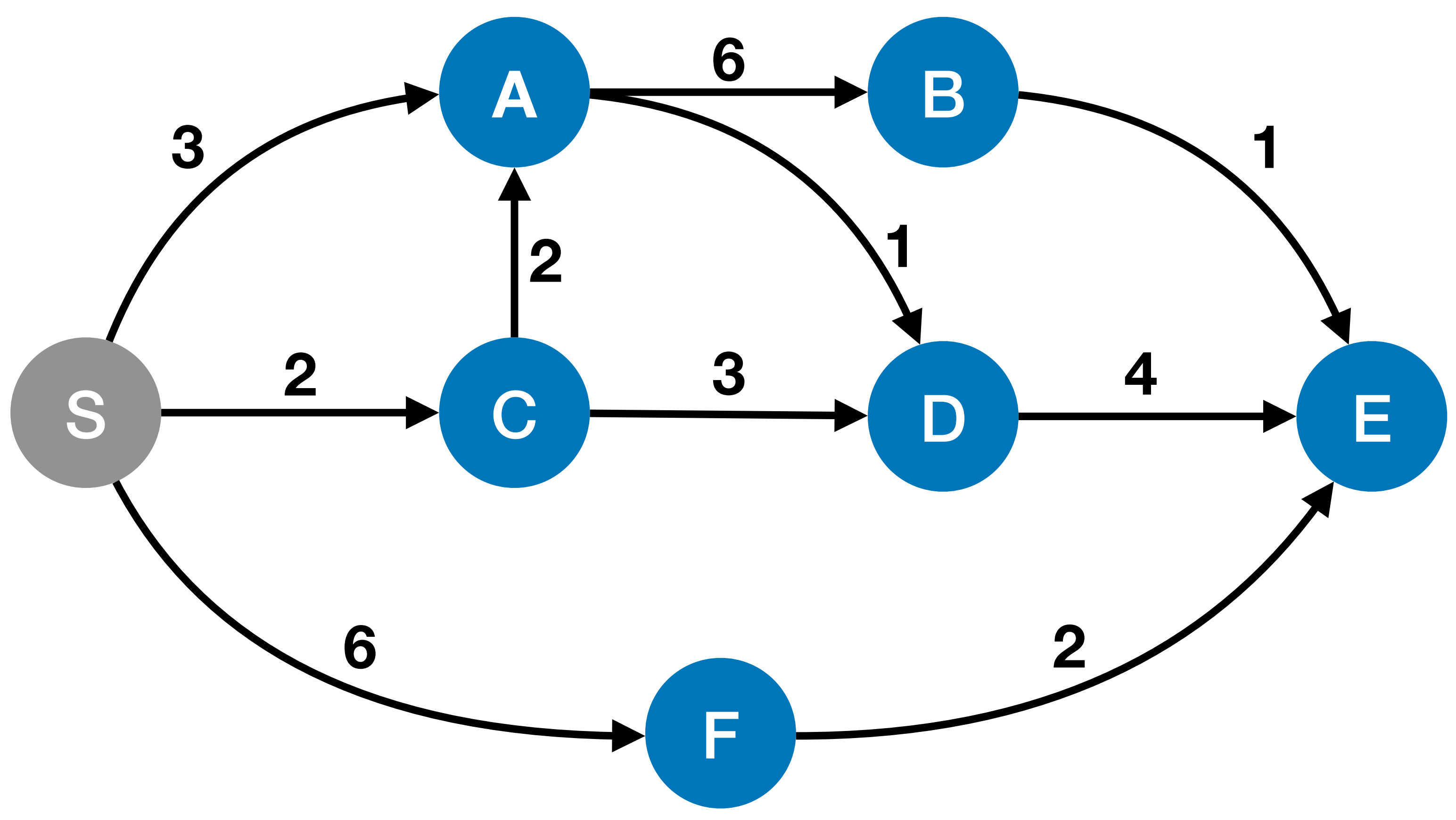



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]        Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node

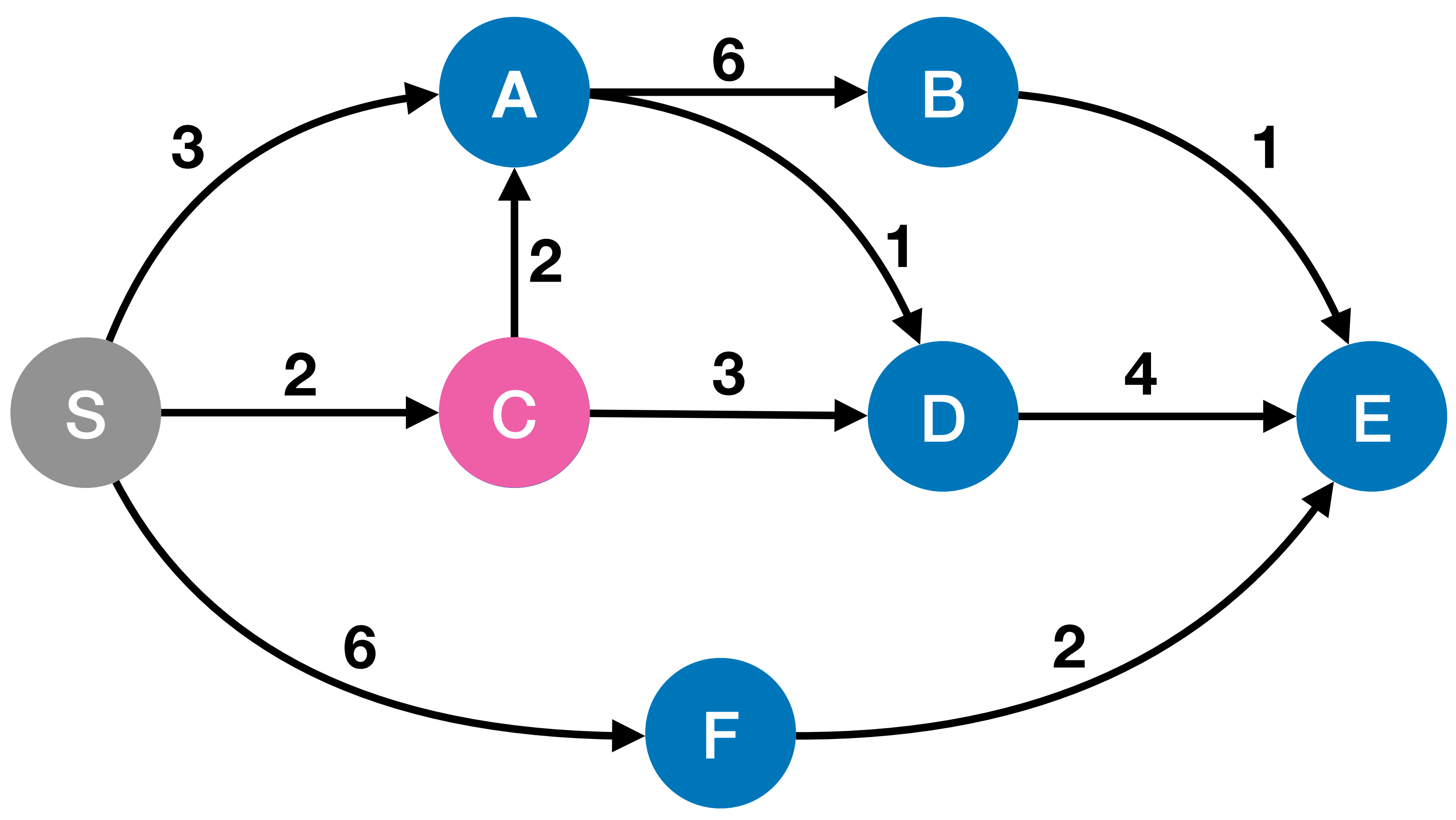



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]        Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node
- This time, it is node (C).

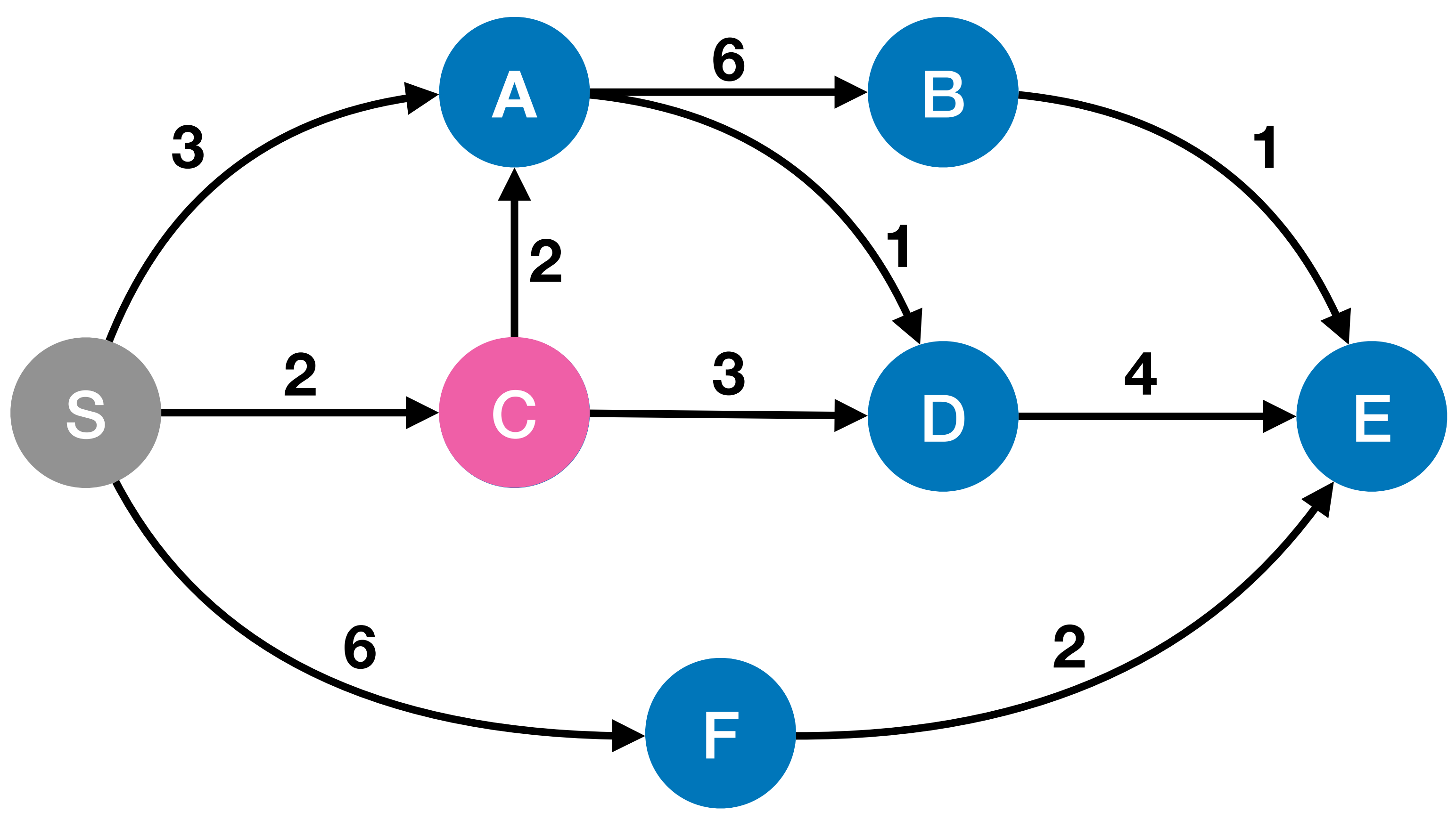



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]  Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]

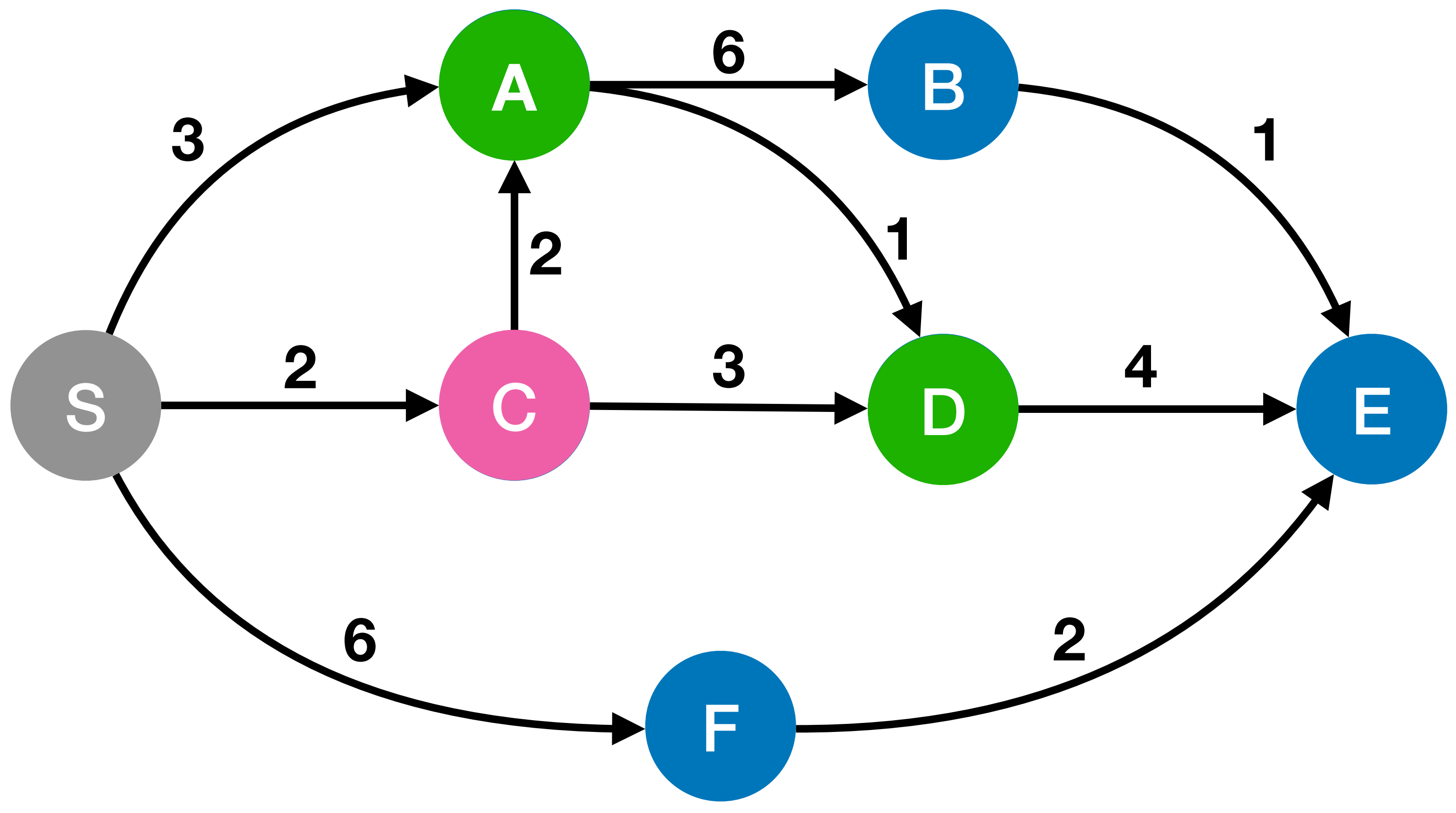
Unexplored = [A, C, F, D, B, E ]



Iterative step - Begin Iter 2

# Dijkstra's algorithm

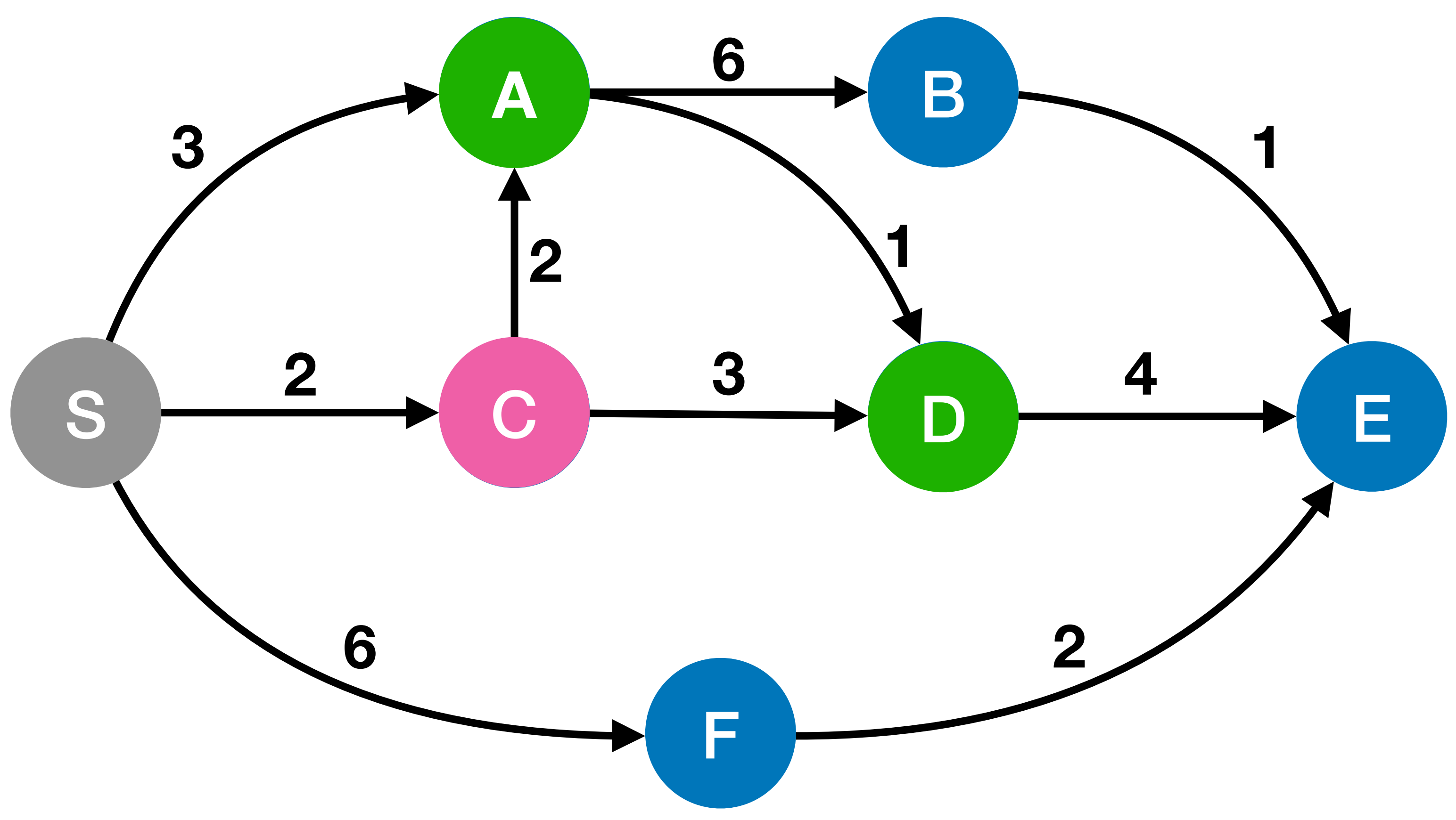
- For the current node, examine its unexplored neighbors
- Current node → C; unexplored neighbors → {A & D}




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]    Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm



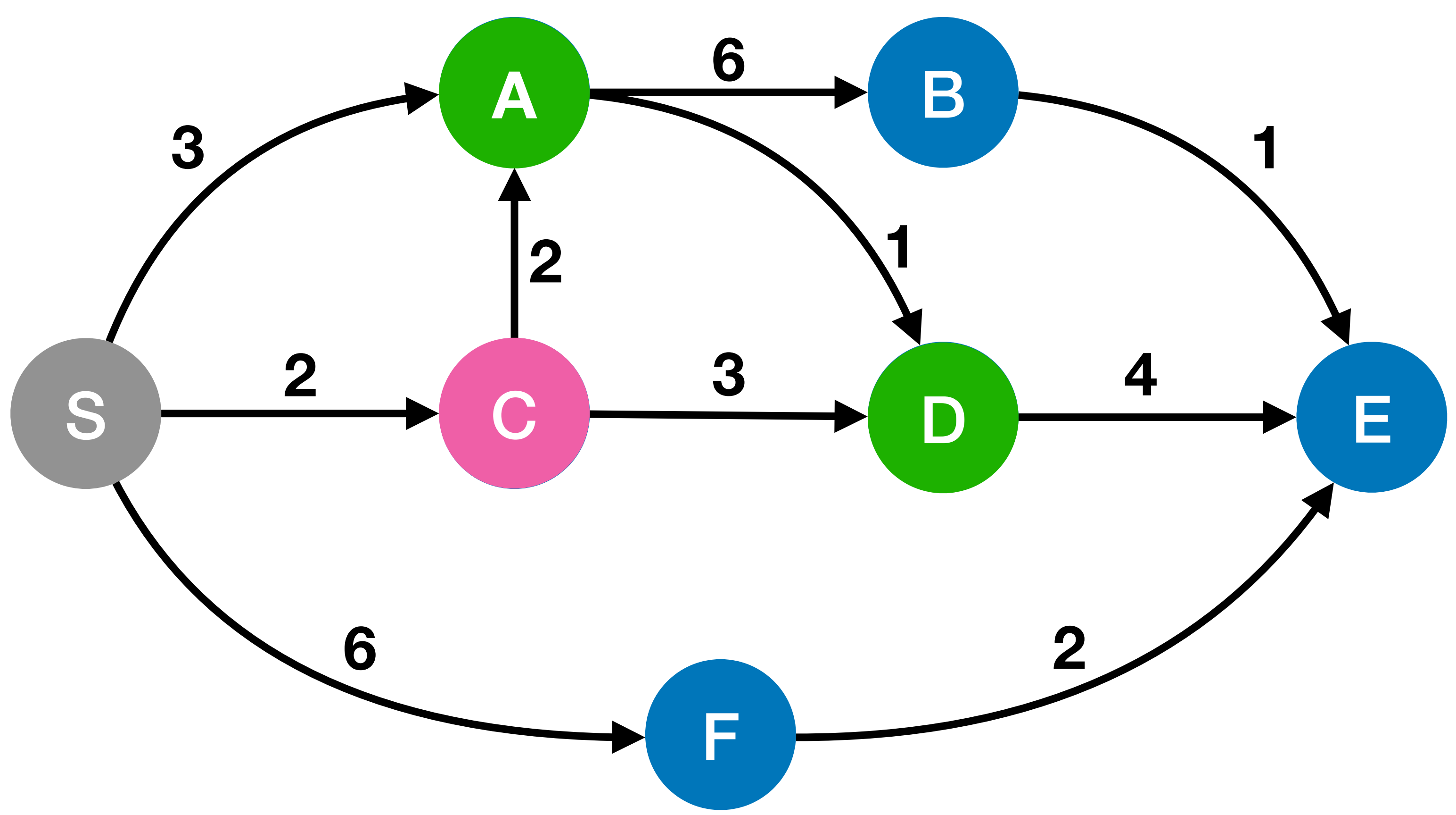

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]

Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.

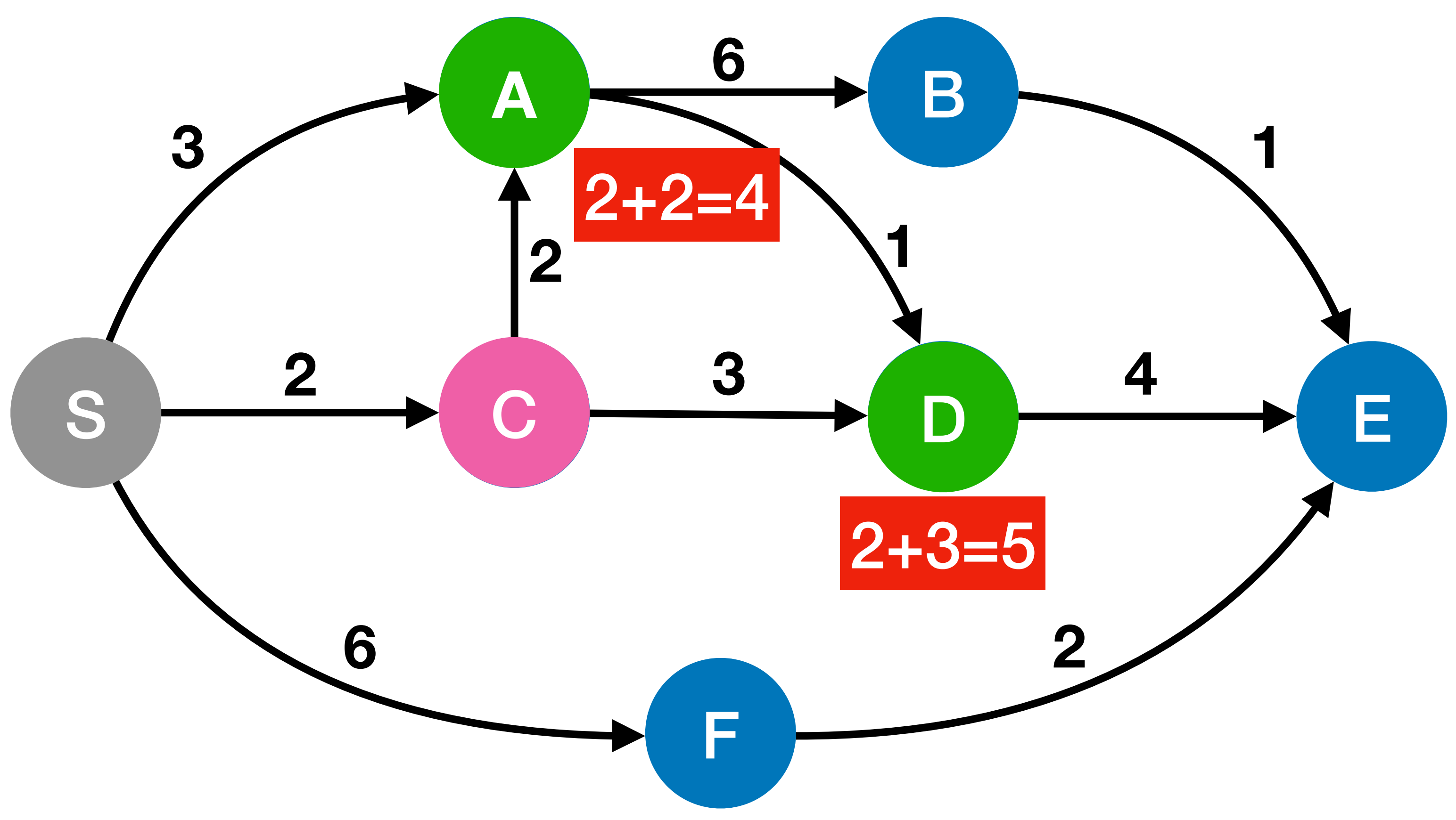



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]     Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.

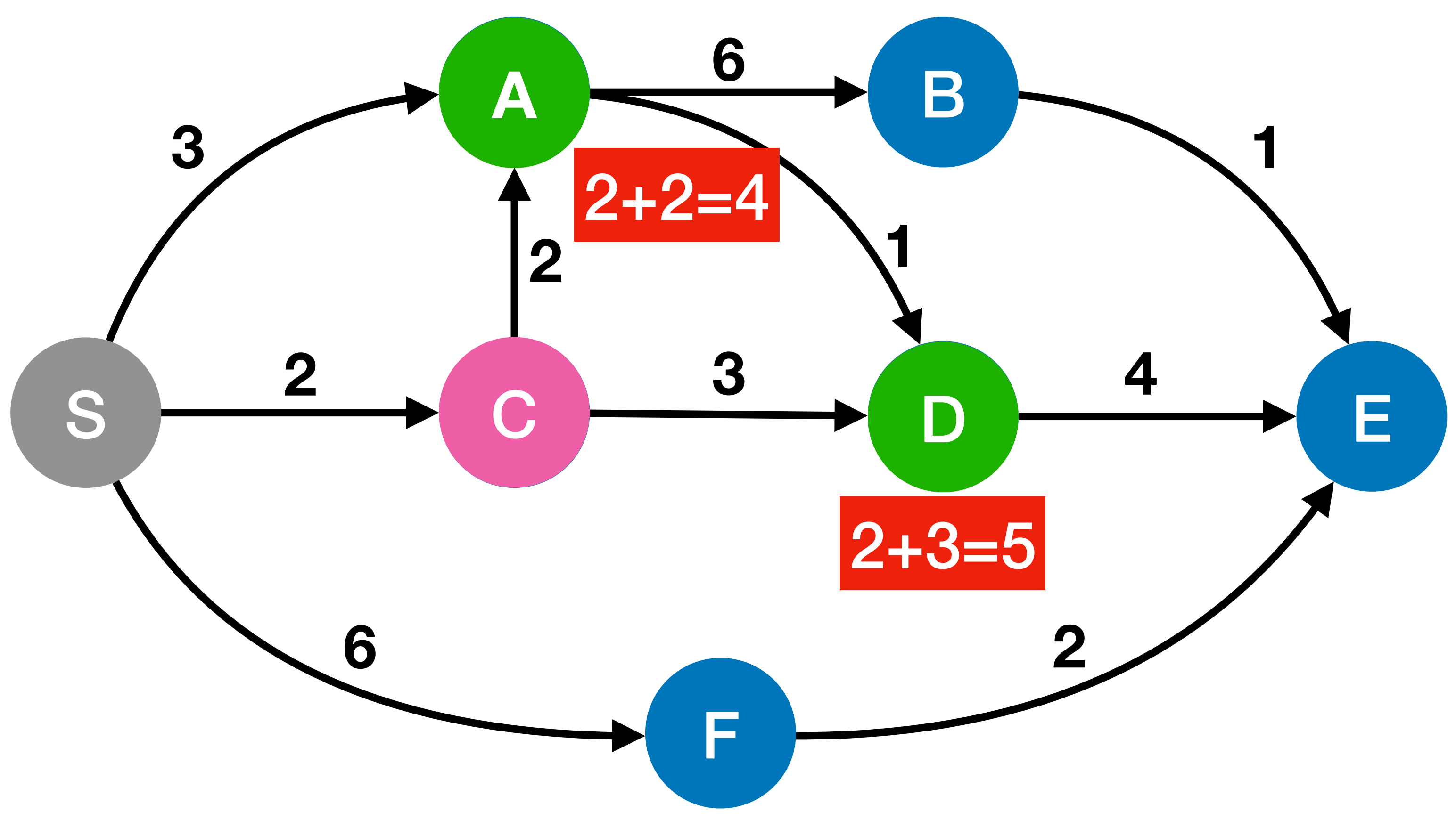



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ] Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.

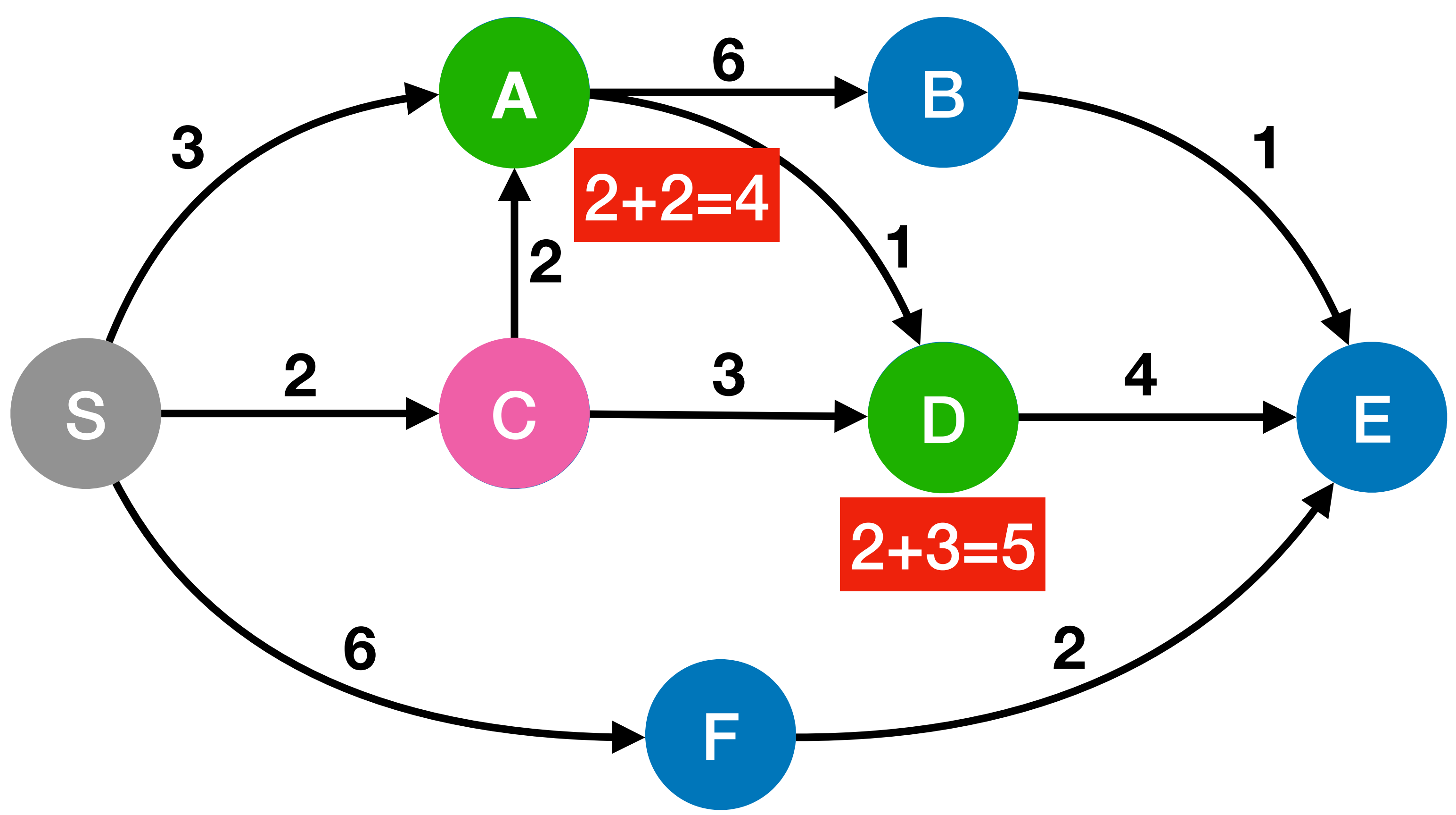



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | ∞ | |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]    Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

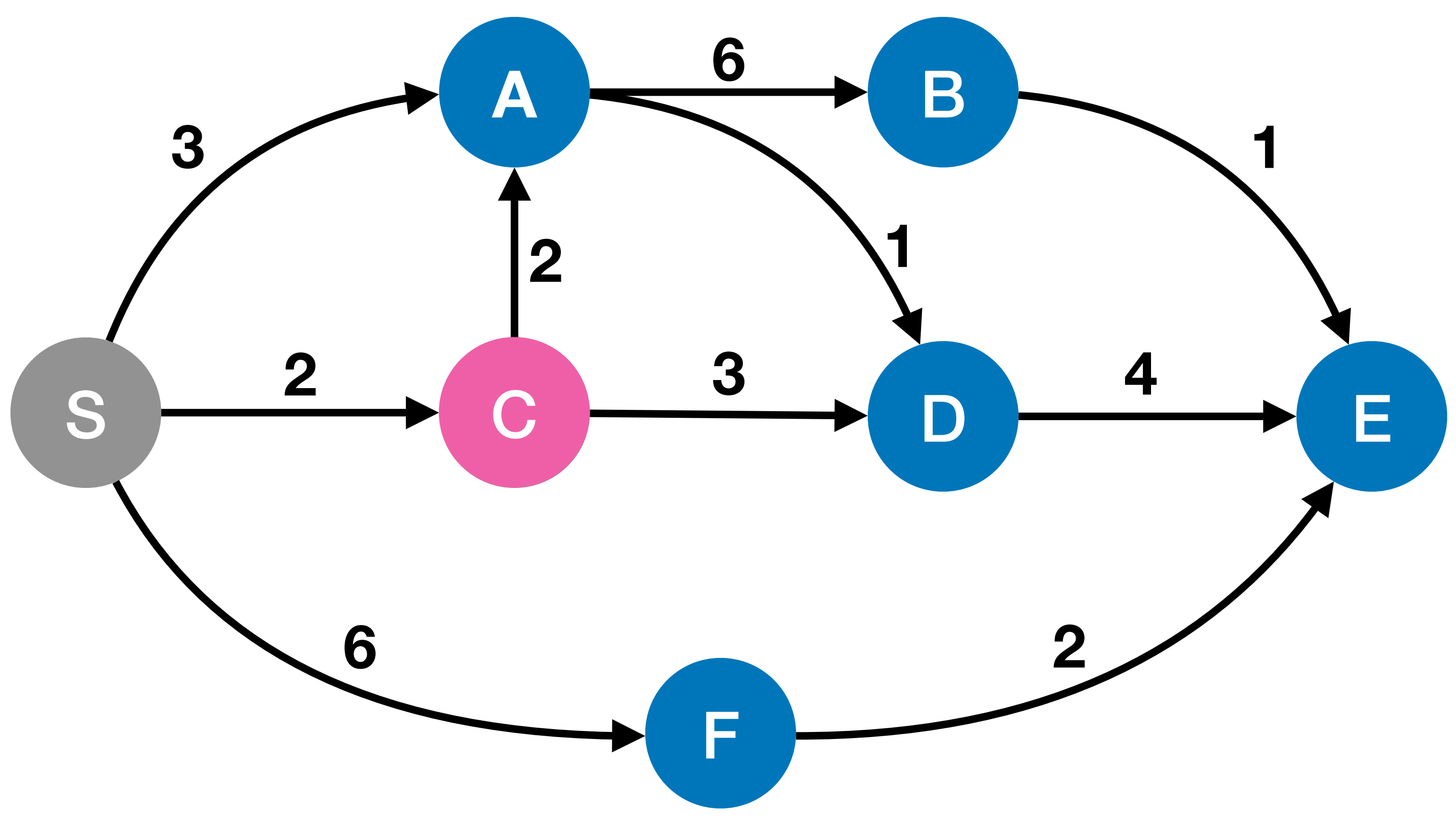
- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S ]    Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

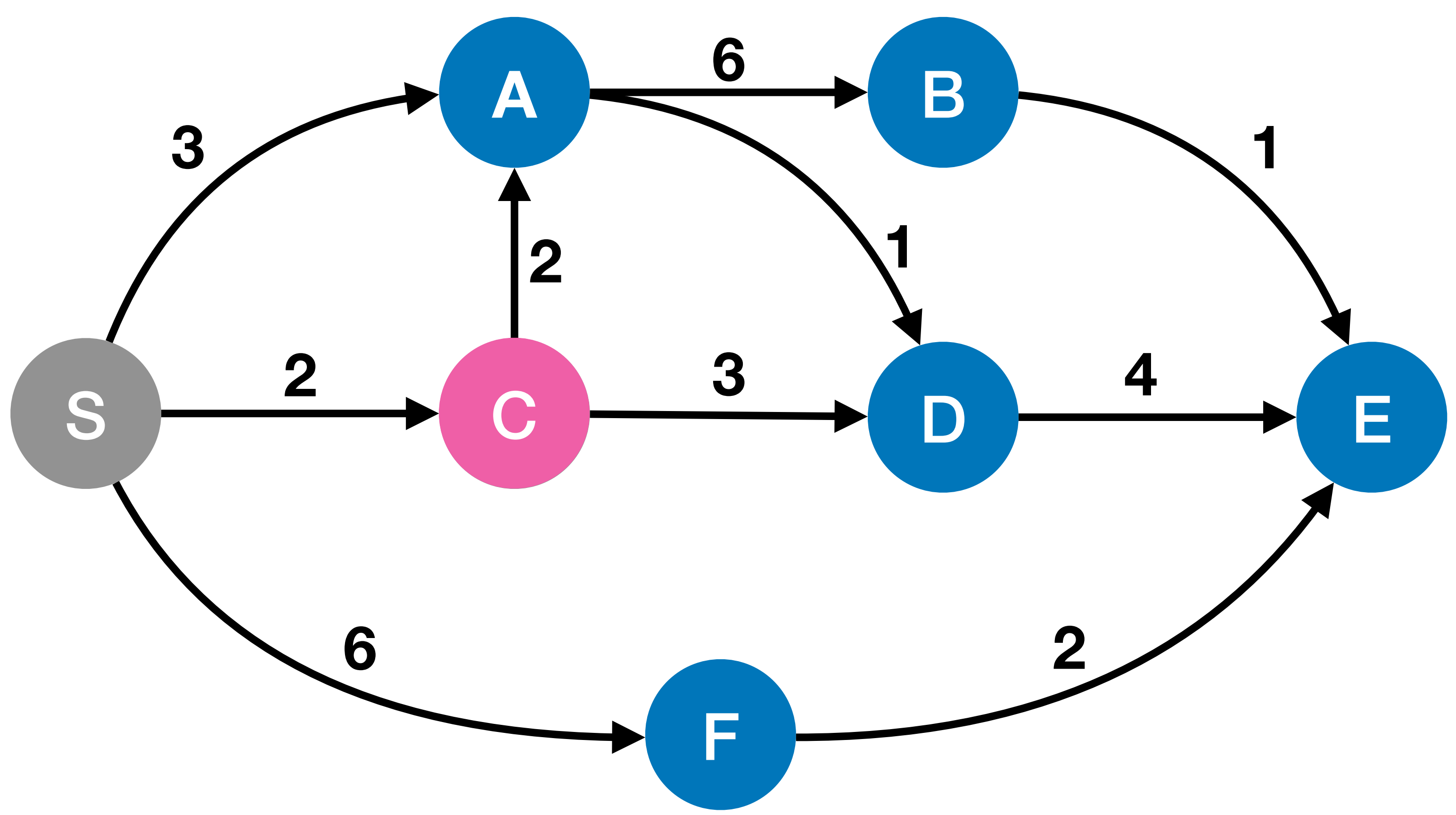



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, ]    Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

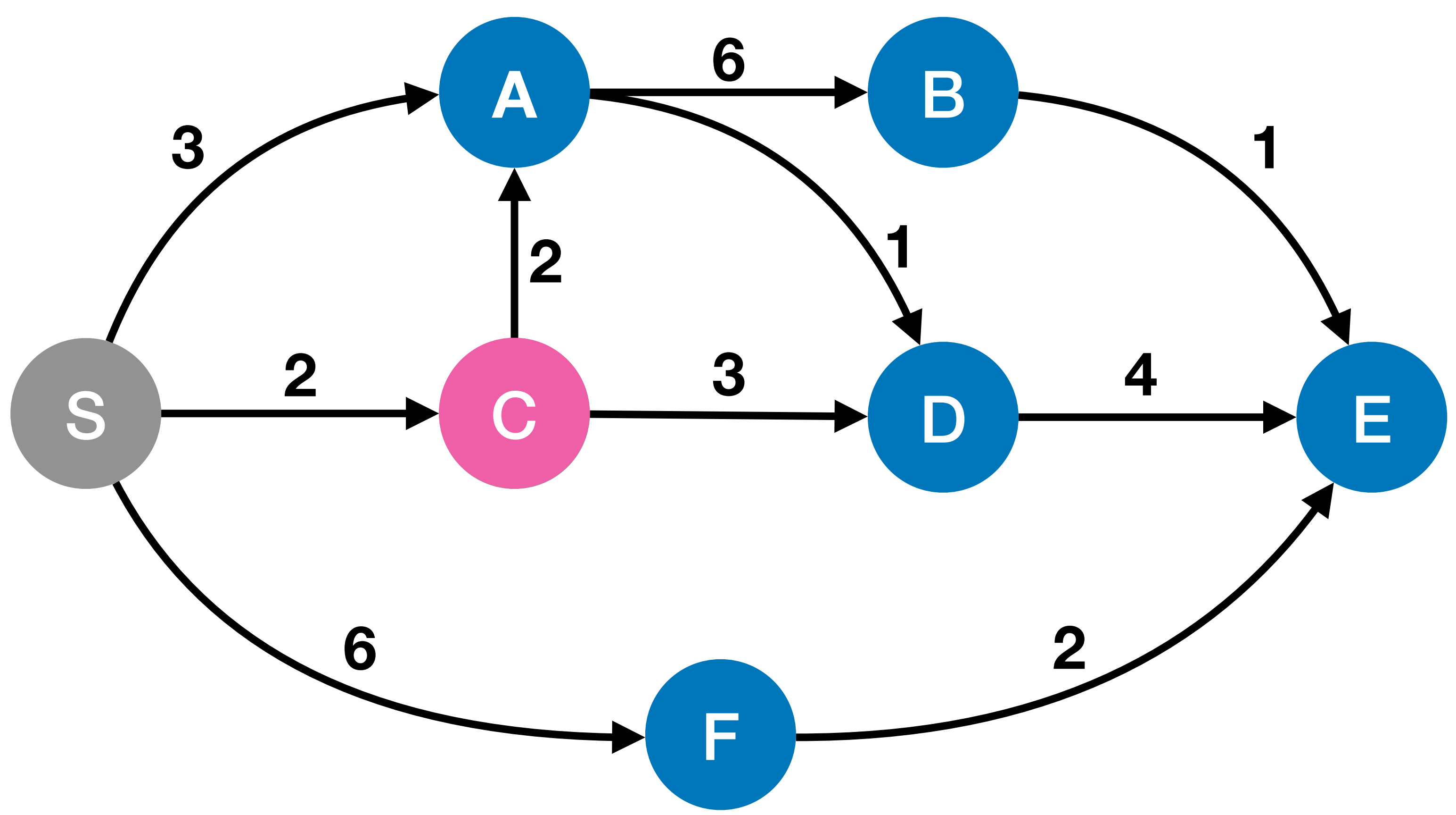



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, ]  Unexplored = [A, C, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

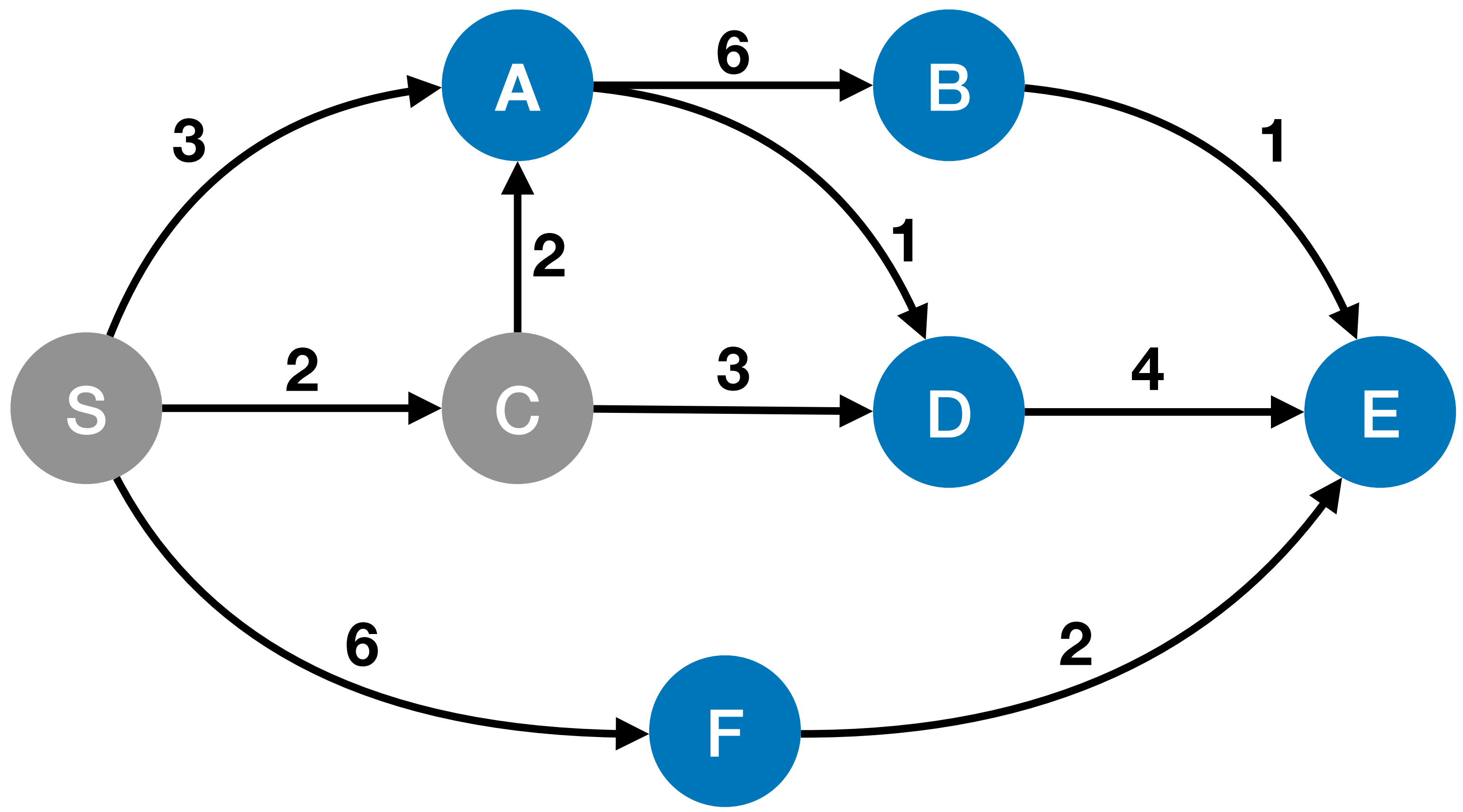



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, ]  Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

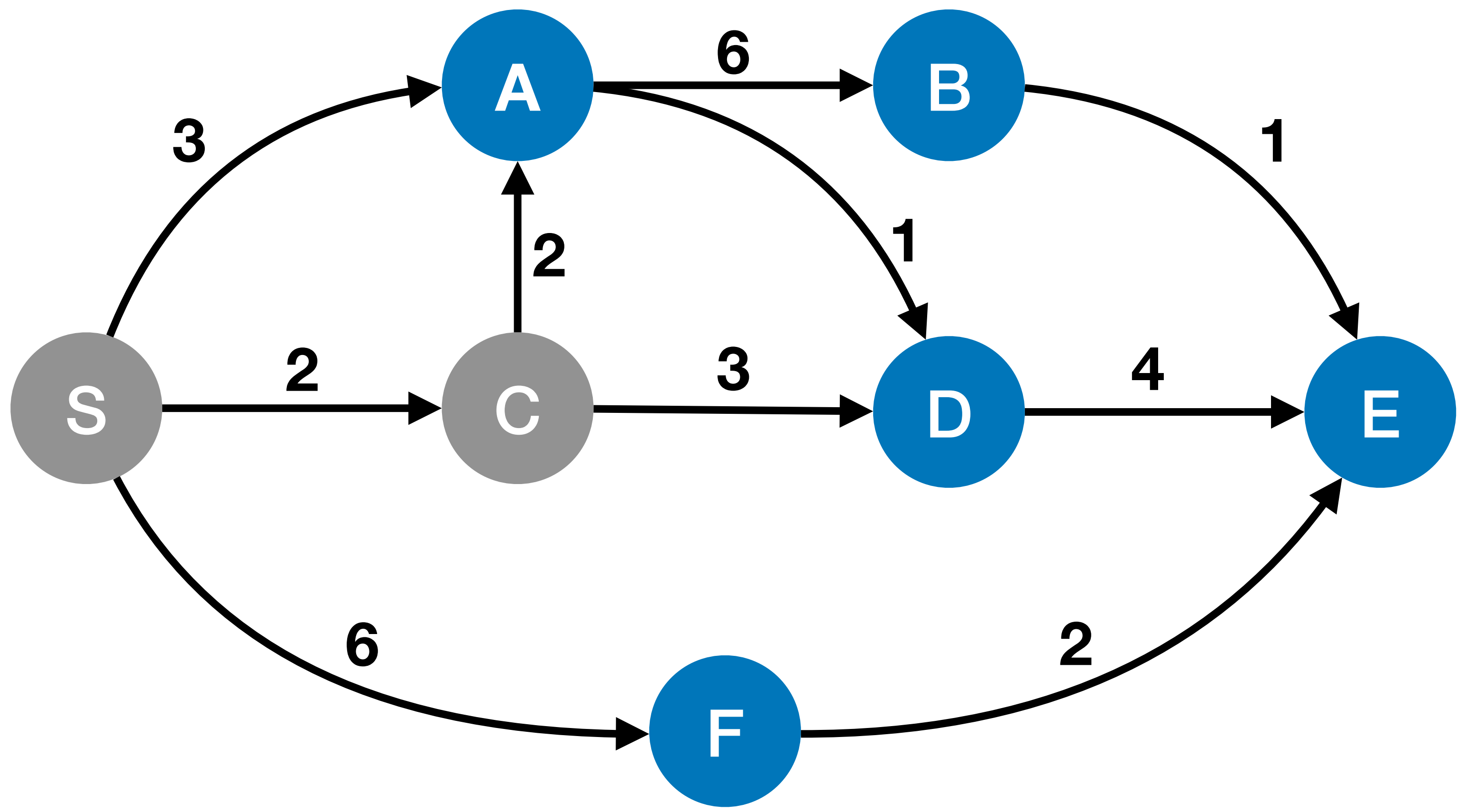



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ] Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes



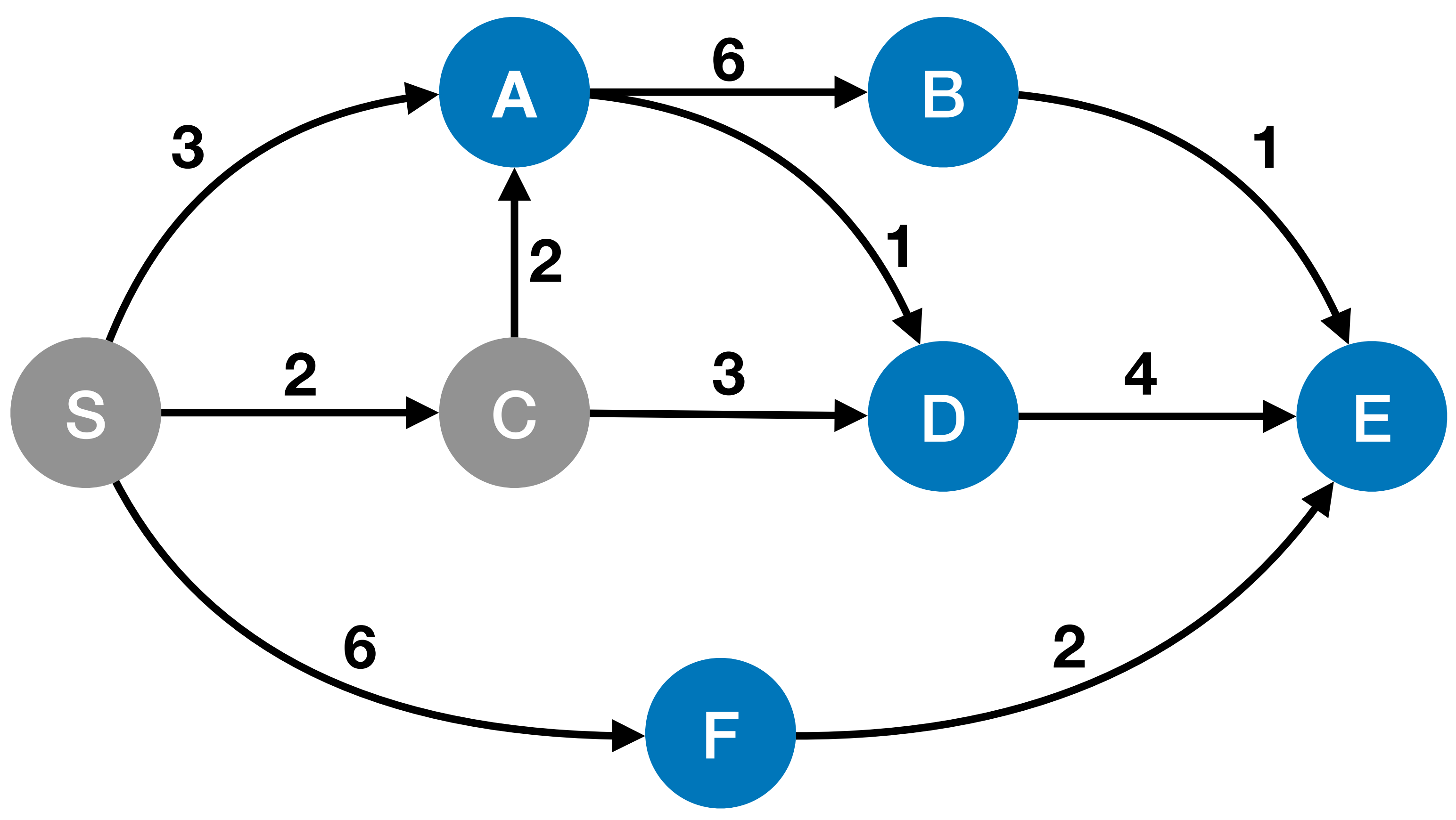

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| **C** | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]     Unexplored = [A, F, D, B, E ]



**Iterative step - End Iter 2**

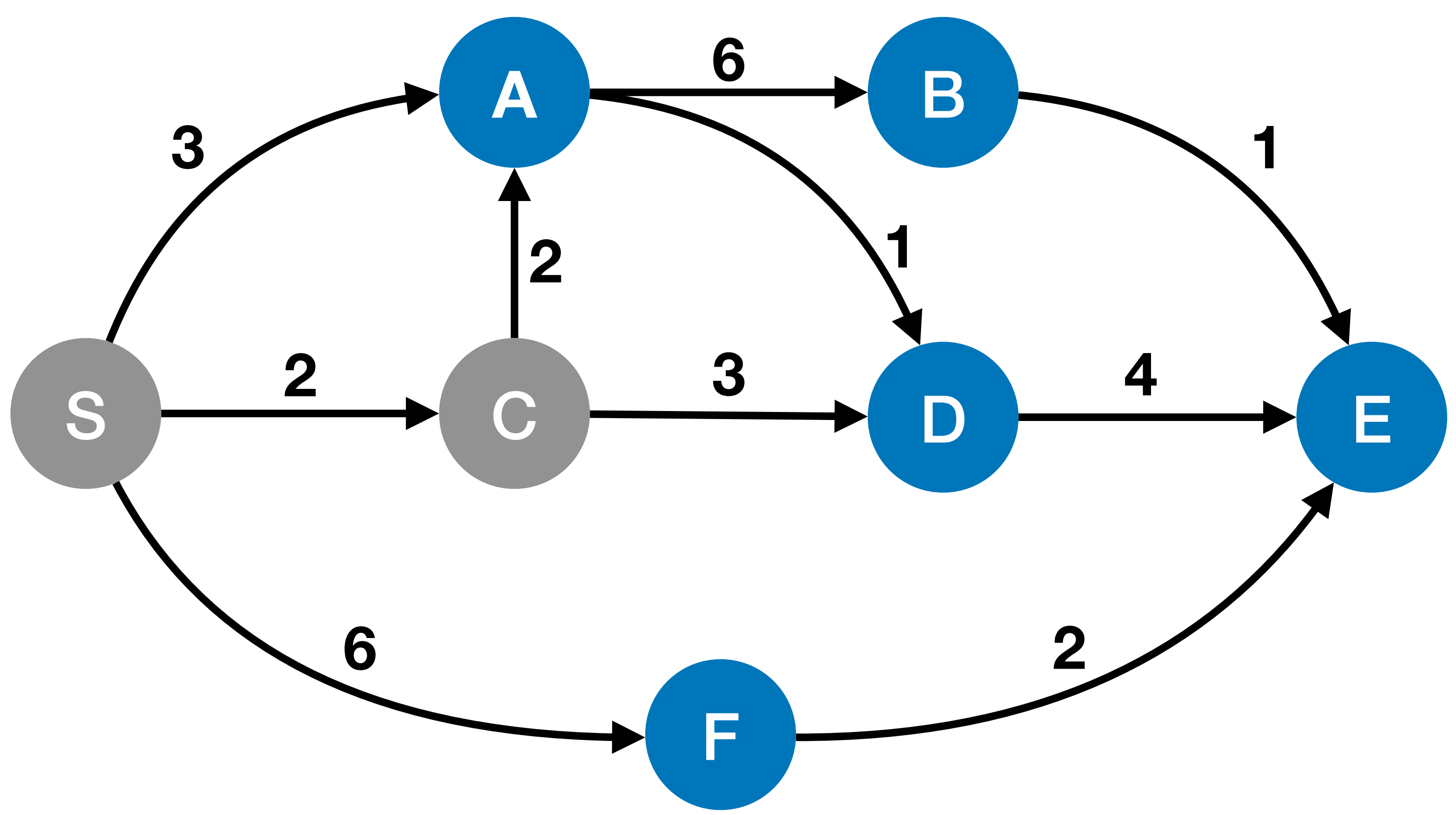
# Dijkstra's algorithm




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]  Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

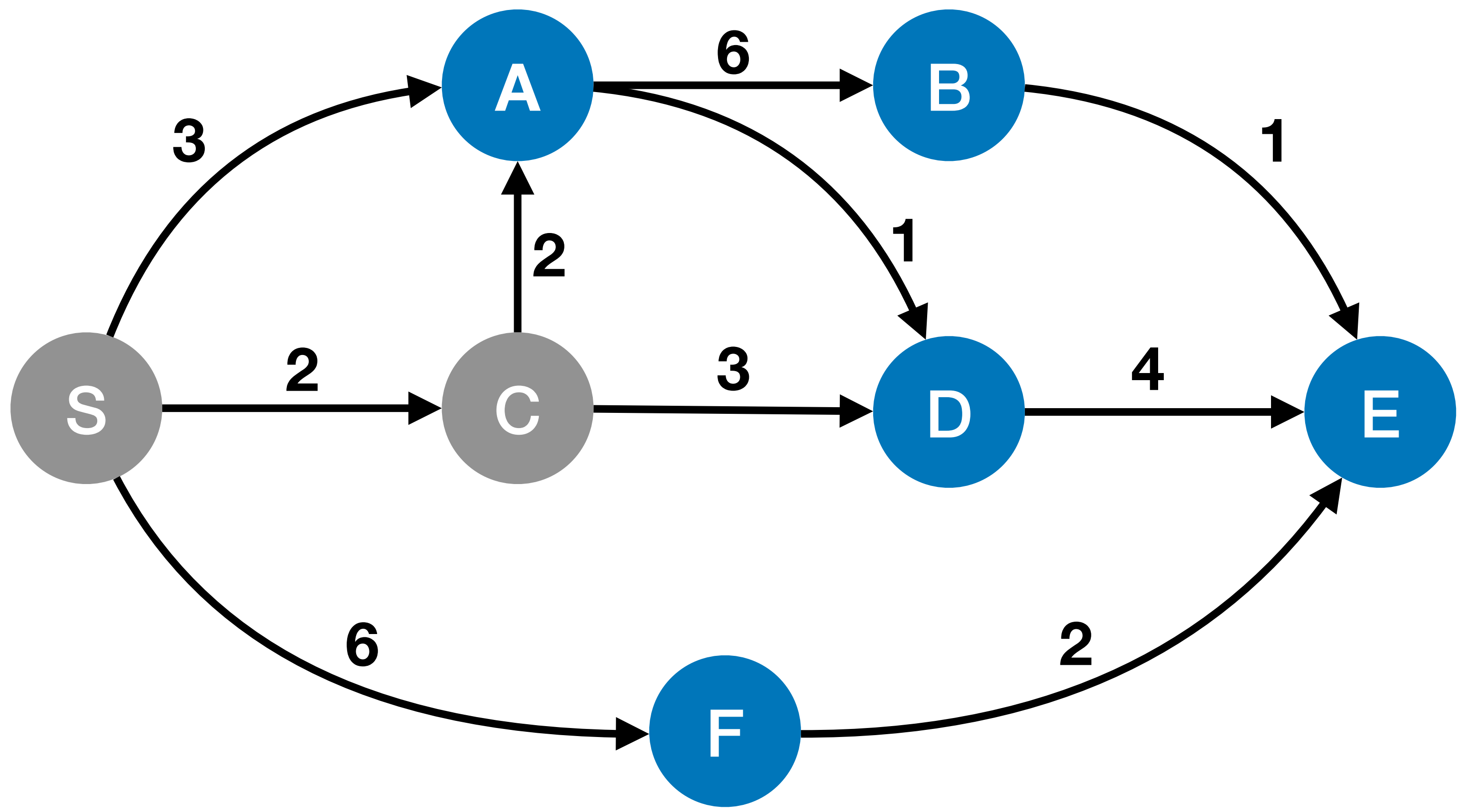



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]    Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node

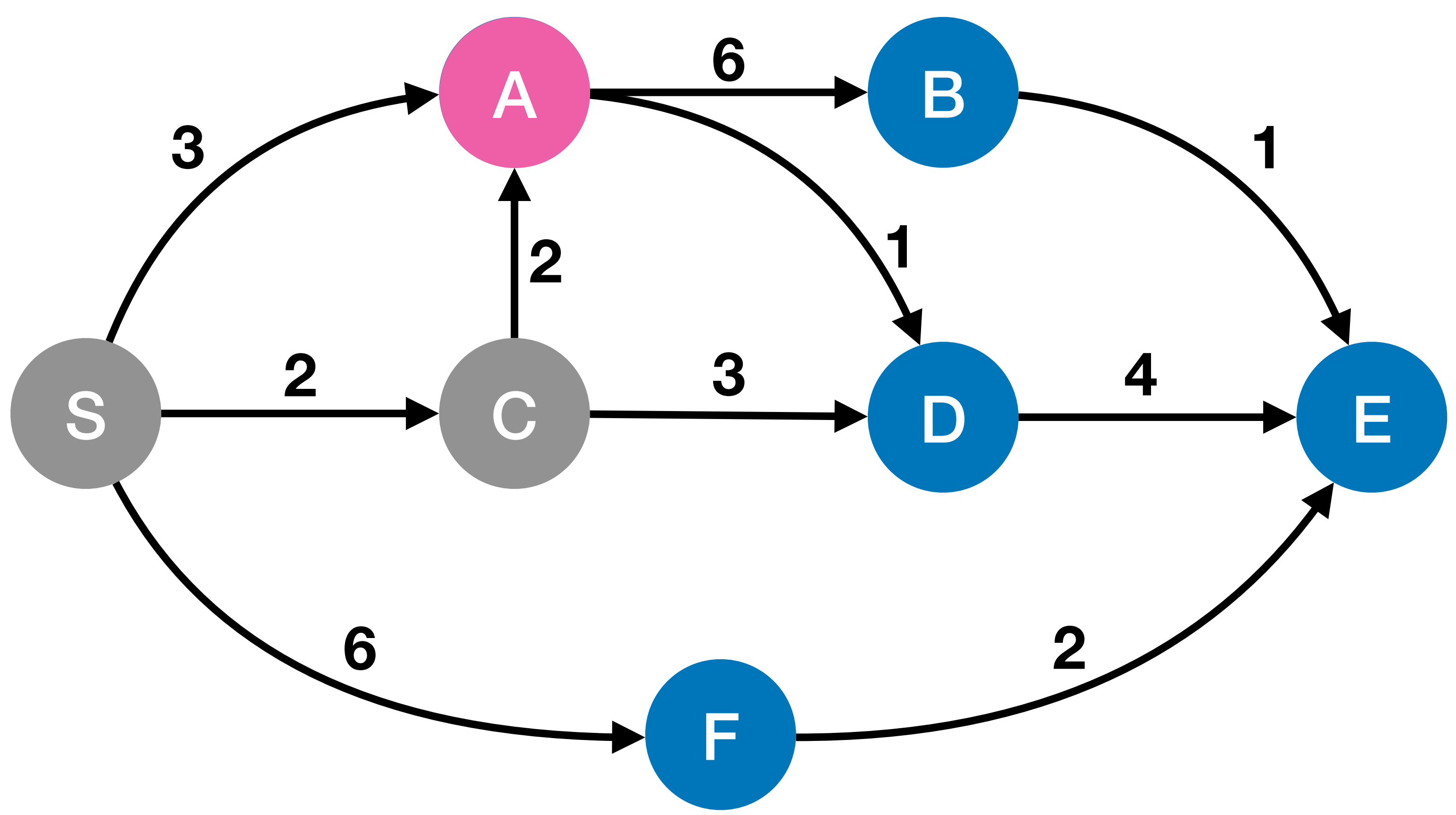



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]     Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node
- This time, it is node (A).

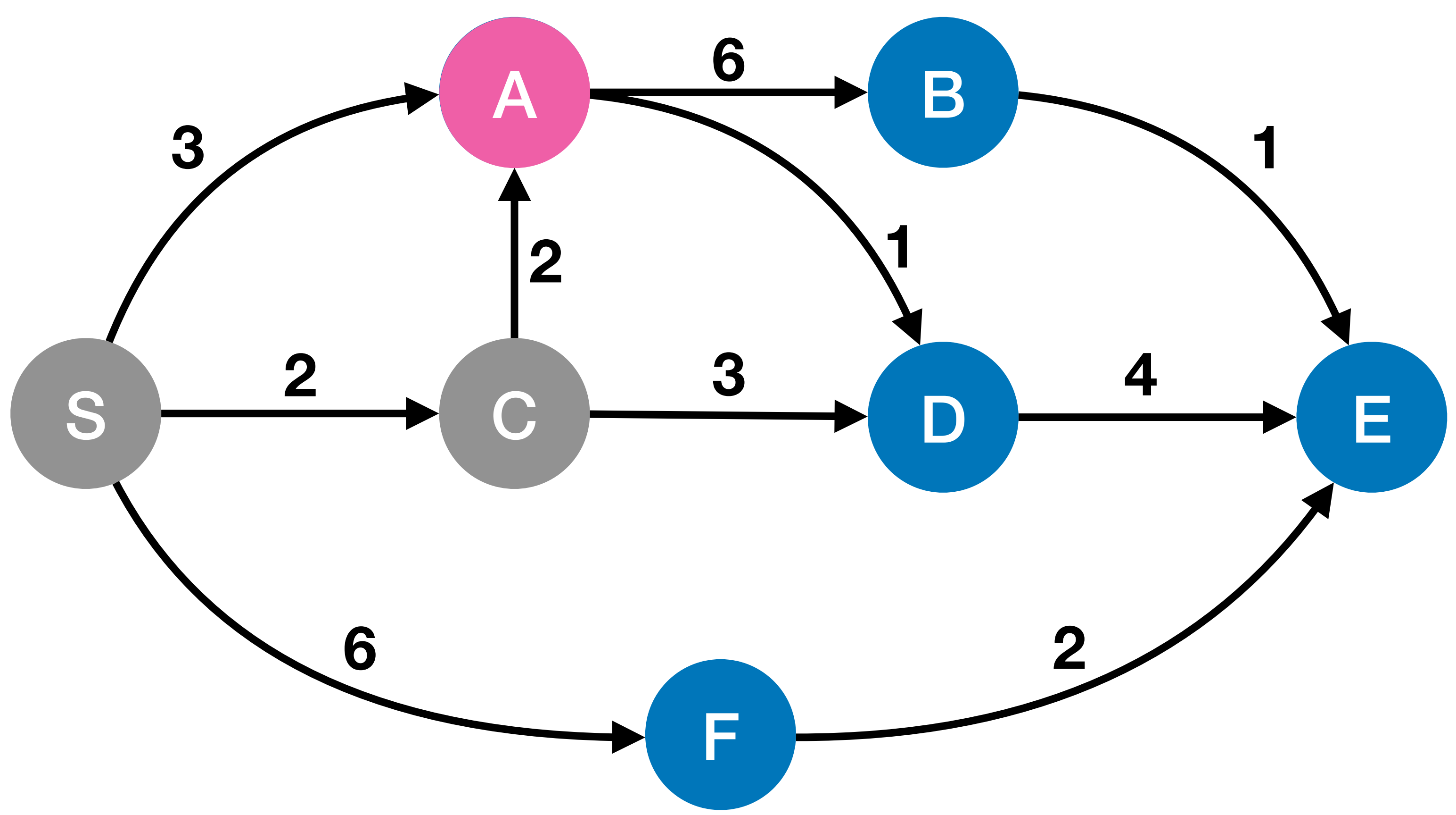



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]    Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors

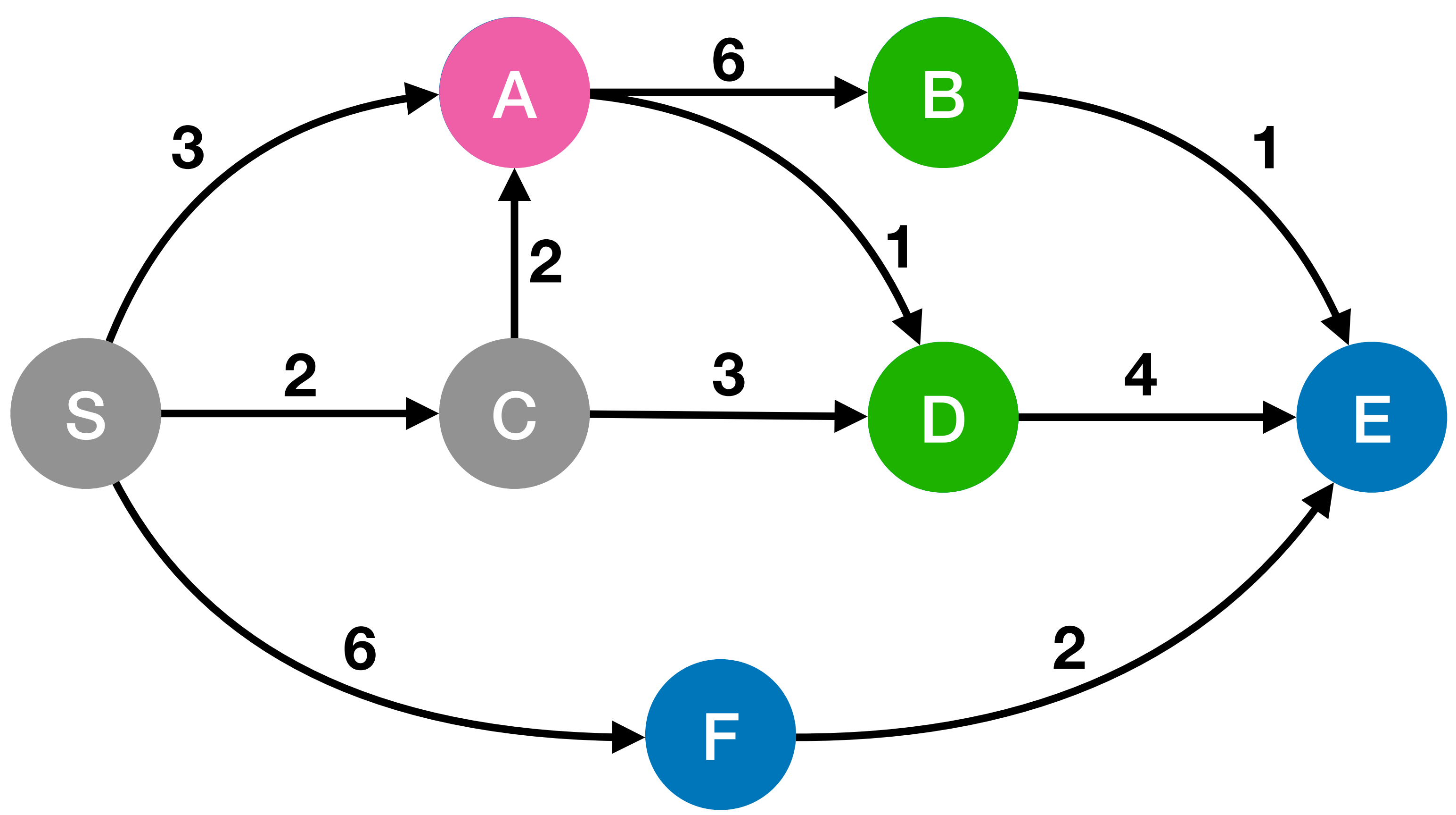



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ] Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

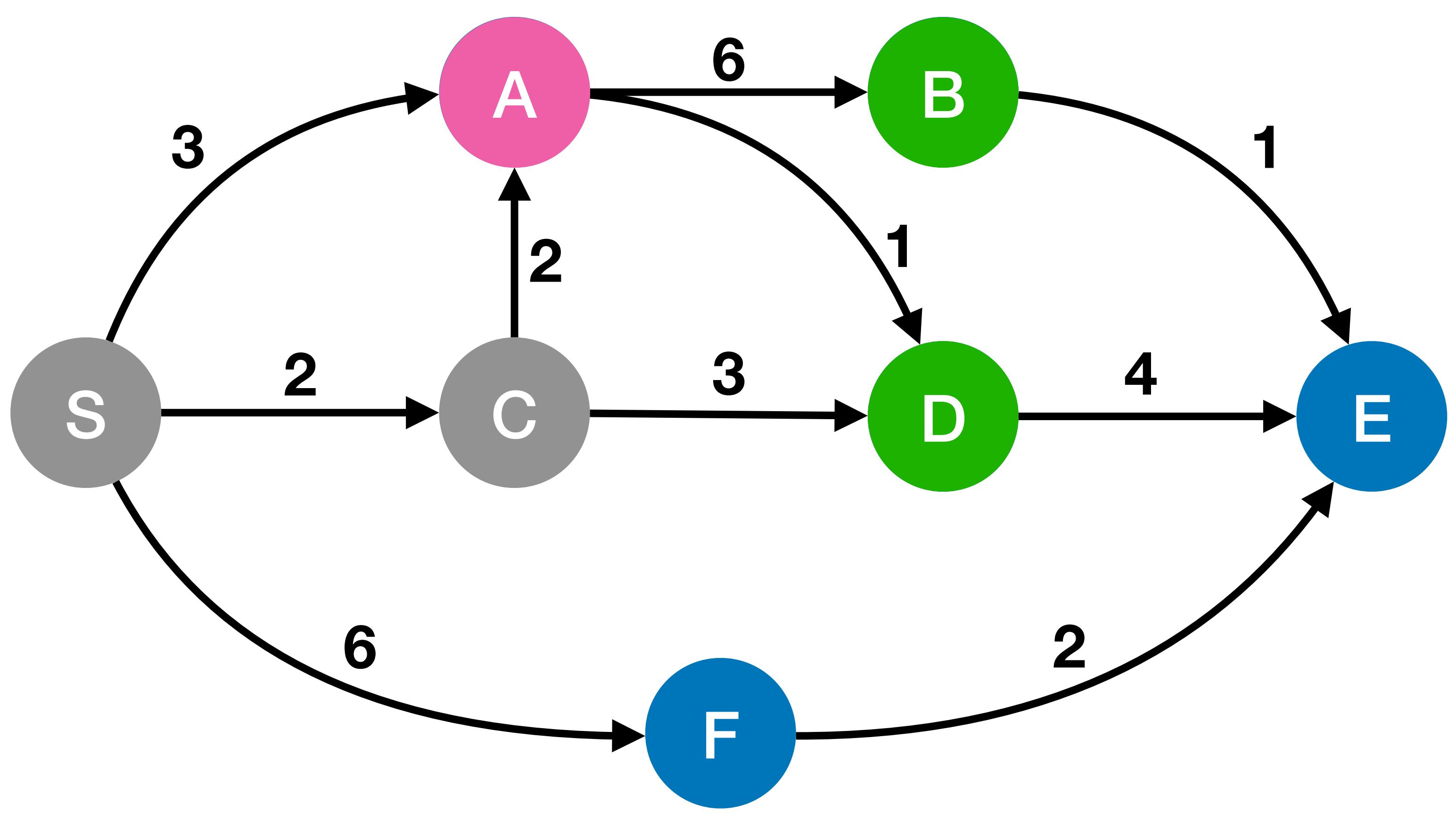
- For the current node, examine its unexplored neighbors
- Current node → A; unexplored neighbors → {B & D}




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]      Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

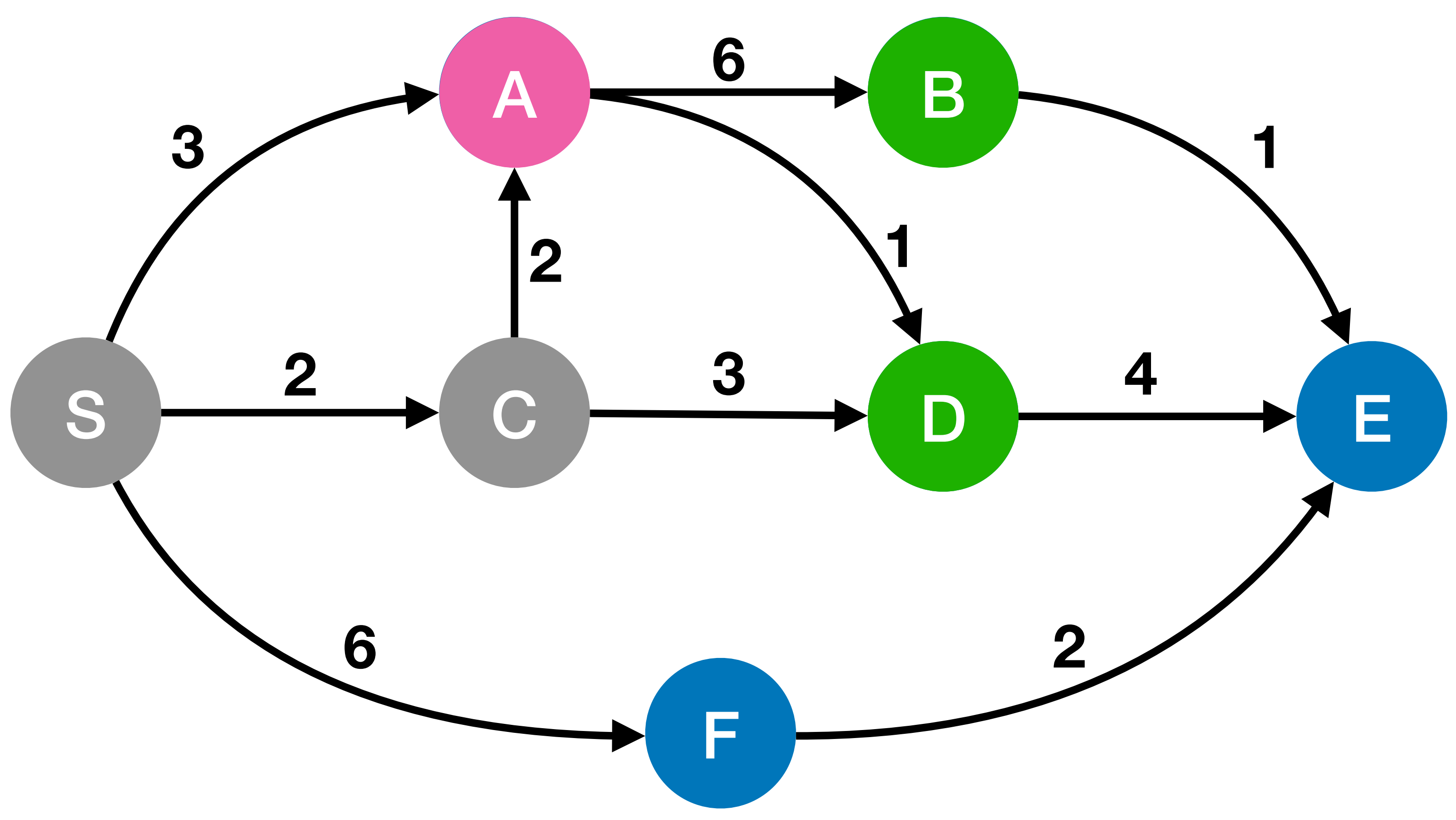



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| **A** | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]          Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.

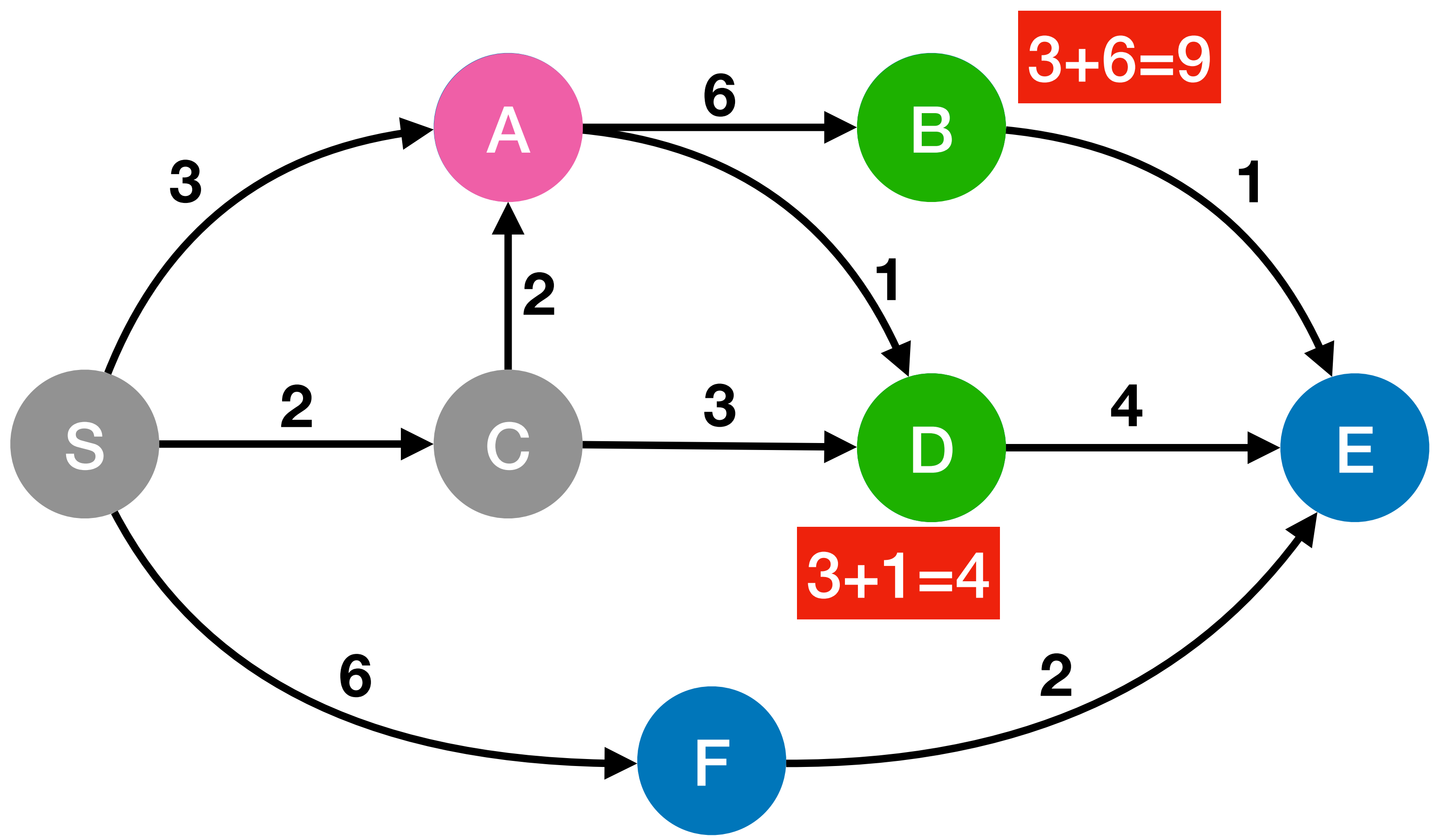



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| **A** | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]        Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.

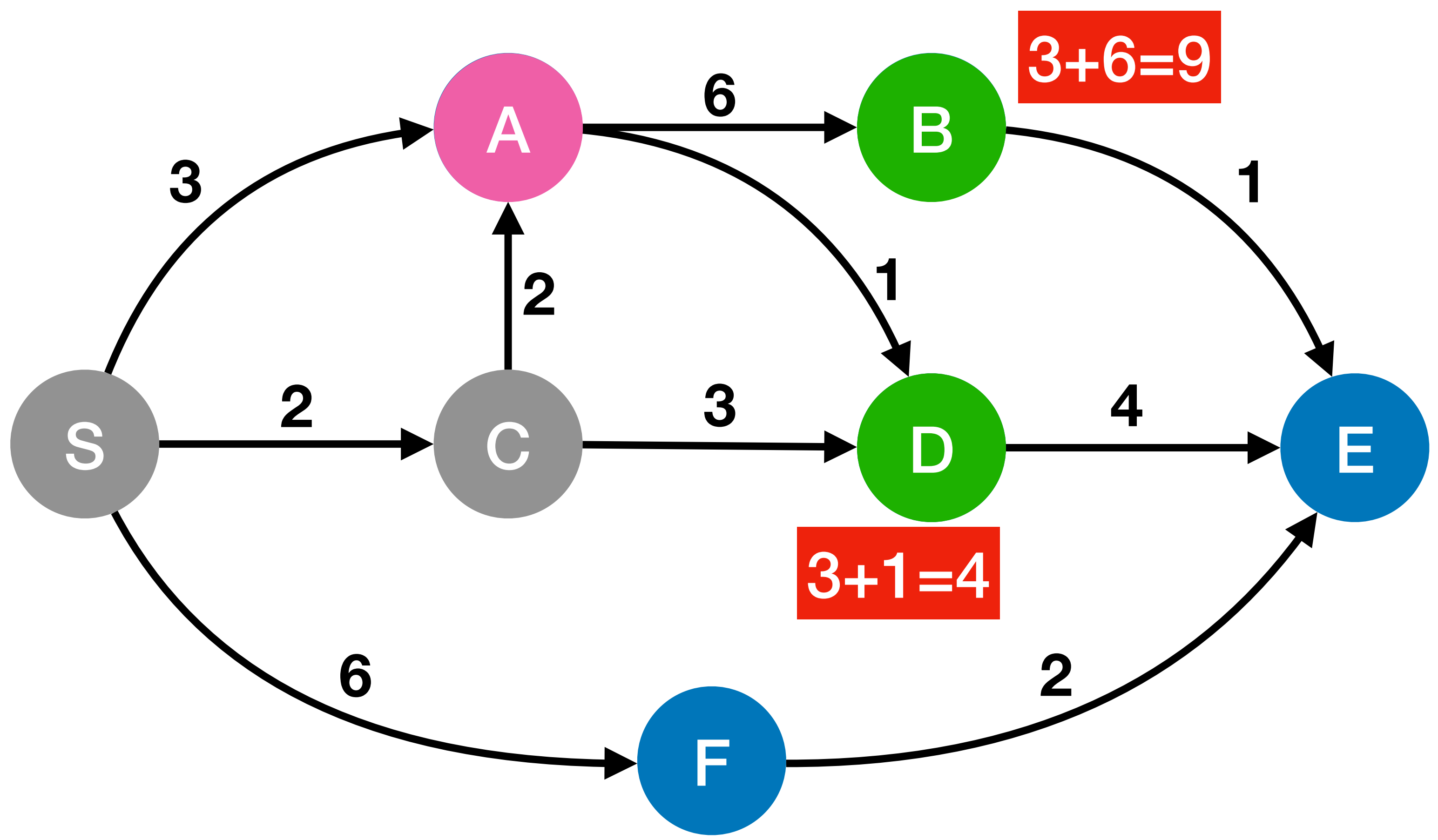



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| **A** | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]    Unexplored = [A, F, D, B, E ]

# Dijkstra’s algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.

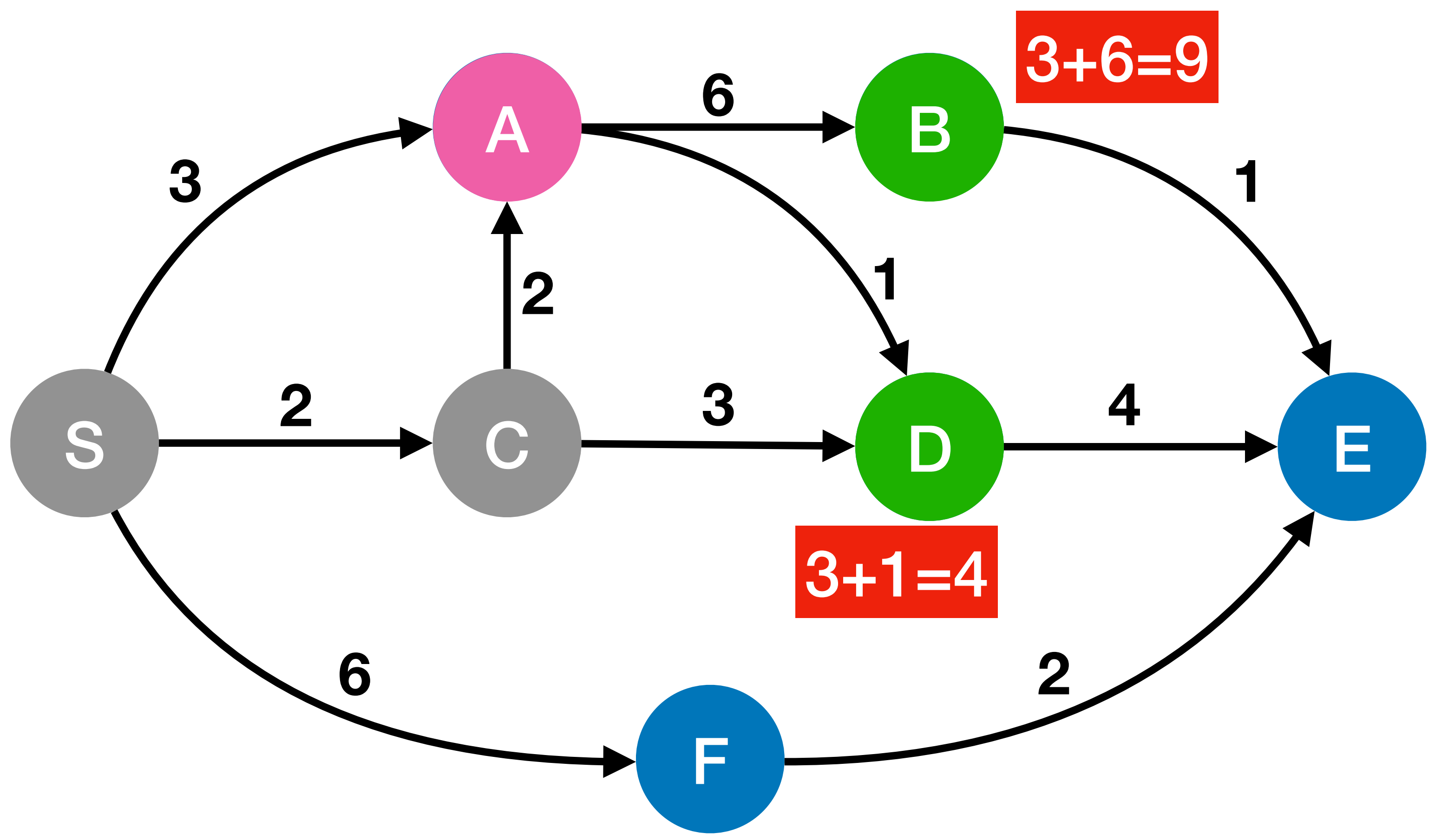



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| **A** | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 5 | C |
| B | ∞ | |
| E | ∞ | |

Settled = [ S, C ]   Unexplored = [A, F, D, B, E ]

# Dijkstra’s algorithm

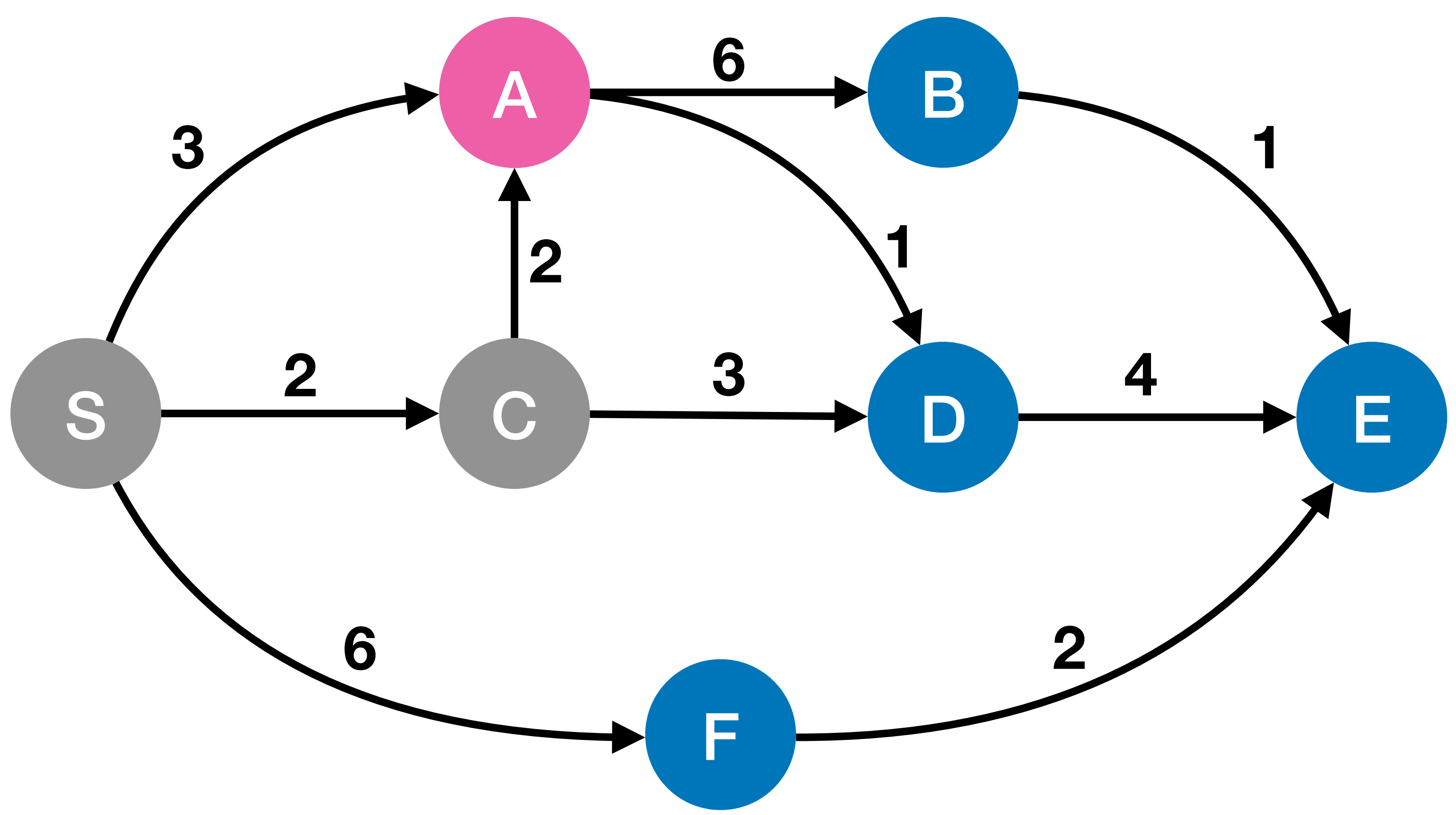
- For the current node, calculate the distance of each unsettled neighbor from the source node via current node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| **A** | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C ]    Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

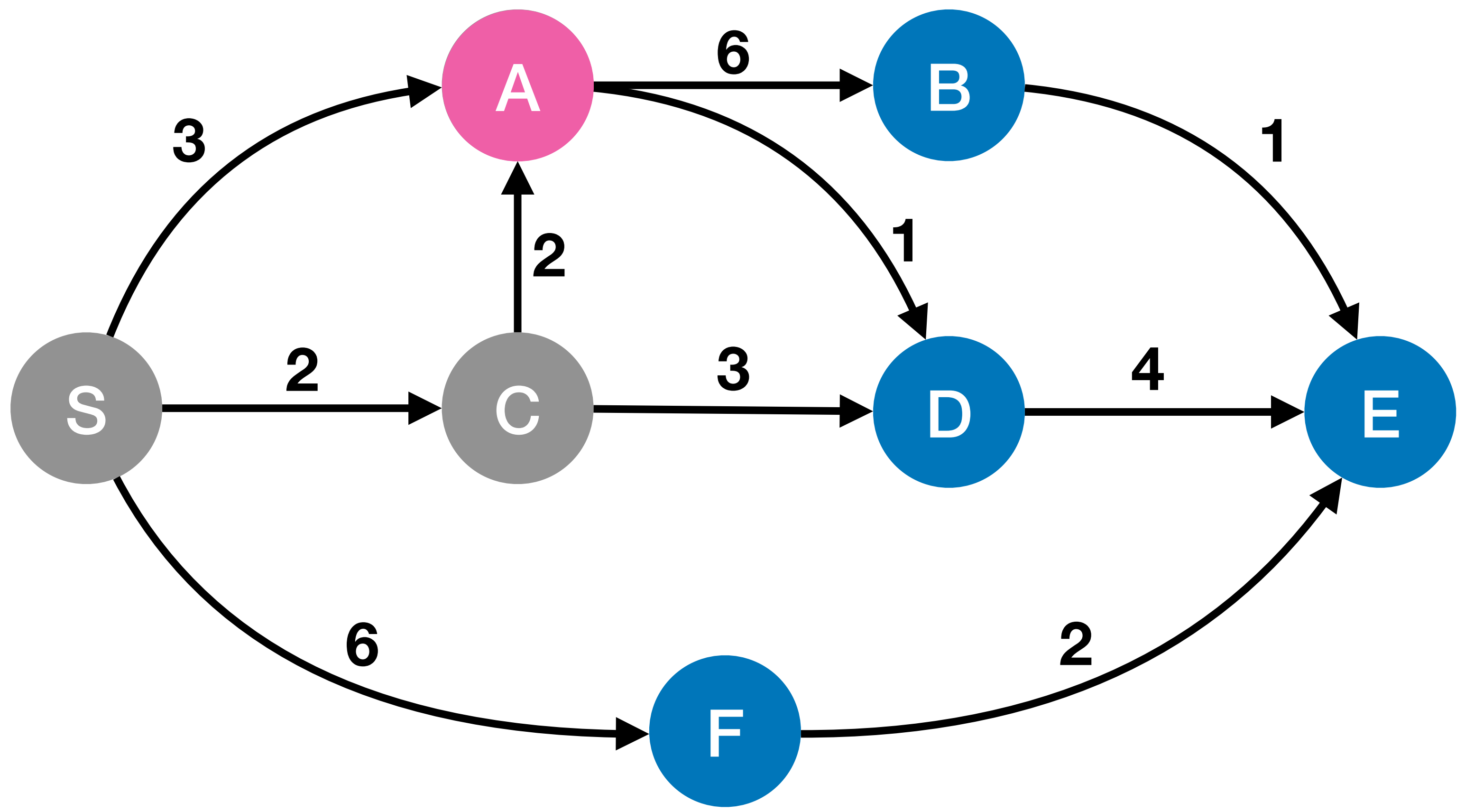



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, ]  Unexplored = [A, F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

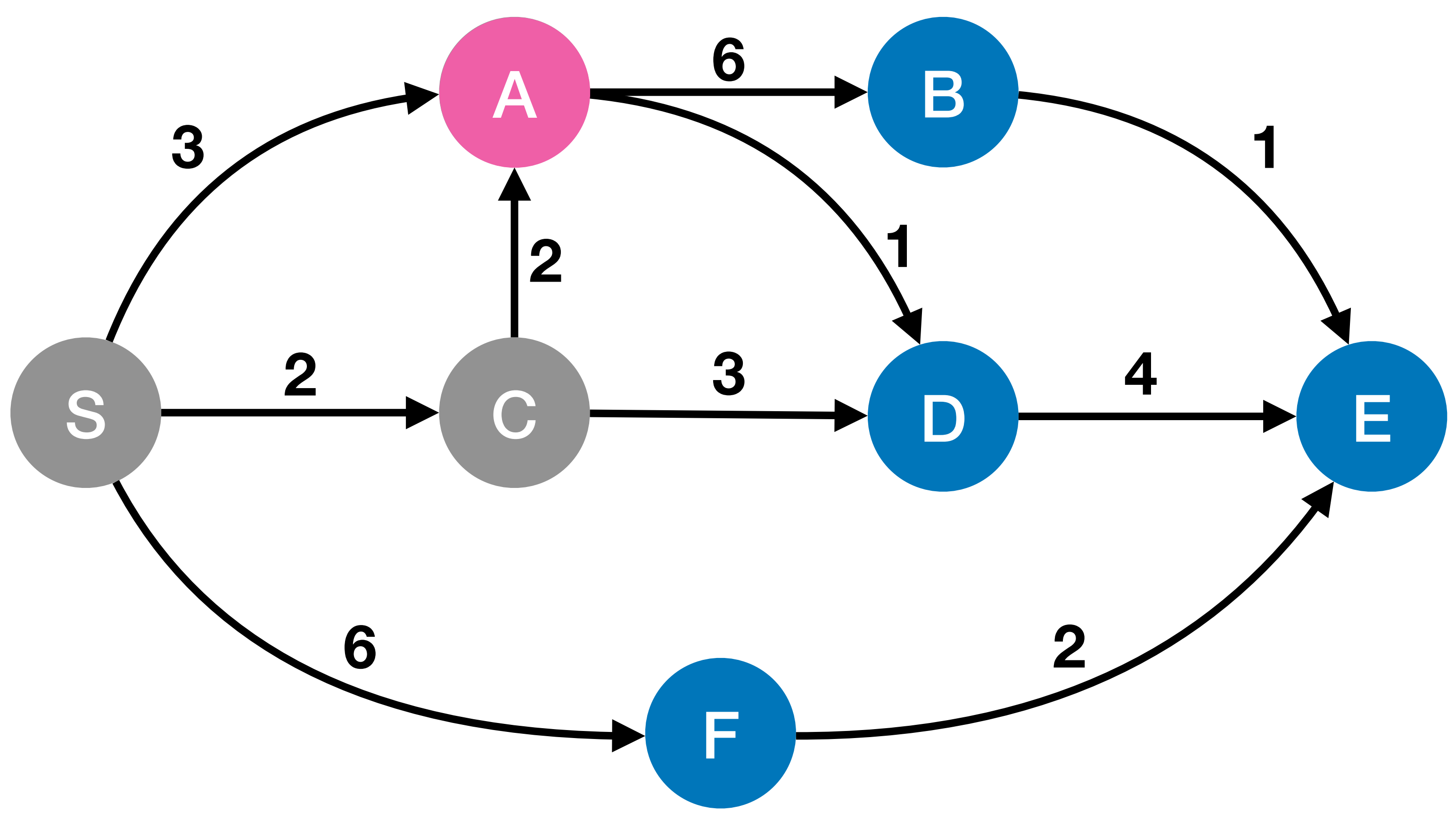



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, ]      Unexplored = [A, F, D, B, E ]

# Dijkstra’s algorithm

- Add the current node to the list of *settled* nodes

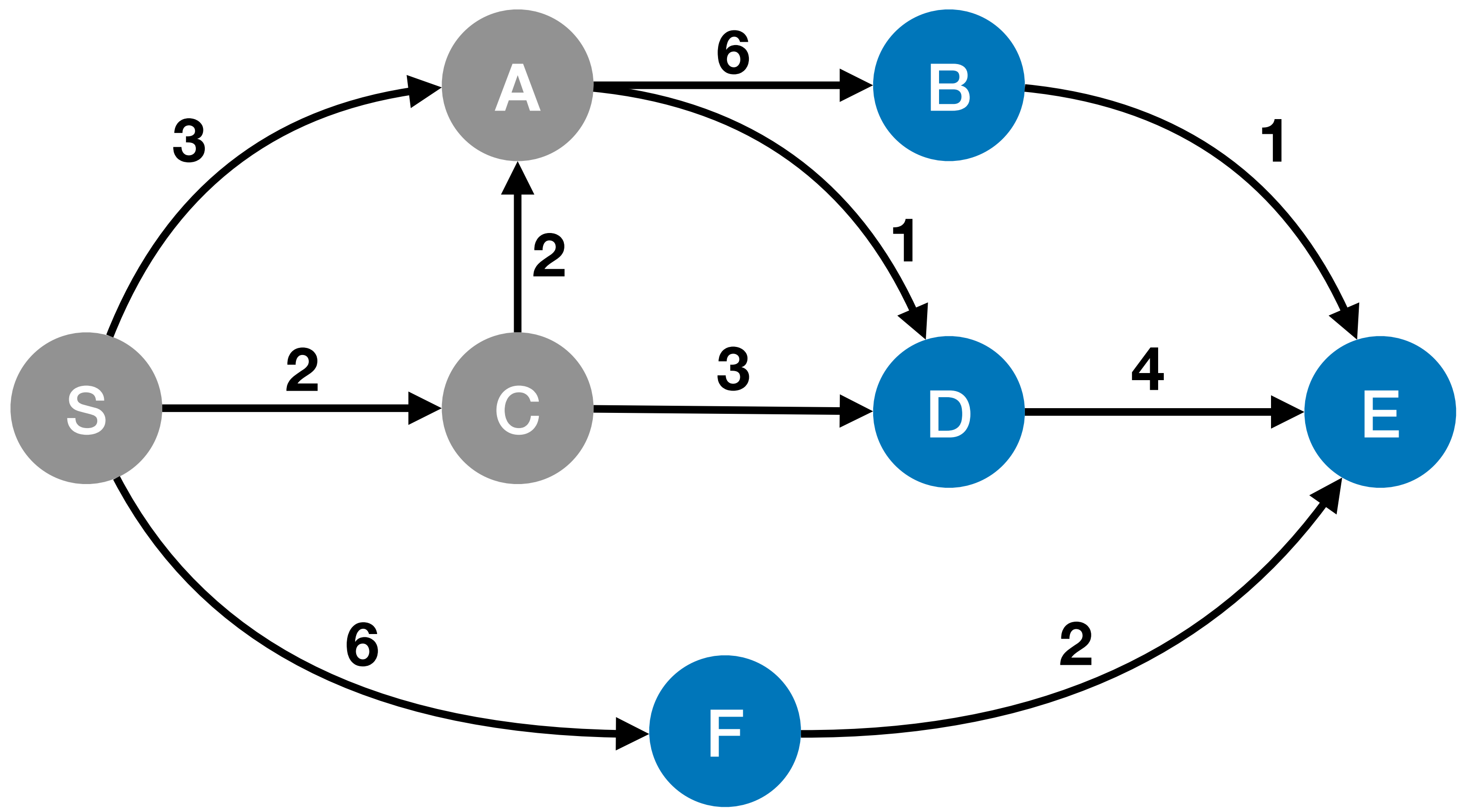



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, ] Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

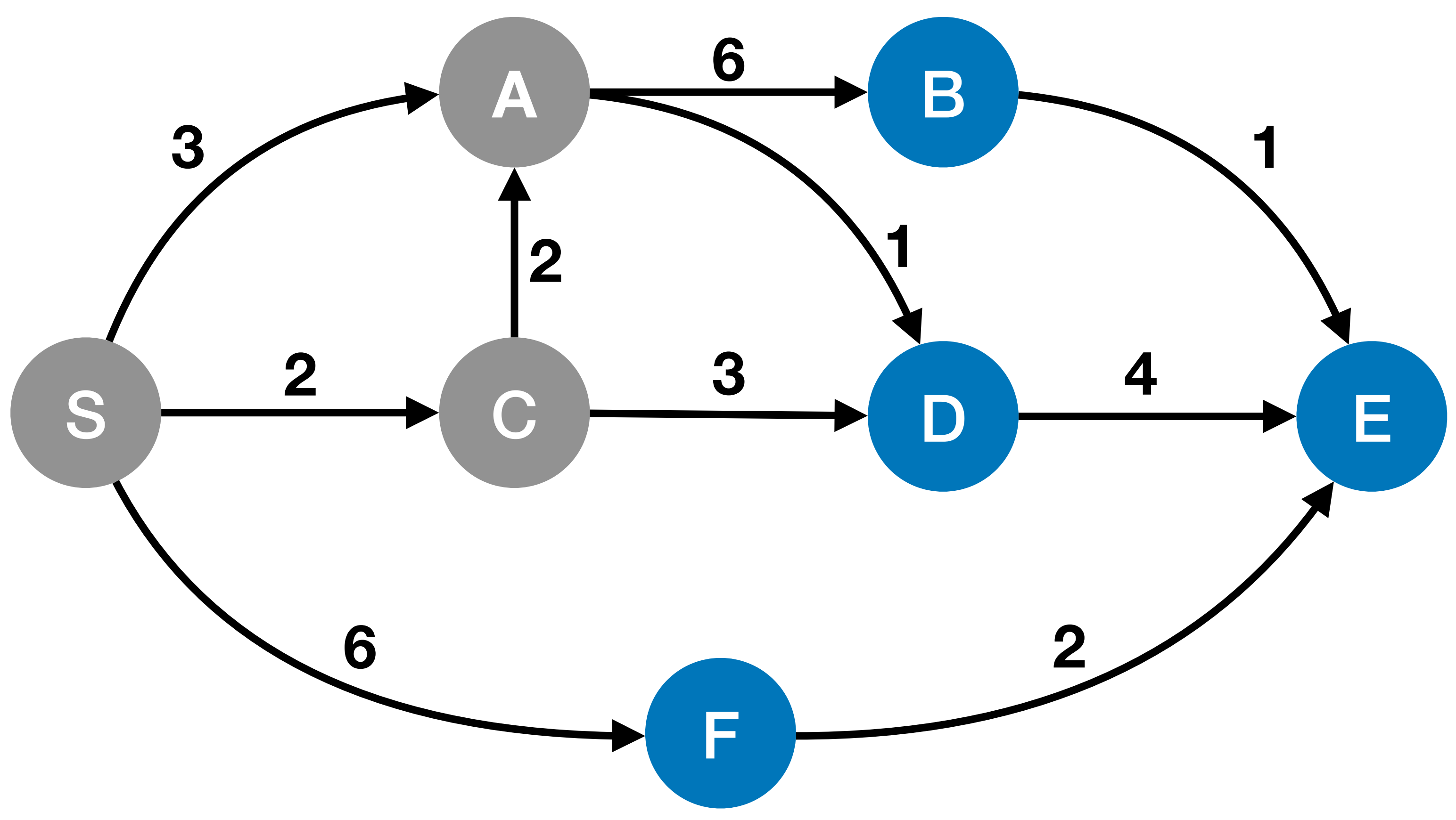



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]    Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

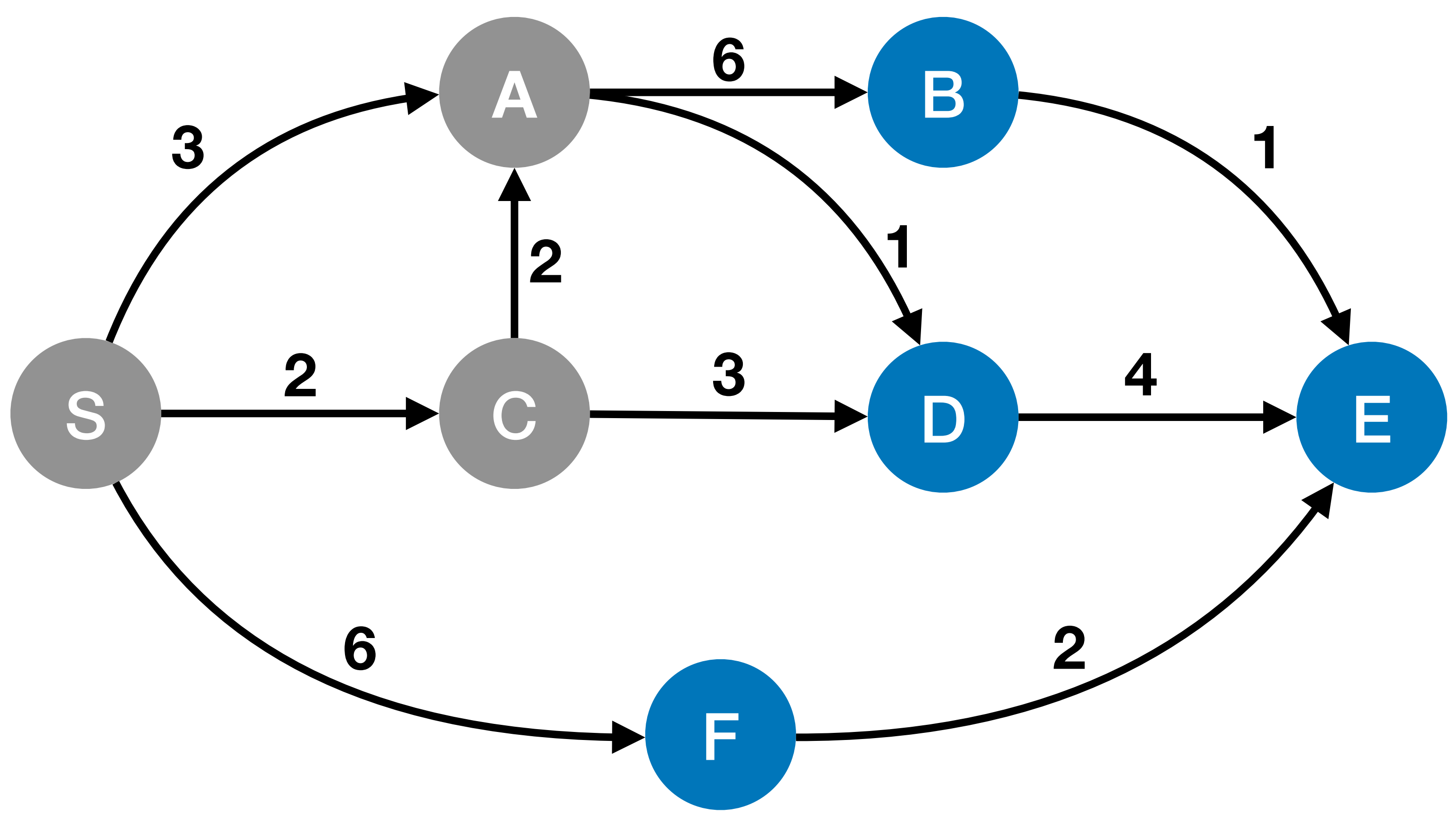
- Add the current node to the list of *settled* nodes




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]     Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

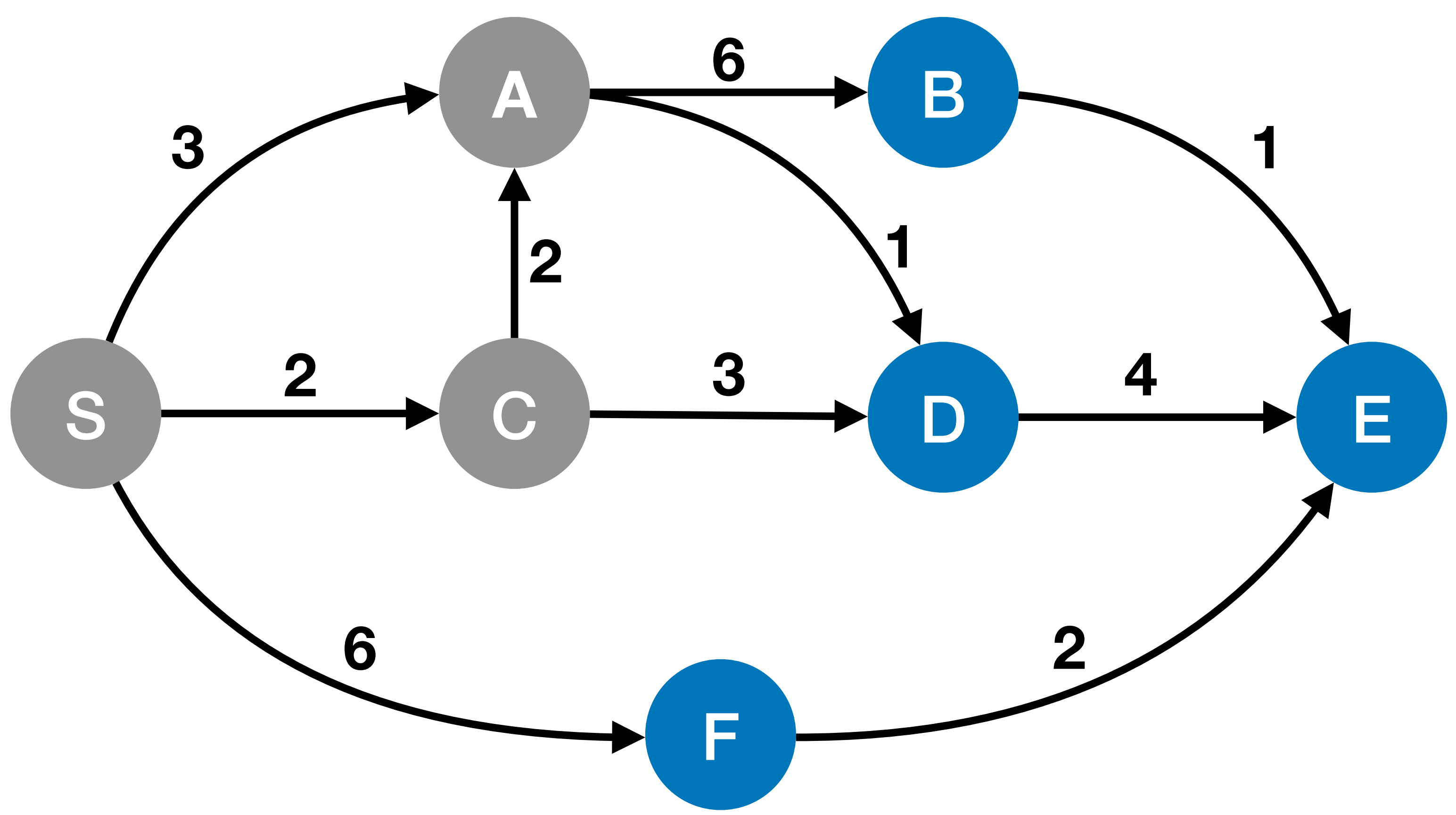



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]

Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

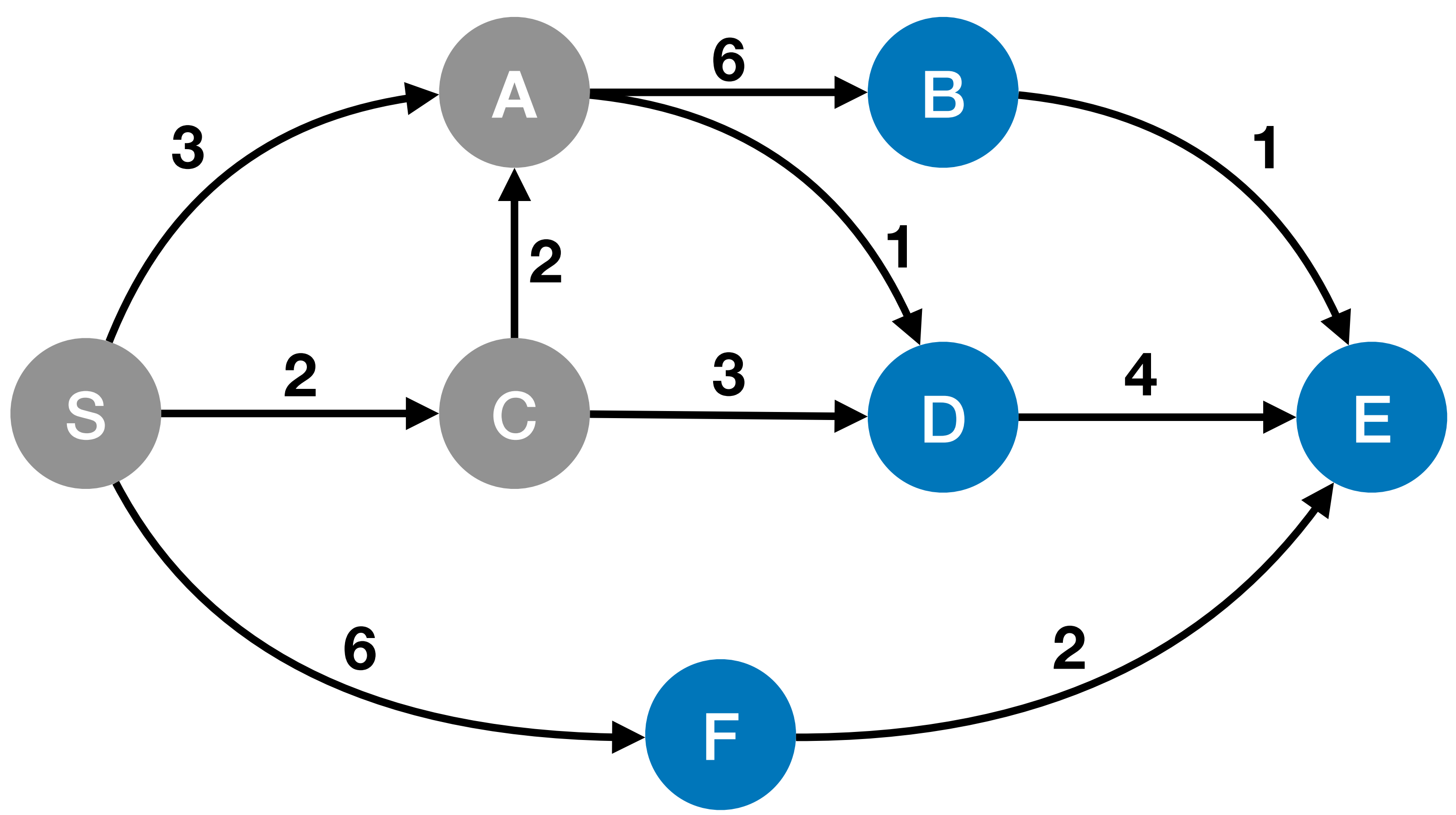



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]    Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node



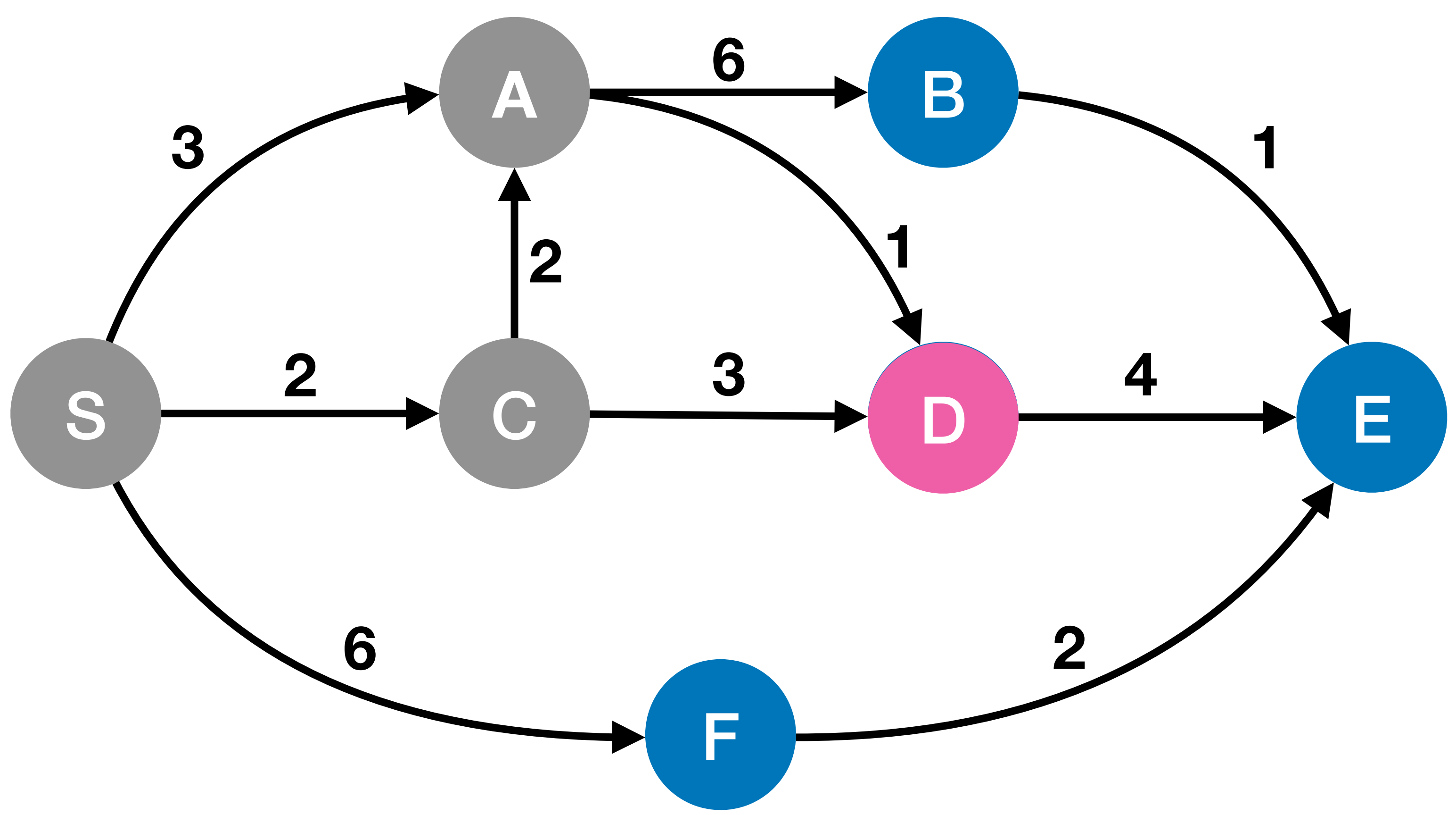

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]

Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node
- This time, it is node (D).

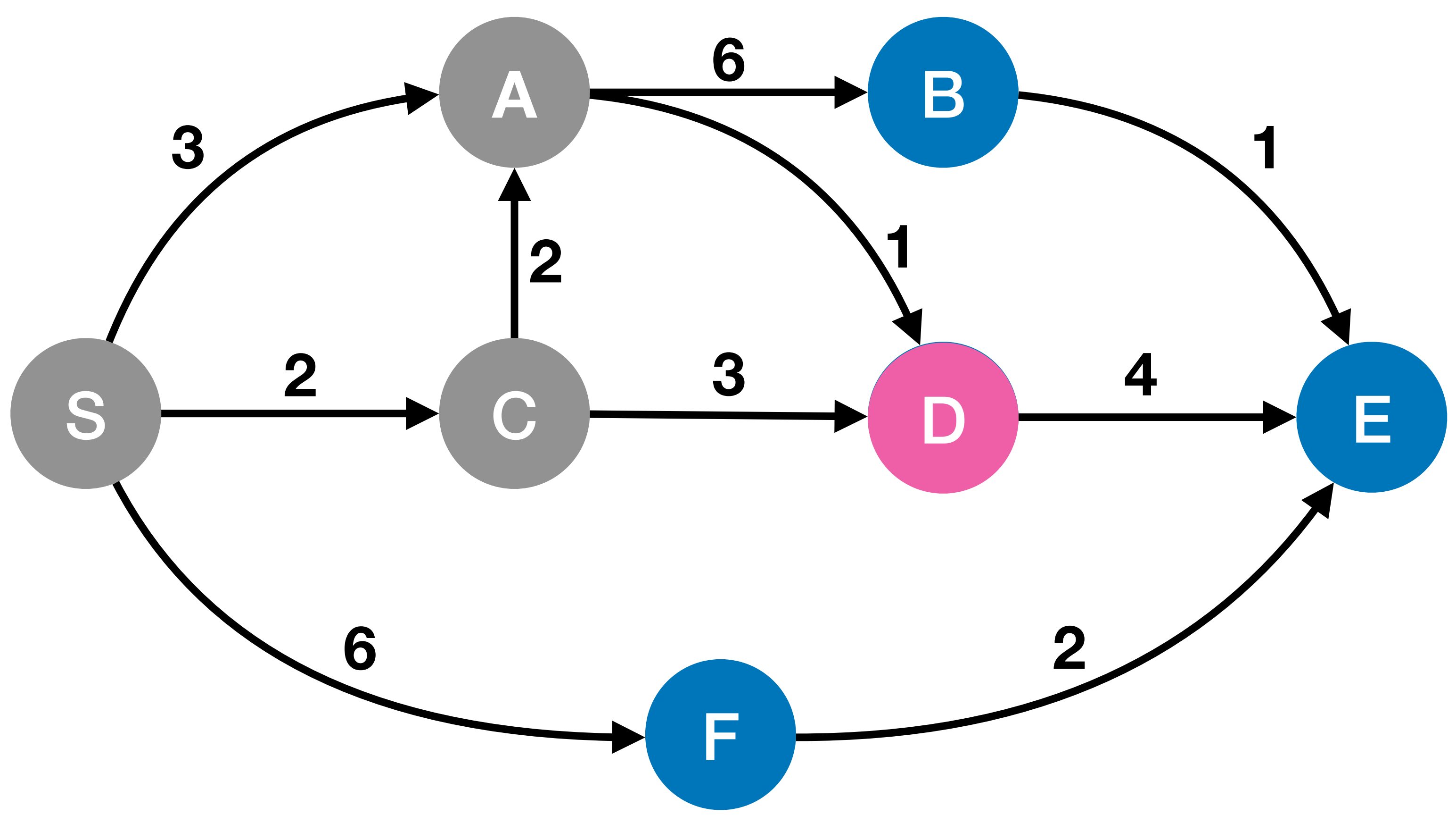



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]    Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors



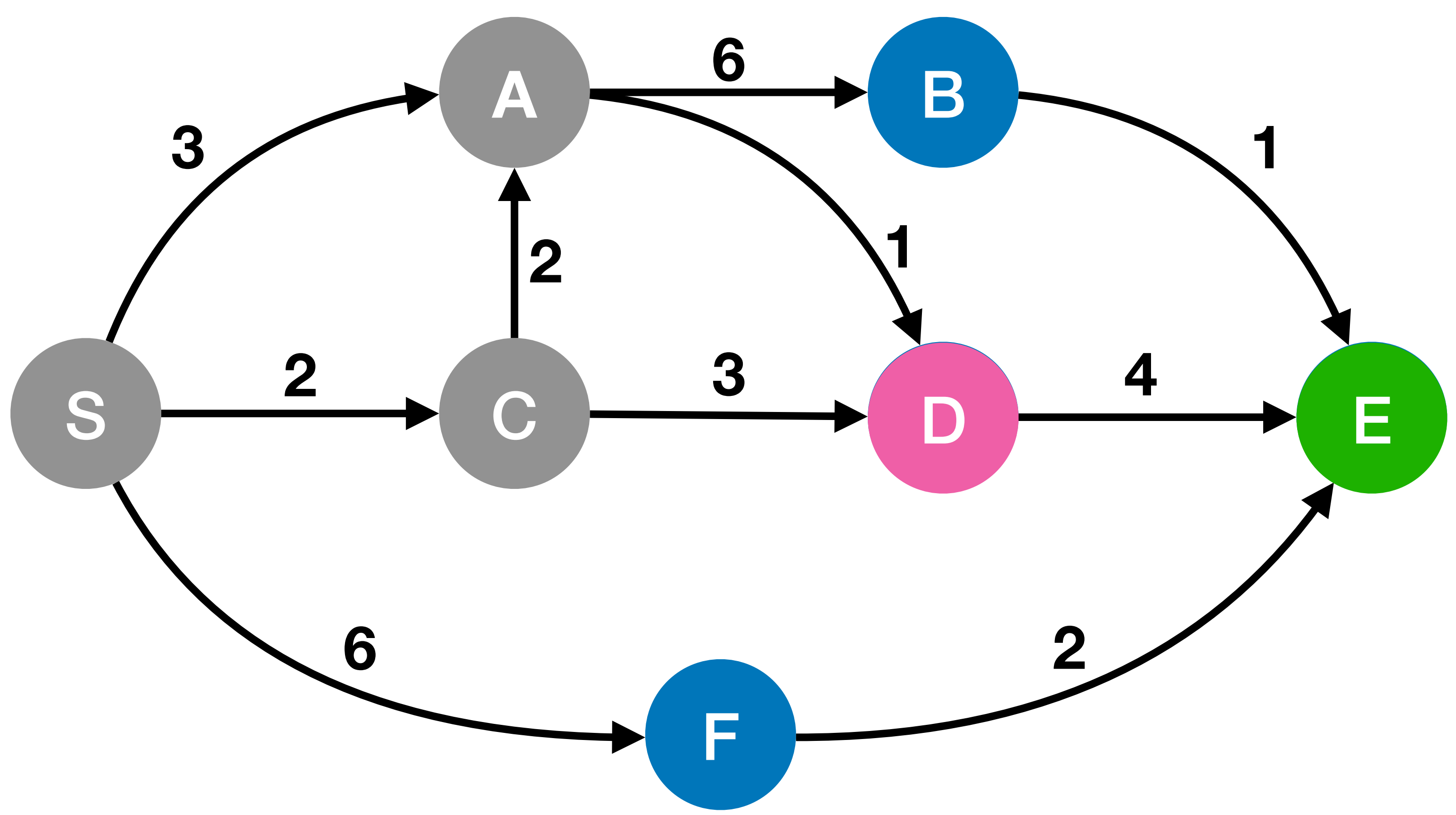

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]

Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → D; unexplored neighbors → {E}

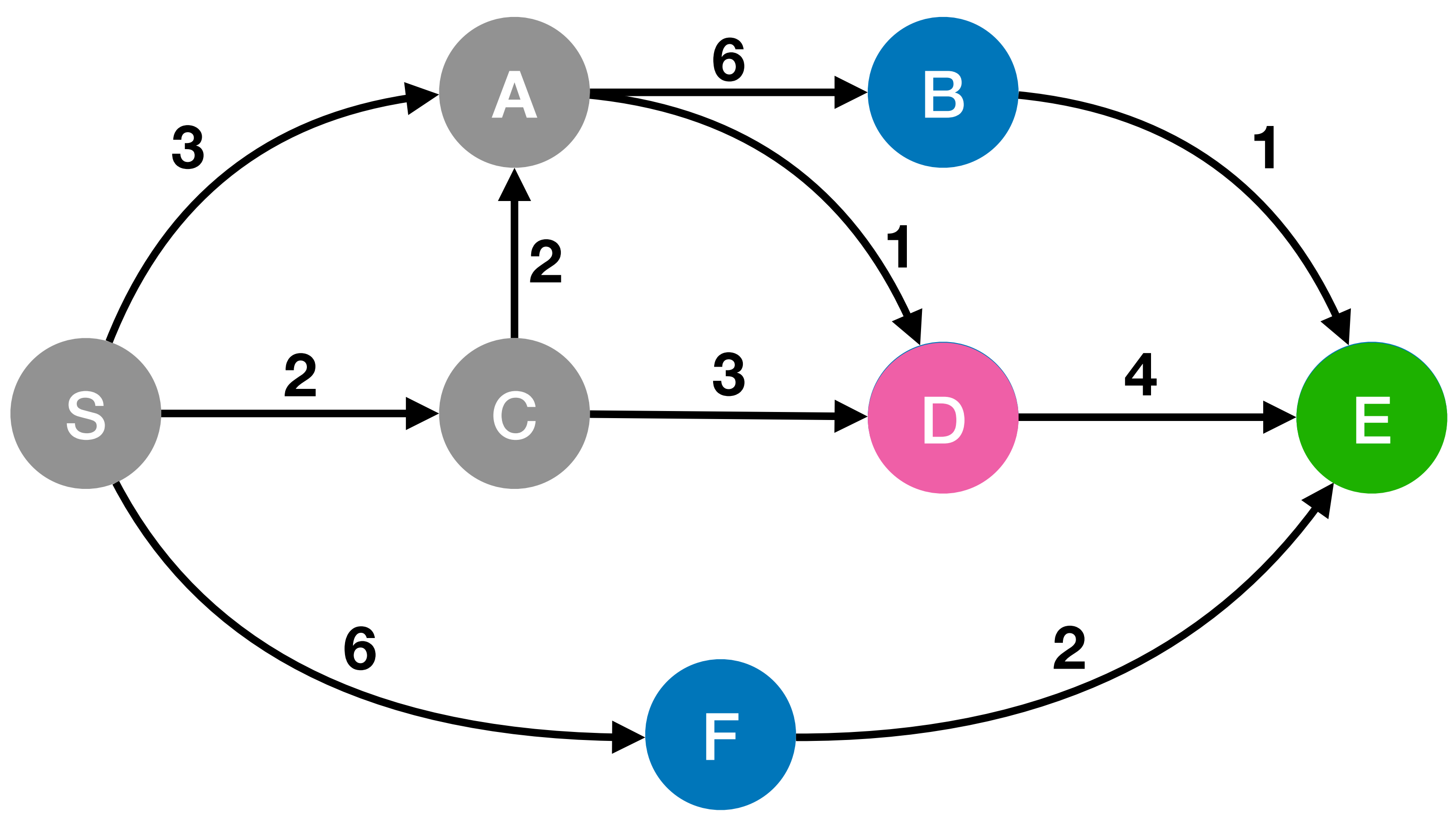



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]

Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

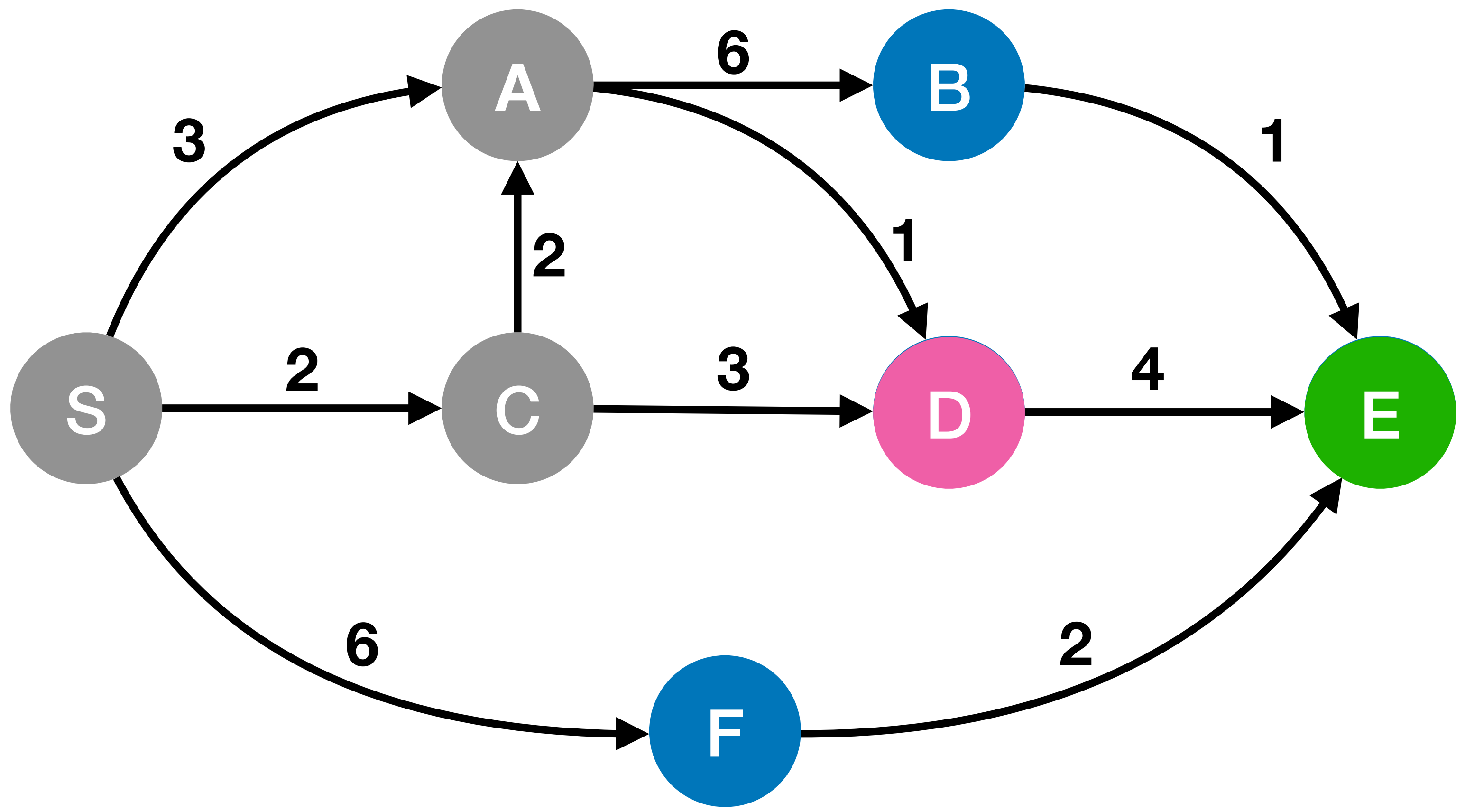



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]    Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node.



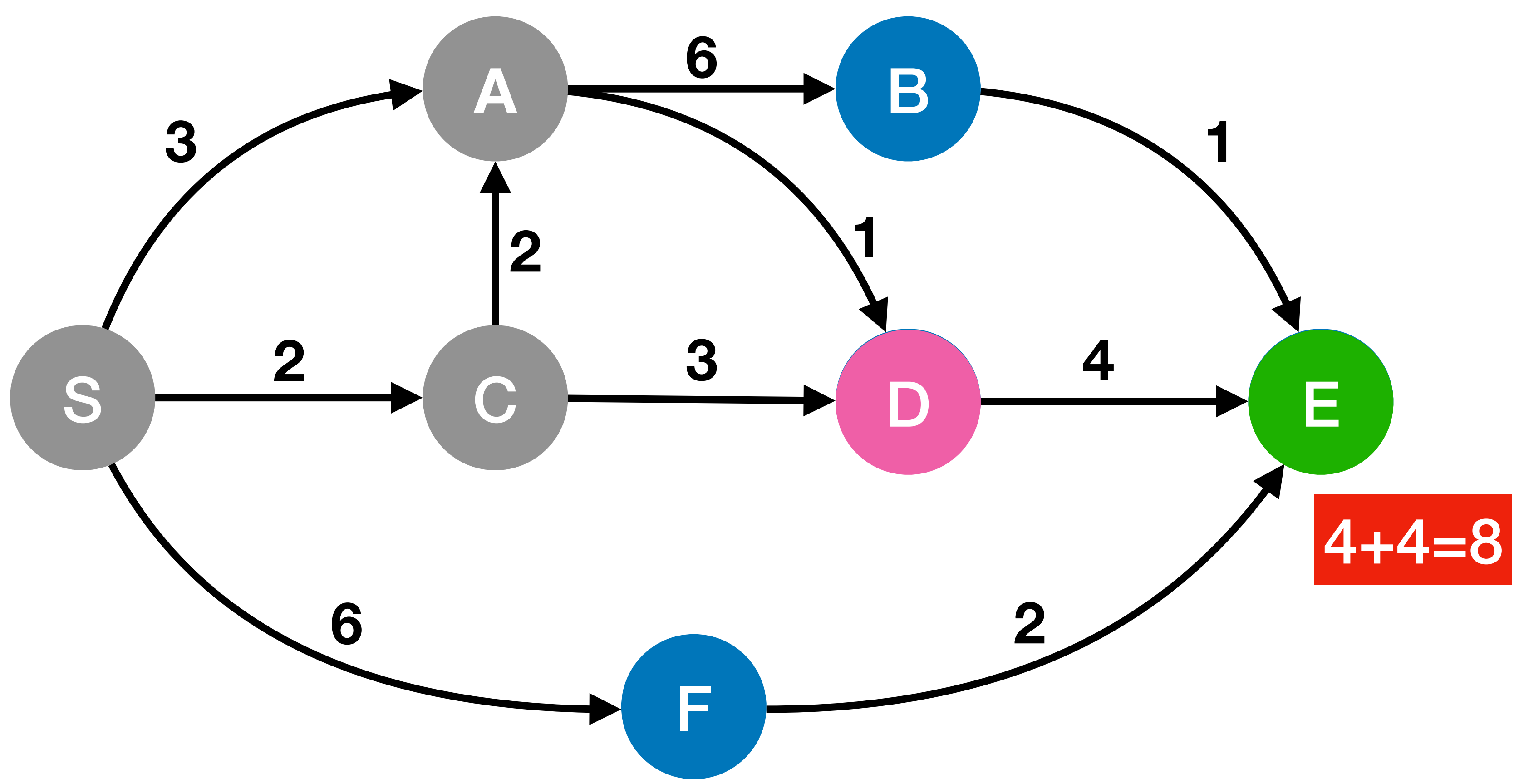

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]

Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node.



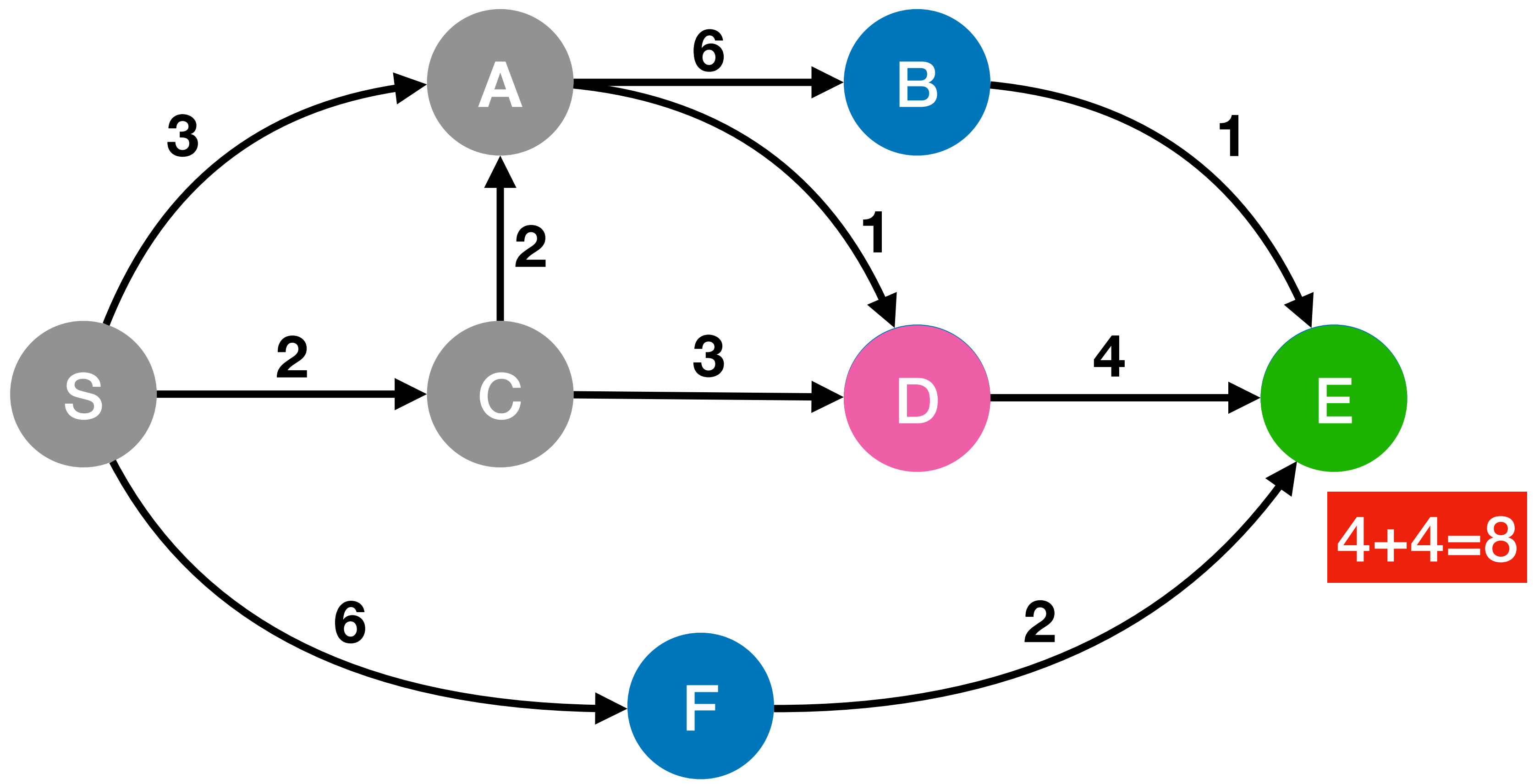

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]

Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.



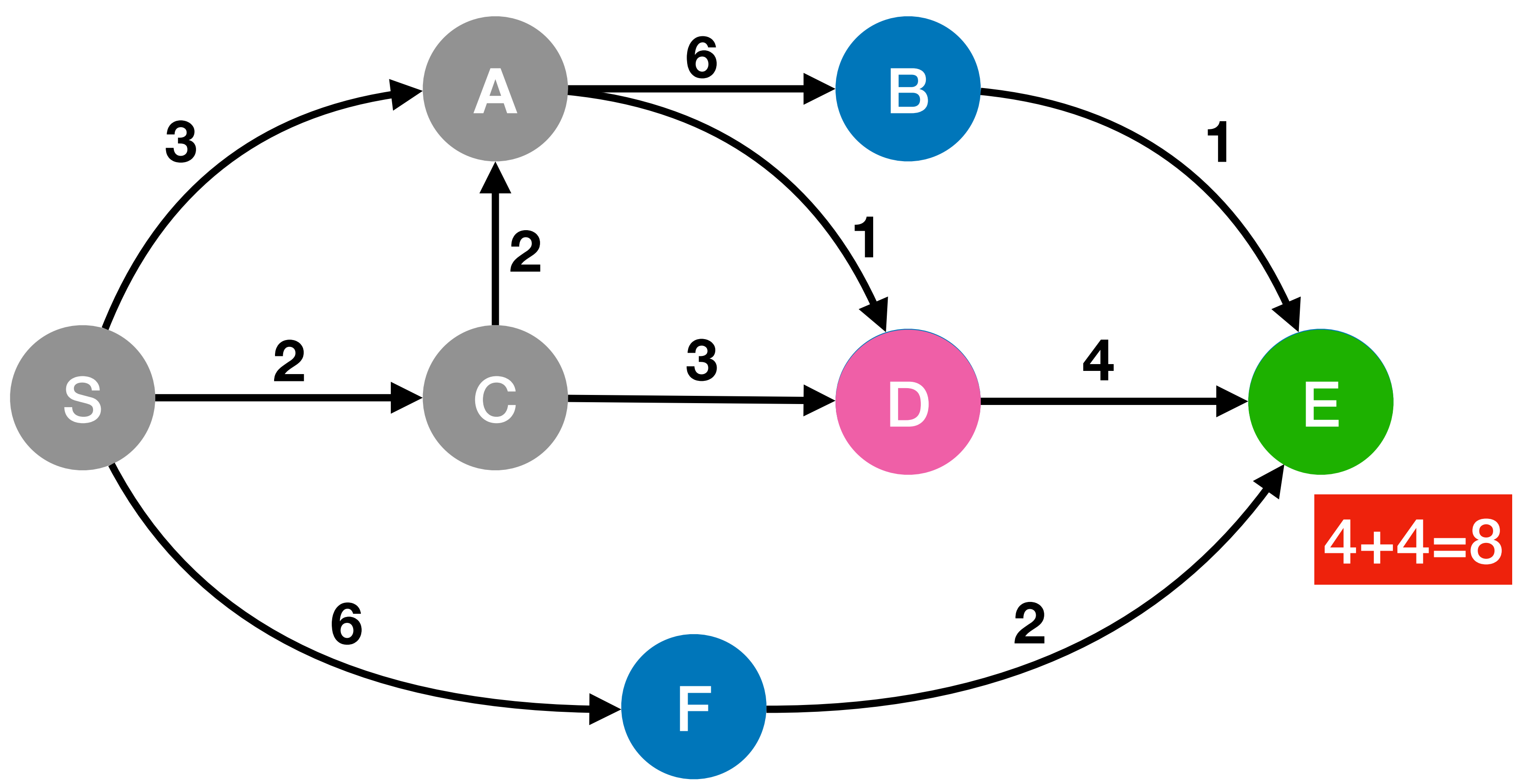

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | ∞ | |

Settled = [ S, C, A ]

Unexplored = [F, D, B, E ]

# Dijkstra’s algorithm

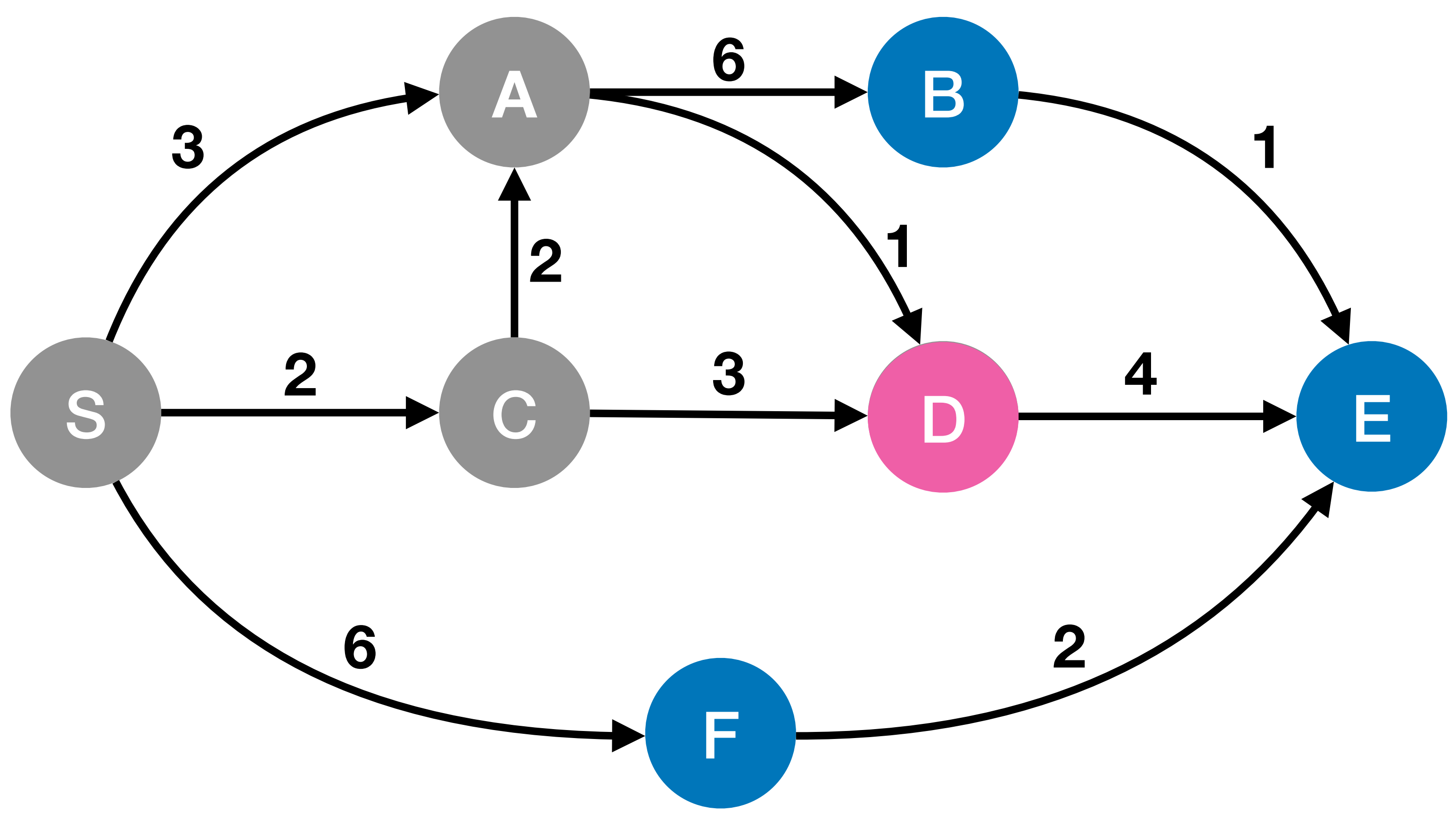
- For the current node, calculate the distance of each unsettled neighbor from the source node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A ]    Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

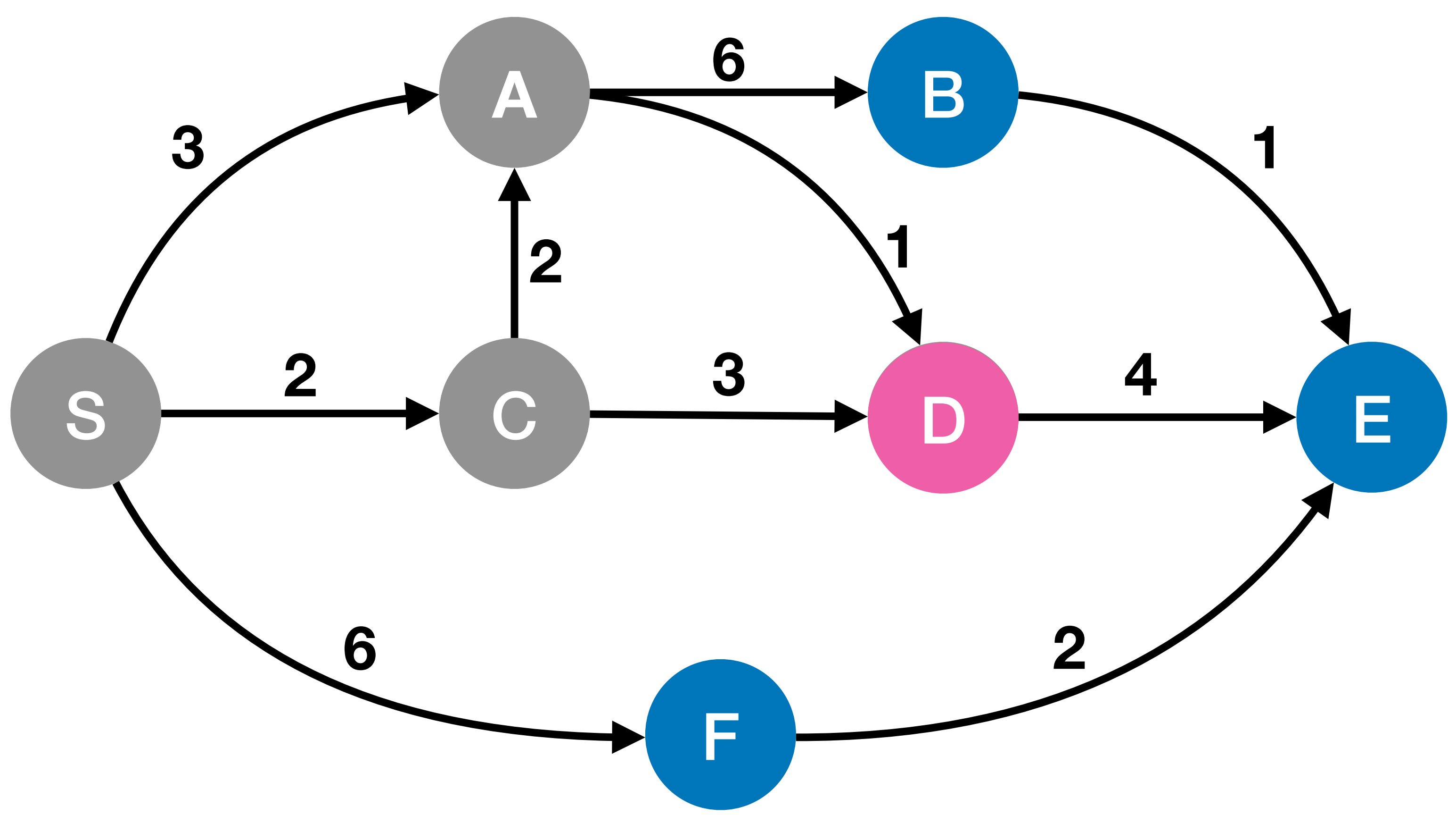



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, ]  Unexplored = [F, D, B, E ]

# Dijkstra’s algorithm

- Add the current node to the list of *settled* nodes

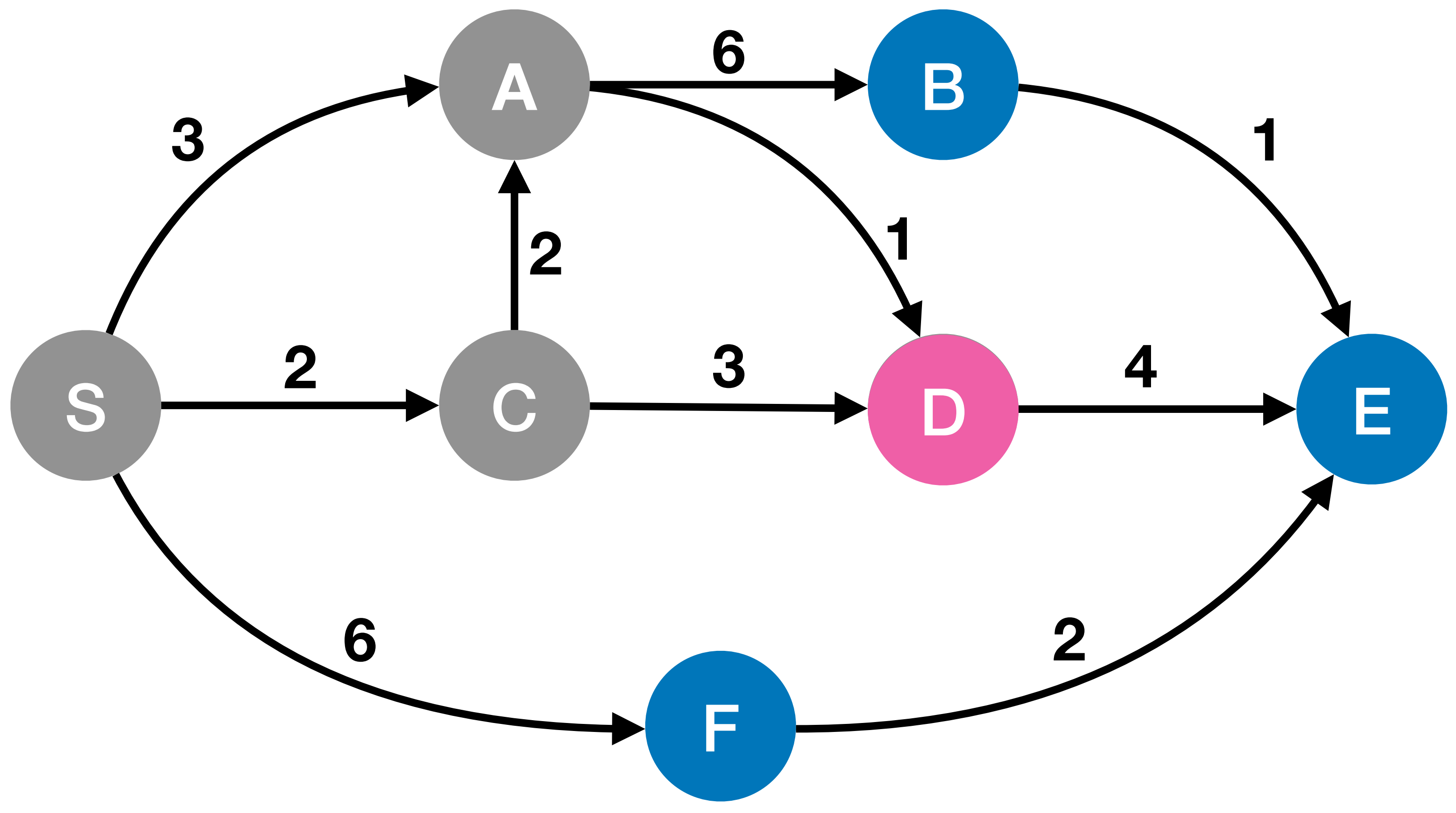



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, ]    Unexplored = [F, D, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

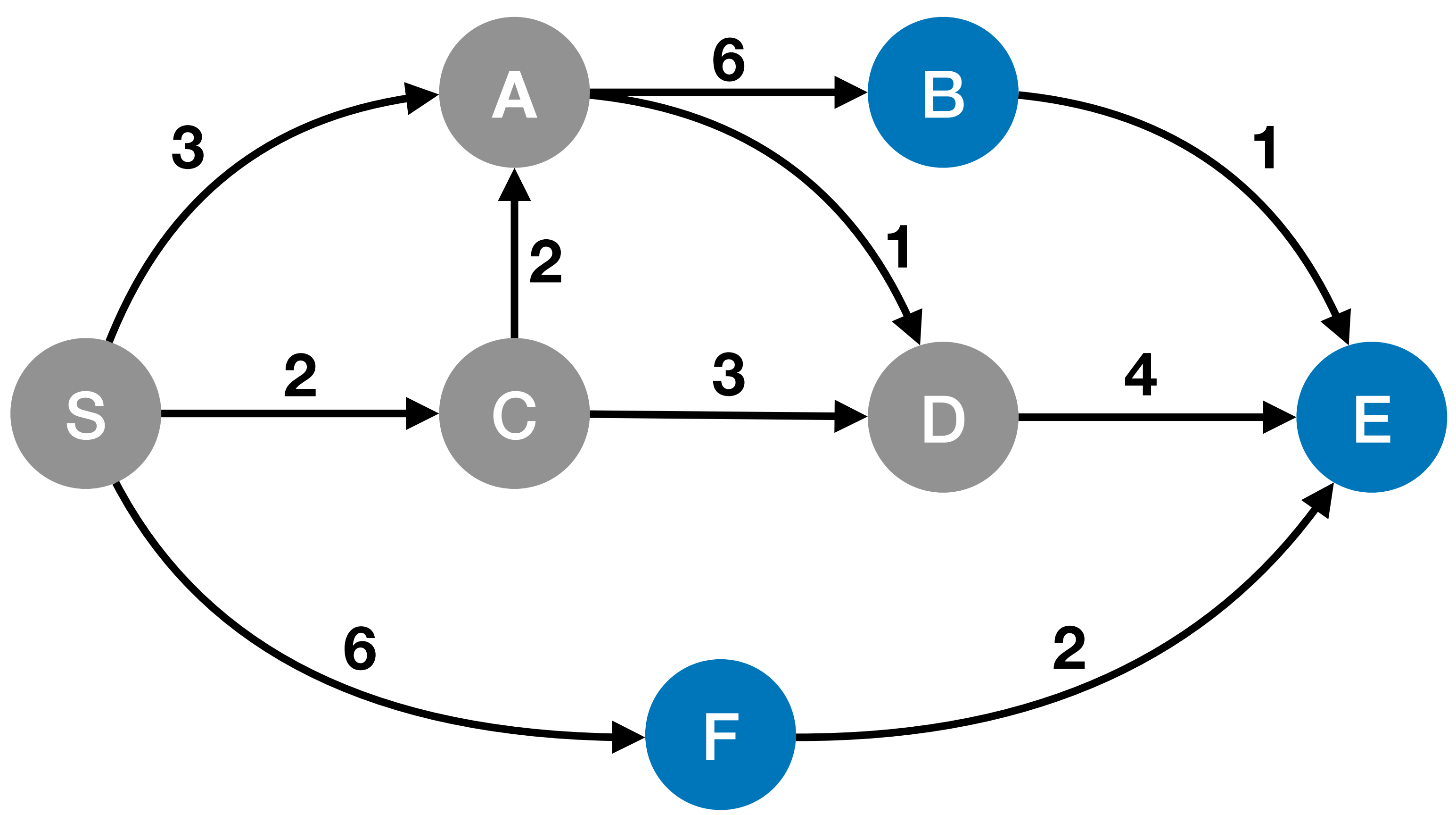



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, ]     Unexplored = [F, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

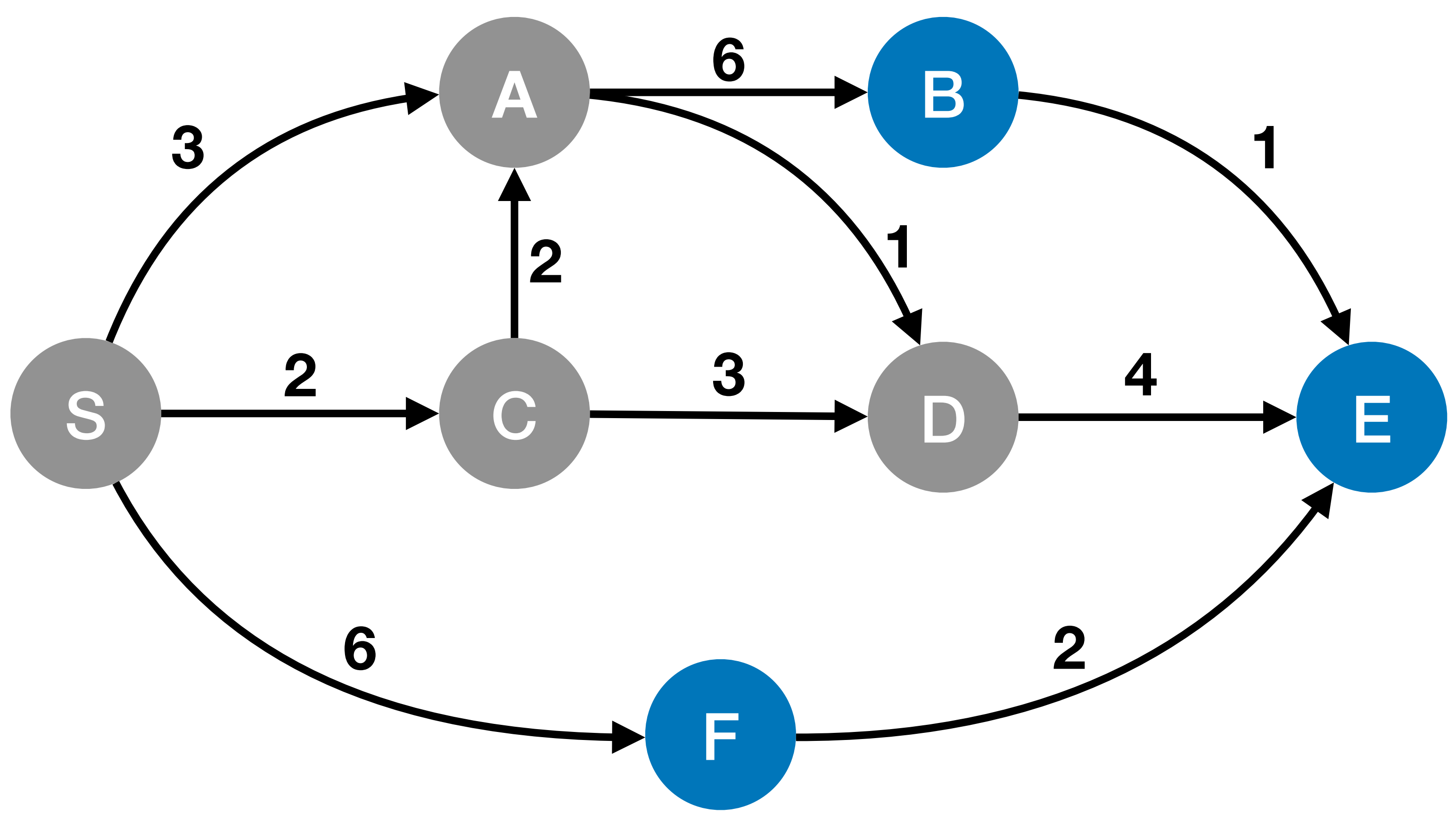



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]    Unexplored = [F, B, E ]

# Dijkstra’s algorithm

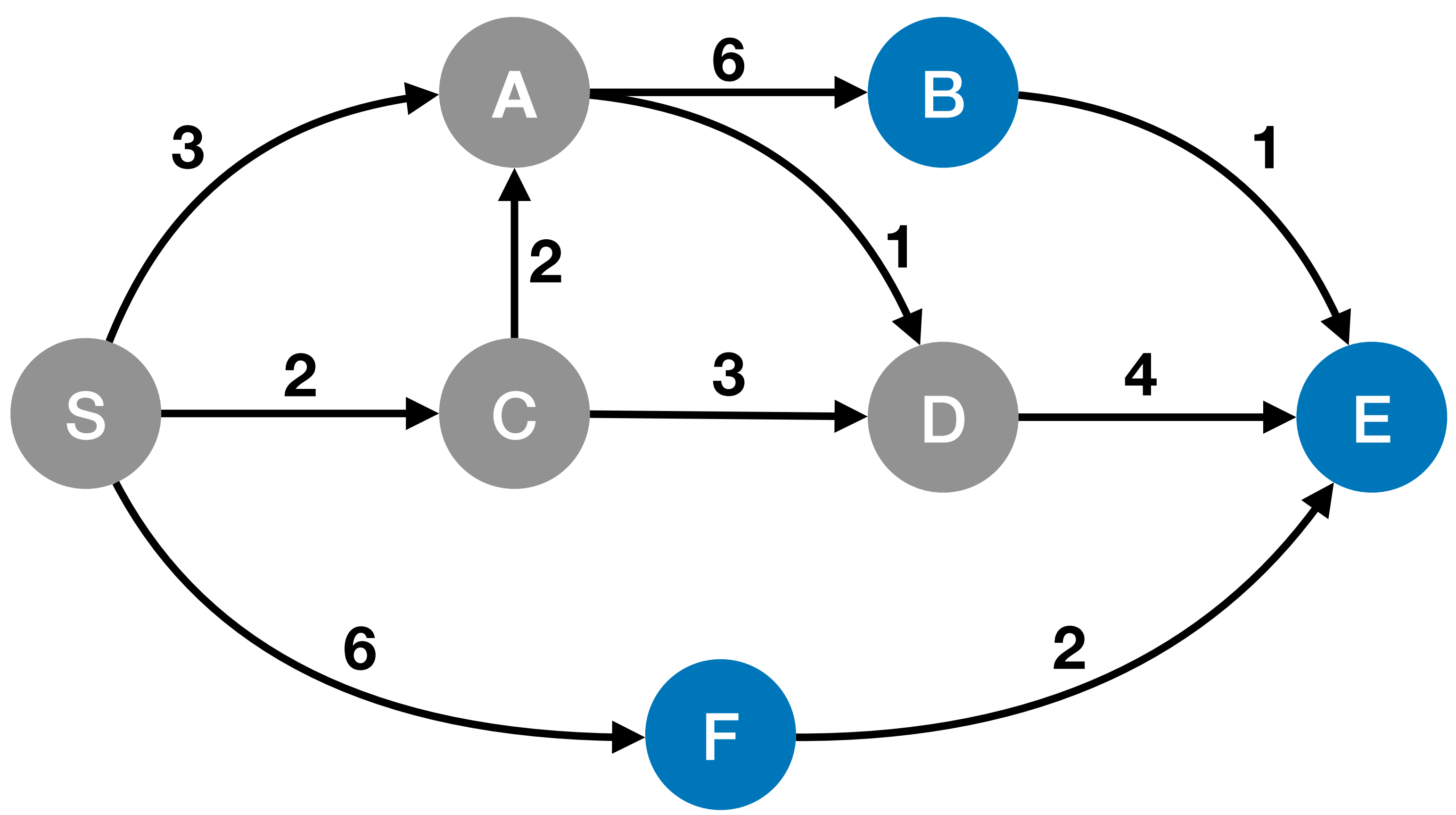
- Add the current node to the list of *settled* nodes




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| **D** | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]    Unexplored = [F, B, E ]

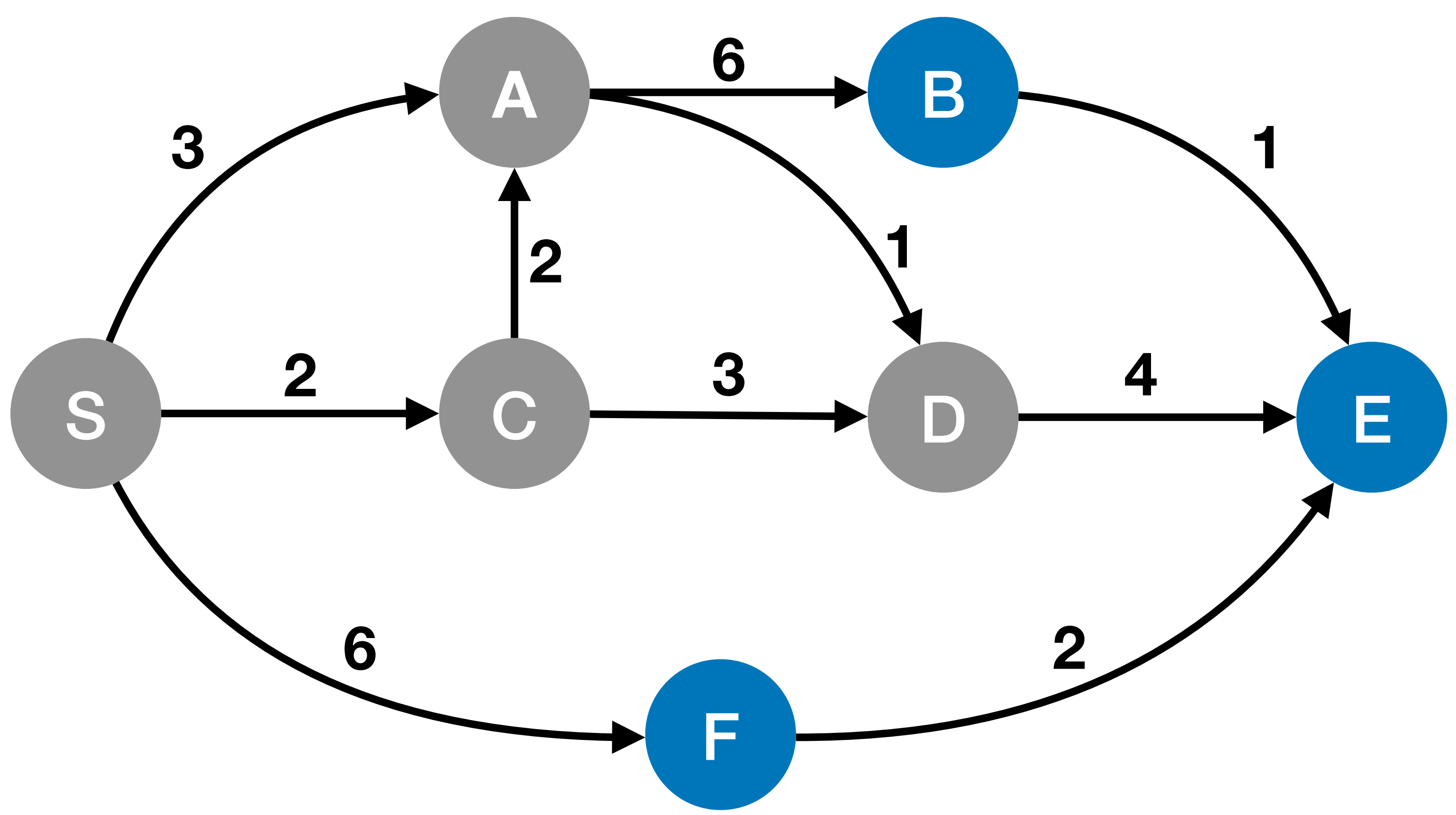
# Dijkstra's algorithm




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]    Unexplored = [F, B, E ]

# Dijkstra's algorithm

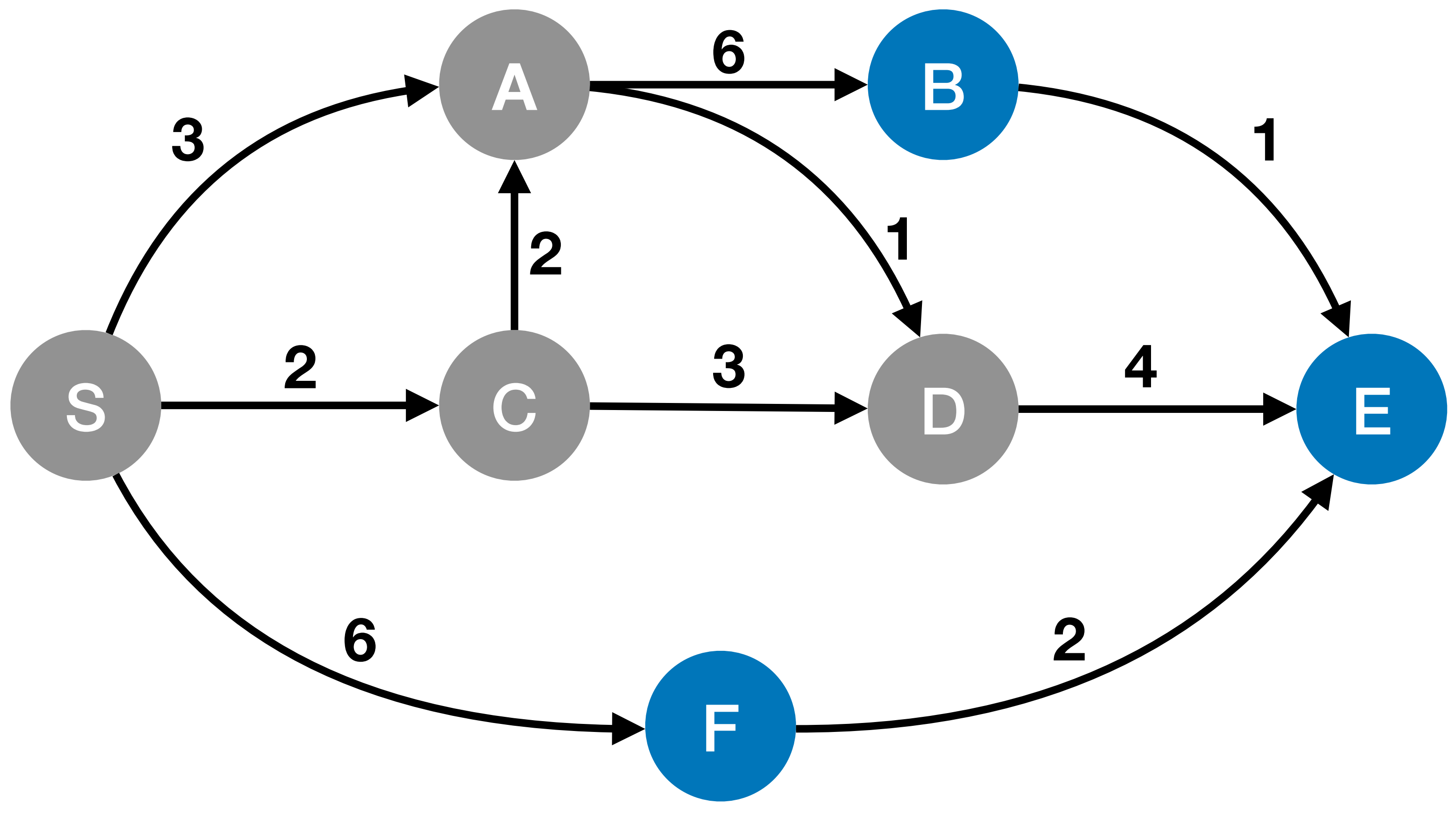



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]    Unexplored = [F, B, E ]

# Dijkstra’s algorithm

- Pick the unsettled node with the smallest known distance from the source node




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

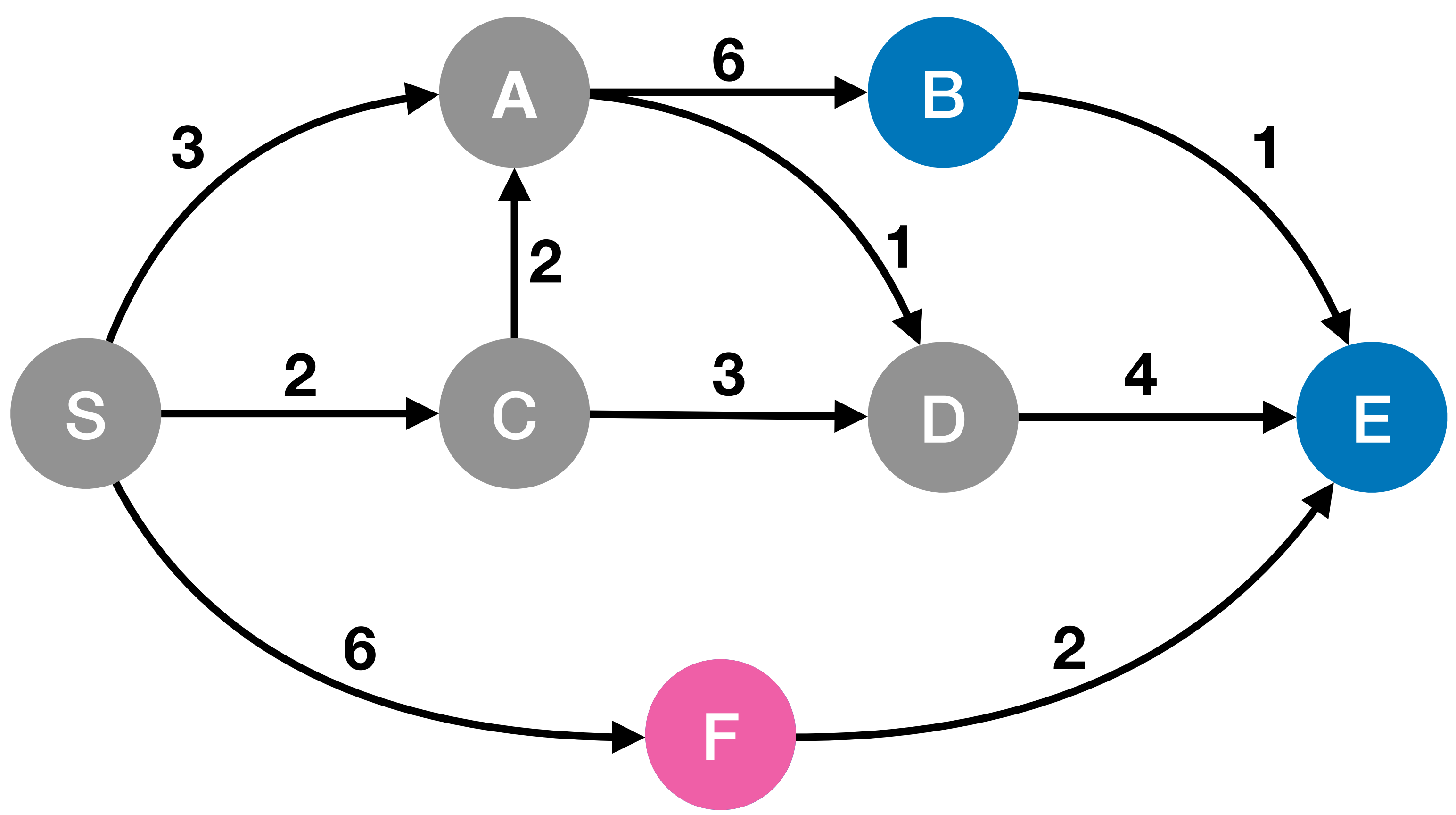
Settled = [ S, C, A, D ]    Unexplored = [F, B, E ]



**Iterative step - Begin Iter 5**

# Dijkstra's algorithm

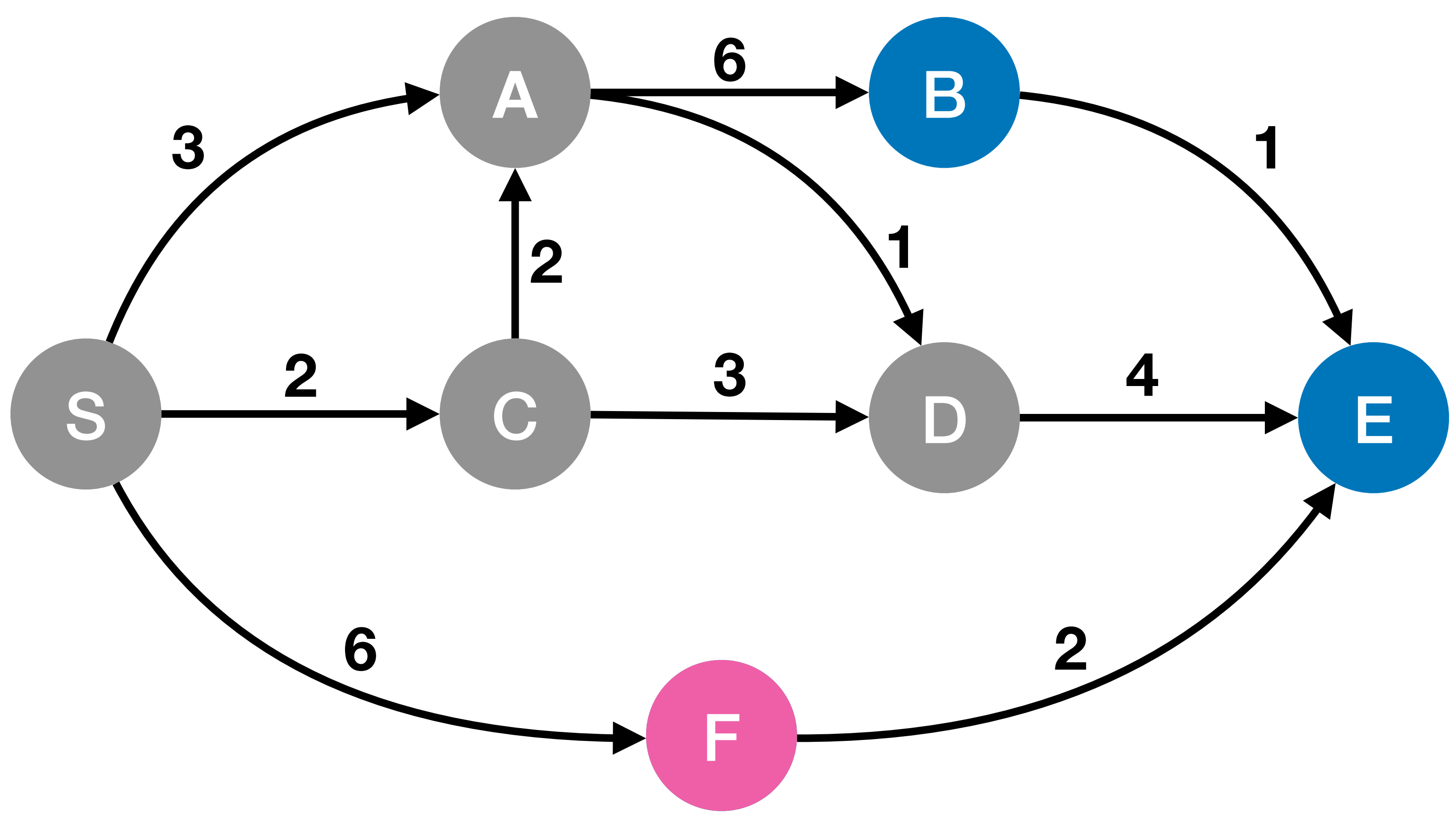
- Pick the unsettled node with the smallest known distance from the source node
- This time, it is node (F).




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ] Unexplored = [F, B, E ]

# Dijkstra's algorithm




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

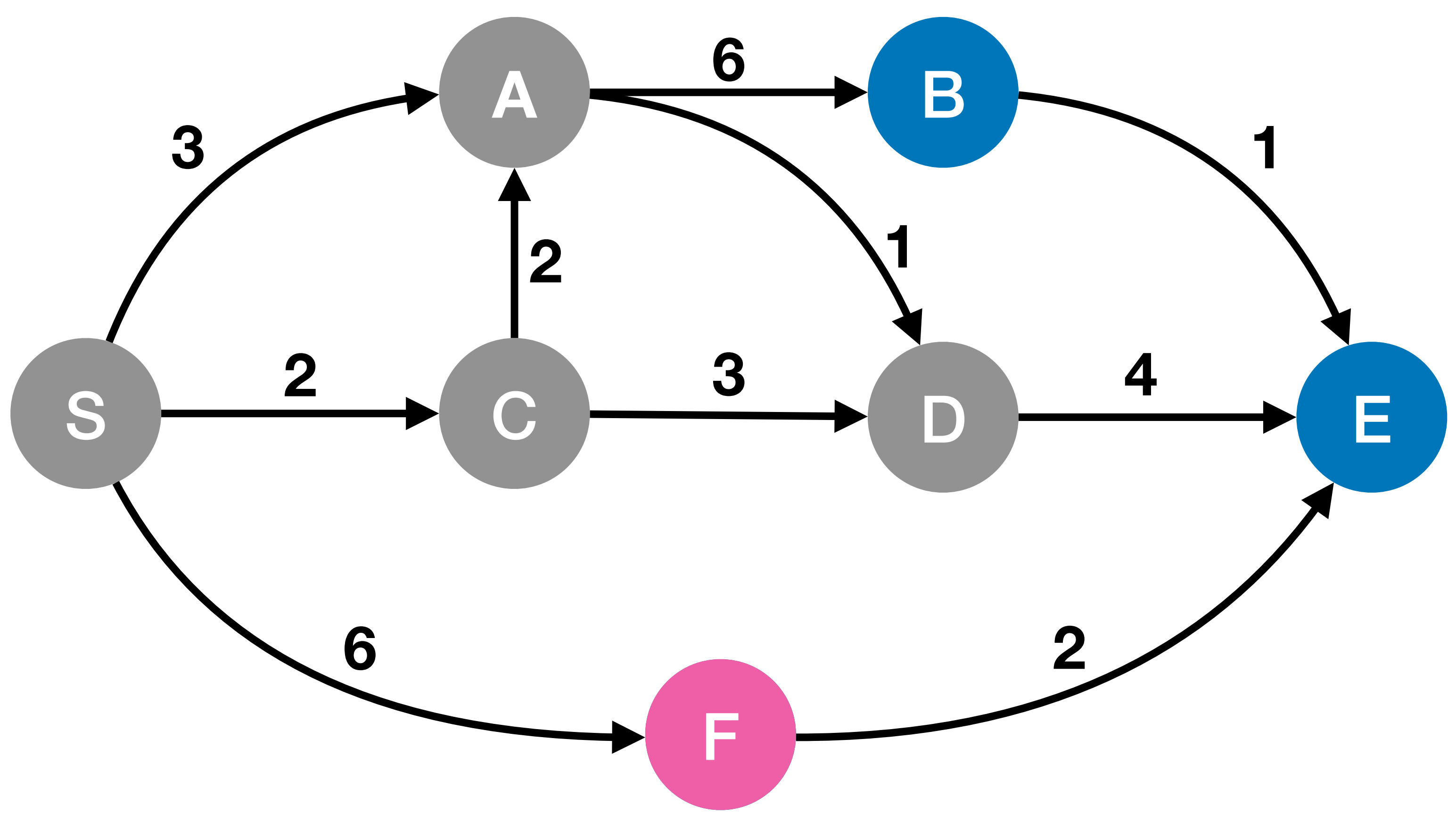
Settled = [ S, C, A, D ]    Unexplored = [F, B, E ]



**Iterative step - Begin Iter 5**

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors

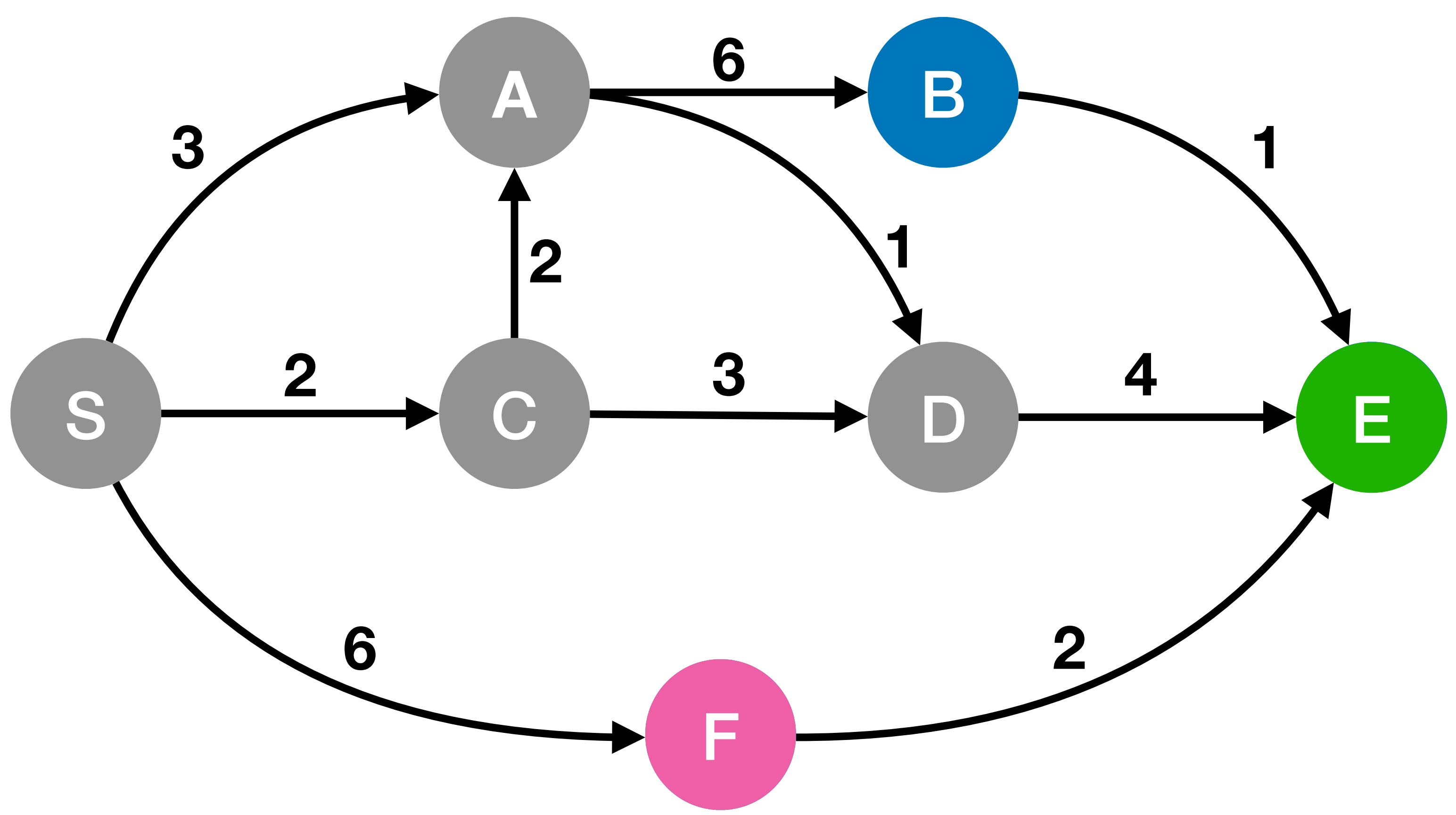



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]　　Unexplored = [F, B, E ]

# Dijkstra's algorithm

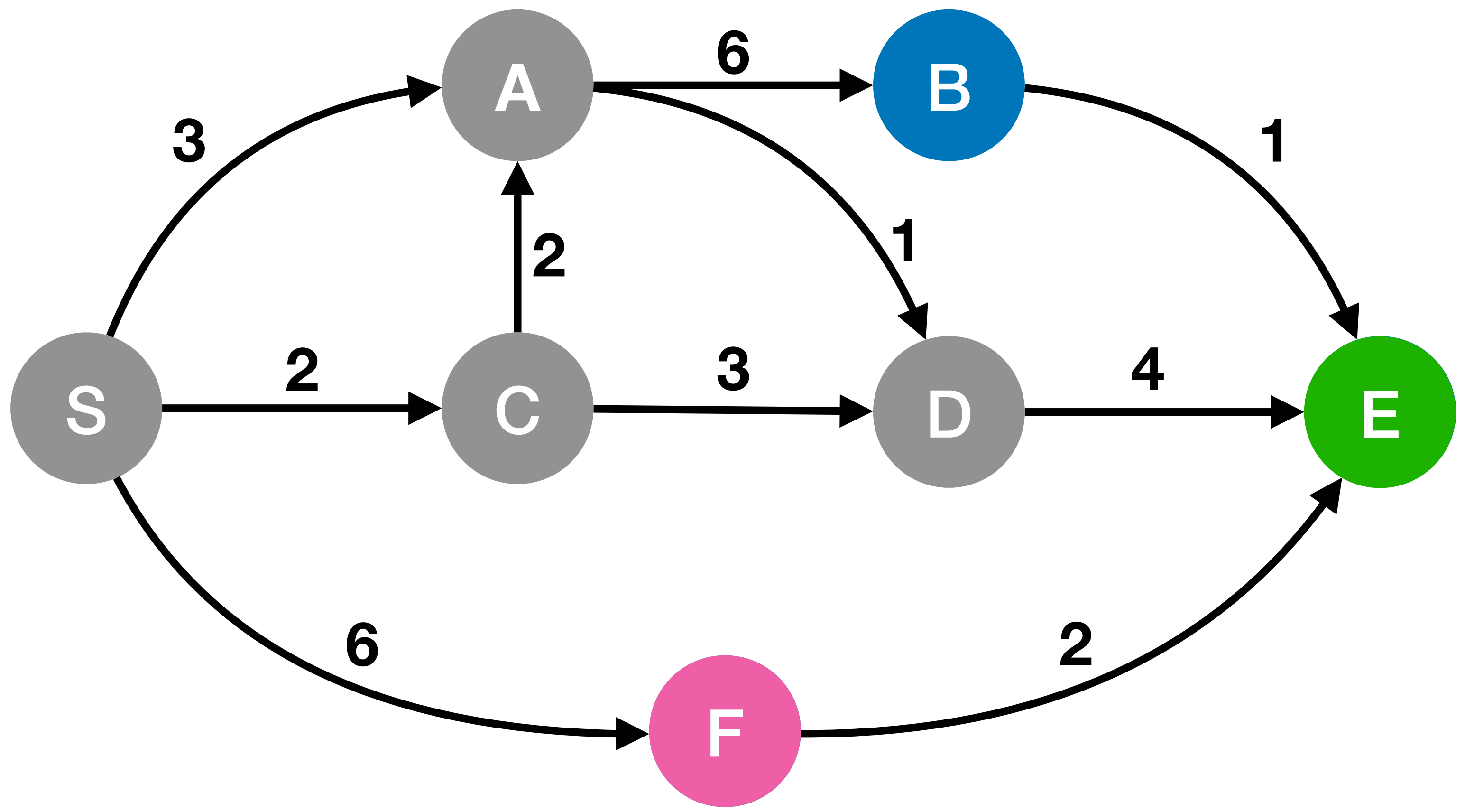
- For the current node, examine its unexplored neighbors
- Current node → F; unexplored neighbors → {E}




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]   Unexplored = [F, B, E ]

# Dijkstra's algorithm

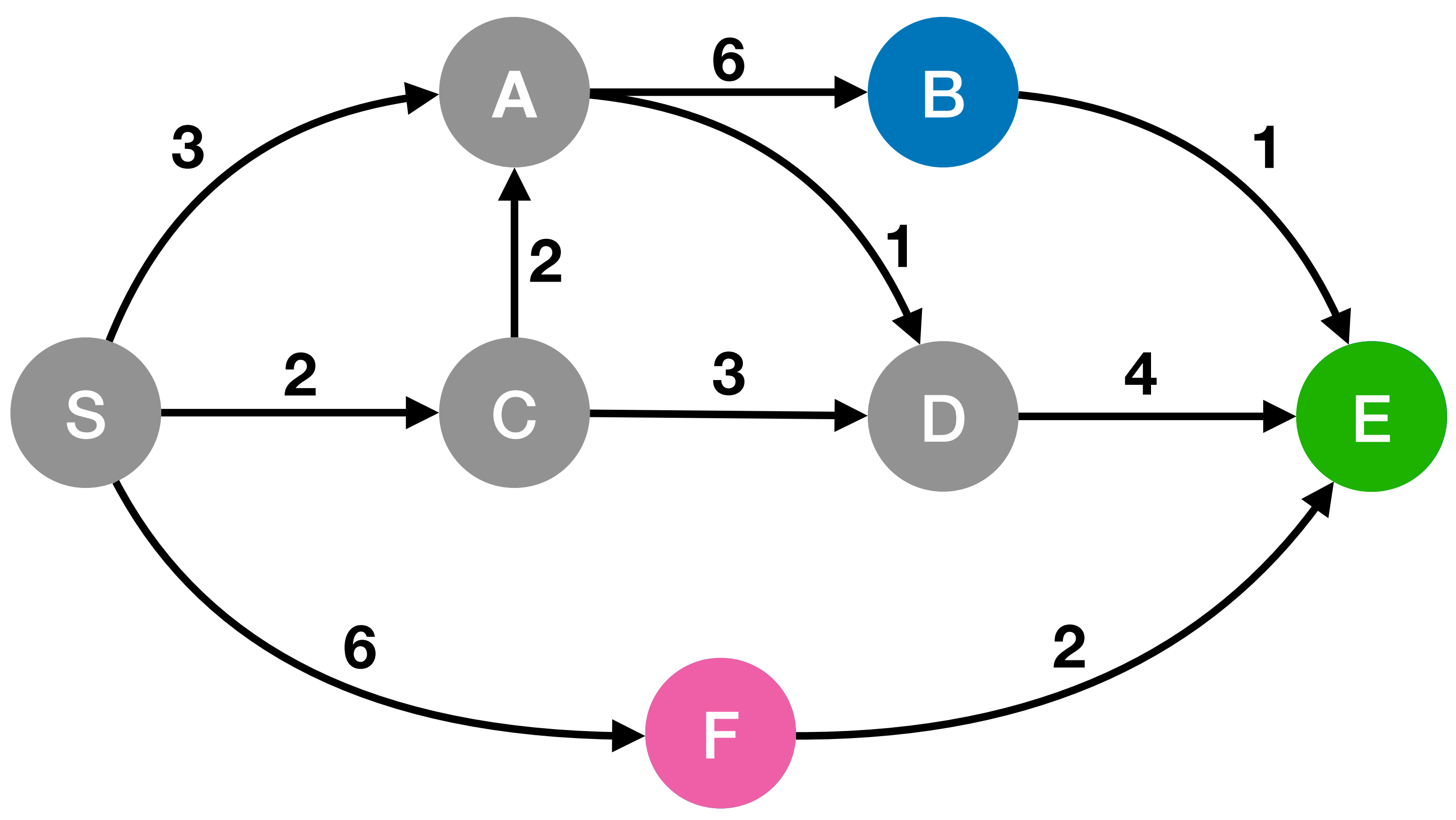



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]    Unexplored = [F, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node.

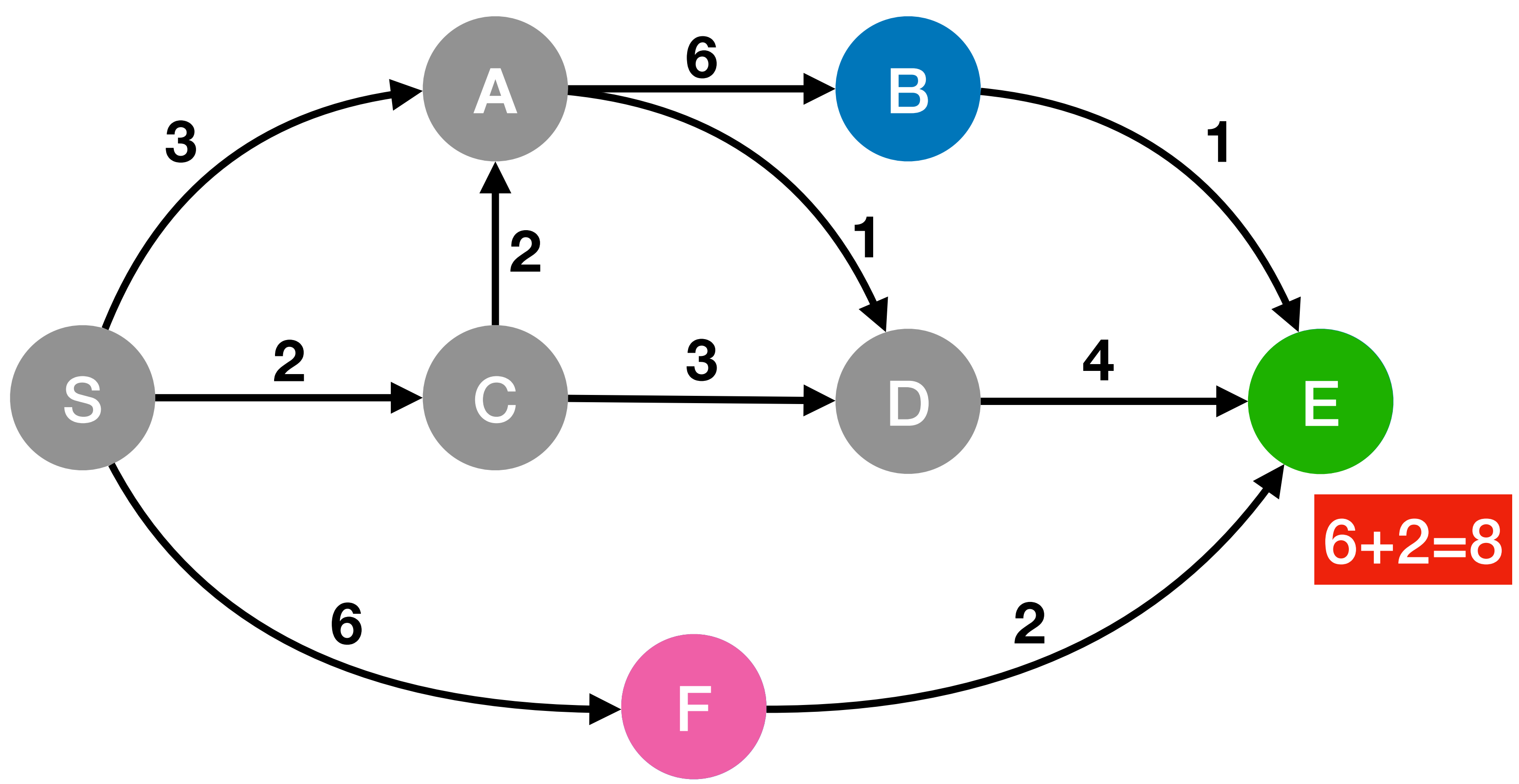



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]  Unexplored = [F, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node.

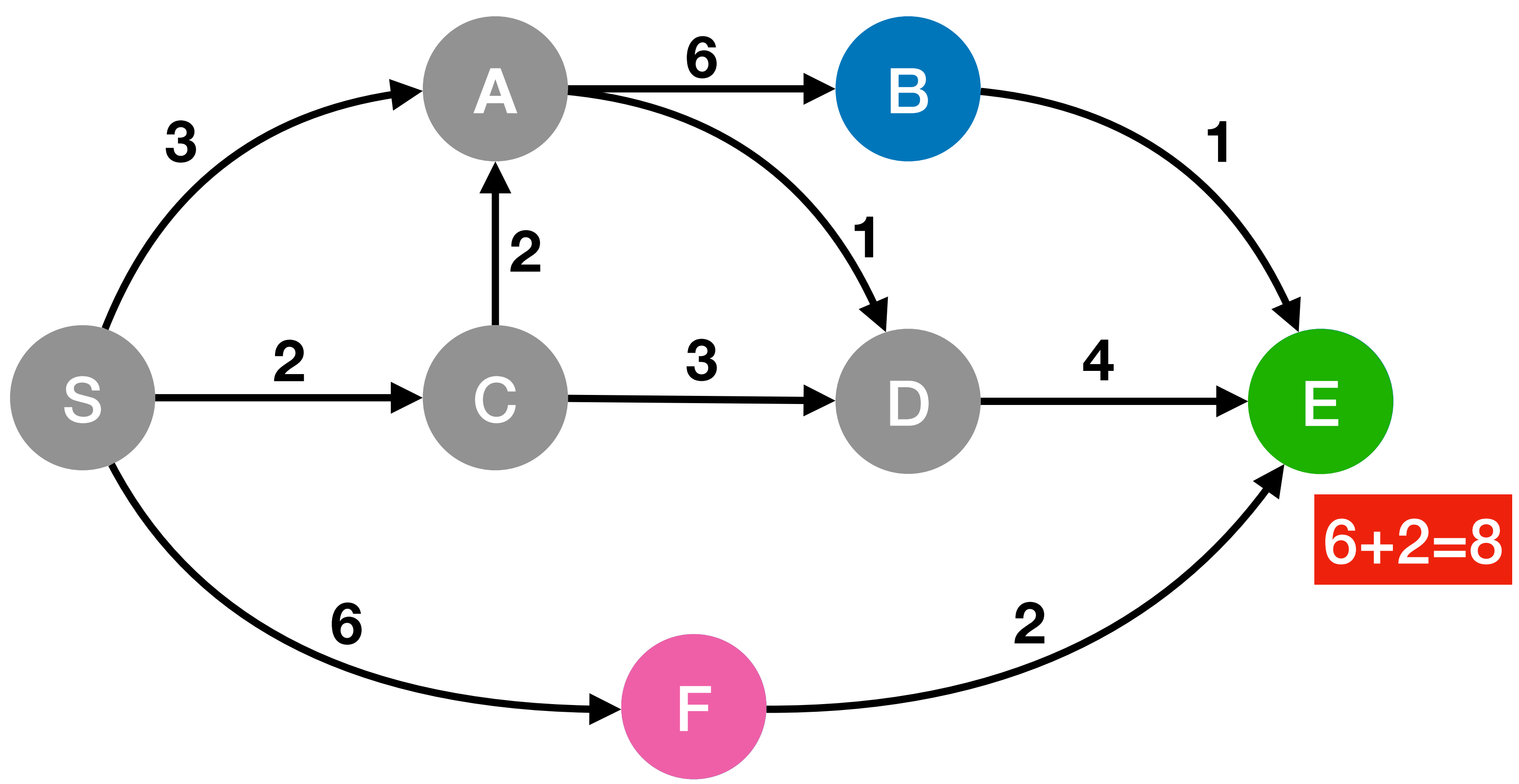



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]      Unexplored = [F, B, E ]

# Dijkstra's algorithm

- For the current node, calculate the distance of each unsettled neighbor from the source node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.

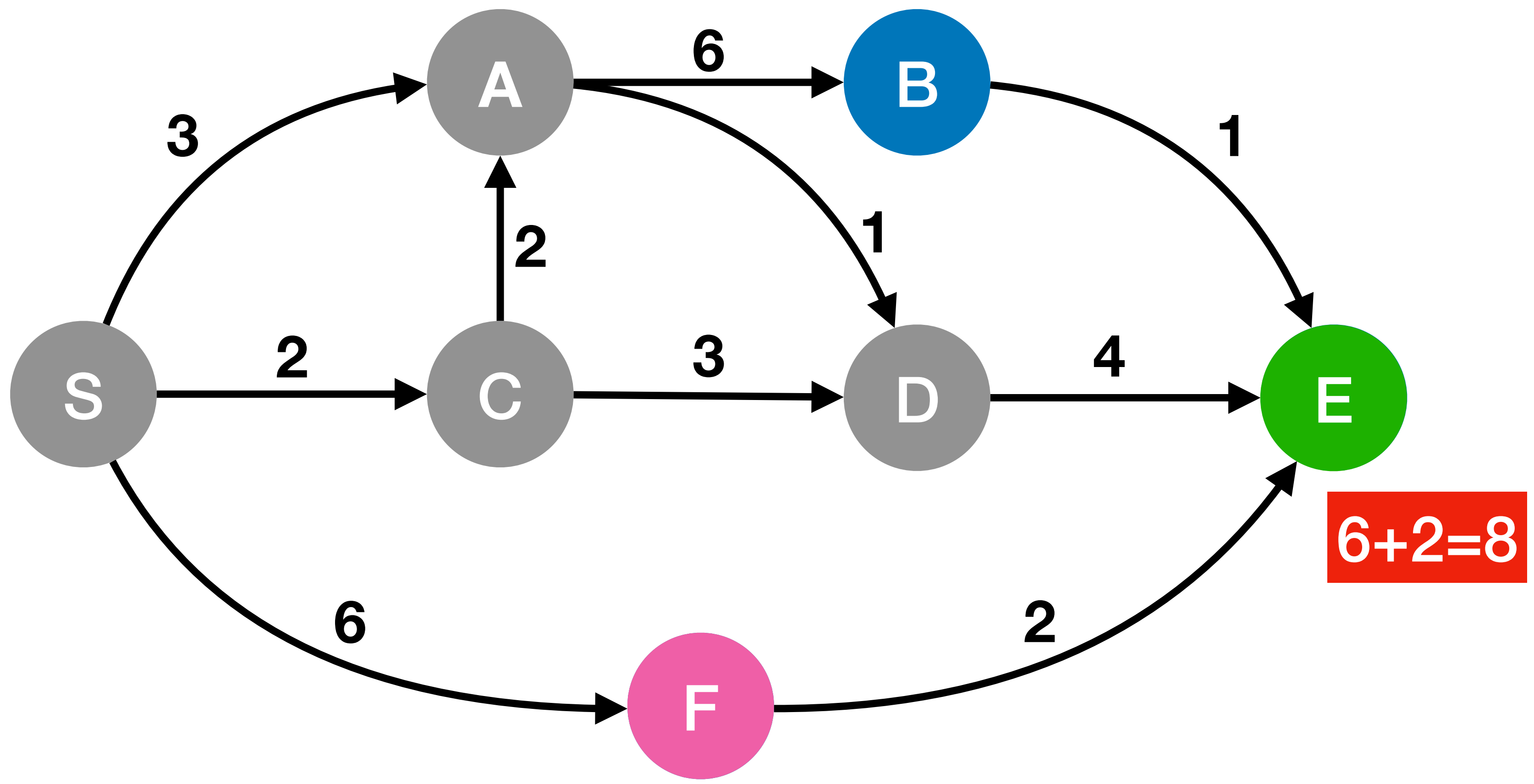



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D |

Settled = [ S, C, A, D ]  Unexplored = [F, B, E ]

# Dijkstra's algorithm

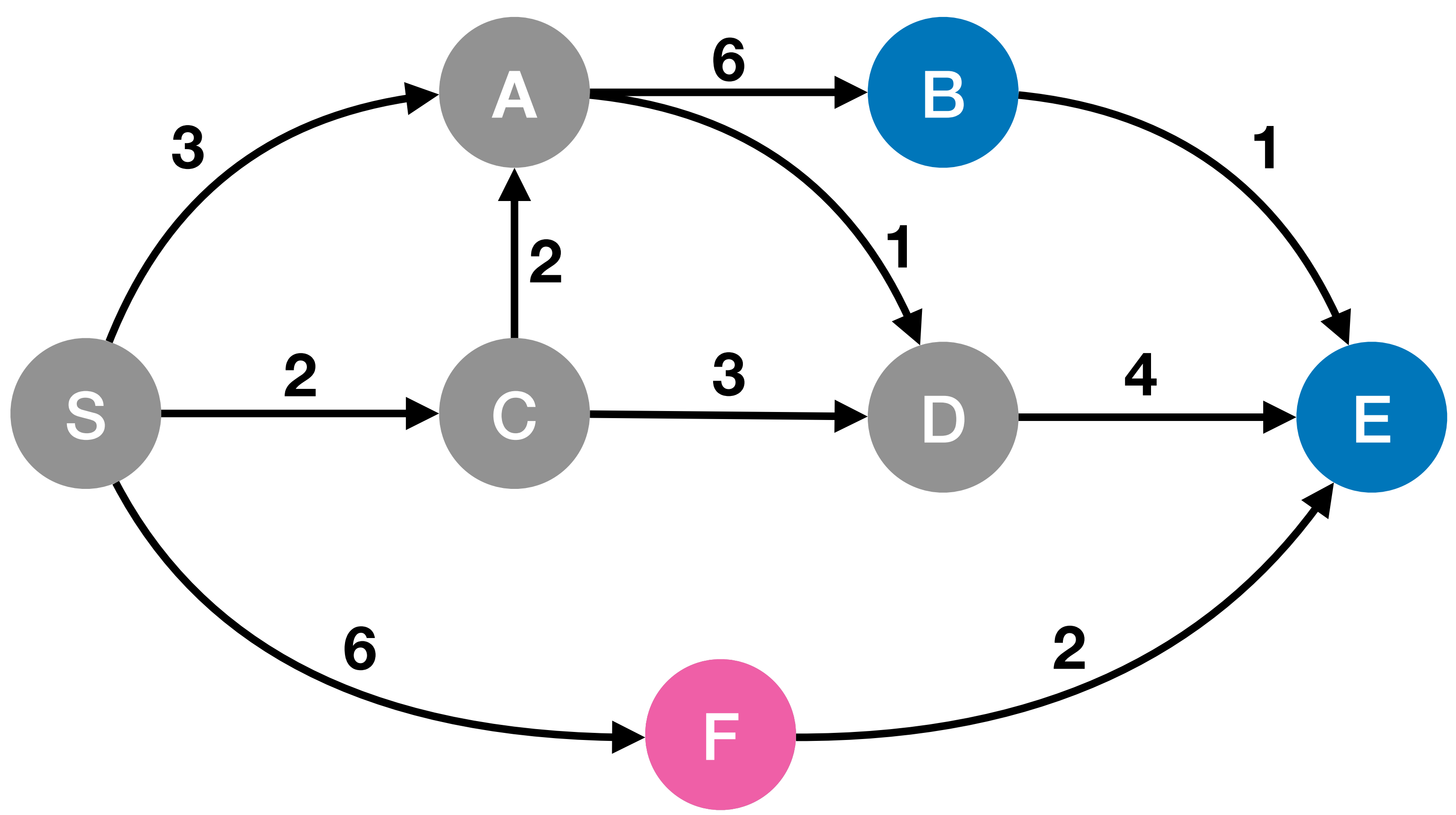
- For the current node, calculate the distance of each unsettled neighbor from the source node.
- If the calculated distance of a node is less than or equal to distance estimate, update the estimate & previous node.




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D ]        Unexplored = [F, B, E ]

# Dijkstra's algorithm

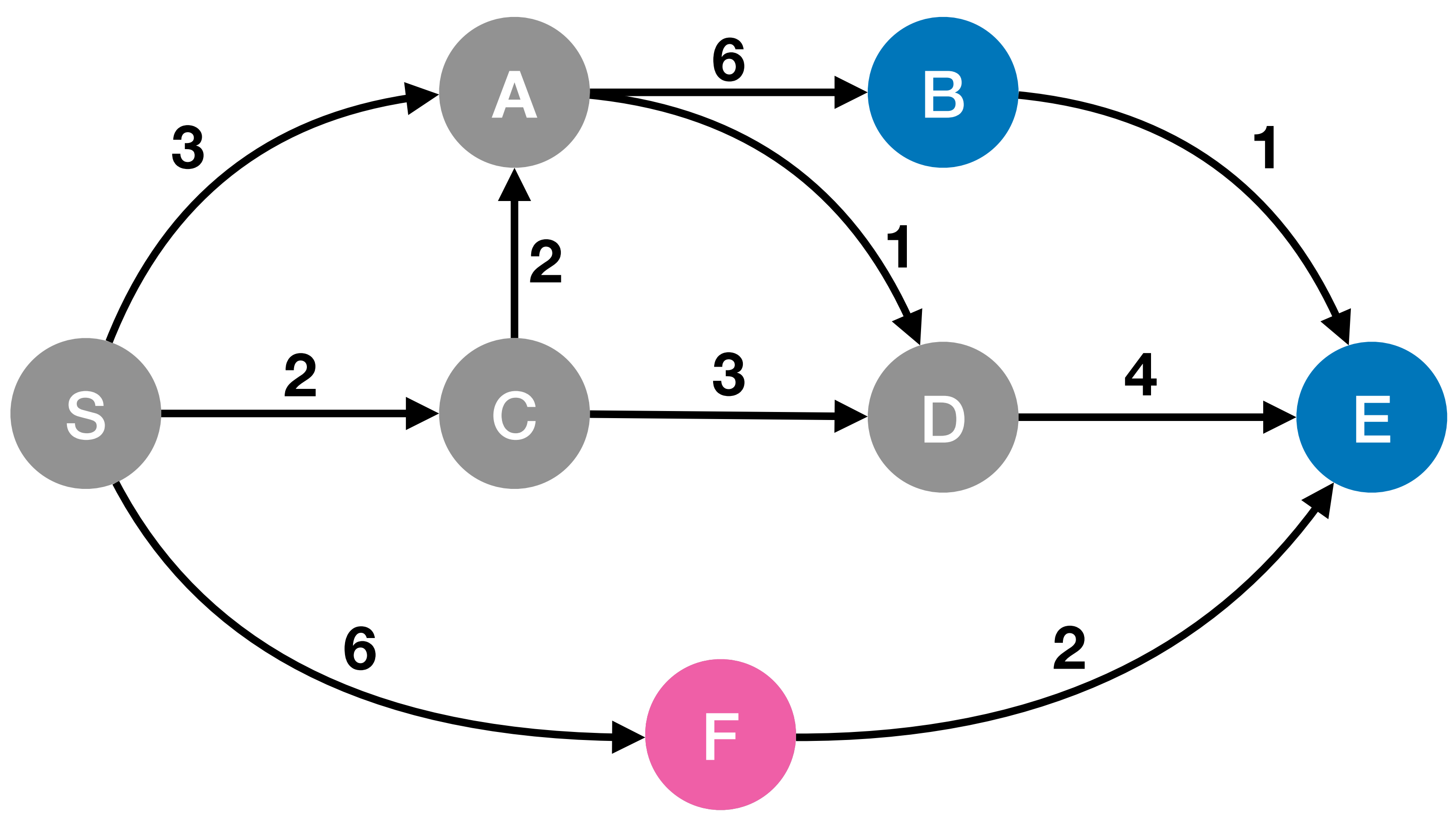



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, ] Unexplored = [F, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

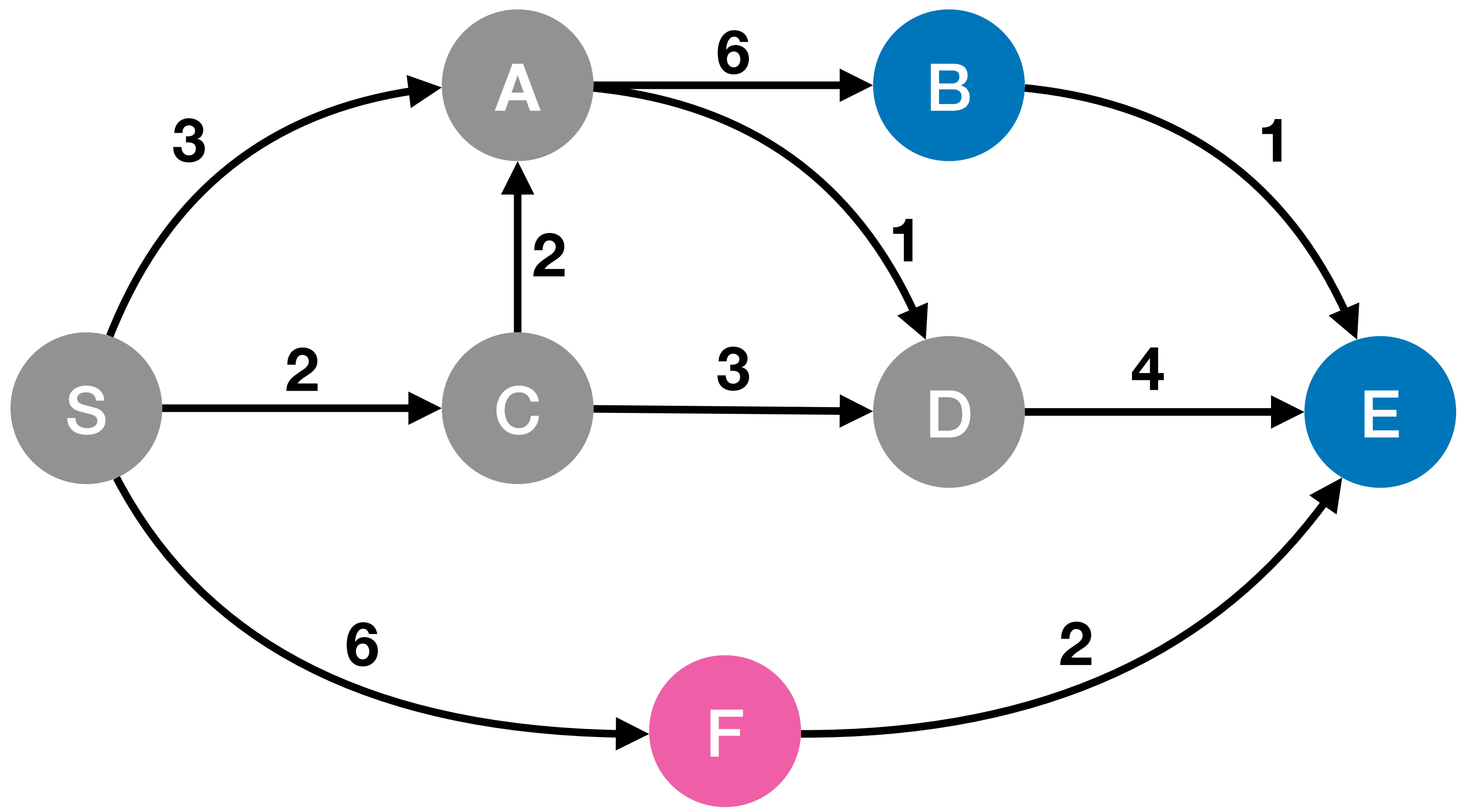



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, ]      Unexplored = [F, B, E ]

# Dijkstra's algorithm

- Add the current node to the list of *settled* nodes

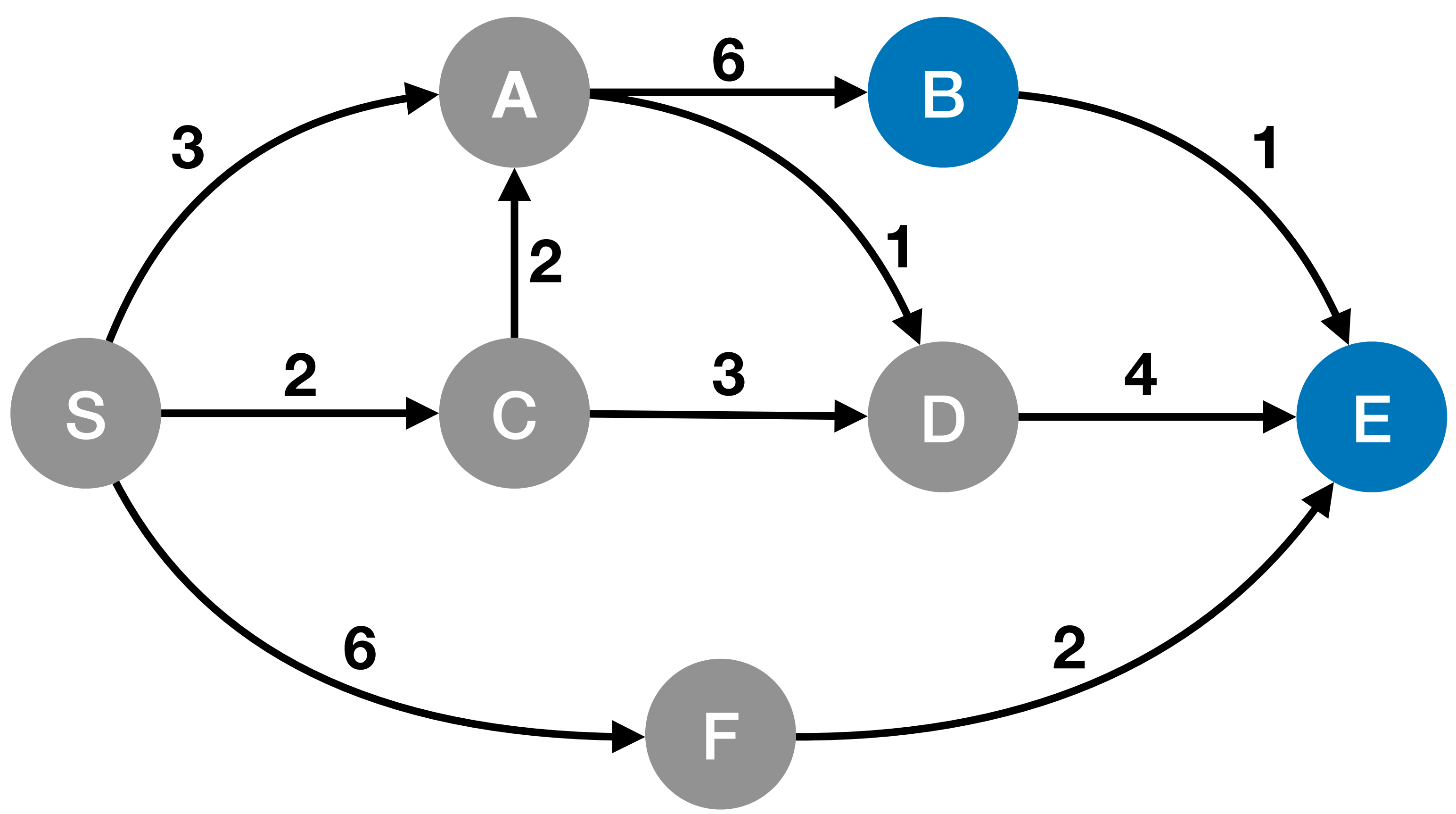



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, ]     Unexplored = [B, E ]

# Dijkstra’s algorithm

- Add the current node to the list of *settled* nodes

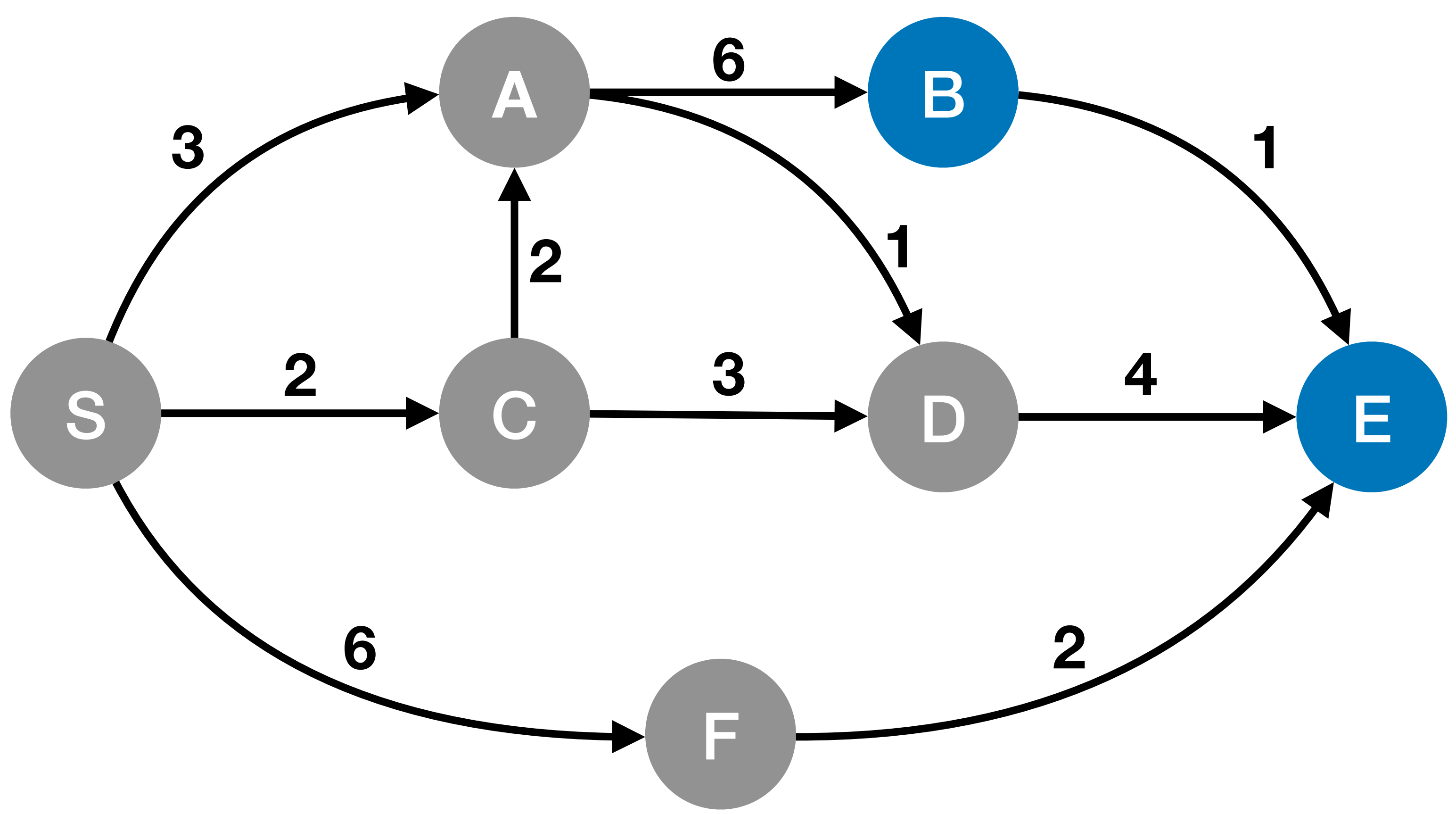



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F ]        Unexplored = [B, E ]

# Dijkstra's algorithm

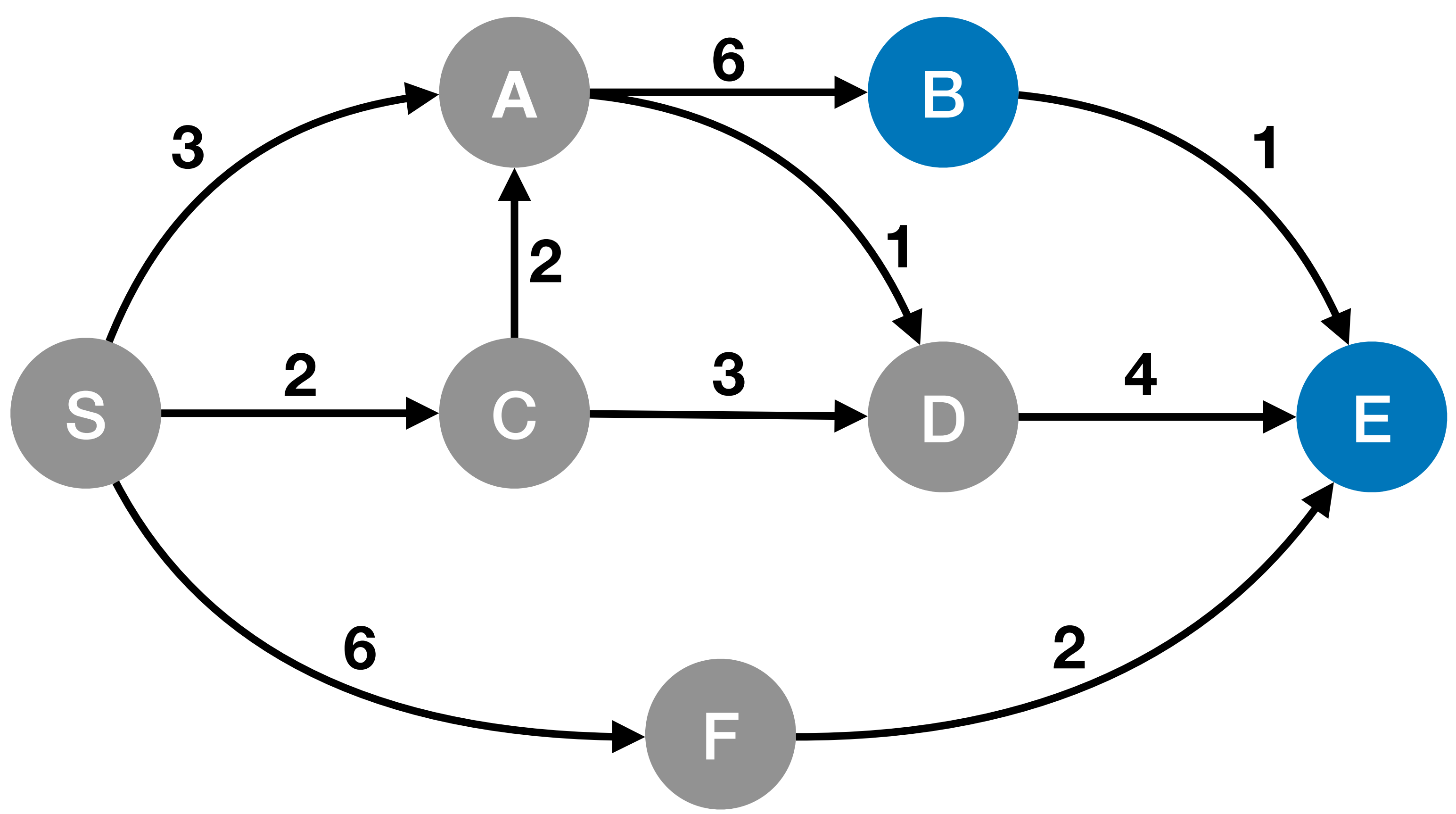
- Add the current node to the list of *settled* nodes




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| **F** | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F ]  Unexplored = [B, E ]

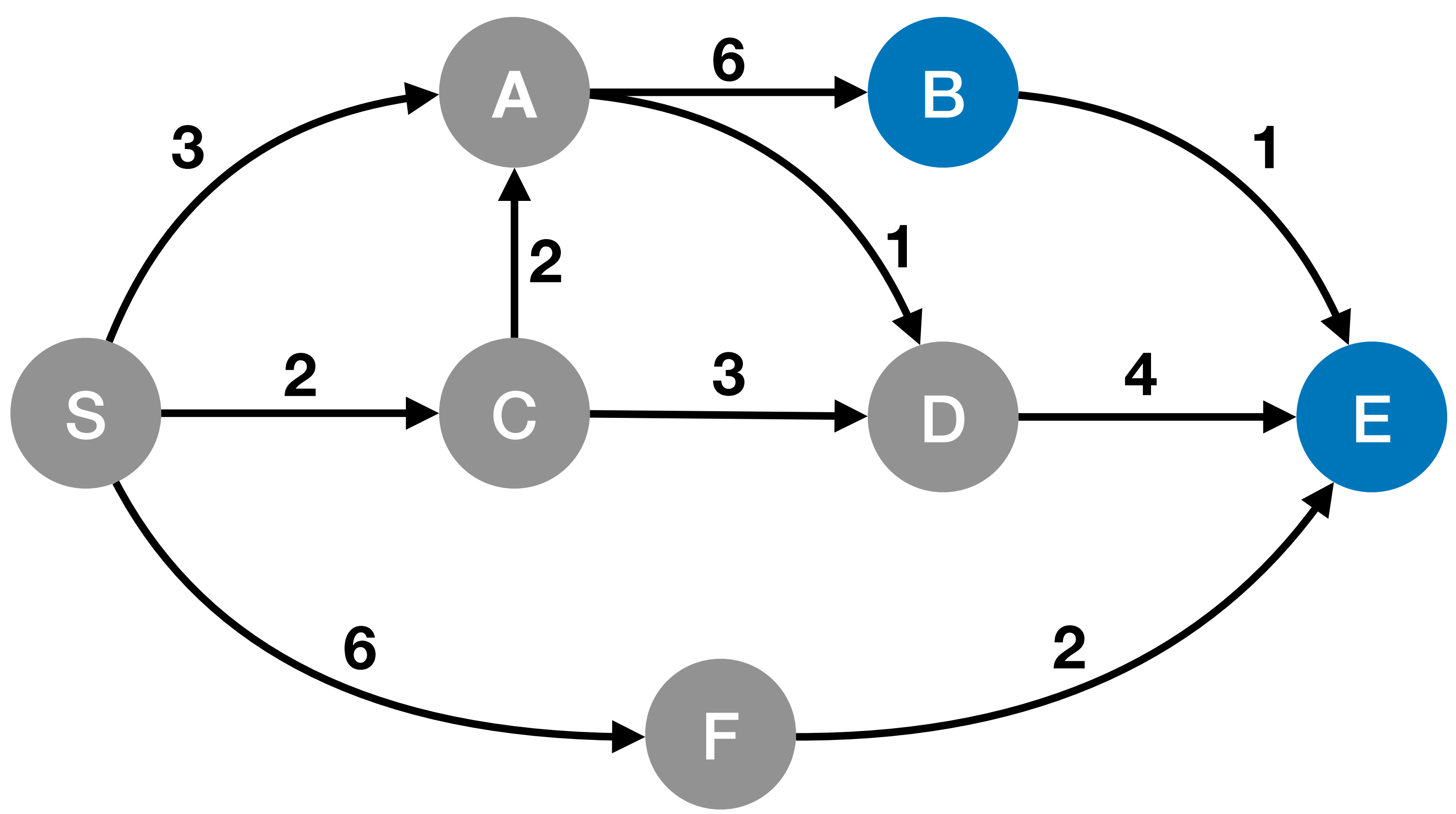
# Dijkstra's algorithm




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F ]     Unexplored = [B, E ]

# Dijkstra's algorithm




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

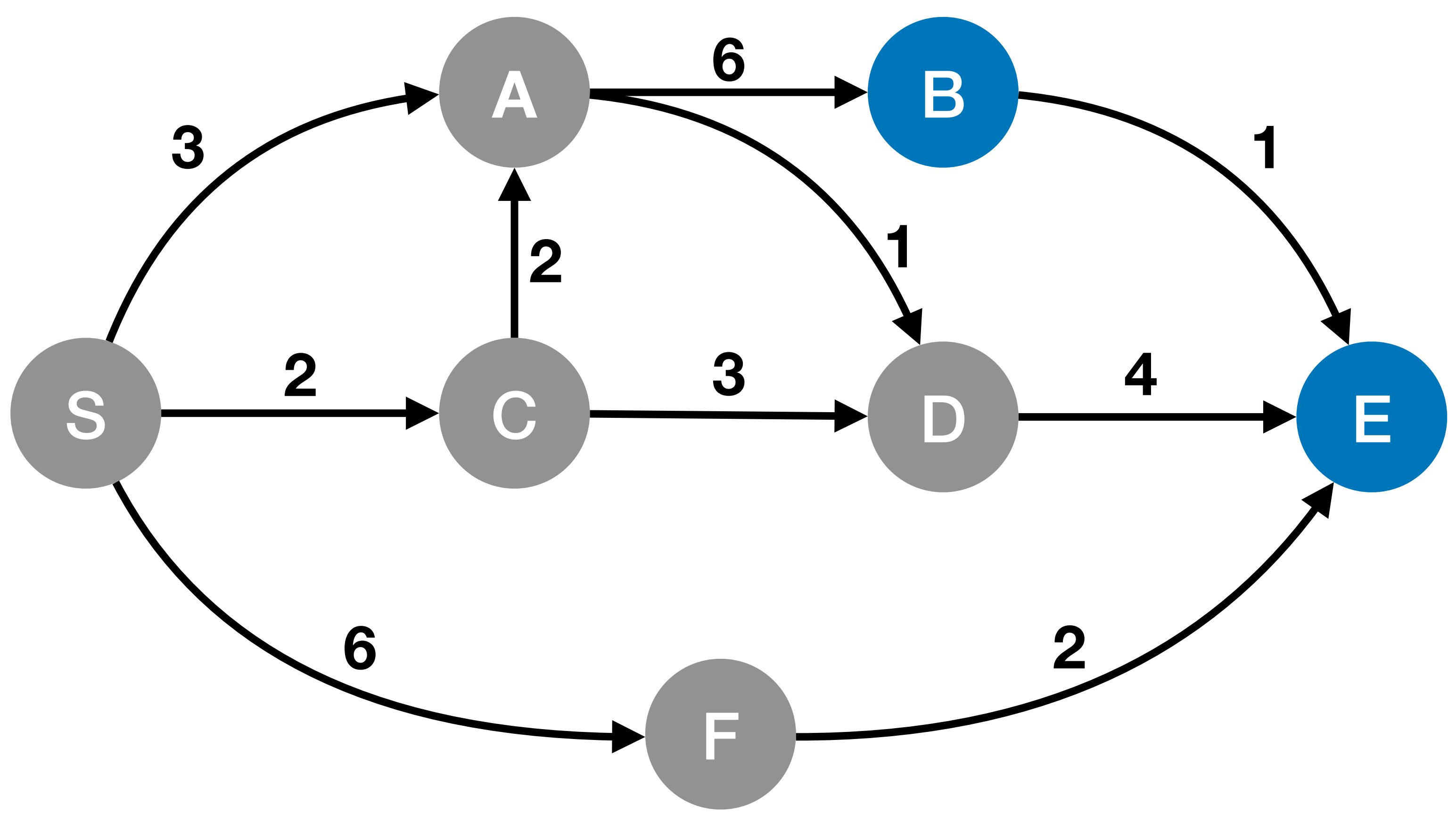
Settled = [ S, C, A, D, F ]    Unexplored = [B, E ]



Iterative step - Begin Iter 6

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node



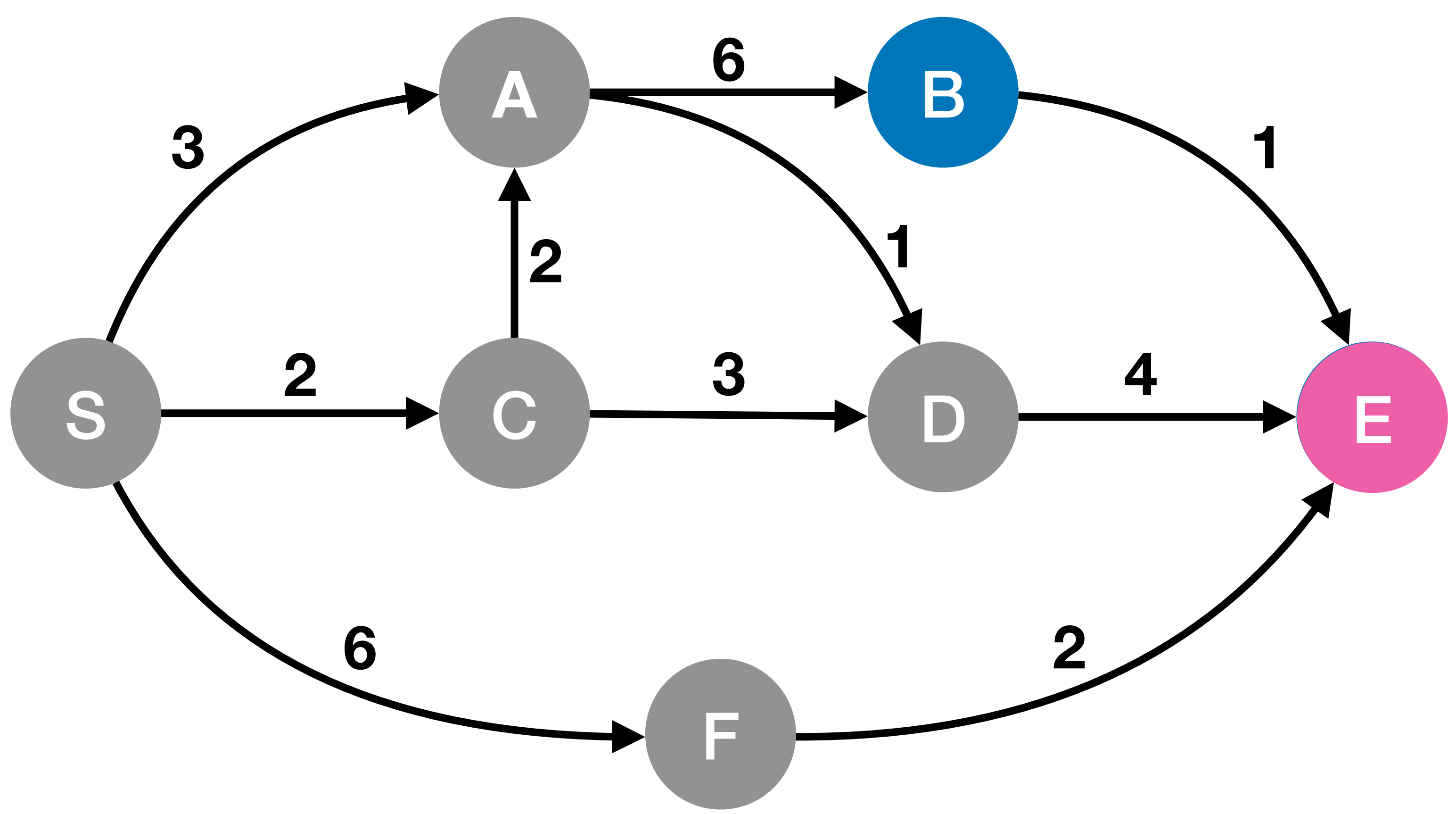

| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F ]        Unexplored = [B, E ]


Iterative step - Begin Iter 6

# Dijkstra’s algorithm

- Pick the unsettled node with the smallest known distance from the source node
- This time, it is node (E).

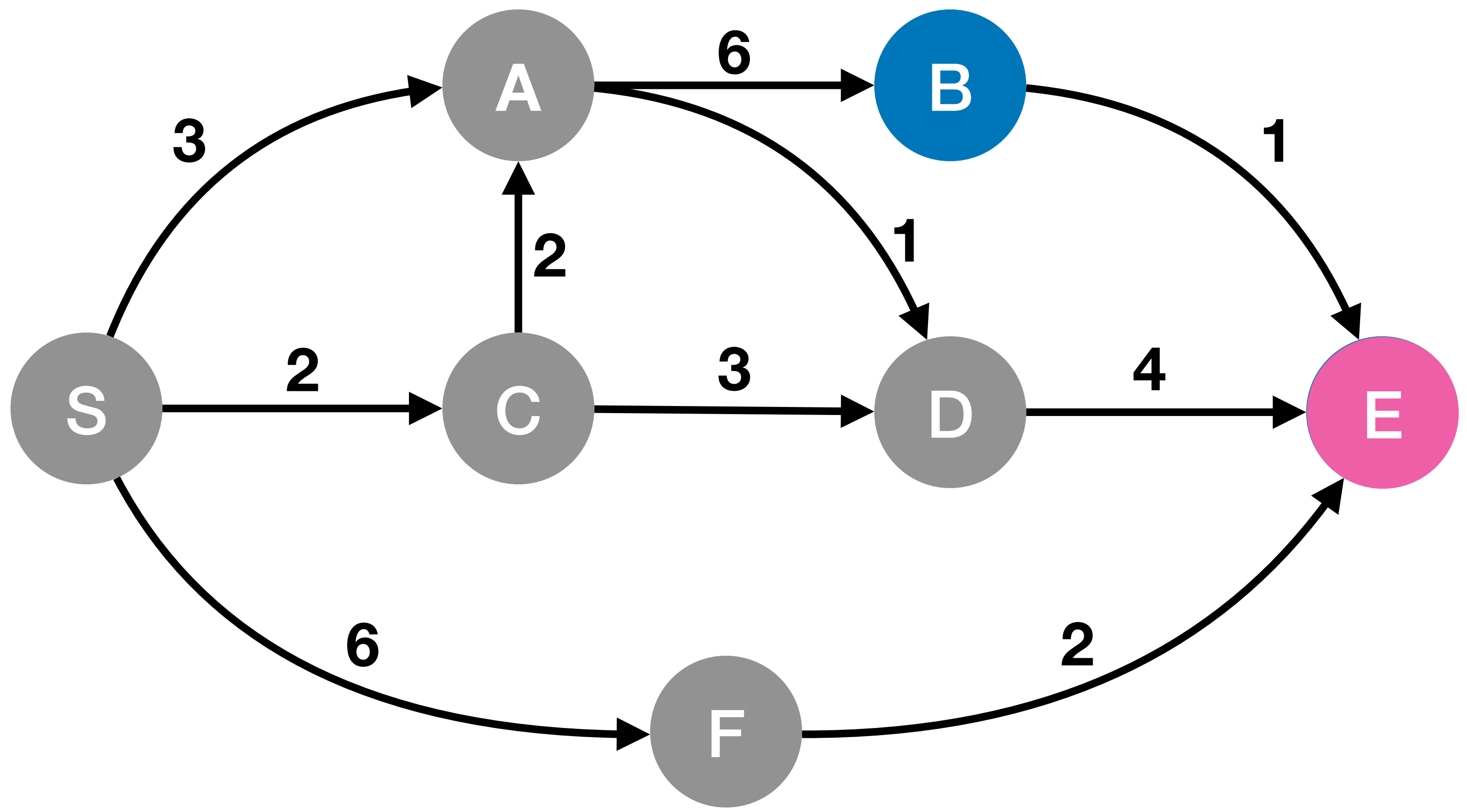



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F ]     Unexplored = [B, E ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

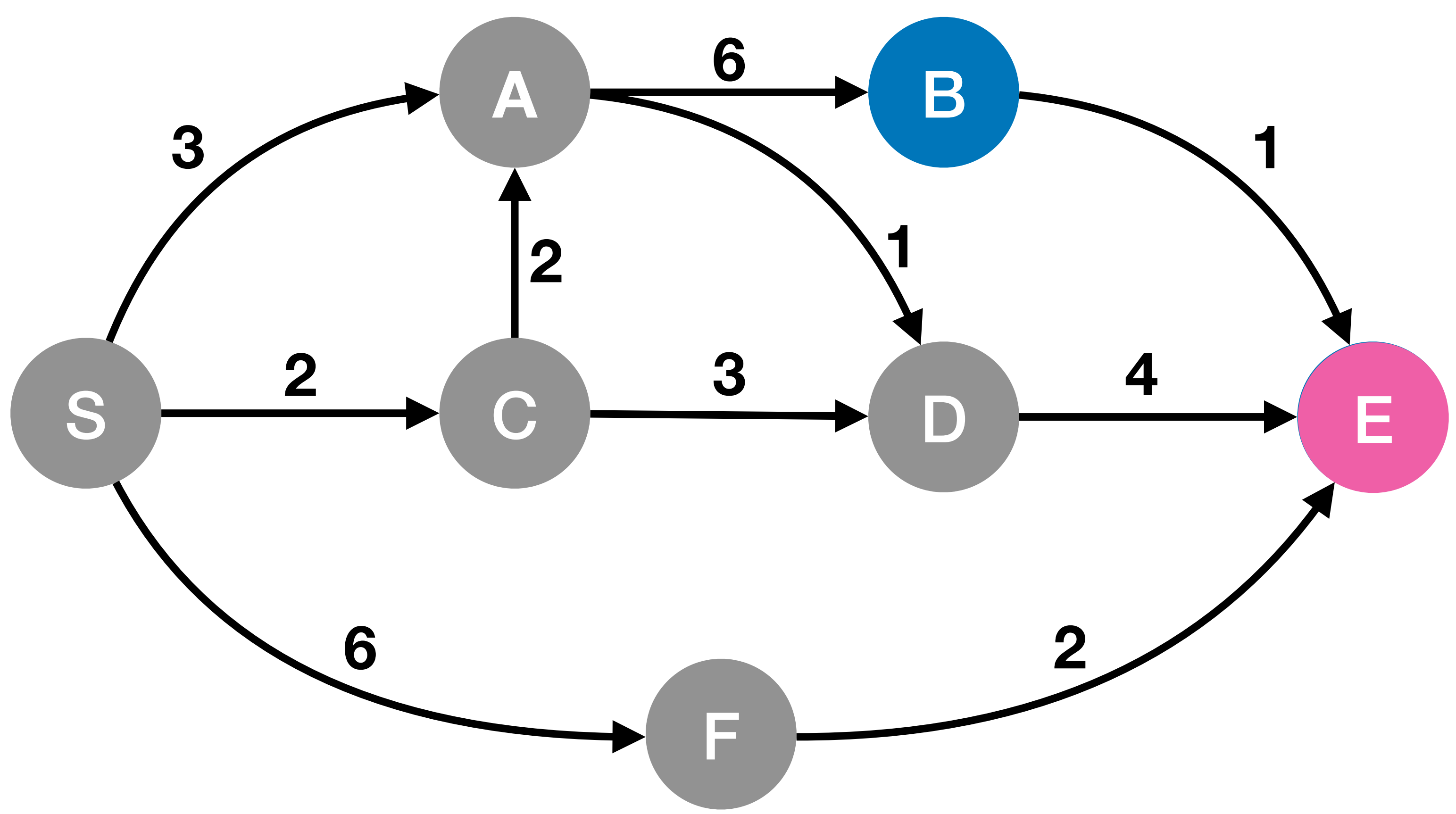
Settled = [ S, C, A, D, F ]        Unexplored = [B, E ]



Iterative step - Begin Iter 6

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → E; unexplored neighbors → {}




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

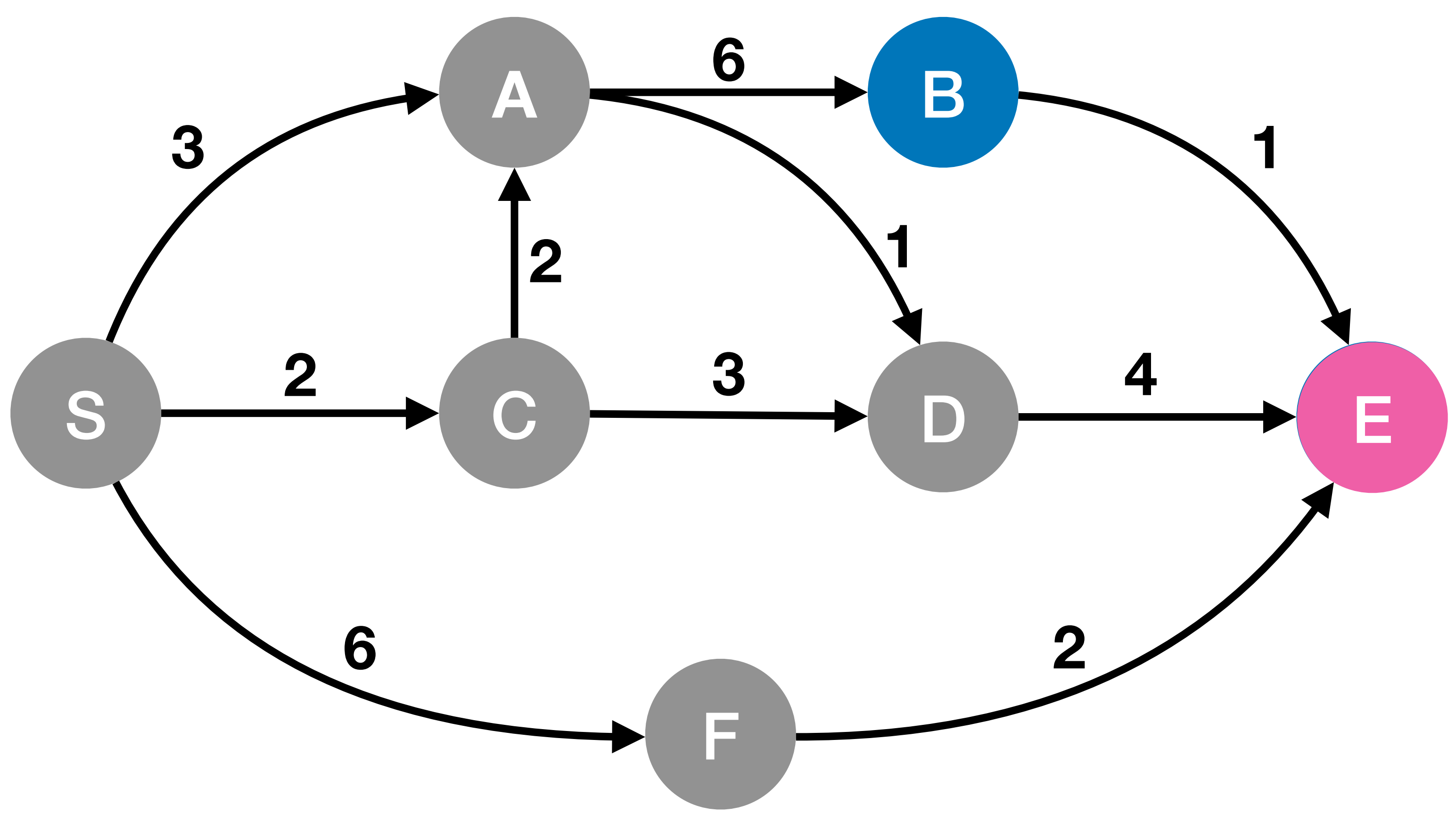
Settled = [ S, C, A, D, F ]        Unexplored = [B, E ]



Iterative step - Begin Iter 6

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → E; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

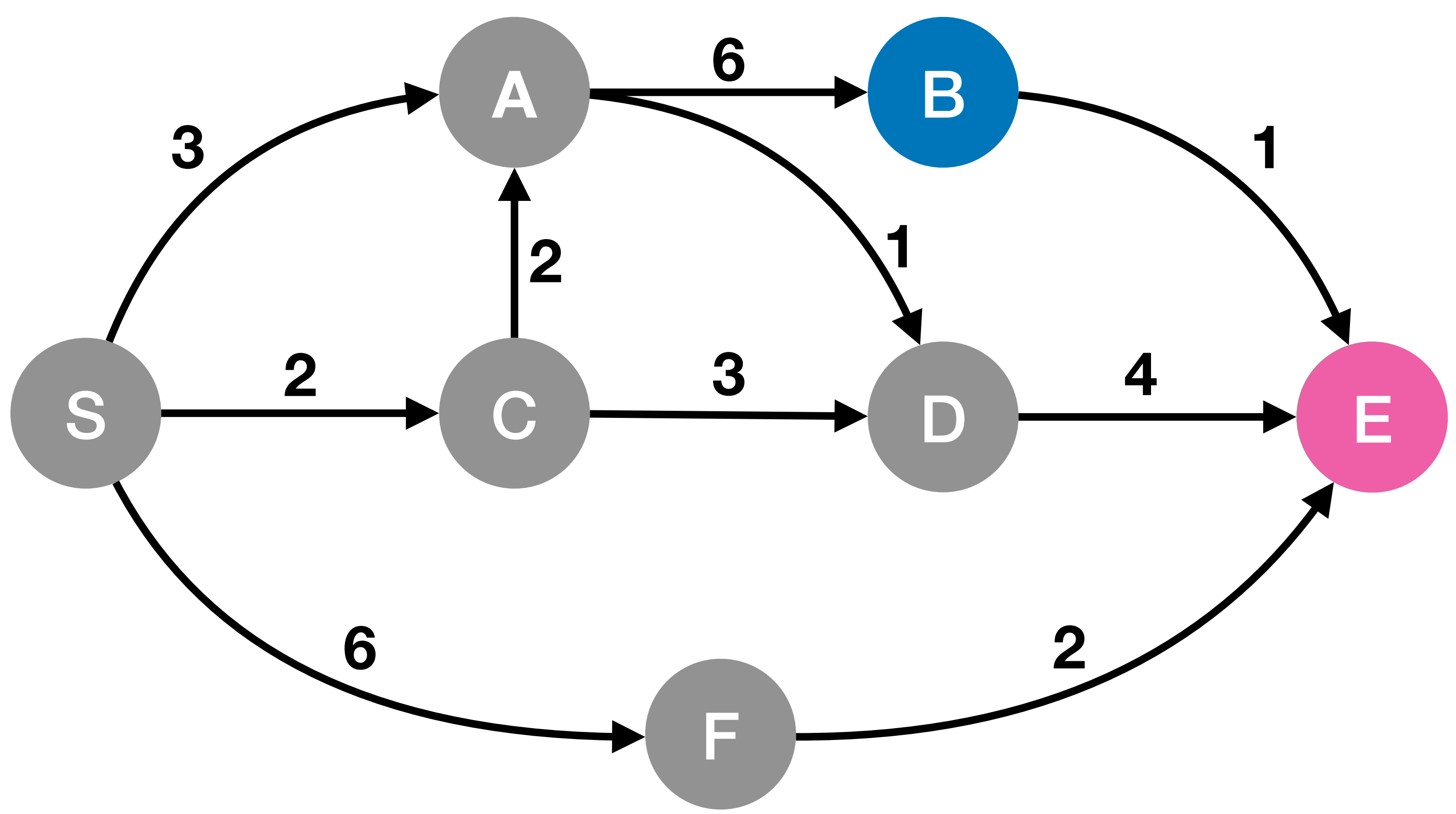



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F ]    Unexplored = [B, E ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → E; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

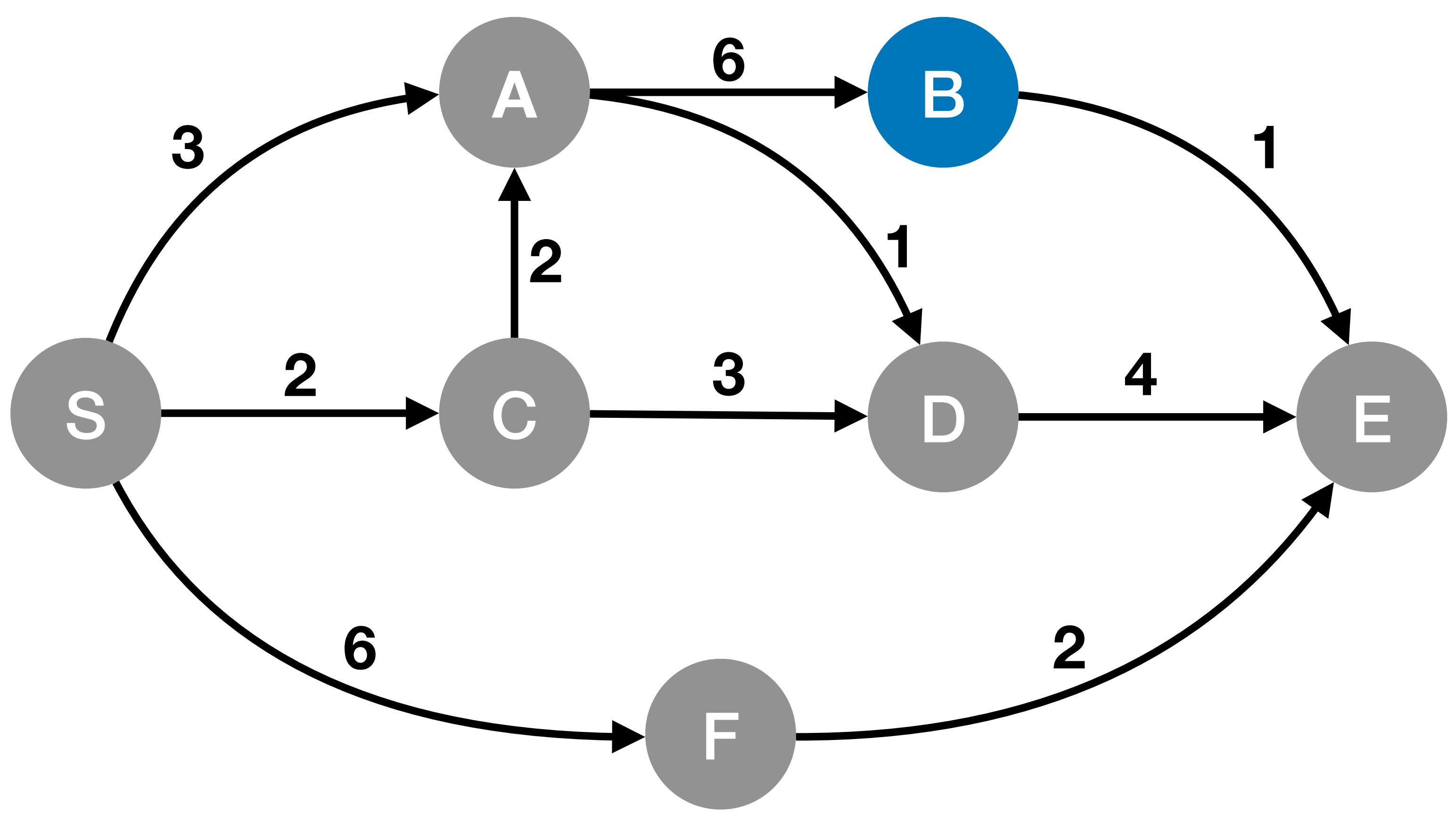



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]    Unexplored = [B ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → E; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

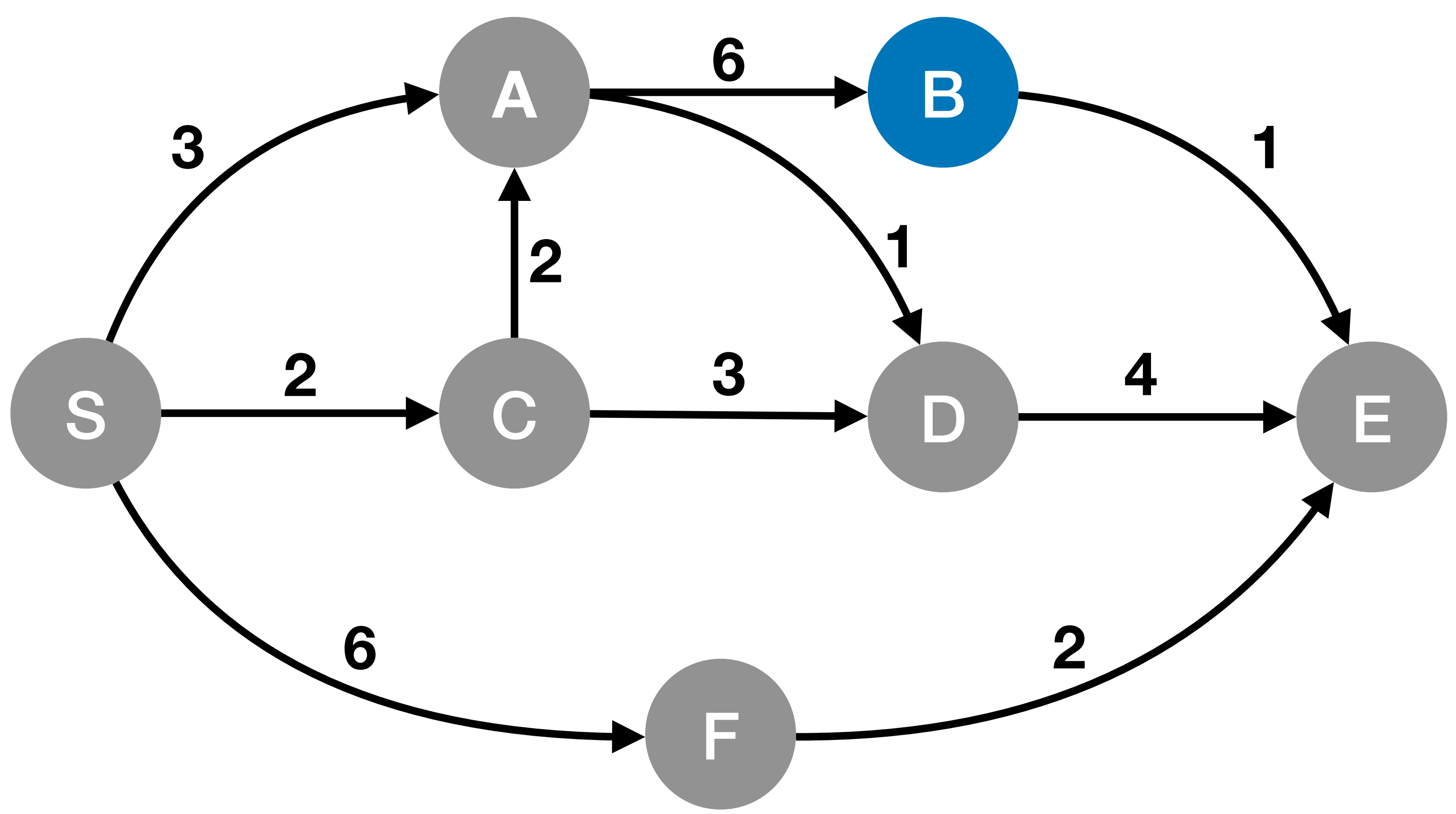



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]  Unexplored = [B ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → E; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

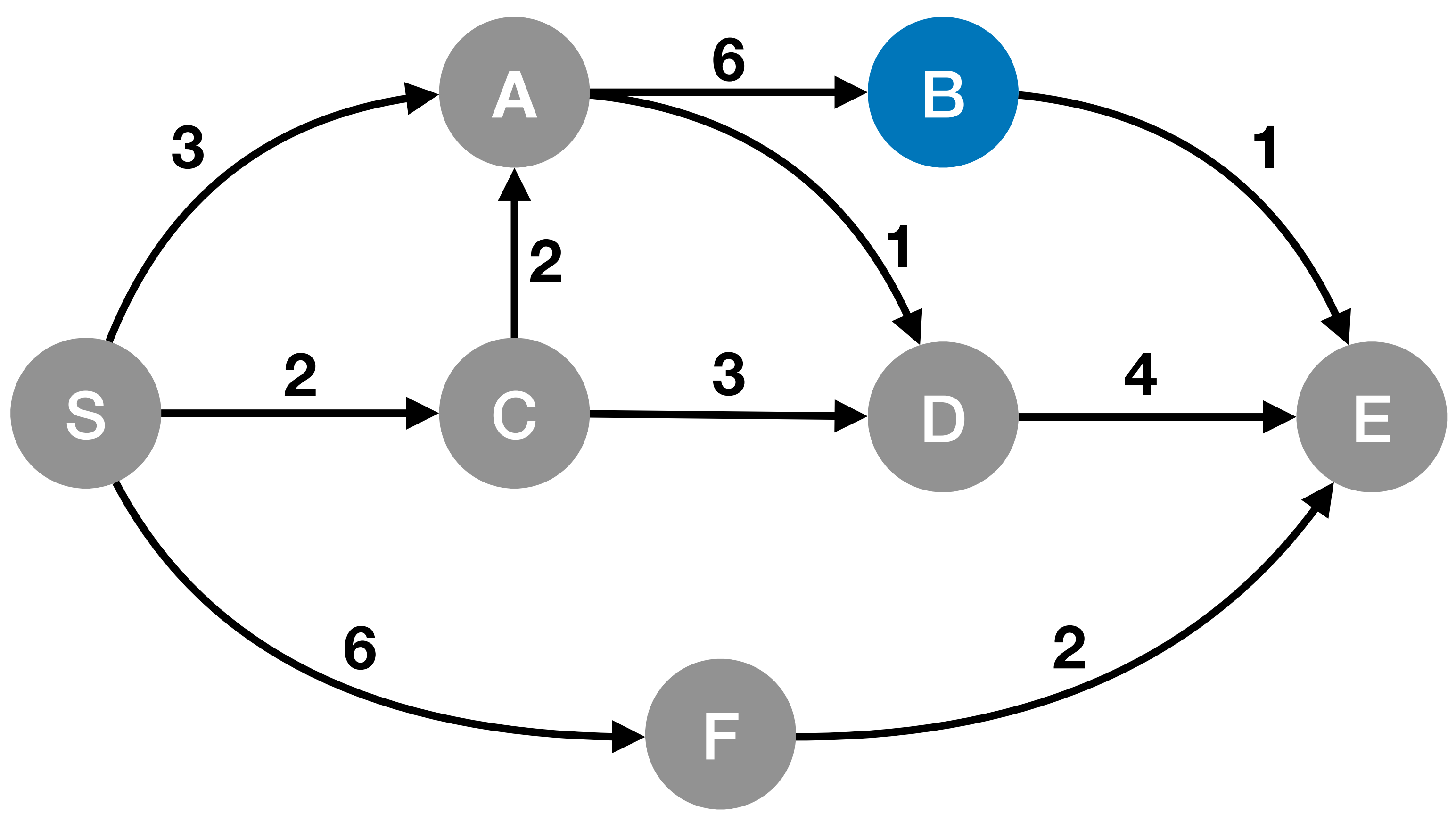



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]

Unexplored = [B ]

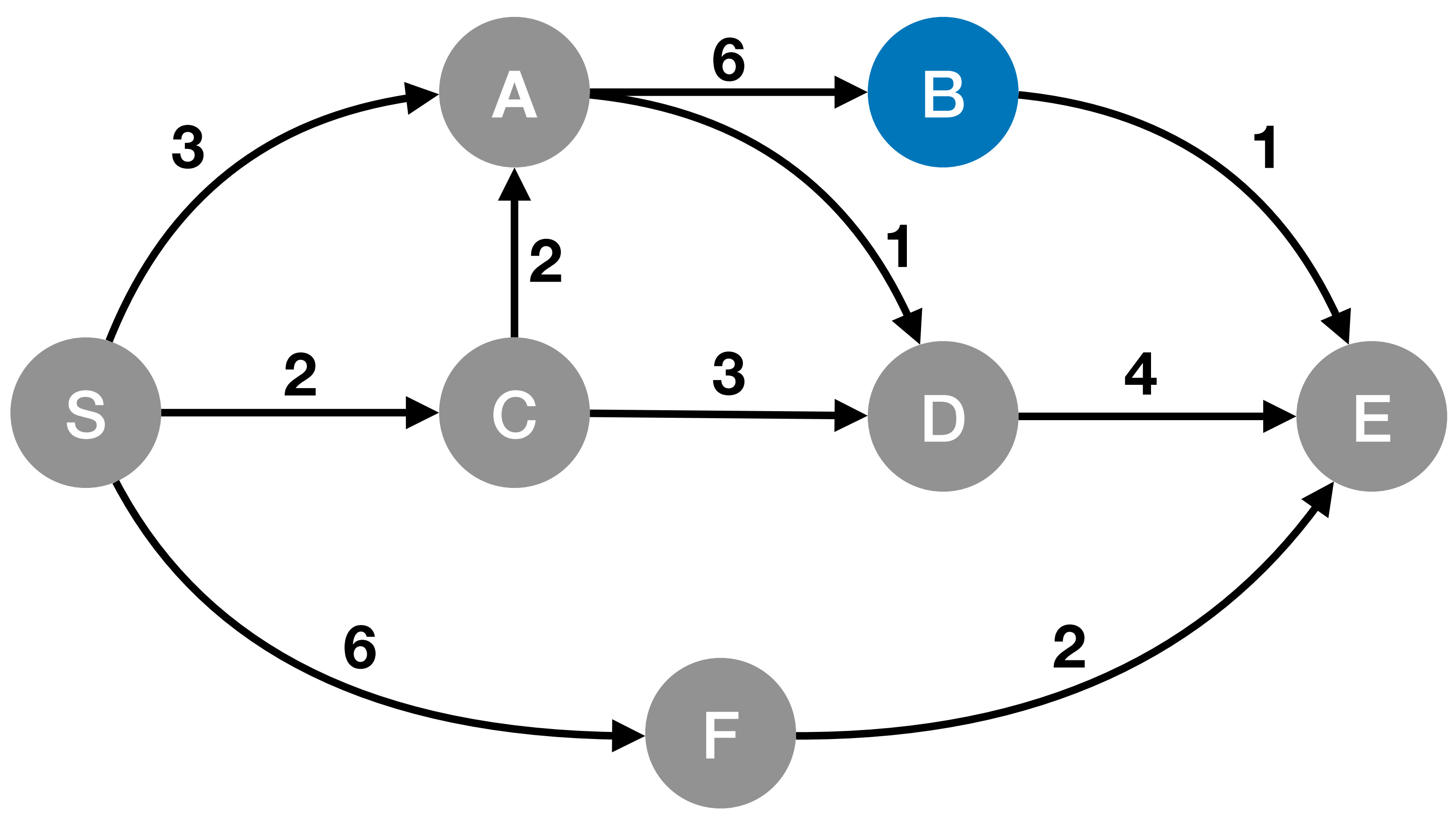
# Dijkstra's algorithm




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]        Unexplored = [B ]

# Dijkstra's algorithm

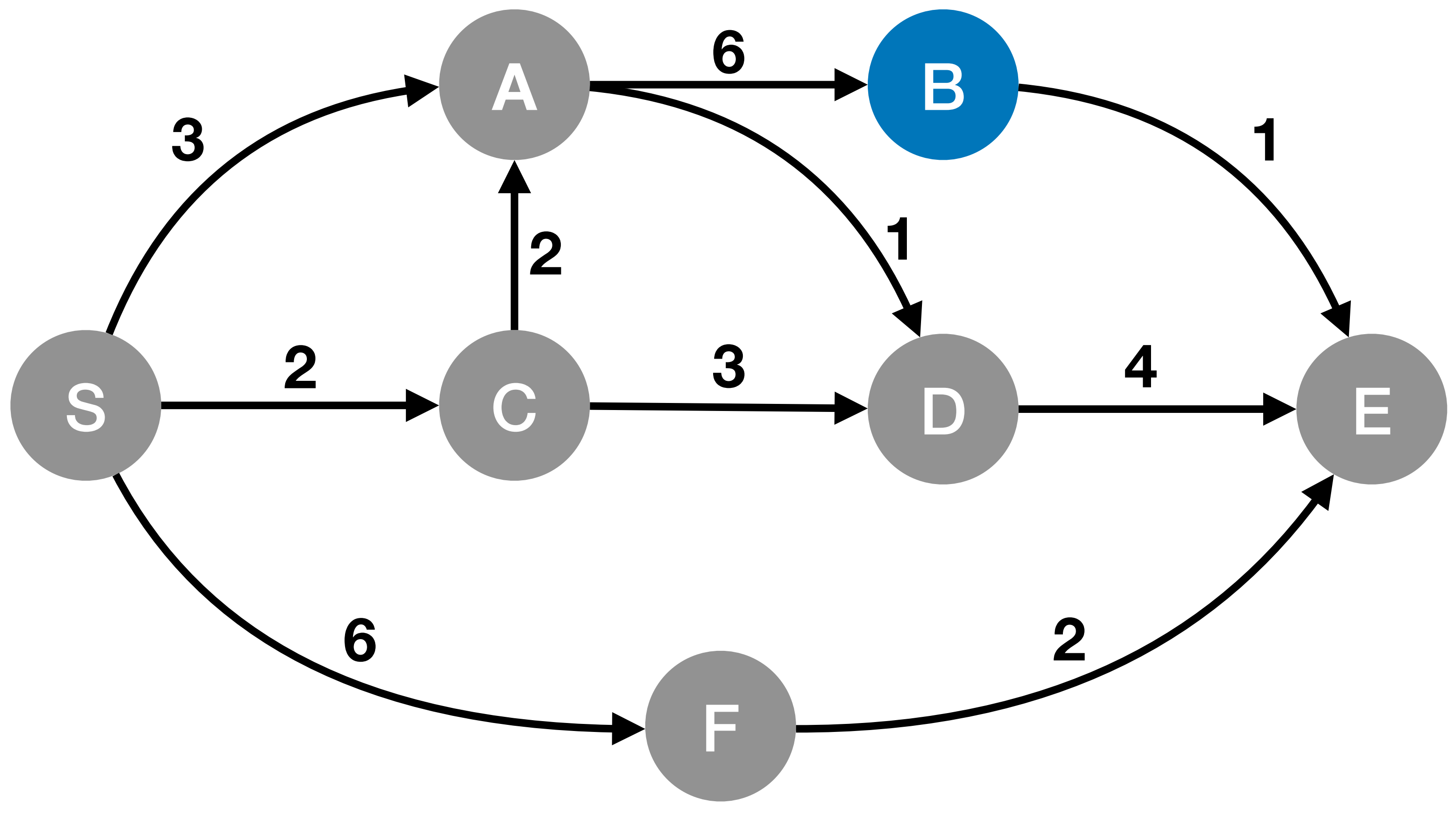



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]    Unexplored = [B ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node

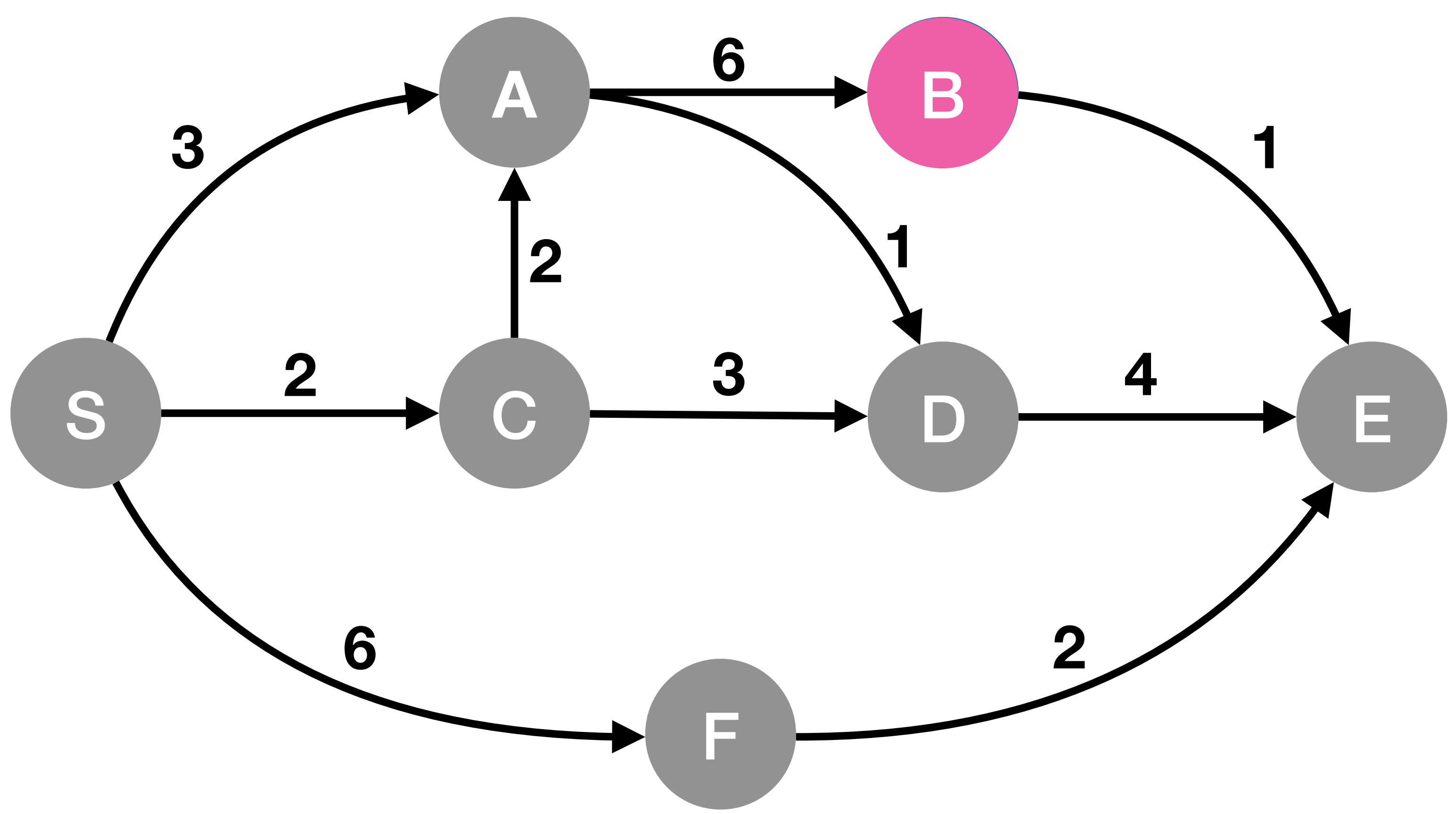



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]        Unexplored = [B ]

# Dijkstra's algorithm

- Pick the unsettled node with the smallest known distance from the source node
- This time, it is node (B).

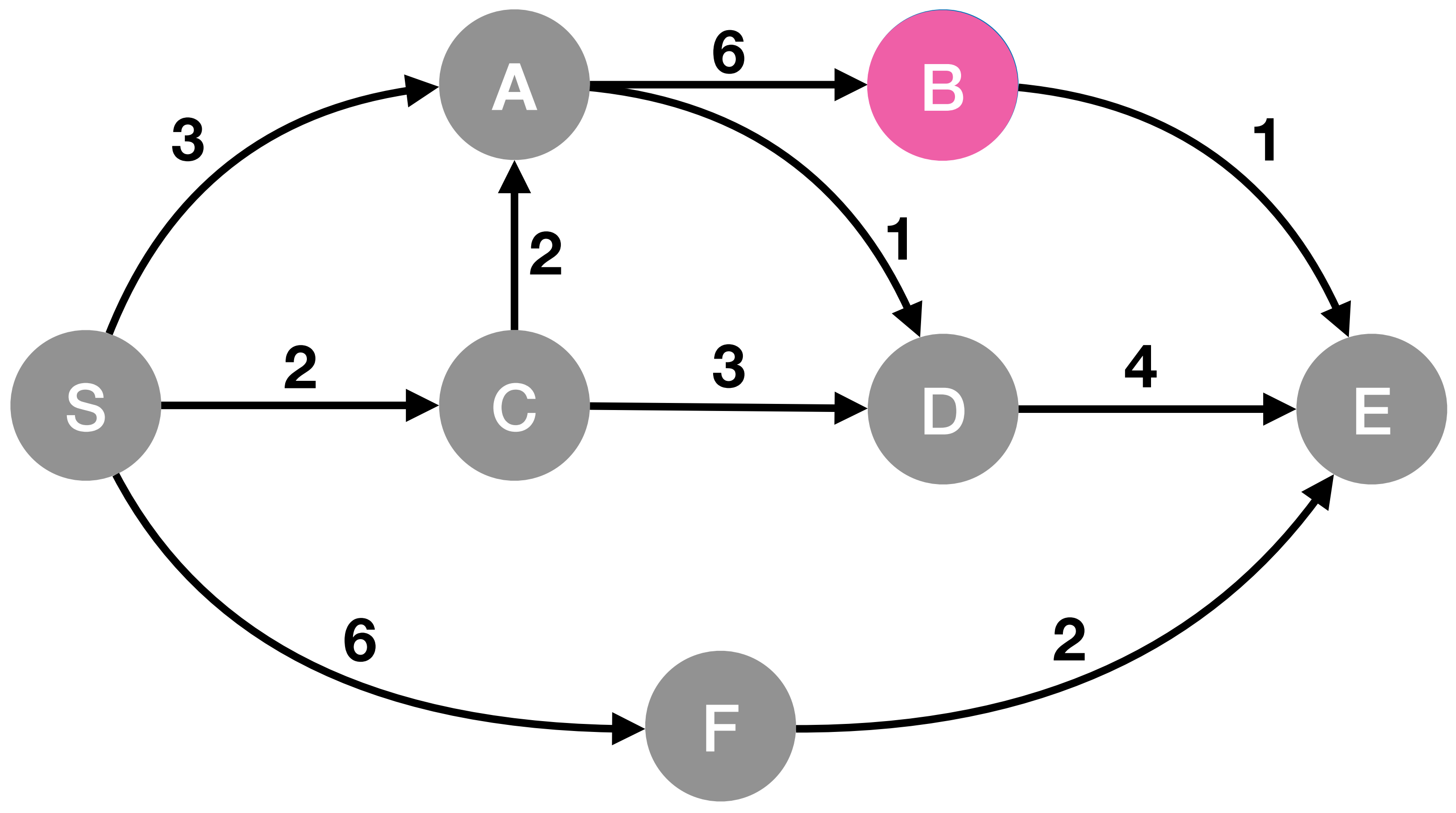



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]    Unexplored = [B ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors

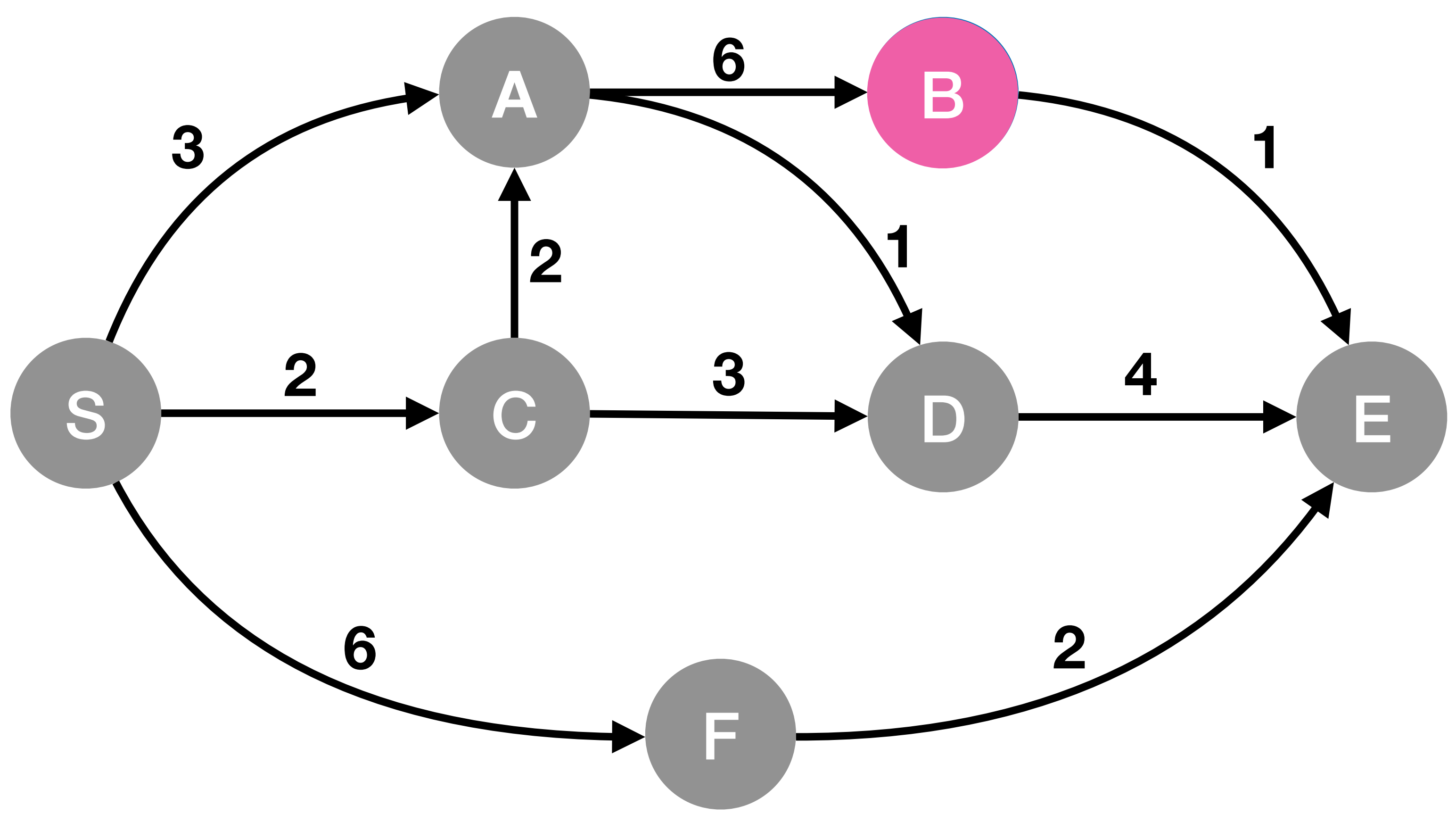



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]     Unexplored = [B ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → B; unexplored neighbors → {}

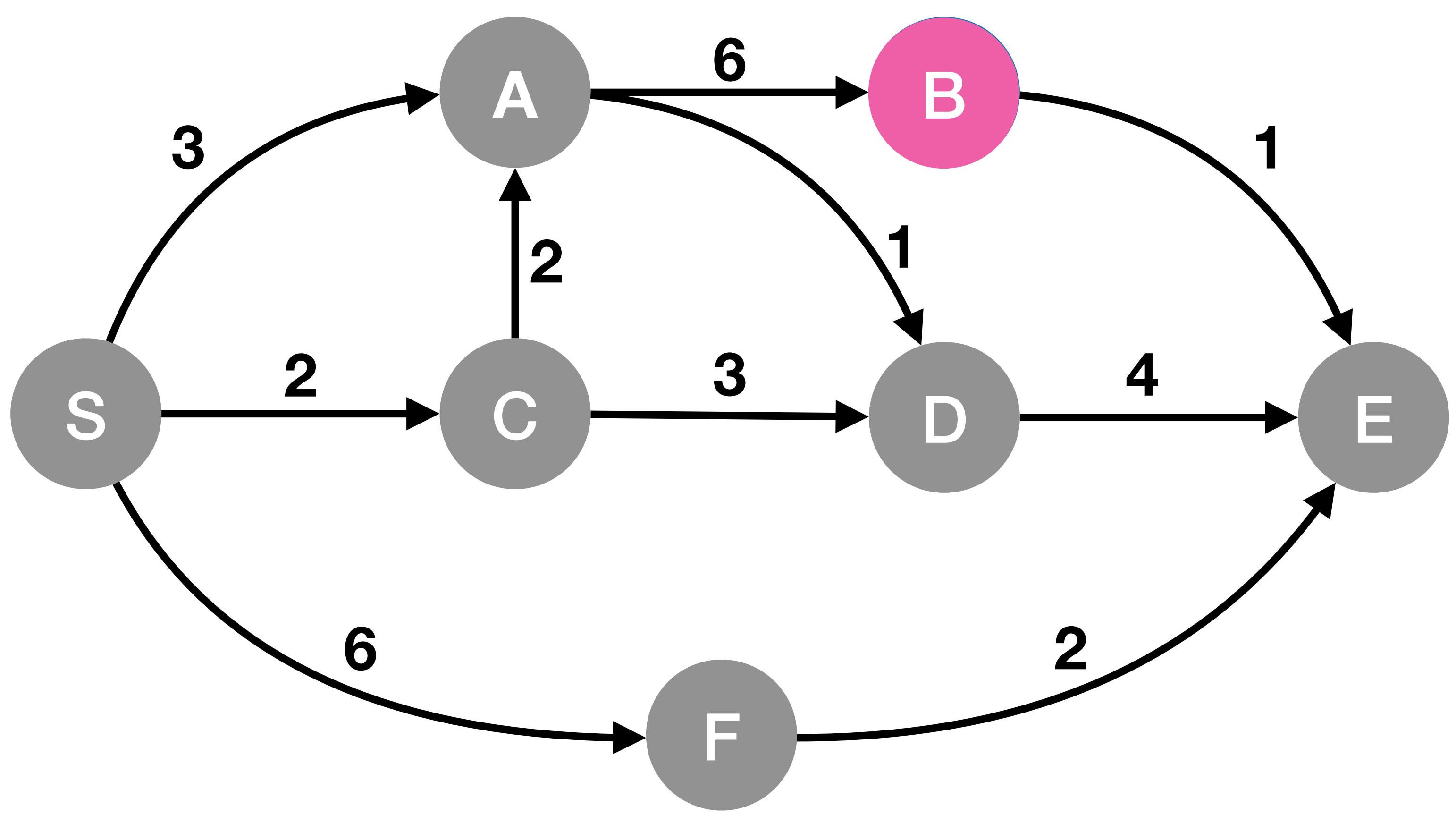



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]    Unexplored = [B ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → B; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

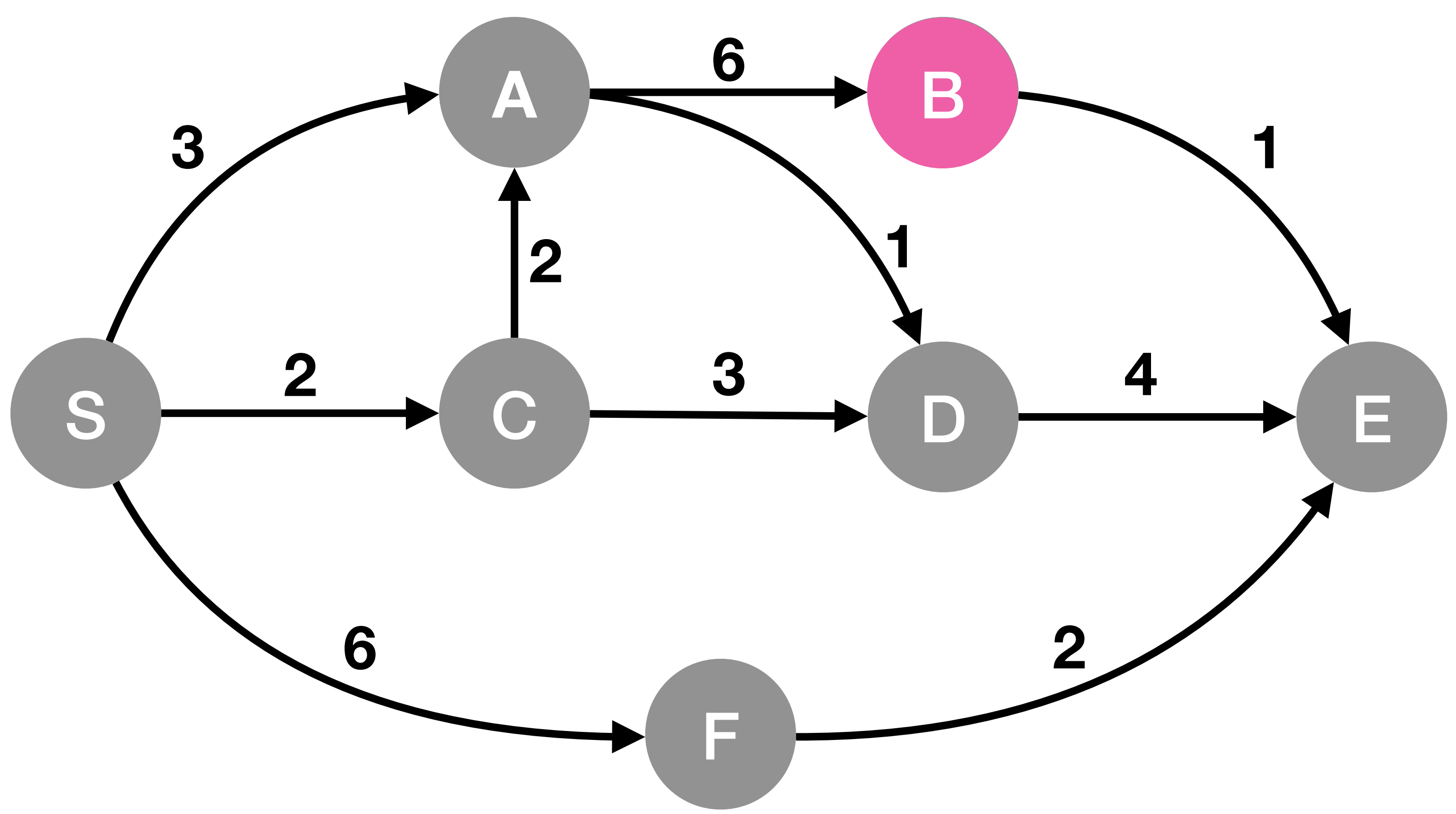



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E ]  Unexplored = [B ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → B; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

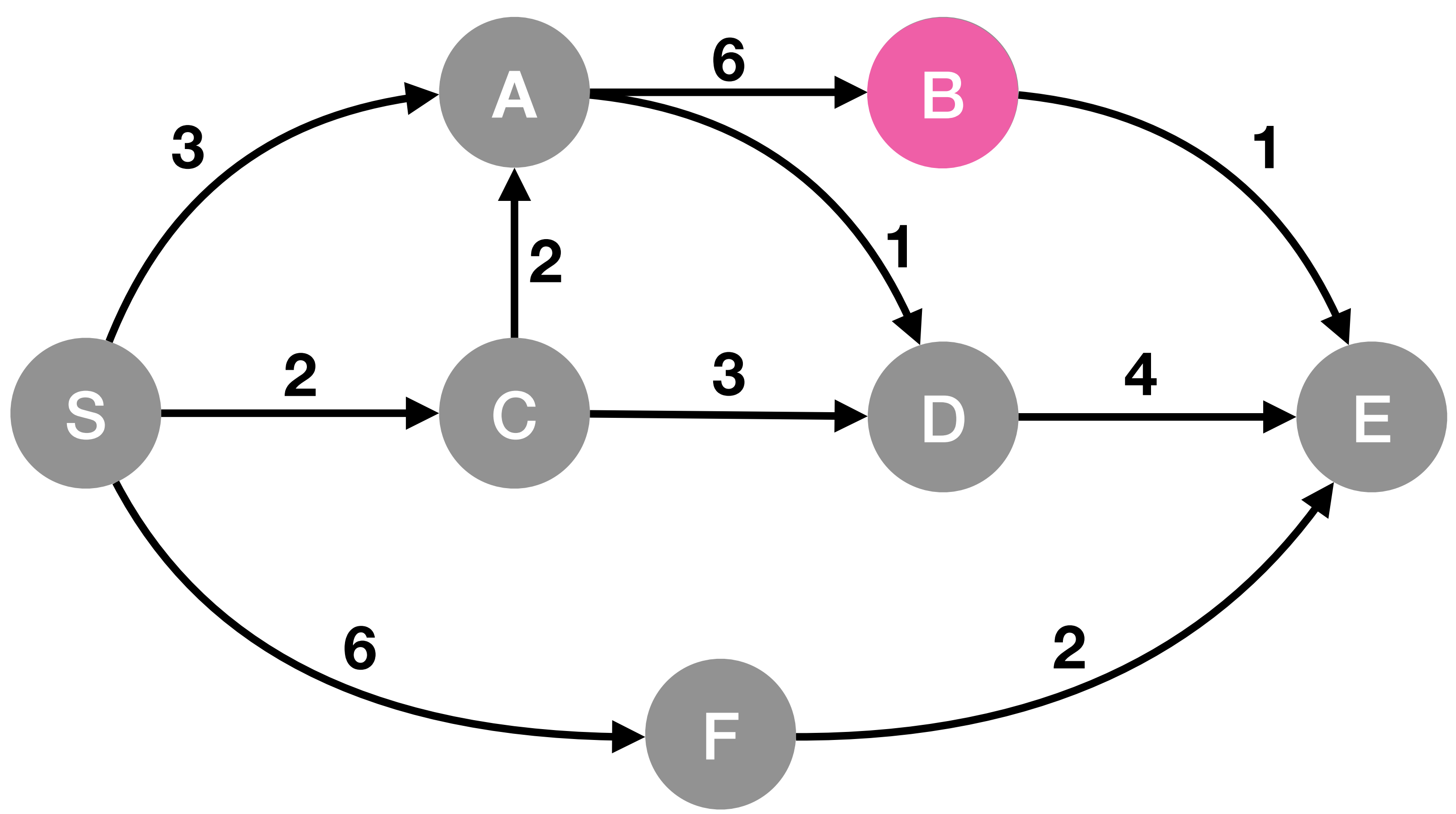



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E, ]    Unexplored = [B ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → B; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

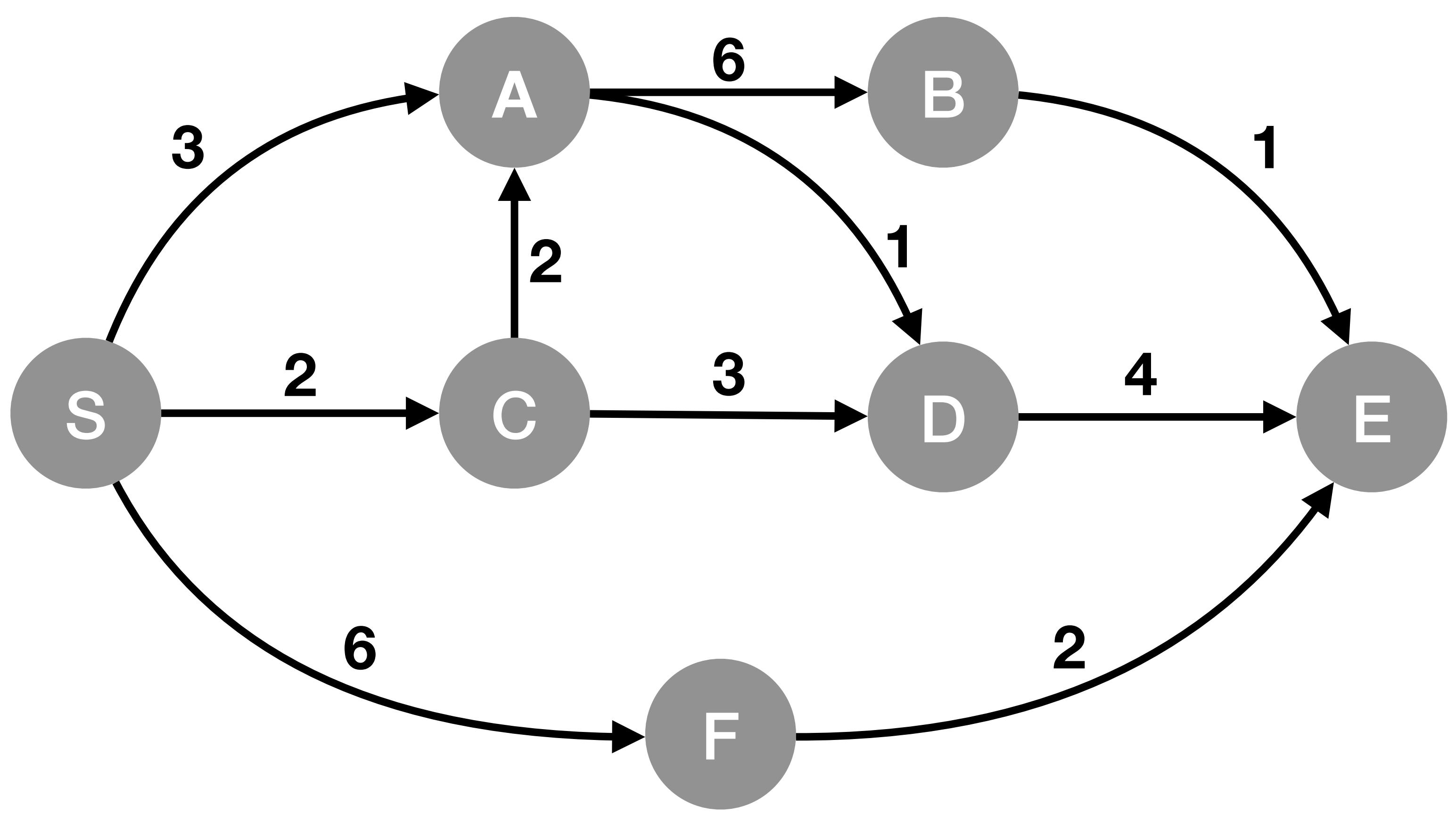



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E, ] Unexplored = [ ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → B; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

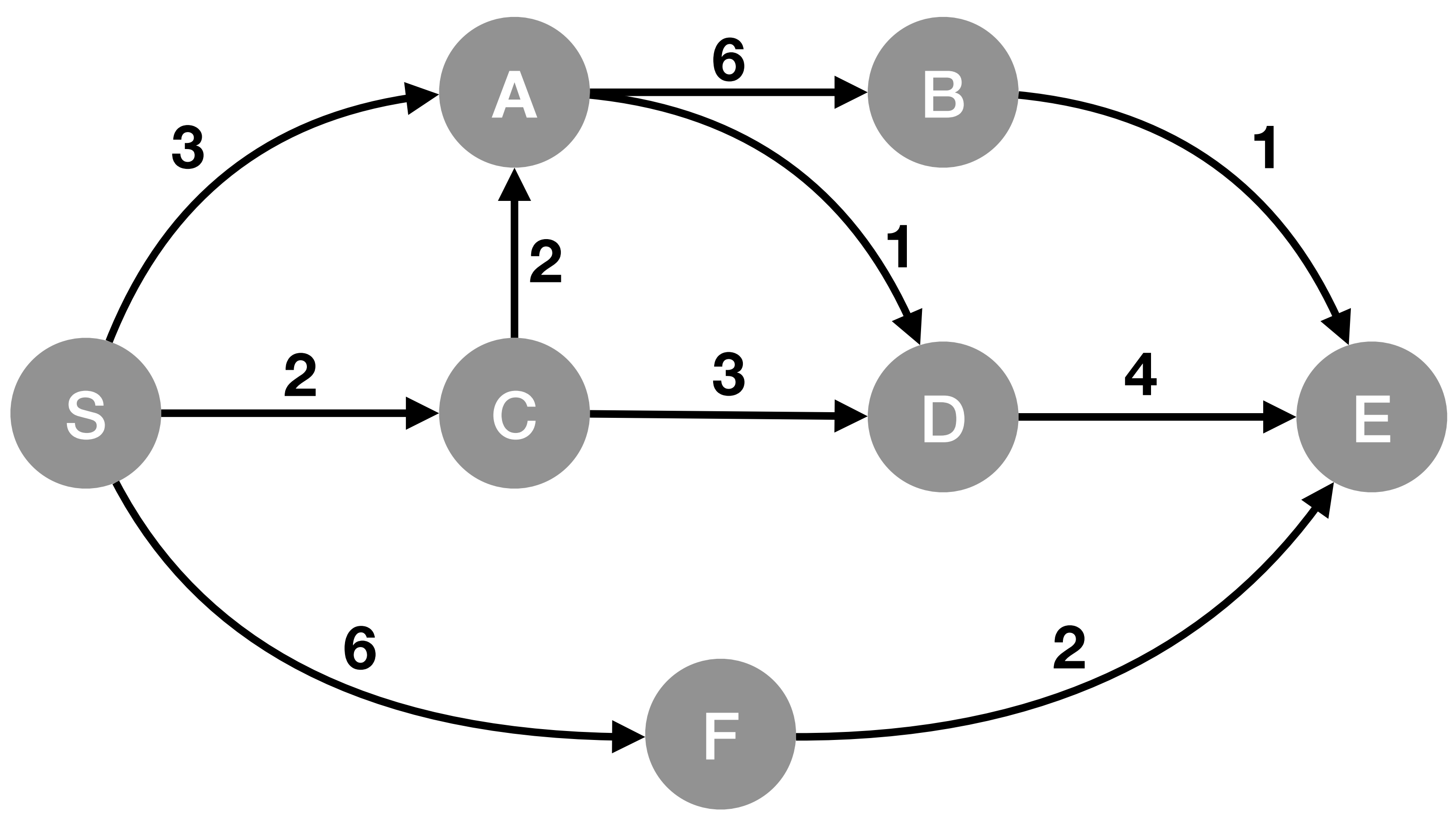



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E, B ]   Unexplored = [ ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → B; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

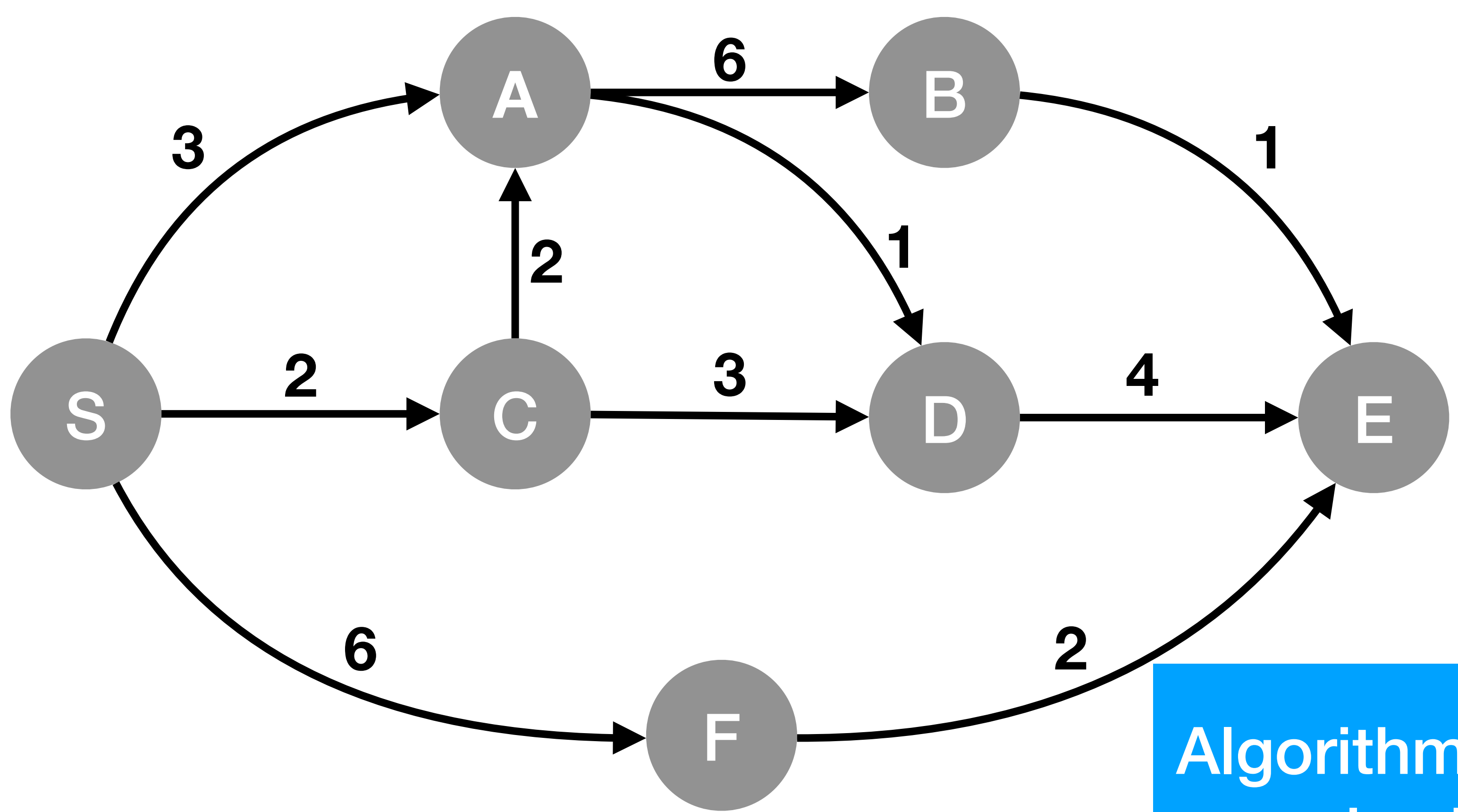



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E, B ]     Unexplored = [ ]

# Dijkstra's algorithm

- For the current node, examine its unexplored neighbors
- Current node → B; unexplored neighbors → {}
- Add the current node to the list of *settled* nodes

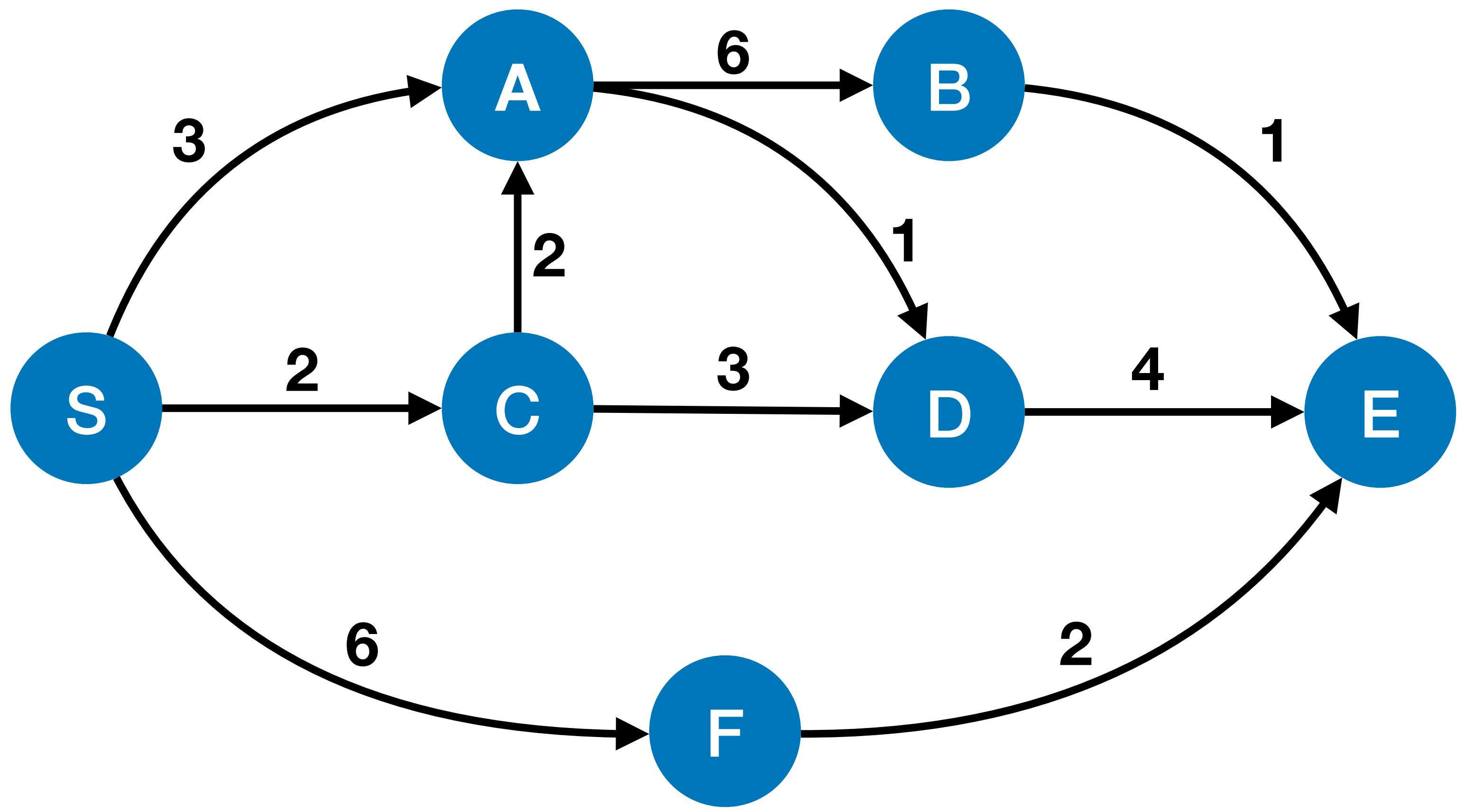



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Algorithm terminates when all nodes have been settled.

Settled = [ S, C, A, D, F, E, B ]   Unexplored = [ ]

# Dijkstra's algorithm

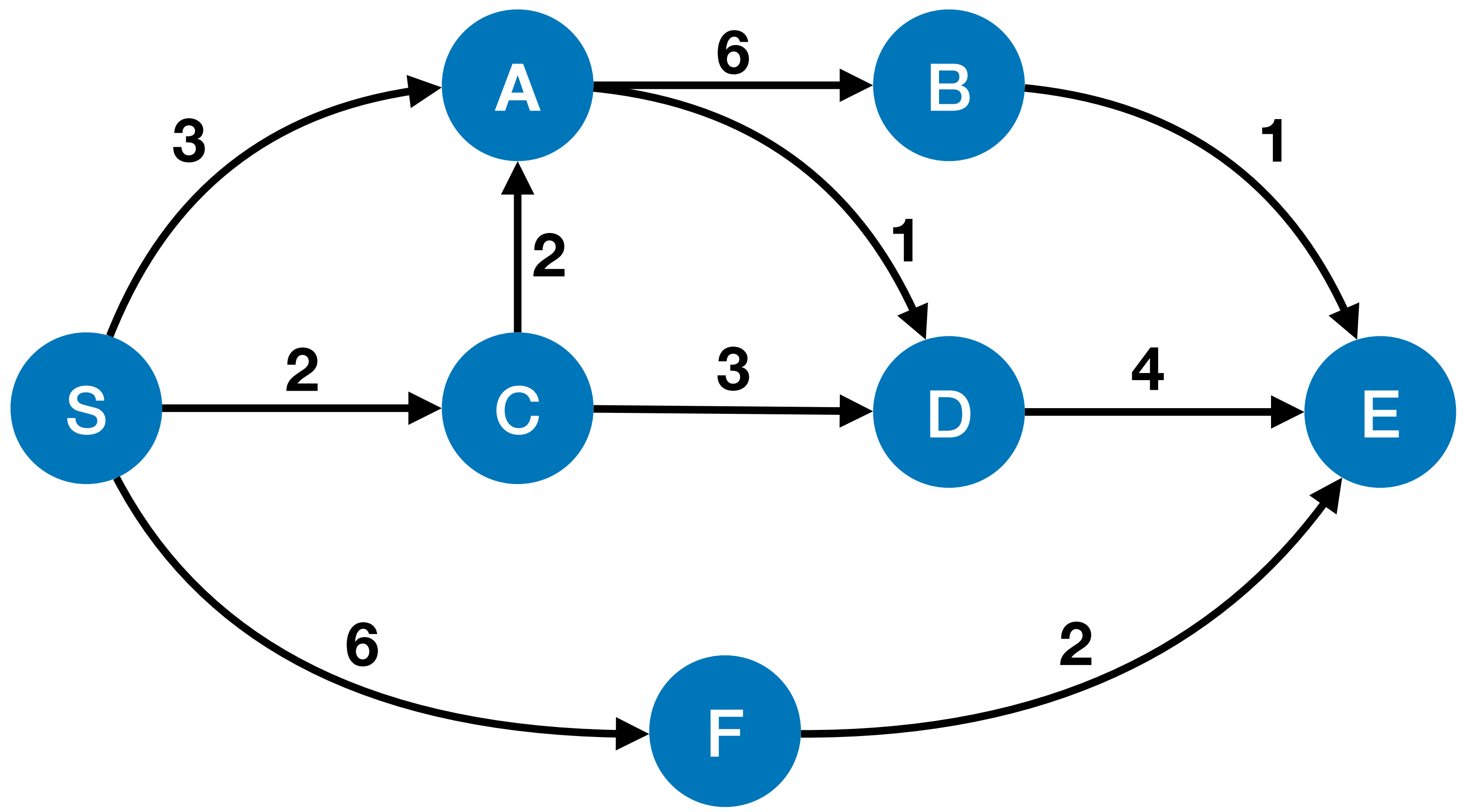



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E, B ]    Unexplored = [ ]

# Dijkstra's algorithm

- We have the distance from source node S to ***every*** other node

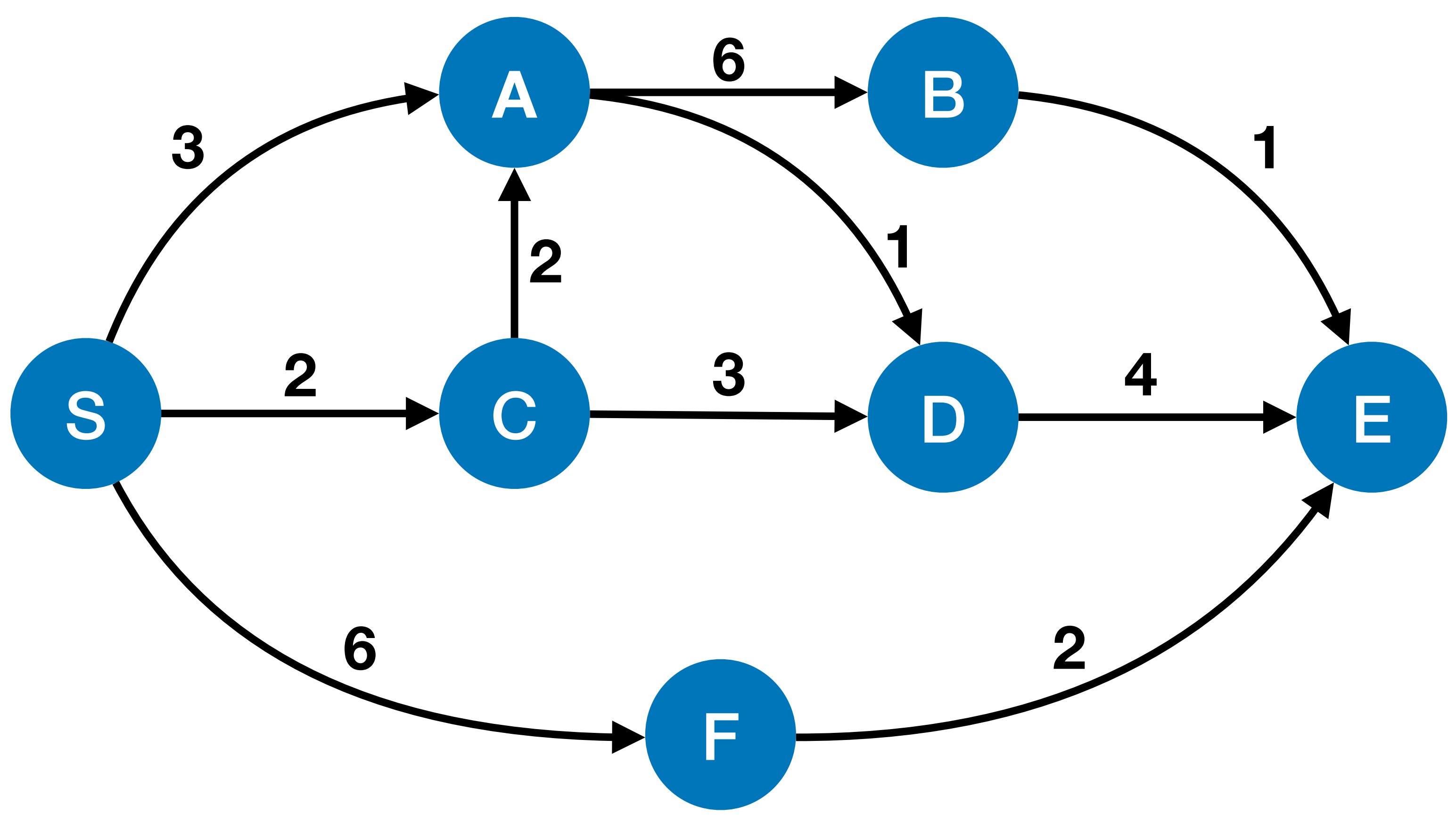



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E, B ]    Unexplored = [ ]

# Dijkstra's algorithm

- We have the distance from source node S to ***every*** other node
- We also have the path which achieves this distance!

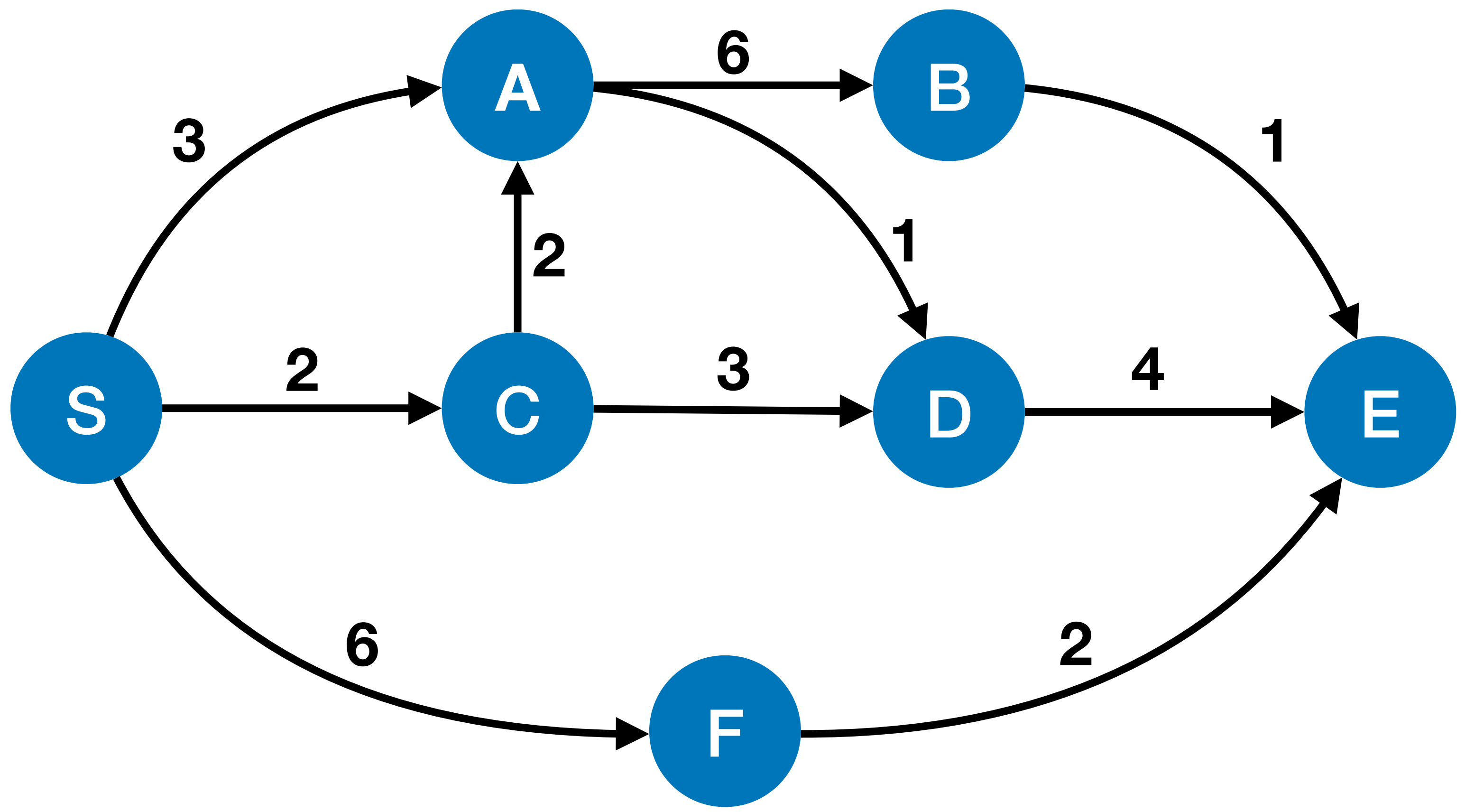



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ S, C, A, D, F, E, B ]    Unexplored = [ ]

# Dijkstra's pseudocode

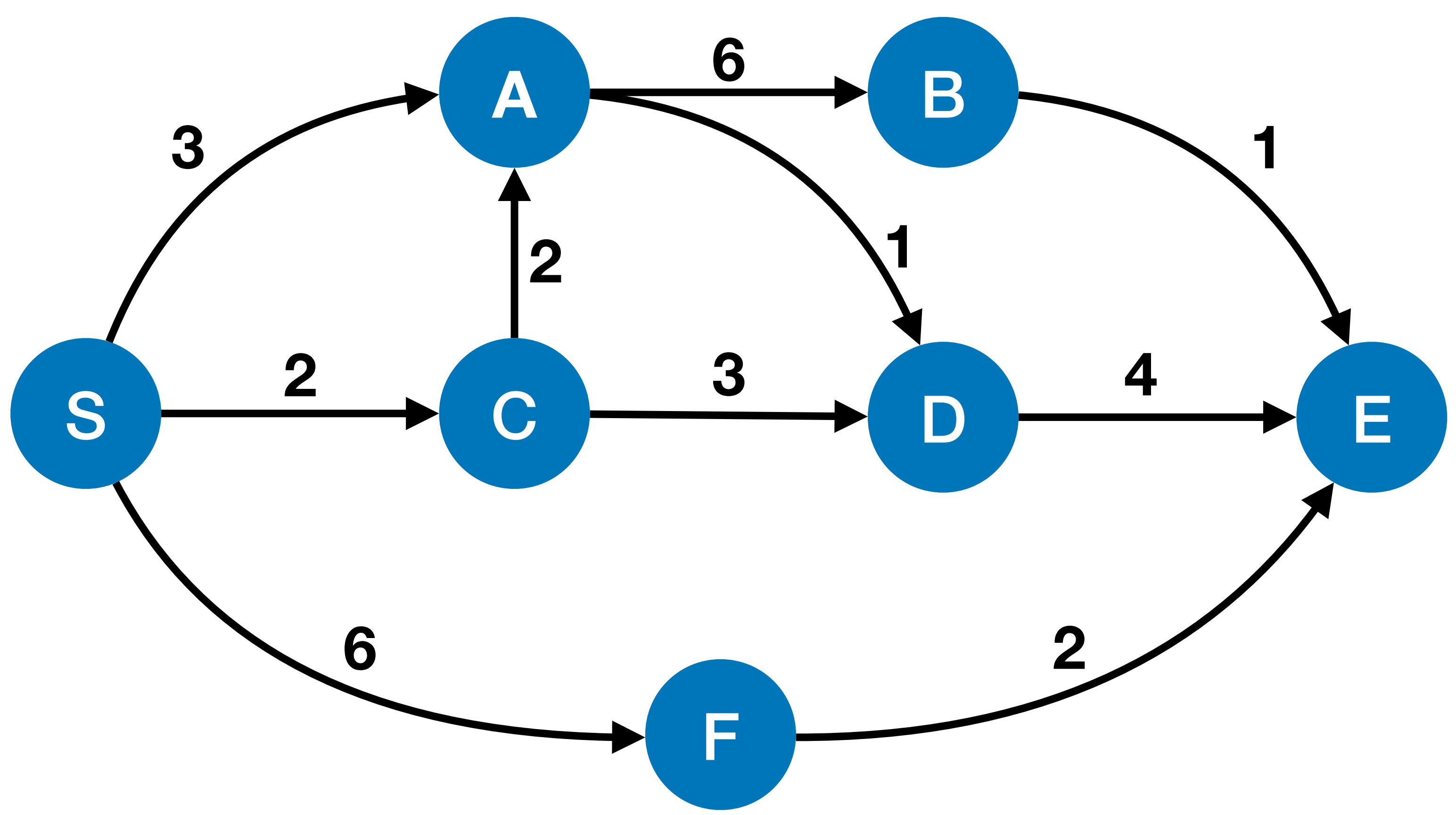
Let the graph be $G = (V, E, w)$. Denote:




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

# Dijkstra's pseudocode

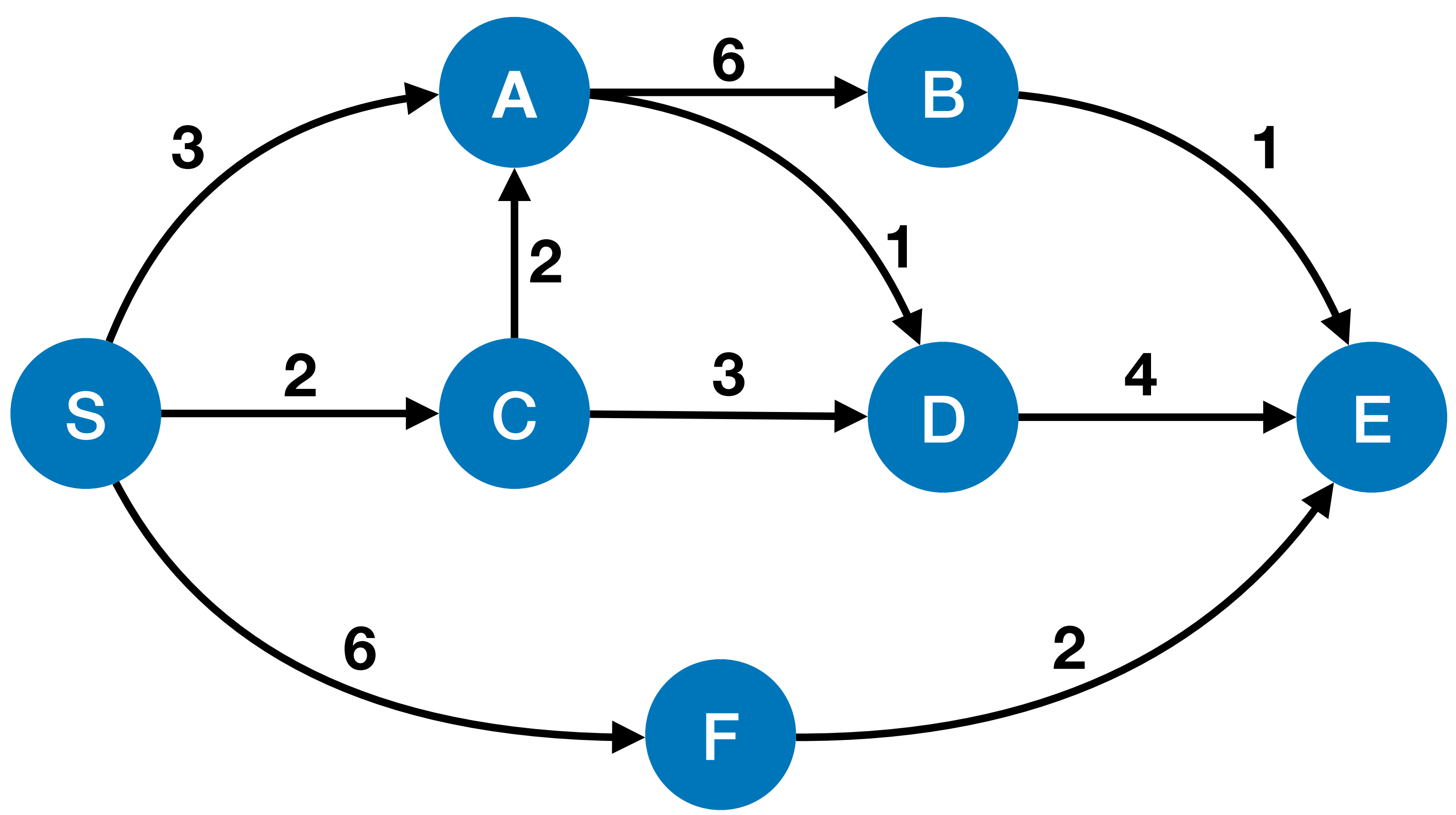
Let the graph be $G = (V, E, w)$. Denote: Source vertex with $s$.




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

# Dijkstra's pseudocode

Let the graph be $G = (V, E, w)$. Denote: Source vertex with $s$.

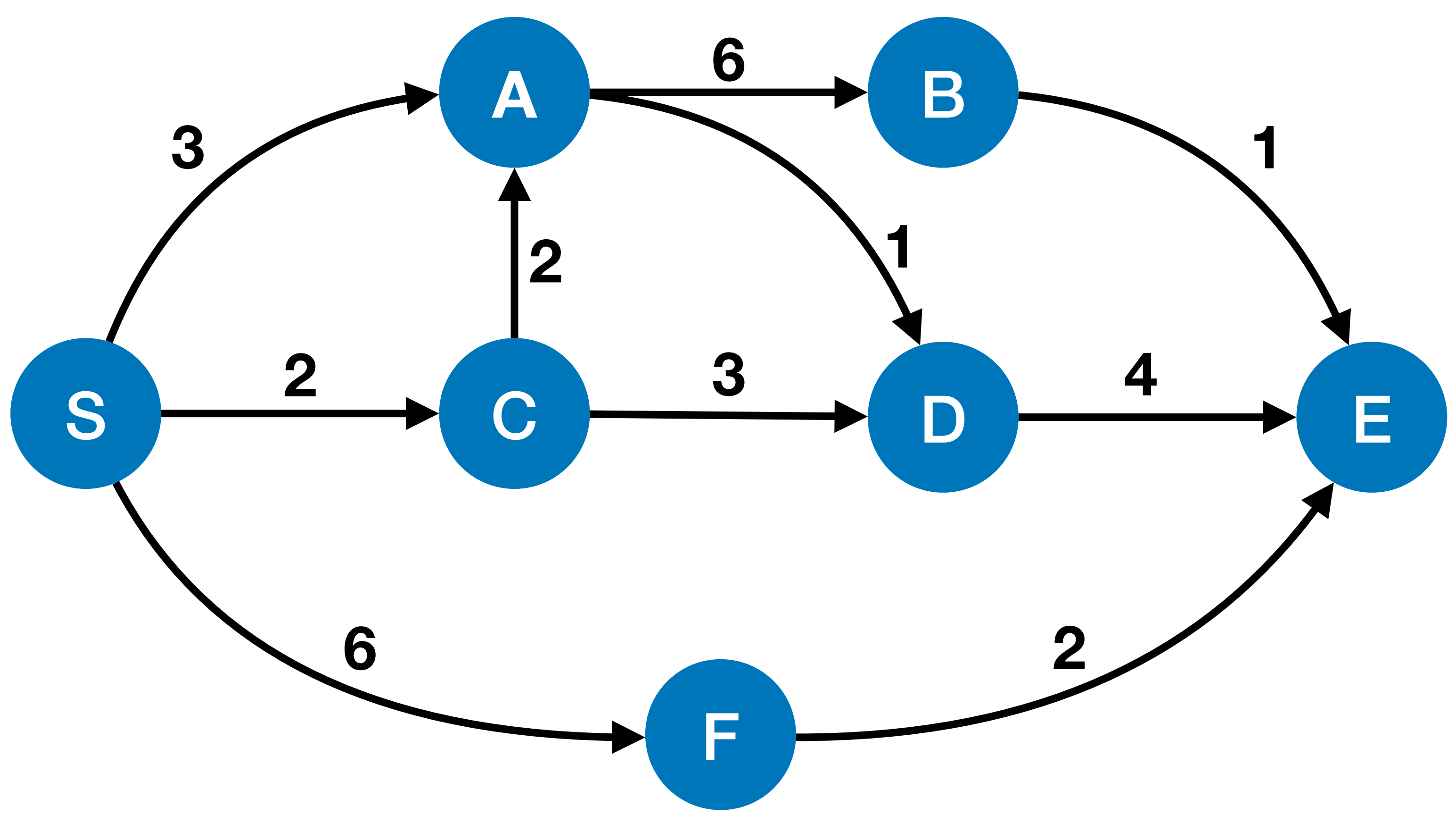



| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Distance estimate with $d(v)$

# Dijkstra's pseudocode

Let the graph be $G = (V, E, w)$. Denote: Source vertex with $s$.




| Node | Distance estimate | Previous node |
|---|---|---|
| S | 0 | |
| A | 3 | S |
| C | 2 | S |
| F | 6 | S |
| D | 4 | A |
| B | 9 | A |
| E | 8 | D or F |

Settled = [ … ]  Settled vertices with $X$  Distance estimate with $d(v)$

# Dijkstra's pseudocode

**Dijkstra(G, s)**

# Dijkstra's pseudocode

**Dijkstra(G, s)**

**Initialization steps**

# Dijkstra's pseudocode

**Dijkstra(G, s)**

**Initialization steps**

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$

# Dijkstra's pseudocode

**Dijkstra(G, s)**

**Initialization steps**

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$
- Set $d(s) = 0$, $X = \{\}$

# Dijkstra's pseudocode

**Dijkstra(G, s)**

**Initialization steps**

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$
- Set $d(s) = 0$, $X = \{\}$

**Iterative steps**

# Dijkstra's pseudocode

**Dijkstra(G, s)**

**Initialization steps**

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$
- Set $d(s) = 0, X = \{\}$

**Iterative steps**

While $X \neq V$

# Dijkstra's pseudocode

**Dijkstra(G, s)**

## Initialization steps

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$
- Set $d(s) = 0, X = \{\,\}$

## Iterative steps

While $X \neq V$

- Pick $u = \arg\min d(x)$ over $x \notin X$

# Dijkstra's pseudocode

**Dijkstra(G, s)**

**Initialization steps**

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$
- Set $d(s) = 0, X = \{\}$

**Iterative steps**

While $X \neq V$

- Pick $u = \arg\min d(x)$ over $x \notin X$
- $\forall\ (u, v) \in E$ such that $v \notin X$ do **Update(u, v)**

# Dijkstra's pseudocode

**Dijkstra(G, s)**

**Initialization steps**

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$
- Set $d(s) = 0, X = \{\}$

**Iterative steps**

While $X \neq V$

- Pick $u = \arg\min d(x)$ over $x \notin X$
- $\forall \ (u, v) \in E$ such that $v \notin X$ do **Update(u, v)**
- Set $X = X \cup \{u\}$

# Dijkstra's pseudocode

**Dijkstra(G, s)**

### Initialization steps

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$
- Set $d(s) = 0, X = \{\}$

**Update(u,v)**

- If $d(v) > d(u) + w(u, v)$
  - Set $d(v) = d(u) + w(u, v)$

### Iterative steps

While $X \neq V$

- Pick $u = \arg\min d(x)$ over $x \notin X$
- $\forall \ (u, v) \in E$ such that $v \notin X$ do **Update(u, v)**
- Set $X = X \cup \{u\}$

# Dijkstra's pseudocode

**Dijkstra(G, s)**

**Initialization steps**

- $\forall u \in V \backslash \{s\}$ set $d(u) = \infty$
- Set $d(s) = 0, X = \{\}$

**Update(u,v)**

- If $d(v) > d(u) + w(u, v)$
    - Set $d(v) = d(u) + w(u, v)$

**Key Observation**

For each $x \in R, d(x) = \delta(x)$

While $X \neq V$

- Pick $u = \arg\min d(x)$ over $x \notin X$
- $\forall\ (u, v) \in E$ such that $v \notin X$ do **Update(u, v)**
- Set $X = X \cup \{u\}$

# Dijkstra's - proof of validity

**Proof**: By induction on the size of $X$

# Dijkstra's - proof of validity

**Proof**: By induction on the size of $X$

- Base case: $|X| = 1$

# Dijkstra's - proof of validity

**Proof**: By induction on the size of $X$

- Base case: $|X| = 1$
  - By initialization, when $|X| = 1$, $X = \{s\}$ and $d(s) = 0 = \delta(s)$

# Dijkstra's - proof of validity

**Proof**: By induction on the size of $X$

- Base case: $|X| = 1$
  - By initialization, when $|X| = 1$, $X = \{s\}$ and $d(s) = 0 = \delta(s)$
- Let $u$ be a vertex just added to $X$ and denote $X = X' \cup \{u\}$.

# Dijkstra's - proof of validity

**Proof**: By induction on the size of $X$

- Base case: $|X| = 1$
  - By initialization, when $|X| = 1$, $X = \{s\}$ and $d(s) = 0 = \delta(s)$
- Let $u$ be a vertex just added to $X$ and denote $X = X' \cup \{u\}$.
  - This implies $u = \operatorname{argmin} d(v)$ over $v \in V \setminus X'$

# Dijkstra's - proof of validity

**Proof**: By induction on the size of $X$

- Base case: $|X| = 1$
  - By initialization, when $|X| = 1$, $X = \{s\}$ and $d(s) = 0 = \delta(s)$
- Let $u$ be a vertex just added to $X$ and denote $X = X' \cup \{u\}$.
  - This implies $u = \text{argmin}\, d(v)$ over $v \in V \backslash X'$
- Inductive hypothesis: $\forall x \in X', d(x) = \delta(x)$

# Dijkstra's - proof of validity

**Proof**: By induction on the size of $X$

- Base case: $|X| = 1$
  - By initialization, when $|X| = 1$, $X = \{s\}$ and $d(s) = 0 = \delta(s)$
- Let $u$ be a vertex just added to $X$ and denote $X = X' \cup \{u\}$.
  - This implies $u = \text{argmin}\, d(v)$ over $v \in V \backslash X'$
- Inductive hypothesis: $\forall x \in X', d(x) = \delta(x)$
- Need to show: $d(u) = \delta(u)$

Have $u = \operatorname{argmin} d(v)$ over $v \in V \backslash X'$. Need to show: $d(u) = \delta(u)$.

# Dijkstra's - proof of validity

**Proof**:

Have $u = \operatorname{argmin} d(v)$ over $v \in V \setminus X'$. Need to show: $d(u) = \delta(u)$.

# Dijkstra's - proof of validity

**Proof**:

Suppose $\exists$ a path $Q : s \to u$
such that,

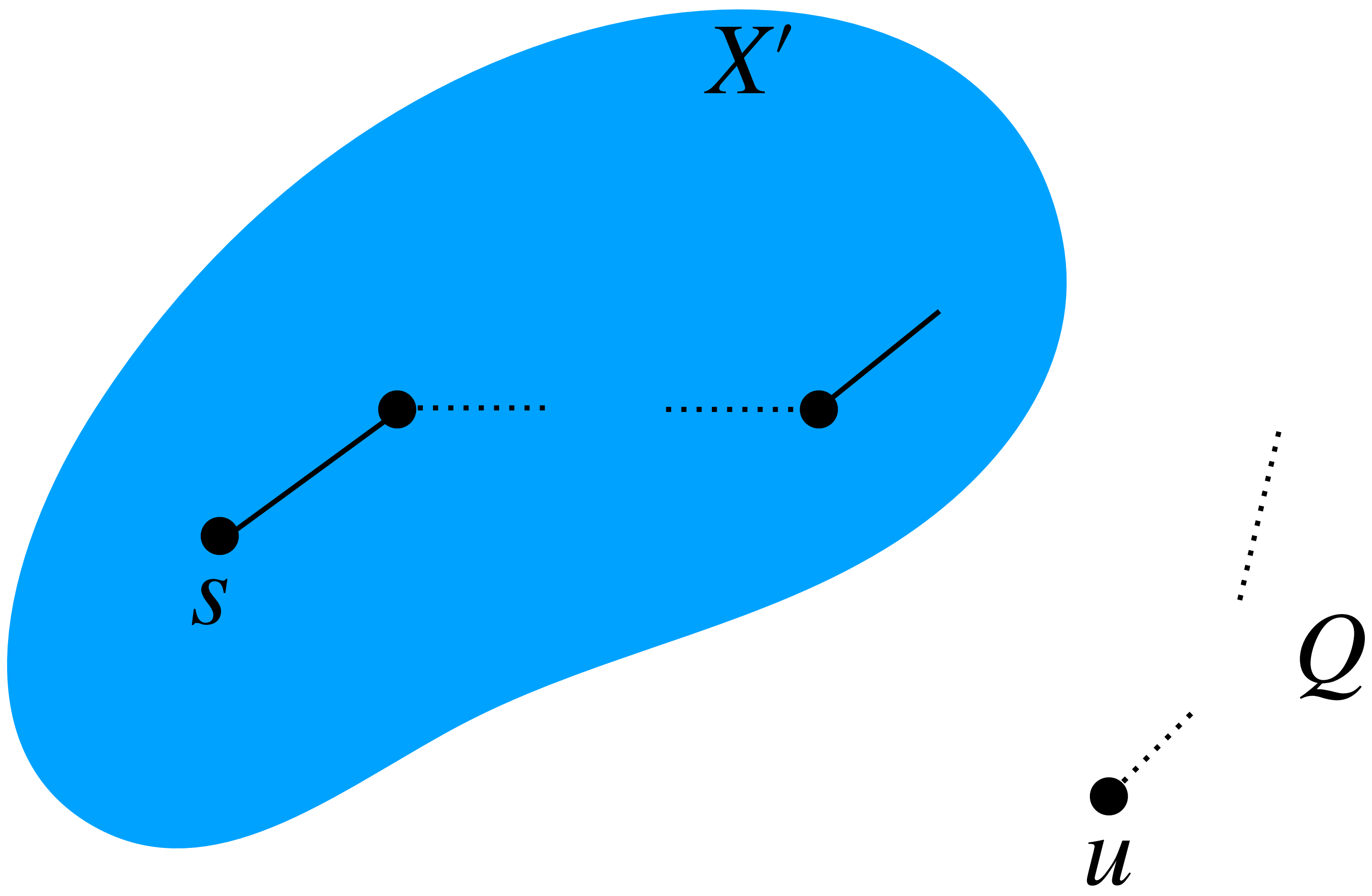
Have $u = \text{argmin}\, d(v)$ over $v \in V \backslash X'$. Need to show: $d(u) = \delta(u)$.

# Dijkstra's - proof of validity

**Proof**:

Suppose ∃ a path $Q : s \to u$ such that,

$$\delta(u) = l(Q) < d(u)$$



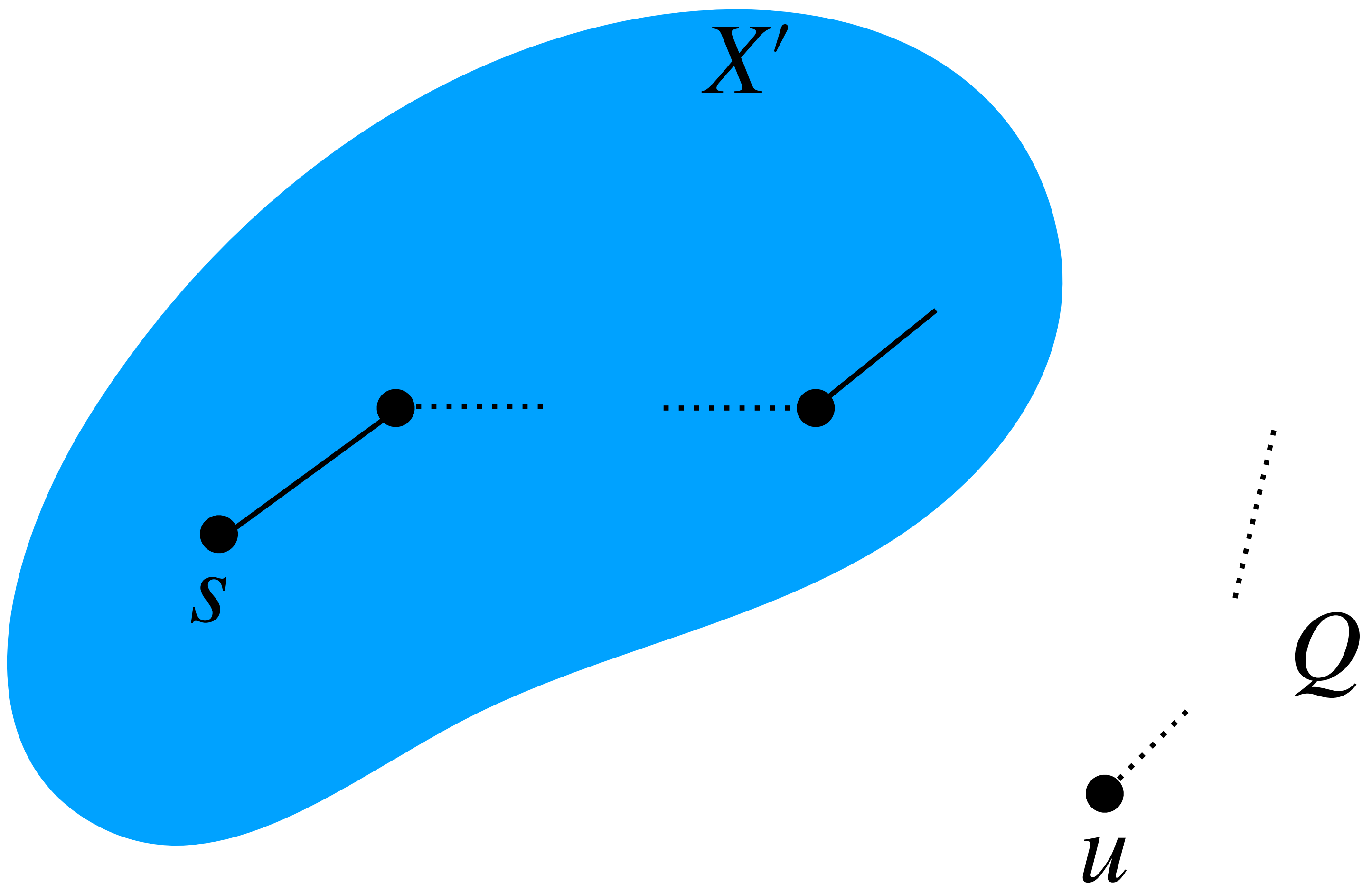
# Dijkstra's - proof of validity

**Proof**:

Suppose $\exists$ a path $Q : s \to u$ such that,

$$\delta(u) = l(Q) < d(u)$$

Then $Q$ must leave $X'$ to get to $u$.



# Dijkstra's - proof of validity

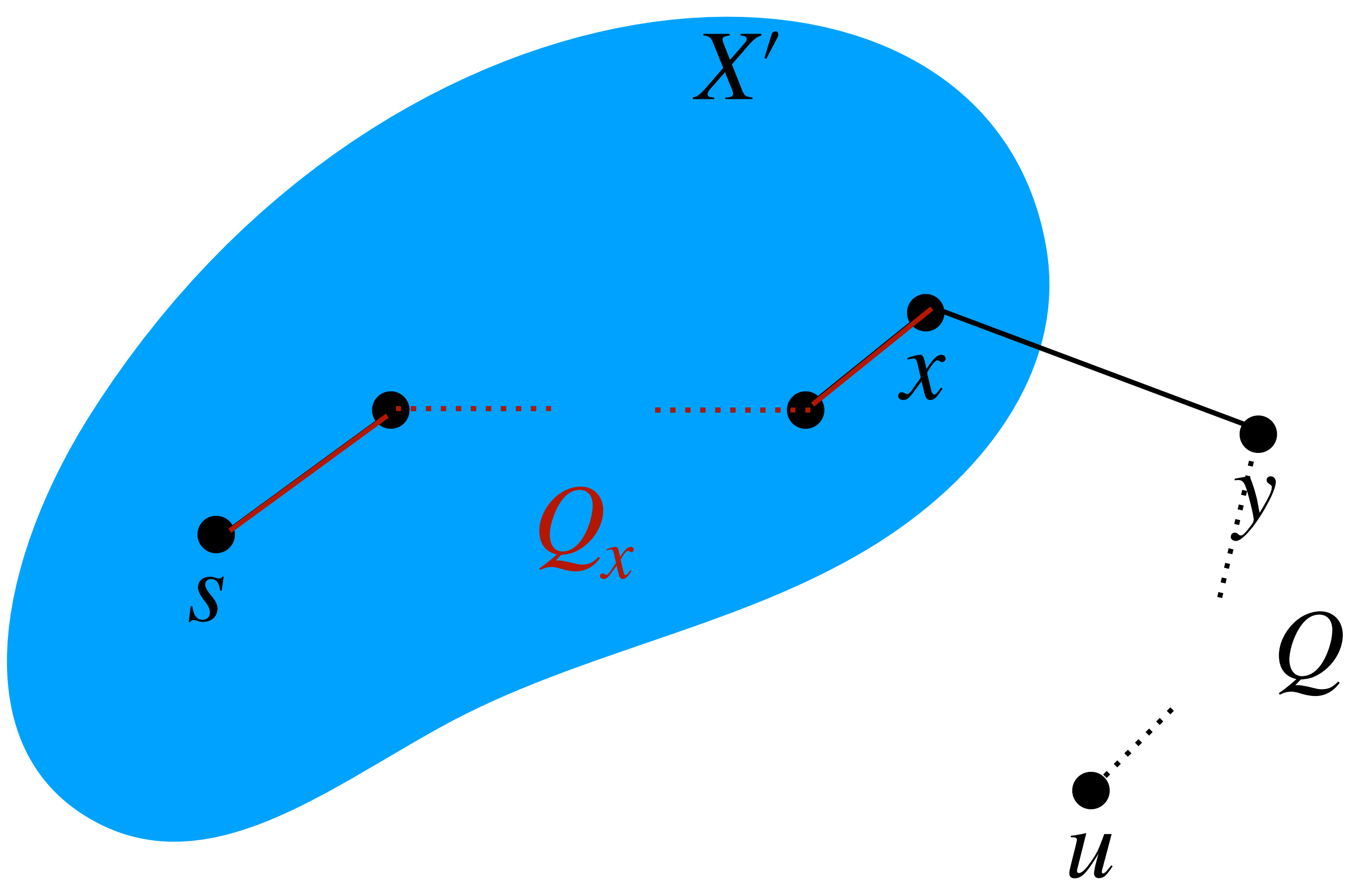
**Proof**:

Suppose $\exists$ a path $Q : s \to u$ such that,

$$\delta(u) = l(Q) < d(u)$$

Then $Q$ must leave $X'$ to get to $u$.

Let $x$-$y$ be the edge by which $Q$ leaves $X'$ the first time and $Q_x$ the subpath of $Q$ until $x$.



Have $u = \text{argmin}\, d(v)$ over $v \in V \backslash X'$. Need to show: $d(u) = \delta(u)$.

# Dijkstra's - proof of validity

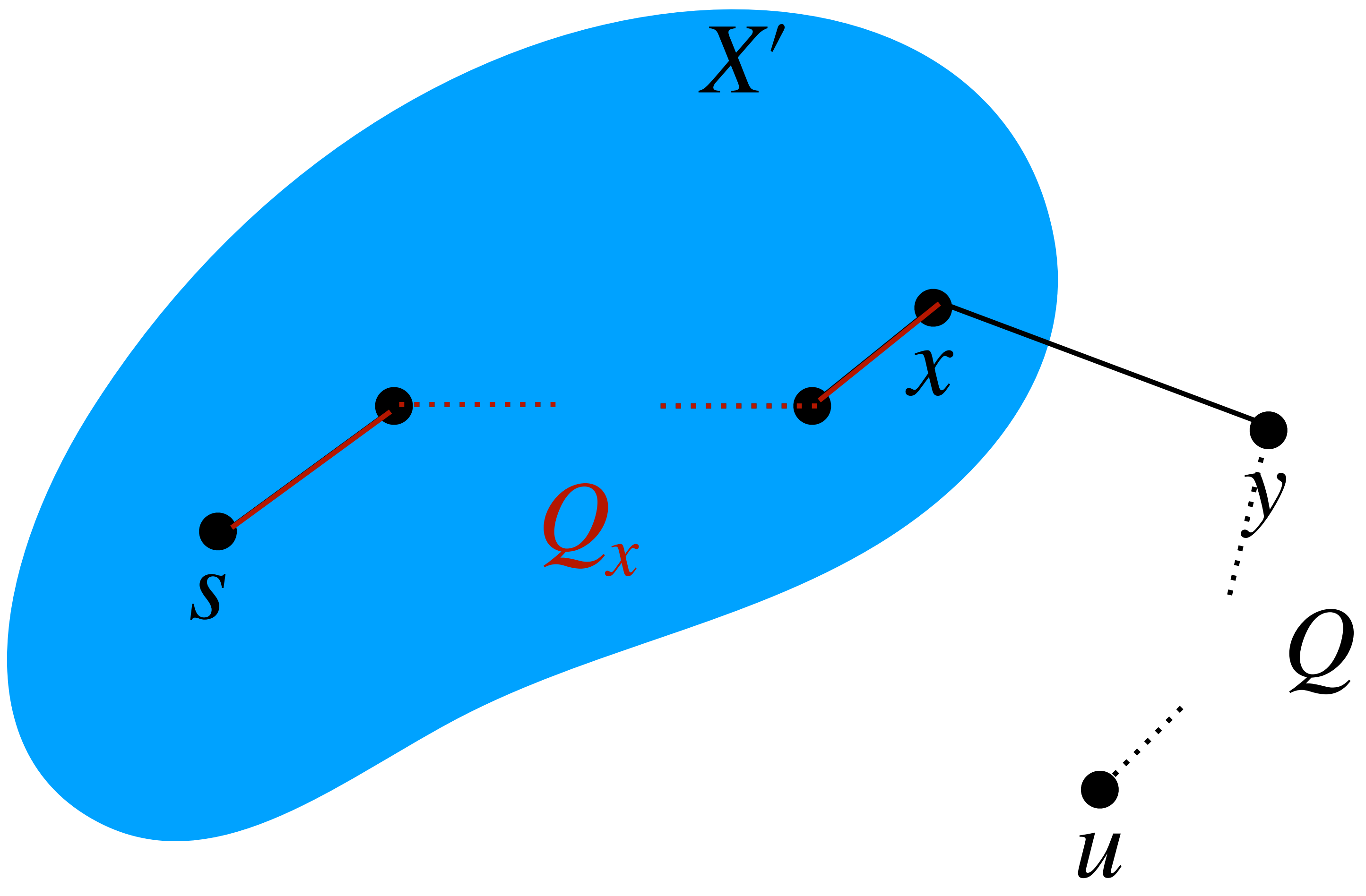
**Proof**:

Suppose $\exists$ a path $Q : s \to u$ such that,

$$\delta(u) = l(Q) < d(u)$$

Then $Q$ must leave $X'$ to get to $u$.

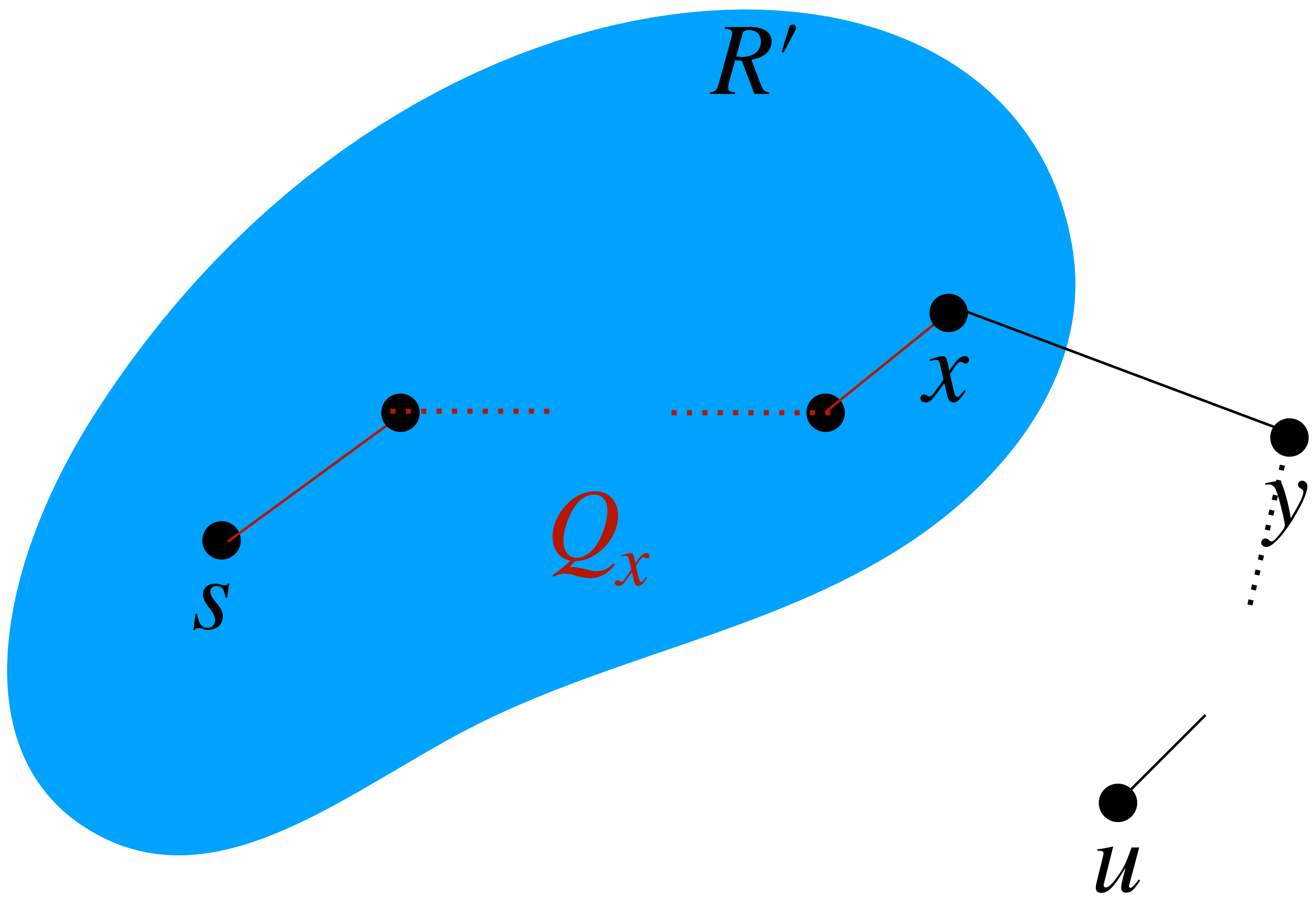
Let $x$-$y$ be the edge by which $Q$ leaves $X'$ the first time and $Q_x$ the subpath of $Q$ until $x$.




$$l(Q_x) + w(x, y) \leq l(Q)$$

Have $u = \operatorname{argmin} d(v)$ over $v \in V \backslash R'$. Need to show: $d(u) = \delta(u)$. Assumed $\delta(u) = l(Q) < d(u)$.

# Dijkstra's - proof of validity



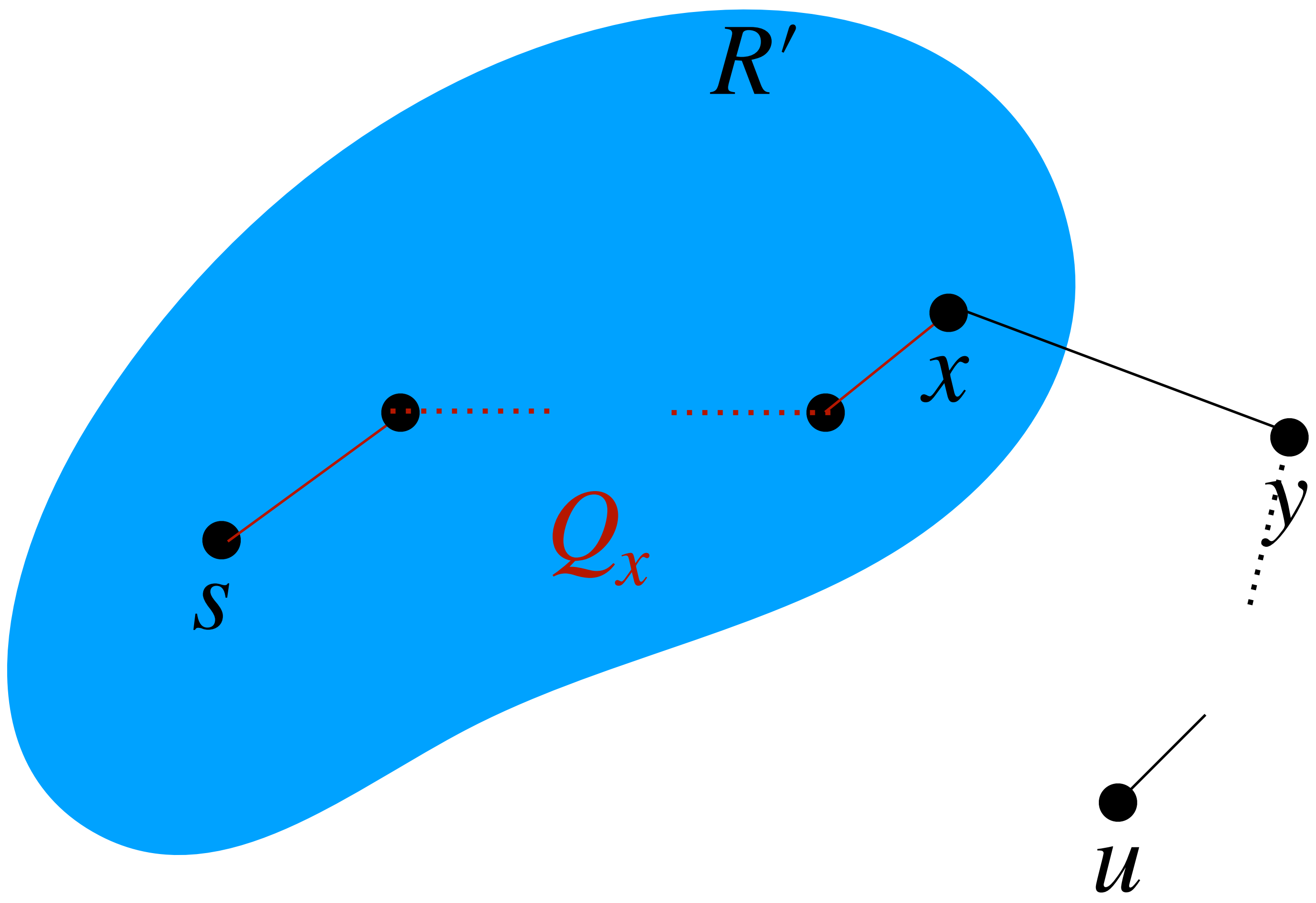

$$l(Q_x) + w(x, y) \leq l(Q)$$

Have $u = \arg\min d(v)$ over $v \in V \backslash R'$. Need to show: $d(u) = \delta(u)$. Assumed $\delta(u) = l(Q) < d(u)$.

# Dijkstra's - proof of validity

... by inductive hypothesis

$$d(x) \leq l(Q_x)$$



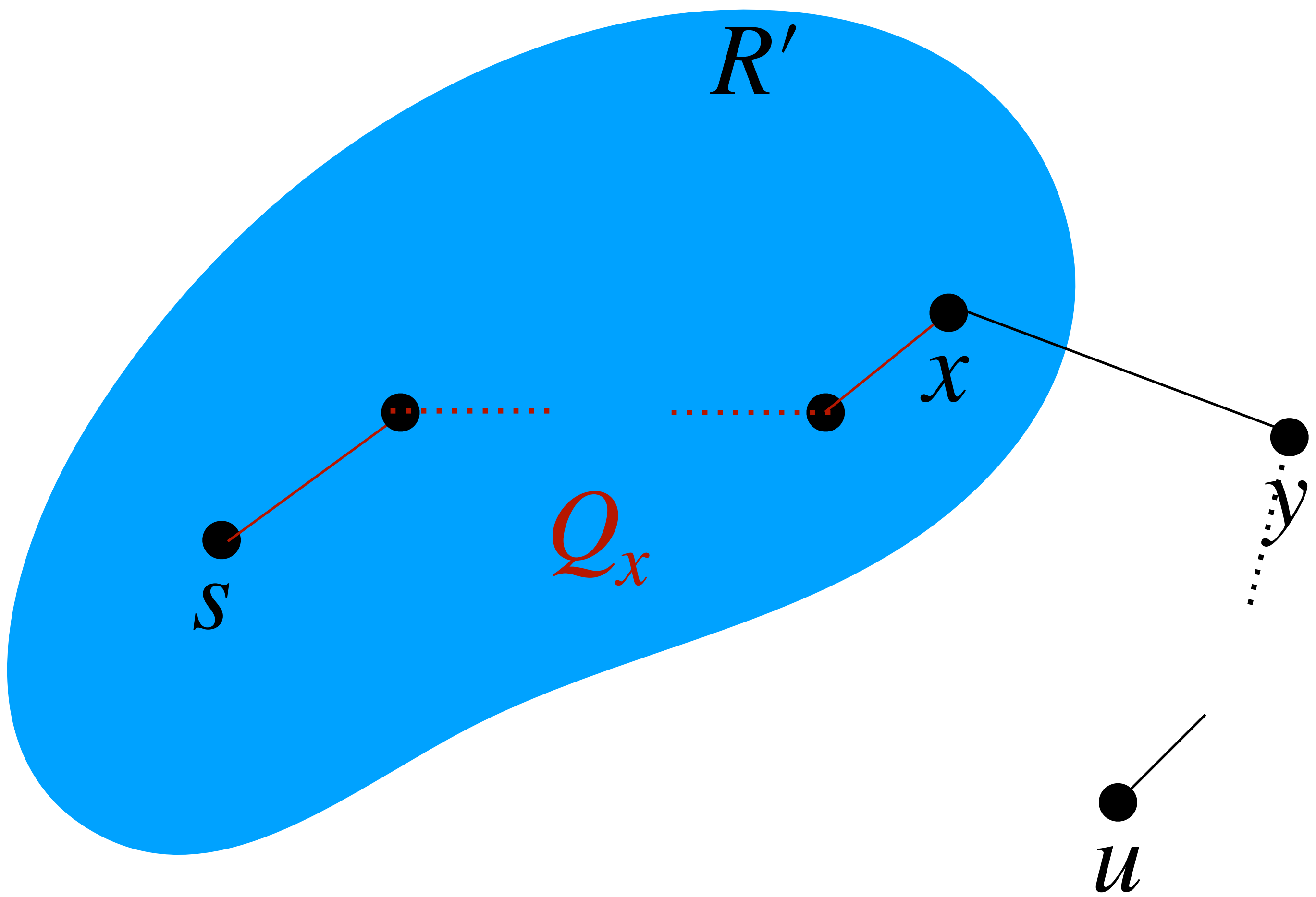

$$l(Q_x) + w(x, y) \leq l(Q)$$

Have $u = \text{argmin}\, d(v)$ over $v \in V \backslash R'$. Need to show: $d(u) = \delta(u)$. Assumed $\delta(u) = l(Q) < d(u)$.

# Dijkstra's - proof of validity

… by inductive hypothesis

$$d(x) \leq l(Q_x)$$




$$l(Q_x) + w(x, y) \leq l(Q)$$

$$d(x) + w(x, y) \leq l(Q)$$

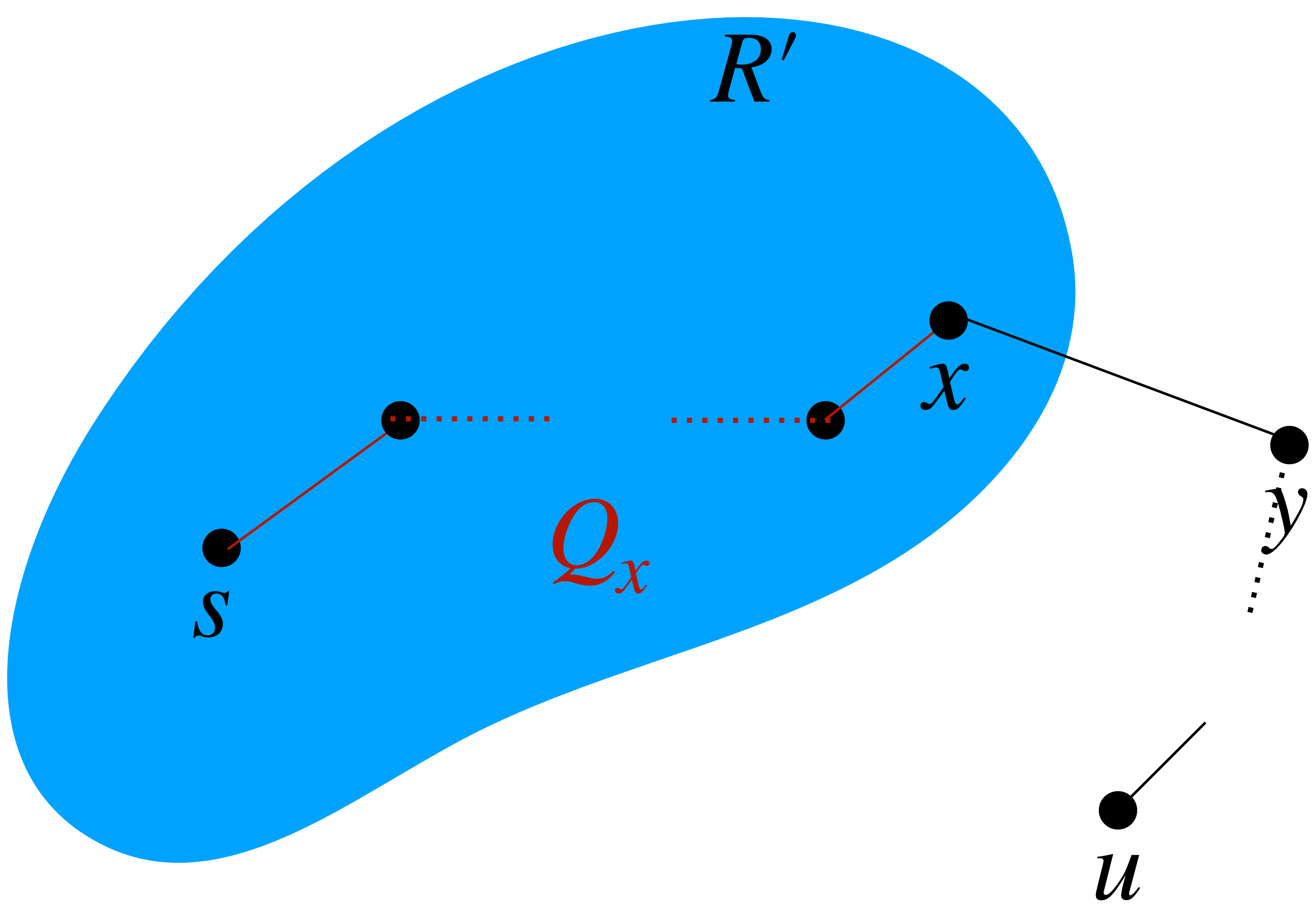
Have $u = \operatorname{argmin} d(v)$ over $v \in V \setminus R'$. Need to show: $d(u) = \delta(u)$. Assumed $\delta(u) = l(Q) < d(u)$.

# Dijkstra's - proof of validity

… by inductive hypothesis

$$d(x) \leq l(Q_x)$$

Since $(x, y) \in E$ and $x \in X'$




$$l(Q_x) + w(x, y) \leq l(Q)$$

$$d(x) + w(x, y) \leq l(Q)$$

Have $u = \text{argmin}\, d(v)$ over $v \in V \backslash R'$. Need to show: $d(u) = \delta(u)$. Assumed $\delta(u) = l(Q) < d(u)$.

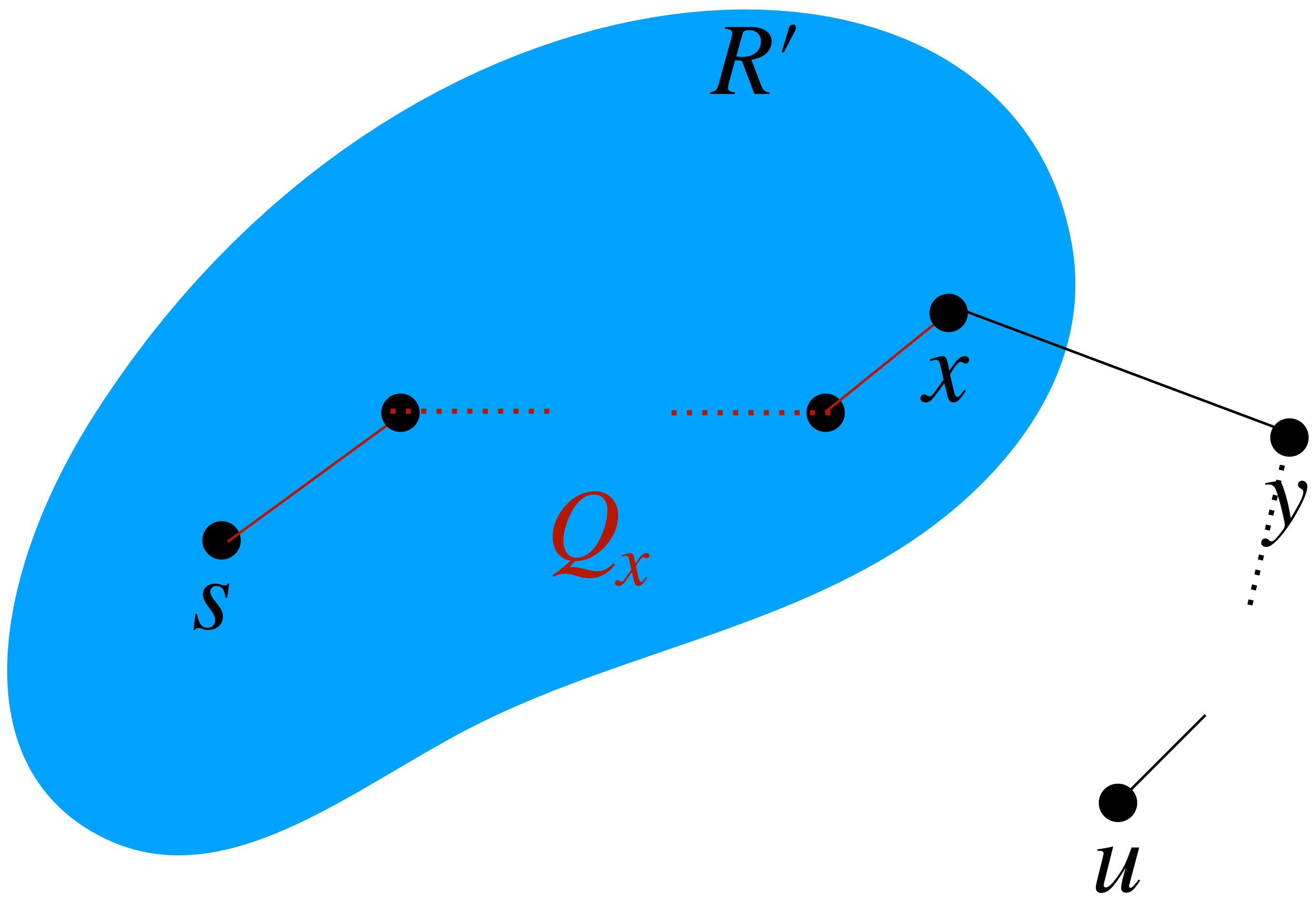
# Dijkstra's - proof of validity

… by inductive hypothesis

$$d(x) \leq l(Q_x)$$

Since $(x, y) \in E$ and $x \in X'$

$$d(y) \leq d(x) + w(x, y)$$




$$l(Q_x) + w(x, y) \leq l(Q)$$

$$d(x) + w(x, y) \leq l(Q)$$

Have $u = \operatorname{argmin} d(v)$ over $v \in V \backslash R'$. Need to show: $d(u) = \delta(u)$. Assumed $\delta(u) = l(Q) < d(u)$.

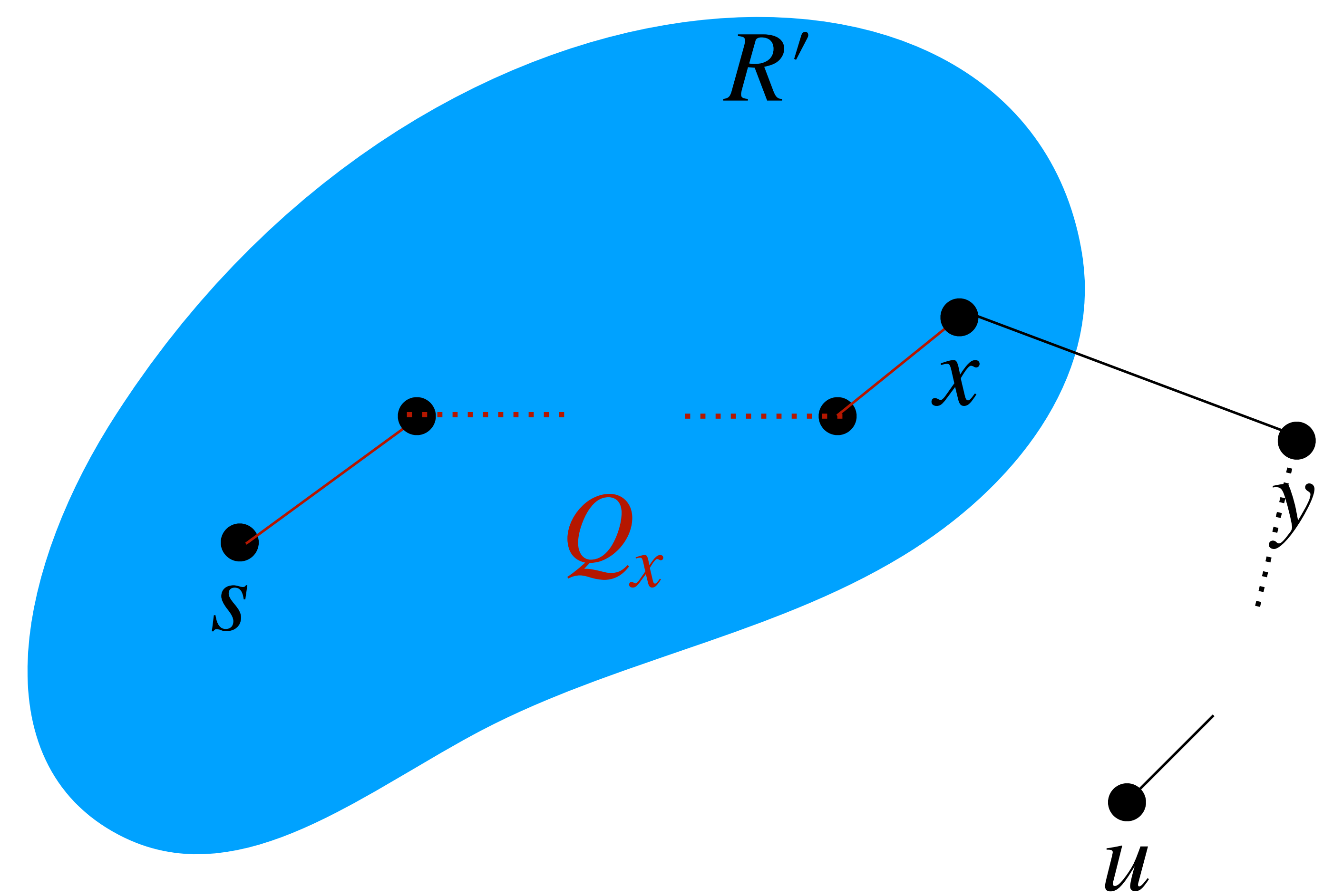
# Dijkstra's - proof of validity

… by inductive hypothesis

$$d(x) \leq l(Q_x)$$

Since $(x, y) \in E$ and $x \in X'$

$$d(y) \leq d(x) + w(x, y)$$




$$l(Q_x) + w(x, y) \leq l(Q)$$

$$d(x) + w(x, y) \leq l(Q)$$

$$d(y) \leq l(Q)$$

# Dijkstra's - proof of validity

… by inductive hypothesis

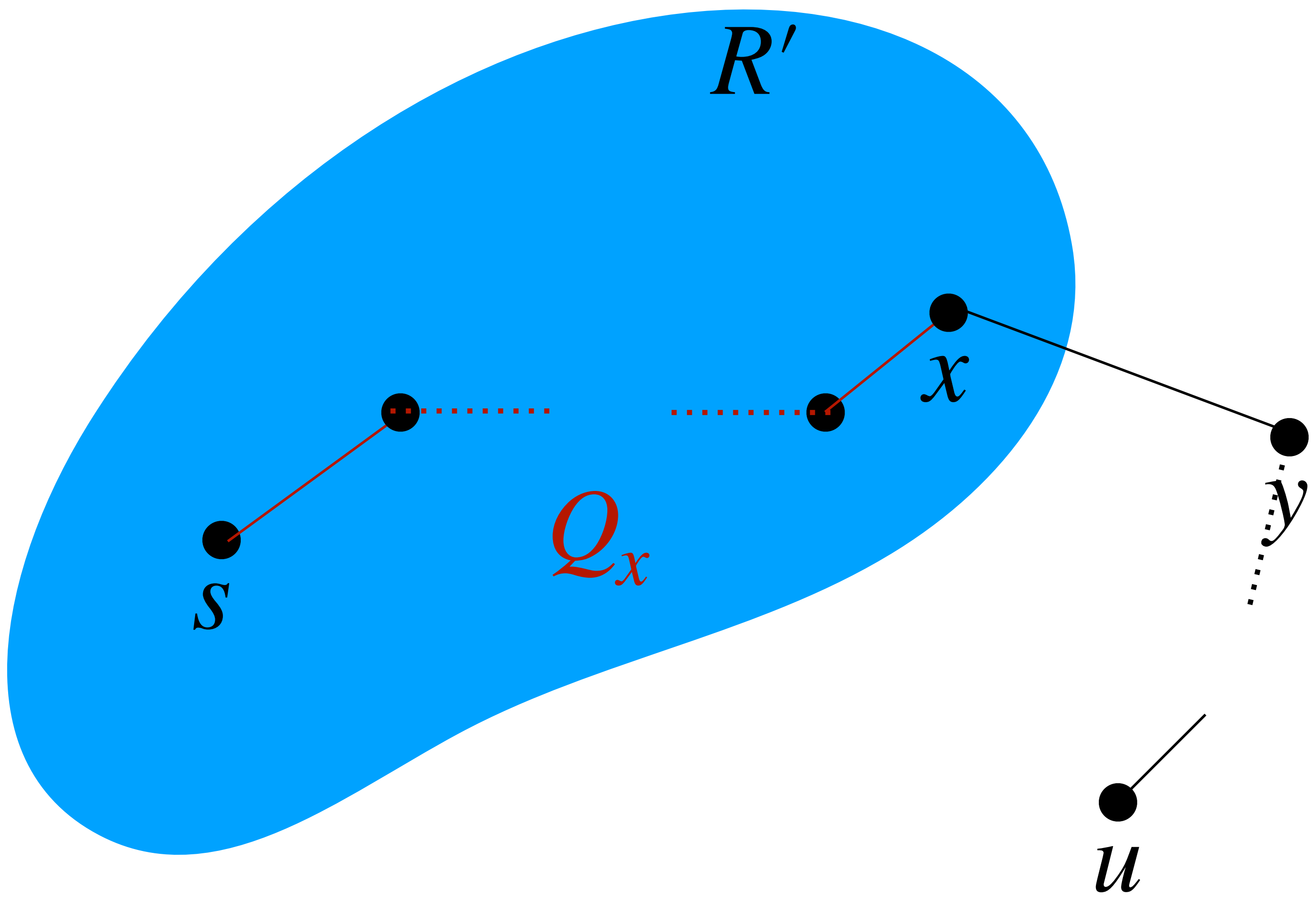
$$d(x) \leq l(Q_x)$$

Since $(x, y) \in E$ and $x \in X'$

$$d(y) \leq d(x) + w(x, y)$$

But $u$ was picked via $\arg\min d(v)$ over vertices not in $X'$

$$d(u) \leq d(y)$$




$l(Q_x) + w(x, y) \leq l(Q)$

$d(x) + w(x, y) \leq l(Q)$

$d(y) \leq l(Q)$

Have $u = \text{argmin}\, d(v)$ over $v \in V \backslash R'$. Need to show: $d(u) = \delta(u)$. Assumed $\delta(u) = l(Q) < d(u)$.

# Dijkstra's - proof of validity

… by inductive hypothesis

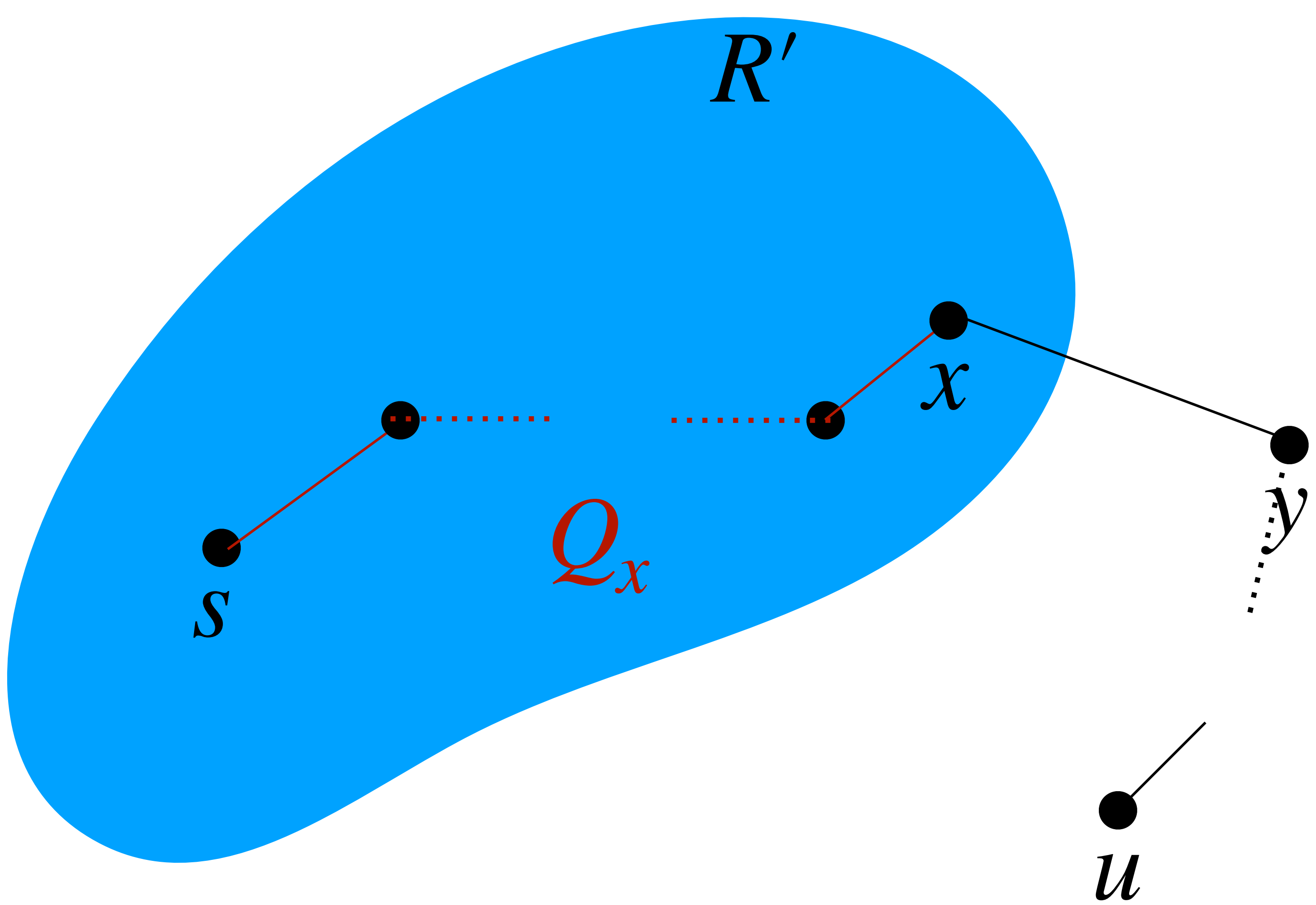
$$d(x) \leq l(Q_x)$$

Since $(x, y) \in E$ and $x \in X'$

$$d(y) \leq d(x) + w(x, y)$$

But $u$ was picked via $\arg\min d(v)$ over vertices not in $X'$

$$d(u) \leq d(y)$$




$$l(Q_x) + w(x, y) \leq l(Q)$$

$$d(x) + w(x, y) \leq l(Q)$$

$$d(y) \leq l(Q)$$

$$d(u) \leq l(Q)$$

Have $u = \text{argmin}\, d(v)$ over $v \in V \backslash R'$. Need to show: $d(u) = \delta(u)$. Assumed $\delta(u) = l(Q) < d(u)$.

# Dijkstra's - proof of validity

… by inductive hypothesis

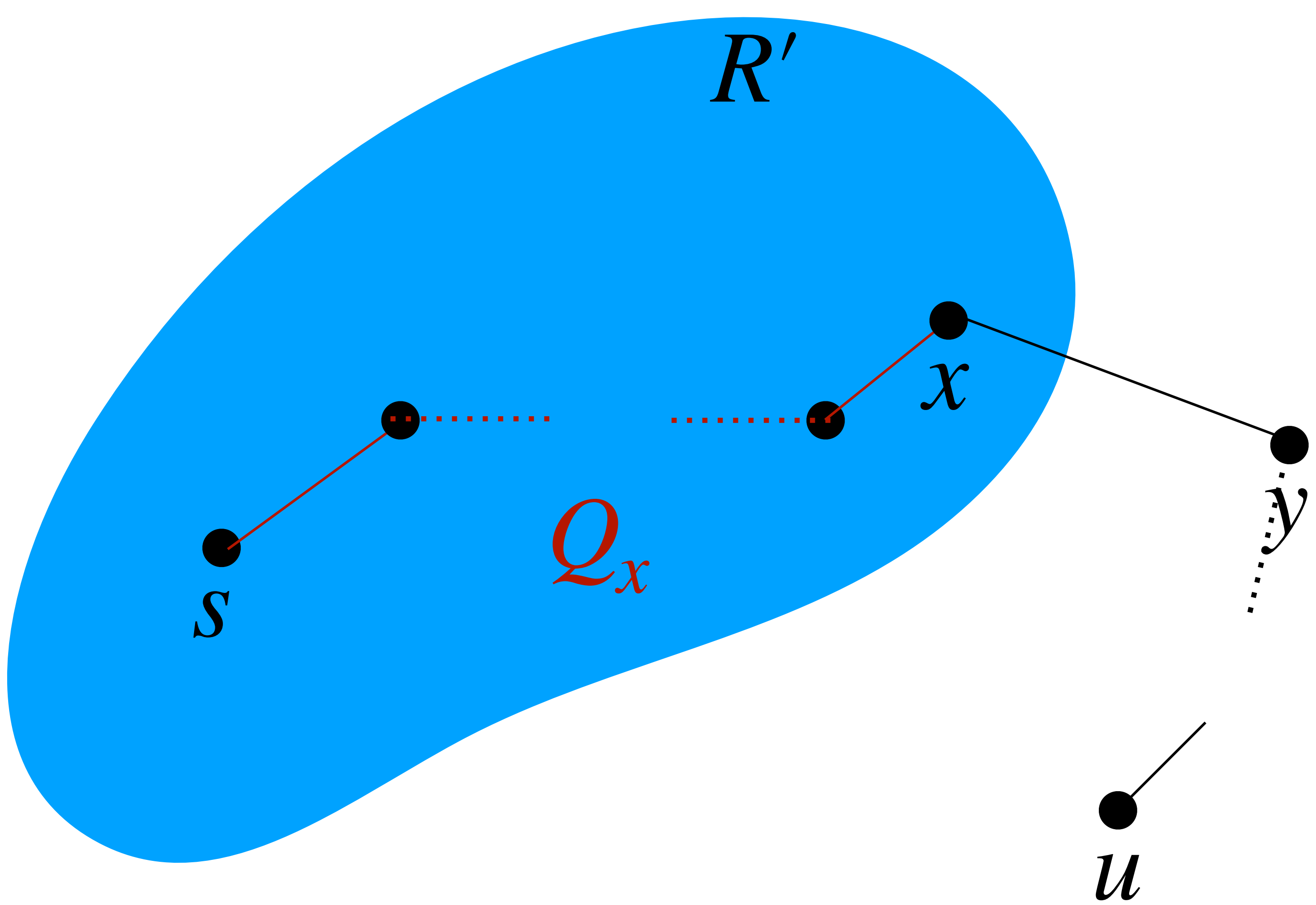
$$d(x) \le l(Q_x)$$

Since $(x, y) \in E$ and $x \in X'$

$$d(y) \le d(x) + w(x, y)$$

But $u$ was picked via $\arg\min d(v)$ over vertices not in $X'$

$$d(u) \le d(y)$$




$$l(Q_x) + w(x, y) \le l(Q)$$

$$d(x) + w(x, y) \le l(Q)$$

$$d(y) \le l(Q)$$

$$d(u) \le l(Q)$$

**Contradicts our assumption!**

# Improved algorithm

- Main work is to compute the $d'(s,u)$ values in each iteration
- $d'(s,u)$ changes from iteration $i$ to $i+1$ only because of the node $v$ that is added to $X$ in iteration $i$ (previous step)

```
Initialize for each node v: dist(s,v) = d'(s,v) = ∞
Initialize X = ∅, d'(s,s) = 0
for i = 1 to |V| do
    // X contains the i−1 closest nodes to s,
    // and the values of d'(s,u) are current
        Let v be node realizing d'(s,v) = min_{u∈V\X} d'(s,u)
        dist(s,v) = d'(s,v)
        X = X ∪ {v}
        Update d'(s,u) for each u in V−X as follows:
            d'(s,u) = min(d'(s,u), dist(s,v) + l(v,u))
```

# Improved algorithm

Running time: $O(m+n^2)$ time.

- $n$ outer iterations and in each iteration following steps take place:

# Improved algorithm

Running time: $O(m+n^2)$ time.

- $n$ outer iterations and in each iteration following steps take place:
  - updating $d'(s, u)$ after $v$ is added takes $O(\deg(v))$ time so ***total*** work is $O(m)$ since a node enters $X$ at most once

# Improved algorithm

Running time: $O(m+n^2)$ time.

- $n$ outer iterations and in each iteration following steps take place:
  - updating $d'(s,u)$ after $v$ is added takes $O(\deg(v))$ time so ***total*** work is $O(m)$ since a node enters $X$ at most once
  - Finding $v$ from $d'(s,u)$ values takes $O(n)$ time

# Dijkstra's Algorithm

- Eliminate $d'(s, u)$ and let dist$(s, u)$ maintain it
- Update dist values after adding $v$ by scanning edges out of $v$

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V\X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + l(v, u))
```

Can use Priority Queues to maintain dist values for even faster running time

# Dijkstra's Algorithm

- Eliminate $d'(s, u)$ and let $\text{dist}(s, u)$ maintain it
- Update dist values after adding $v$ by scanning edges out of $v$

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V\X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + l(v, u))
```

Can use Priority Queues to maintain dist values for even faster running time

- Using heaps and standard priority queues: $O((m + n) \log n)$

# Dijkstra's Algorithm

- Eliminate $d'(s,u)$ and let $\text{dist}(s,u)$ maintain it
- Update dist values after adding $v$ by scanning edges out of $v$

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V\X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + l(v, u))
```

Can use Priority Queues to maintain dist values for even faster running time

- Using heaps and standard priority queues: $O((m+n)\ \log n)$
- Using Fibonacci heaps: $O(m + n\log n)$

# Dijkstra using Priority Queues

## Priority Queues

Data structure to store a set $S$ of $n$ elements where each element $v \in S$ has an associated real/integer key $k(v)$ alongwith that the following operations:

- makePQ: create an empty queue.
- findMin: find the minimum key in $S$.
- extractMin: Remove $v \in S$ with smallest key and return it.
- insert($v, k(v)$): Add new element v with key $k(v)$ to $S$.
- delete($v$): Remove element $v$ from $S$.
- decreaseKey($v, k'(v)$): decrease key of $v$ from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
- meld: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time - decreaseKey is implemented via delete and insert.

# Dijkstra's algorithm using priority queues

```
Q ← makePQ()
insert(Q, (s, 0))
for each node u ≠ s do
    insert(Q, (u, ∞))
X ← ∅
for i = 1 to |V| do
  (v, dist(s, v)) = extractMin(Q)
  X = X ∪ {v}
  for each u in Adj(v) do
        decreaseKey(Q, (u, min(dist(s, u), dist(s, v) + l(v, u))))
```

look at v's neighbour

PQ operations:

- $O(n)$ **insert** operations
- $O(n)$ **extractMin** operations
- $O(m)$ **decreaseKey** operations

# Shortest Path Tree

Dijkstra's alg. finds the shortest path distances from $s$ to $V$.

**Question:** How do we find the paths themselves?

```
Q ← makePQ()
insert(Q, (s, 0))
prev(u) ← null
for each node u ≠ s do
    insert(Q, (u, ∞))
    prev(u) ← null
X ← ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    X = X ∪ {v}
    for each u in Adj(v) do
        if (dist(s, v) + l(v, u) < dist(s, u)) then
            decreaseKey(Q, (u, dist(s, u) + l(v, u)))
            prev(u) = v
```

# Shortest Path Tree

**Lemma**: The edge set $(u, \text{prev}(u))$ is the reverse of a shortest path tree rooted at $s$. For each $u$, the reverse of the path from $u$ to $s$ in the tree is a shortest path from $s$ to $u$.

**Proof Sketch:**

# Shortest Path Tree

**Lemma**: The edge set $(u, \text{prev}(u))$ is the reverse of a shortest path tree rooted at $s$. For each $u$, the reverse of the path from $u$ to $s$ in the tree is a shortest path from $s$ to $u$.

**Proof Sketch:**

- The edge set $\{(u, \text{prev}(u)) \mid u \in V\}$ induces a directed in-tree rooted at $s$ (Why?)

# Shortest Path Tree

**Lemma**: The edge set $(u, \text{prev}(u))$ is the reverse of a shortest path tree rooted at $s$. For each $u$, the reverse of the path from $u$ to $s$ in the tree is a shortest path from $s$ to $u$.

**Proof Sketch:**

- The edge set $\{(u, \text{prev}(u)) \mid u \in V\}$ induces a directed in-tree rooted at $s$ (Why?)
- Use induction on $|X|$ to argue that the obtained tree is a shortest path tree for nodes in $V$.

# Shortest paths *to* s?

Dijkstra's alg. gives shortest paths from $s$ to all nodes in $V$.

How do we find shortest paths from all of $V$ to $s$?

# Shortest paths *to* s?

Dijkstra's alg. gives shortest paths from $s$ to all nodes in $V$.

How do we find shortest paths from all of $V$ to $s$?

- In undirected graphs shortest path from $s$ to $u$ is a shortest path from $u$ to $s$ so there is no need to distinguish.

# Shortest paths *to* s?

Dijkstra's alg. gives shortest paths from $s$ to all nodes in $V$.

How do we find shortest paths from all of $V$ to $s$?

- In undirected graphs shortest path from $s$ to $u$ is a shortest path from $u$ to $s$ so there is no need to distinguish.
- In directed graphs, use Dijkstra's algorithm in $G^{rev}$!