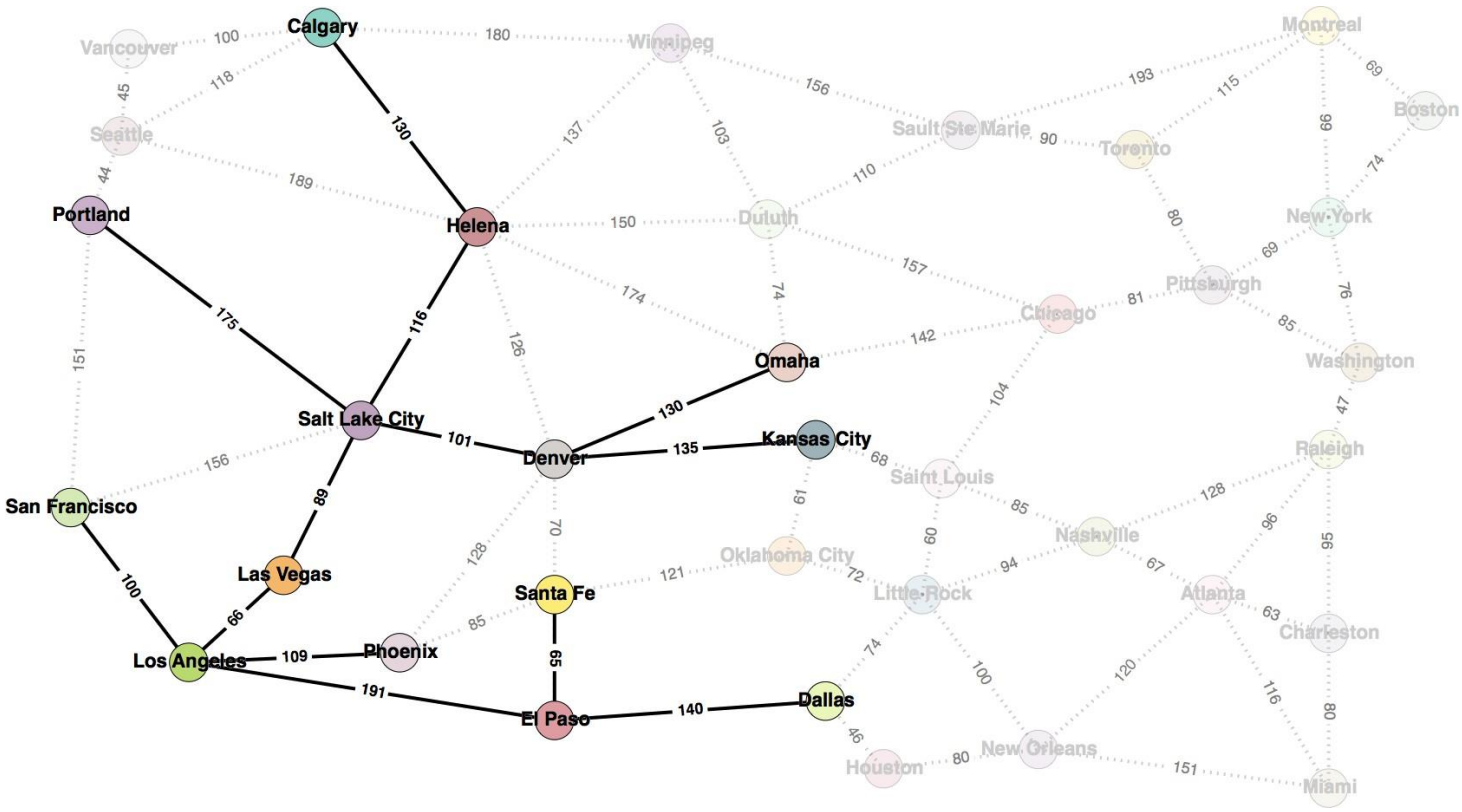
# Artificial Intelligence

## Search Agents

## Uninformed search



# Uninformed search

**Use no domain knowledge!**

**Strategies:**

1. Breadth-first search (BFS): Expand shallowest node

# Uninformed search

**Use no domain knowledge!**

**Strategies:**

1. Breadth-first search (BFS): Expand shallowest node
2. Depth-first search (DFS): Expand deepest node

# Uninformed search

**Use no domain knowledge!**

**Strategies:**

1. Breadth-first search (BFS): Expand shallowest node
2. Depth-first search (DFS): Expand deepest node
3. Depth-limited search (DLS): Depth first with depth limit

# Uninformed search

**Use no domain knowledge!**

**Strategies:**

1. Breadth-first search (BFS): Expand shallowest node
2. Depth-first search (DFS): Expand deepest node
3. Depth-limited search (DLS): Depth first with depth limit
4. Iterative-deepening search (IDS): DLS with increasing limit

# Uninformed search

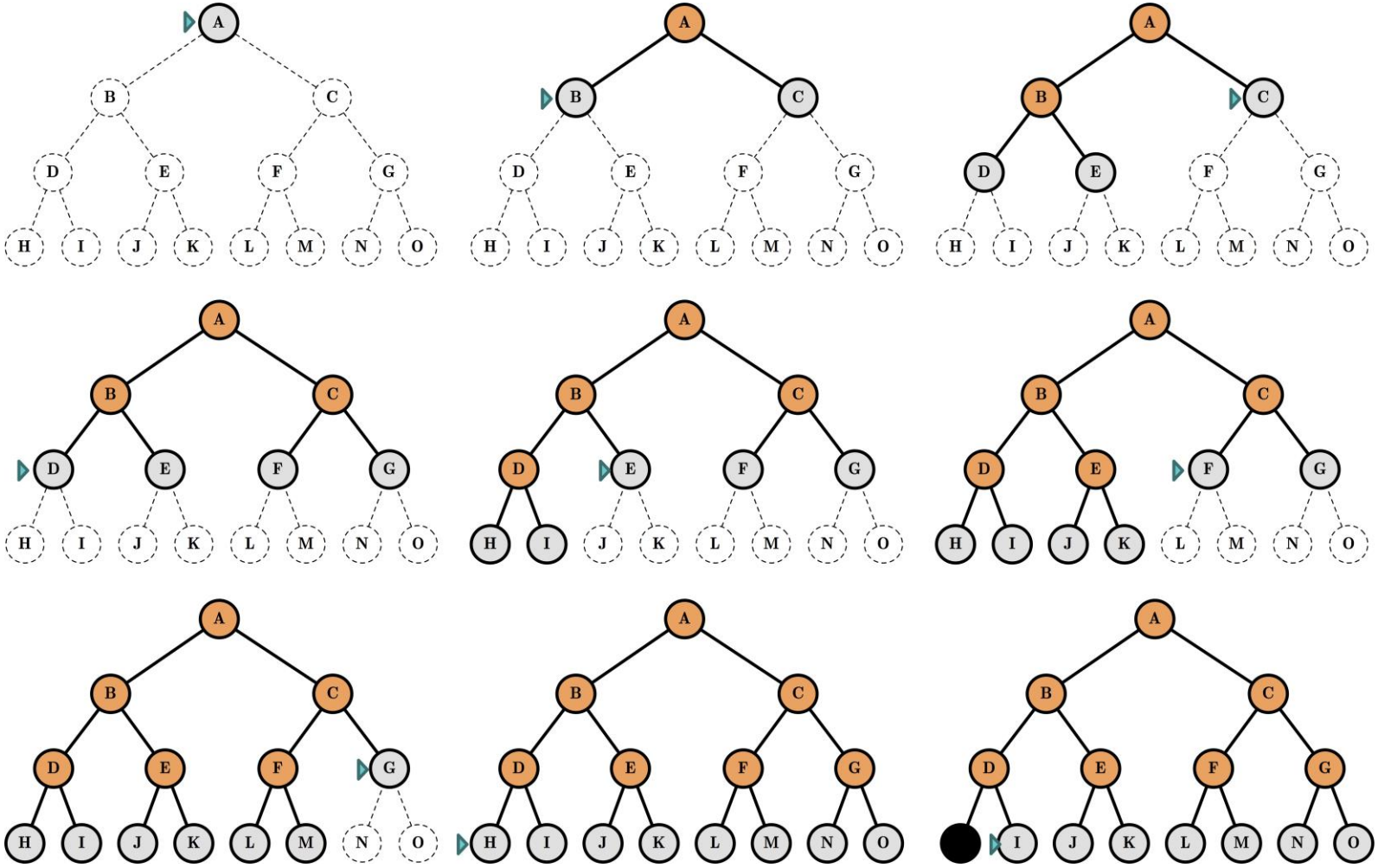
**Use no domain knowledge!**

**Strategies:**

1. Breadth-first search (BFS): Expand shallowest node
2. Depth-first search (DFS): Expand deepest node
3. Depth-limited search (DLS): Depth first with depth limit
4. Iterative-deepening search (IDS): DLS with increasing limit
5. Uniform-cost search (UCS): Expand least cost node

# Breadth-first search (BFS)

**BFS: Expand shallowest first.**



# BFS search

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

# BFS Criteria

BFS criteria?

# BFS

- **Complete** Yes (if $b$ is finite)
- **Time** $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- **Space** $O(b^d)$
  Note: If the *goal test* is applied at expansion rather than generation then $O(b^{d+1})$
- **Optimal** Yes (if cost = 1 per step).
- **implementation**: fringe: FIFO (Queue)

**Question: If time and space complexities are exponential, why use BFS?**

# BFS

How bad is BFS?

# BFS

How bad is BFS?

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 milliseconds | 107 kilobytes |
| 4 | 11,110 | 11 milliseconds | 10.6 megabytes |
| 6 | $10^6$ | 1.1 seconds | 1 gigabyte |
| 8 | $10^8$ | 2 minutes | 103 gigabytes |
| 10 | $10^{10}$ | 3 hours | 10 terabytes |
| 12 | $10^{12}$ | 13 days | 1 petabyte |
| 14 | $10^{14}$ | 3.5 years | 99 petabytes |
| 16 | $10^{16}$ | 350 years | 10 exabytes |

Time and Memory requirements for breadth-first search for a branching factor b=10; 1 million nodes per second; 1,000 bytes per node.

# BFS

How bad is BFS?

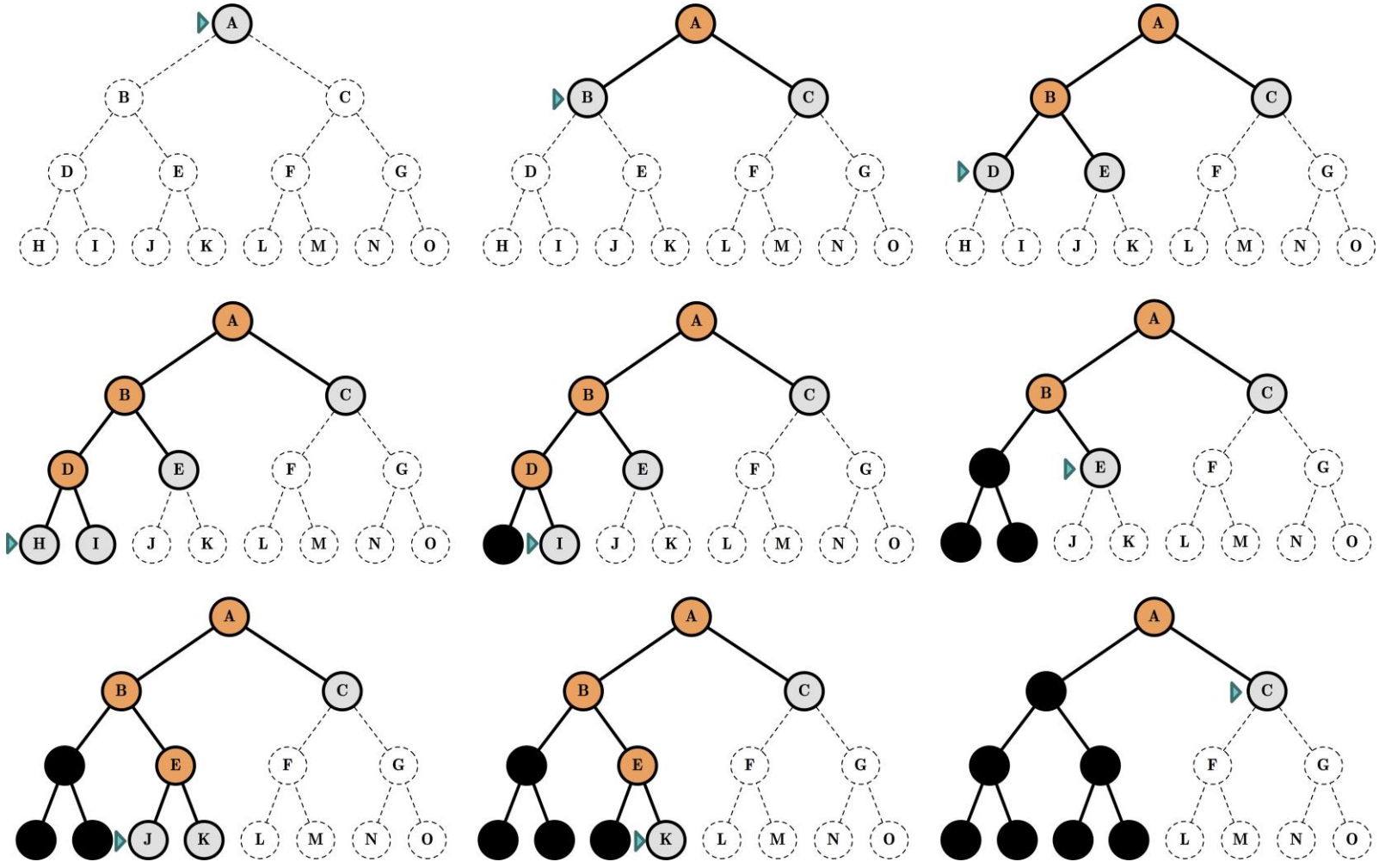
| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 milliseconds | 107 kilobytes |
| 4 | 11,110 | 11 milliseconds | 10.6 megabytes |
| 6 | $10^6$ | 1.1 seconds | 1 gigabyte |
| 8 | $10^8$ | 2 minutes | 103 gigabytes |
| 10 | $10^{10}$ | 3 hours | 10 terabytes |
| 12 | $10^{12}$ | 13 days | 1 petabyte |
| 14 | $10^{14}$ | 3.5 years | 99 petabytes |
| 16 | $10^{16}$ | 350 years | 10 exabytes |

Time and Memory requirements for breadth-first search for a branching factor b=10; 1 million nodes per second; 1,000 bytes per node.

**Memory requirement + exponential time complexity are the biggest handicaps of BFS!**

# DFS

**DFS: Expand deepest first.**



# DFS search

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

# DFS

DFS criteria?

# DFS

- **Complete** No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path.
  ⇒ complete in finite spaces
- **Time** $O(b^m)$: $1 + b + b^2 + b^3 + \ldots + b^m = O(b^m)$
  bad if $m$ is much larger than $d$
  but if solutions are dense, may be much faster than BFS.
- **Space** $O(bm)$linear space complexity!(needs to store only a single path from the root to a leaf node, **along with the remaining unexpanded sibling nodes for each node on the path, hence the $m$ factor**.)
- **Optimal** No
- **Implementation:** fringe: LIFO (Stack)

# DFS

How bad is DFS?

Recall for BFS...

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 milliseconds | 107 kilobytes |
| 4 | 11,110 | 11 milliseconds | 10.6 megabytes |
| 6 | $10^6$ | 1.1 seconds | 1 gigabyte |
| 8 | $10^8$ | 2 minutes | 103 gigabytes |
| 10 | $10^{10}$ | 3 hours | 10 terabytes |
| 12 | $10^{12}$ | 13 days | 1 petabyte |
| 14 | $10^{14}$ | 3.5 years | 99 petabytes |
| 16 | $10^{16}$ | 350 years | 10 exabytes |

Depth =16.

We go down from 10 exabytes in BFS to . . . in DFS?

# DFS

How bad is DFS?

Recall for BFS...

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 milliseconds | 107 kilobytes |
| 4 | 11,110 | 11 milliseconds | 10.6 megabytes |
| 6 | $10^6$ | 1.1 seconds | 1 gigabyte |
| 8 | $10^8$ | 2 minutes | 103 gigabytes |
| 10 | $10^{10}$ | 3 hours | 10 terabytes |
| 12 | $10^{12}$ | 13 days | 1 petabyte |
| 14 | $10^{14}$ | 3.5 years | 99 petabytes |
| 16 | $10^{16}$ | 350 years | 10 exabytes |

Depth =16.

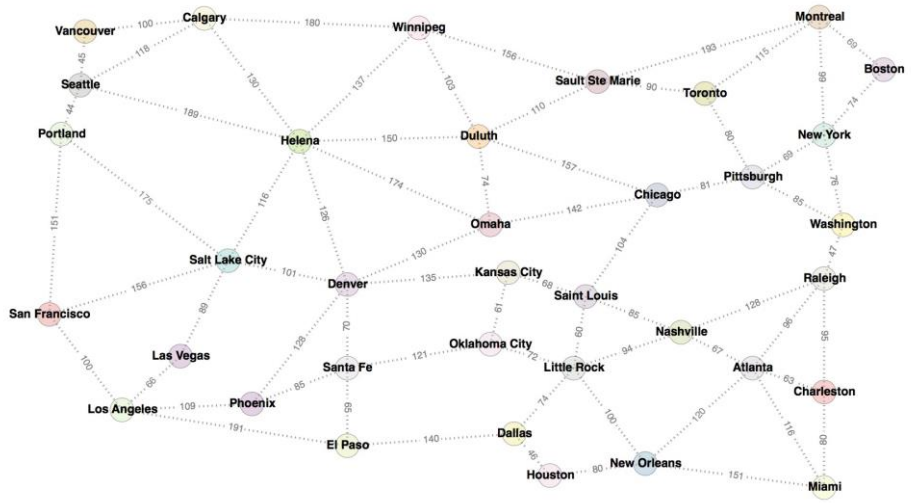
We go down from 10 exabytes in BFS to156kilobytes in DFS!

# Depth-limited search

- DFS with depth limit *l* (nodes at level *l* has no successors).
- Select some limit *L* in depth to explore with DFS
- Iterative deepening: increasing the limit *l*

# Depth-limited search

- If we know some knowledge about the problem, may be we don't need to go to a full depth.




Idea: any city can be reached from another city in at most $L$ steps with $L < 36$.

# Iterative Deepening

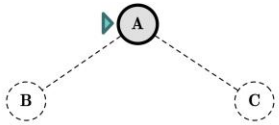
- Combines the benefits of BFS and DFS.
- Idea: Iteratively increase the search limit until the depth of the shallowest solution *d* is reached.
- AppliesDLS with increasing limits.
- The algorithm will stop if a solution is found or if DLS returns a failure (no solution).
- Because most of the nodes are on the bottom of the search tree, it not a big waste to iteratively re-generate the top
- Let's take an example with a depth limit between 0 and 3.

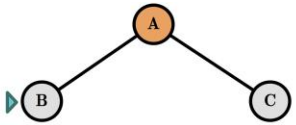
# Iterative Deepening

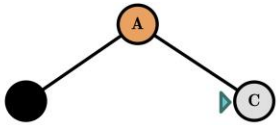
Limit = 0

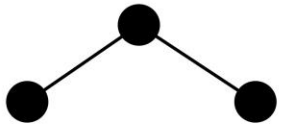
# Iterative Deepening

**Limit = 1**



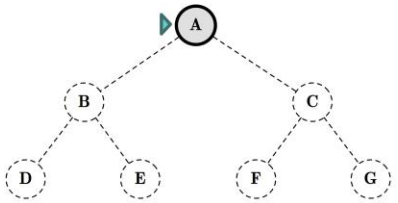



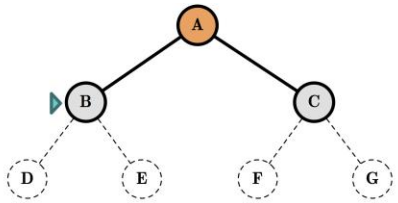



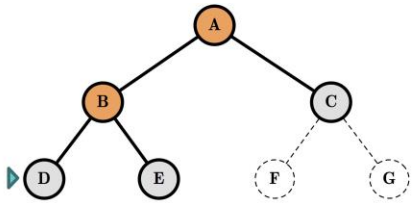



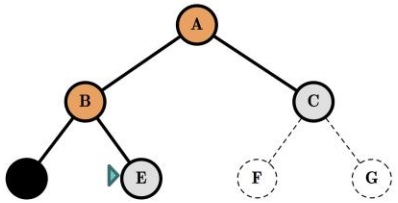
# Iterative Deepening

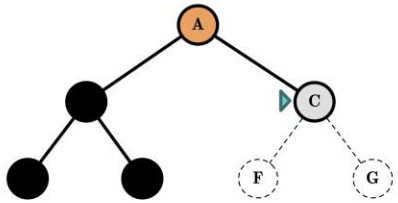
**Limit = 2**

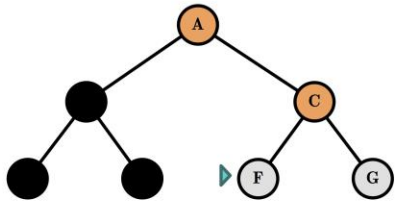


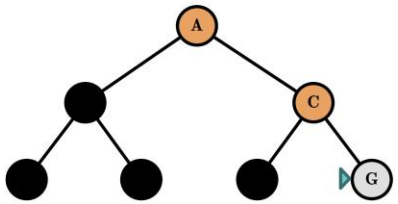



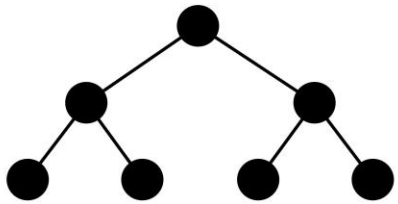



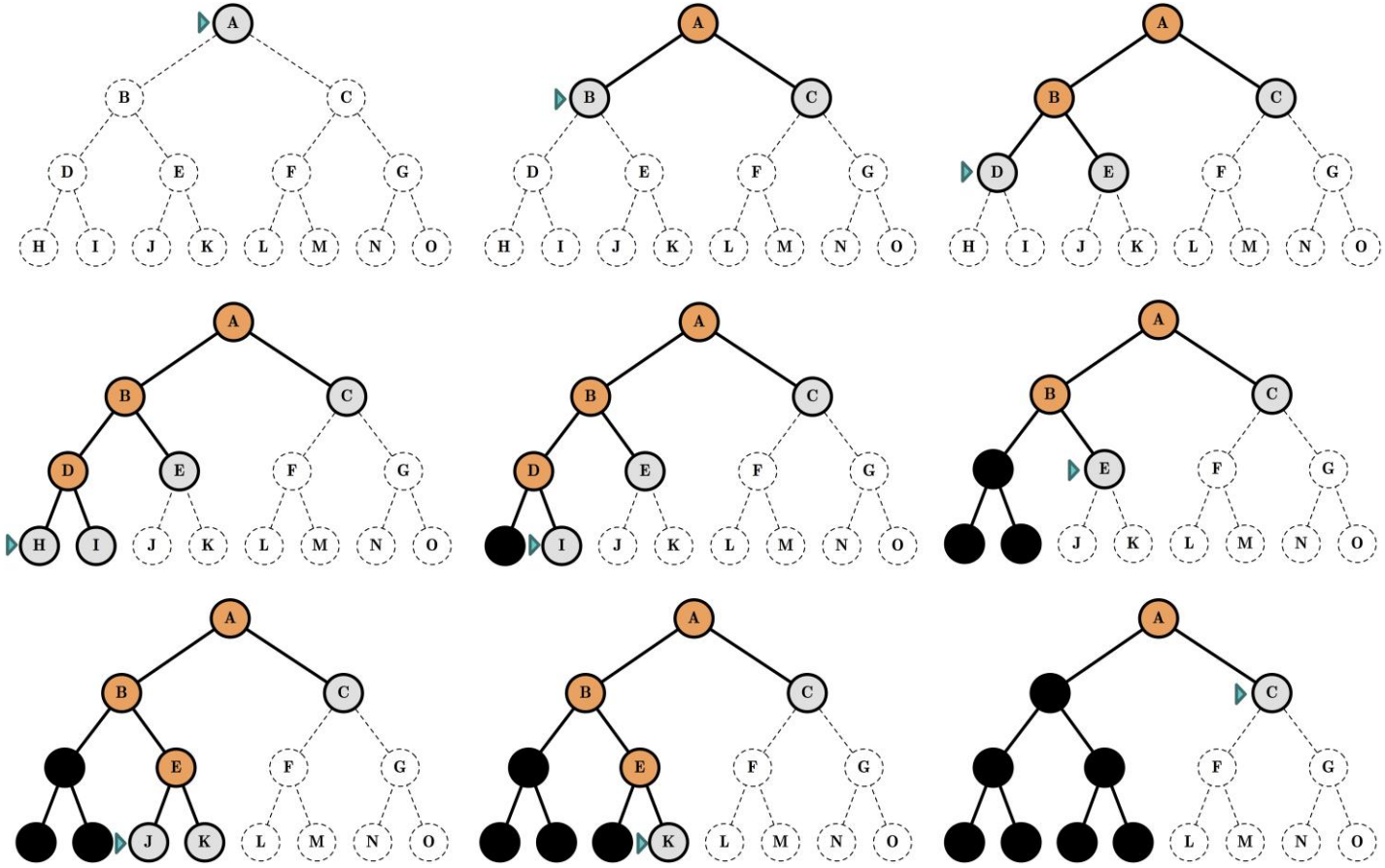



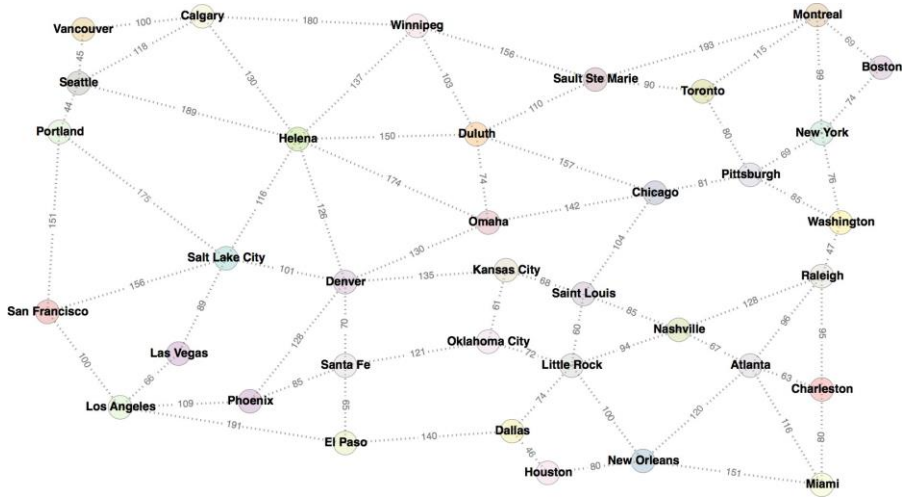



# Iterative Deepening

**Limit = 3**

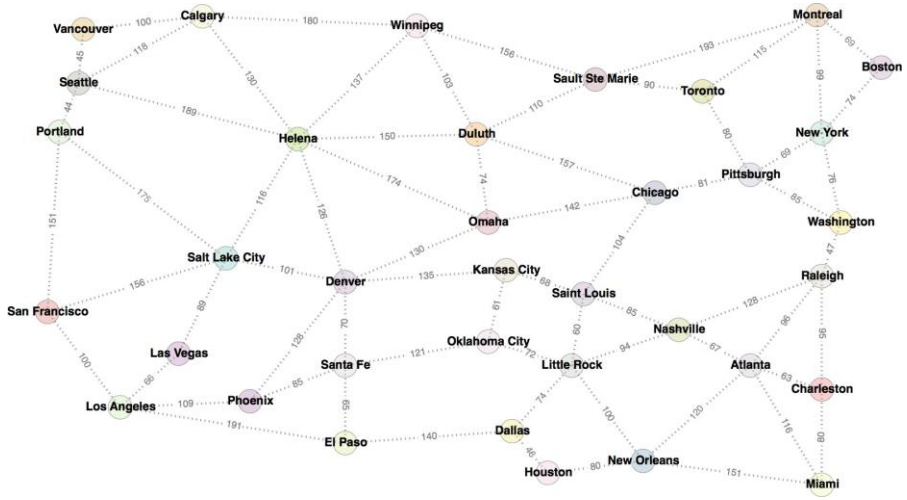


# Uniform-cost search



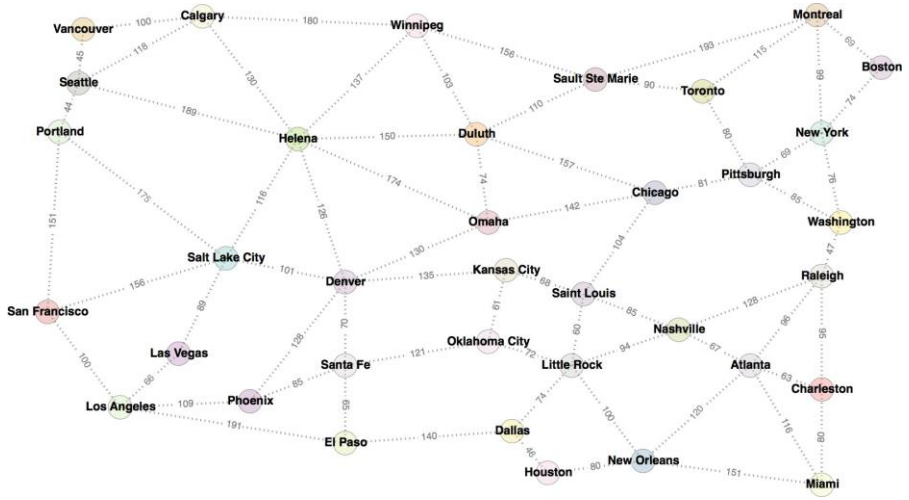

- The arcs in the search graph may have weights (different cost attached). How to leverage this information?

# Uniform-cost search



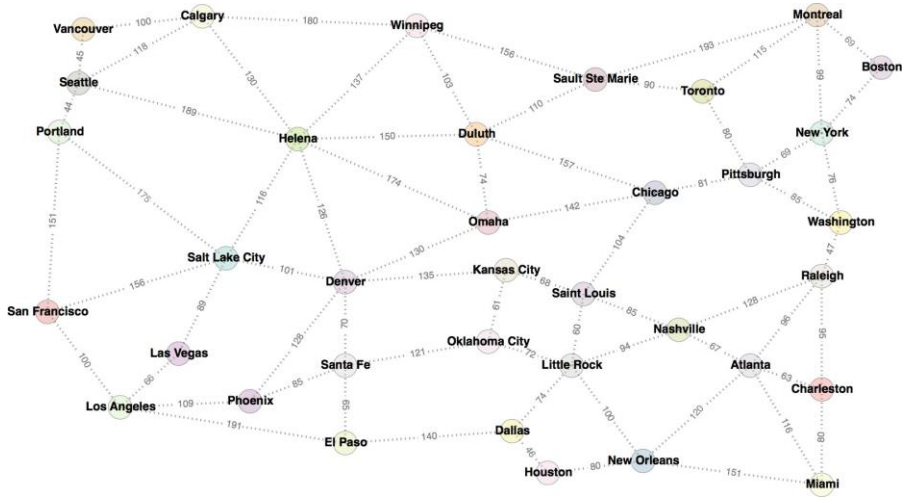

- The arcs in the search graph may have weights (different cost attached). How to leverage this information?
- BFS will find the shortest path which may be costly.
- We want thecheapestnot shallowest solution.

# Uniform-cost search




- The arcs in the search graph may have weights (different cost attached). How to leverage this information?
- BFS will find the shortest path which may be costly.
- We want the cheapest not shallowest solution.
- Modify BFS: Prioritize by cost not depth → **Expand node** *n* **with the lowest path cost** $g(n)$

# Uniform-cost search

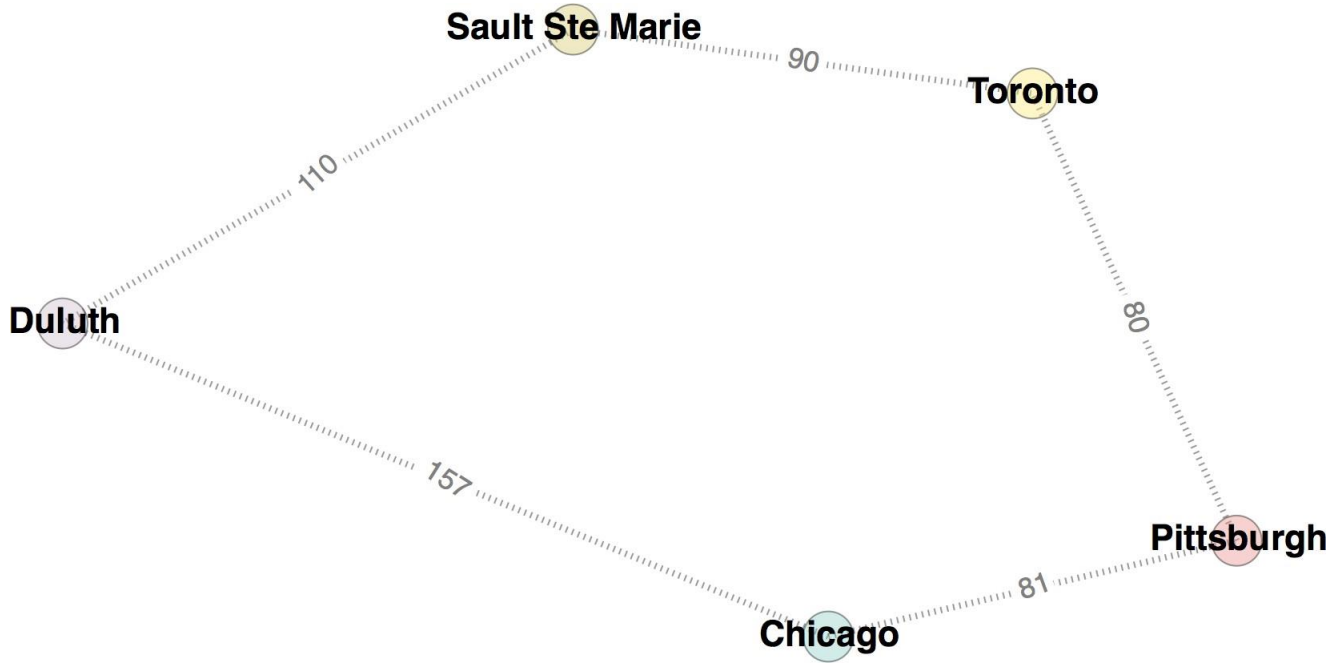



- The arcs in the search graph may have weights (different cost attached). How to leverage this information?
- BFS will find the shortest path which may be costly.
- We want the cheapest not shallowest solution.
- Modify BFS: Prioritize by cost not depth → **Expand node** *n* **with the lowest path cost** $g(n)$
- Explores increasing costs.

# UCS algorithm

```
function UNIFORM-COST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

# Uniform-cost search



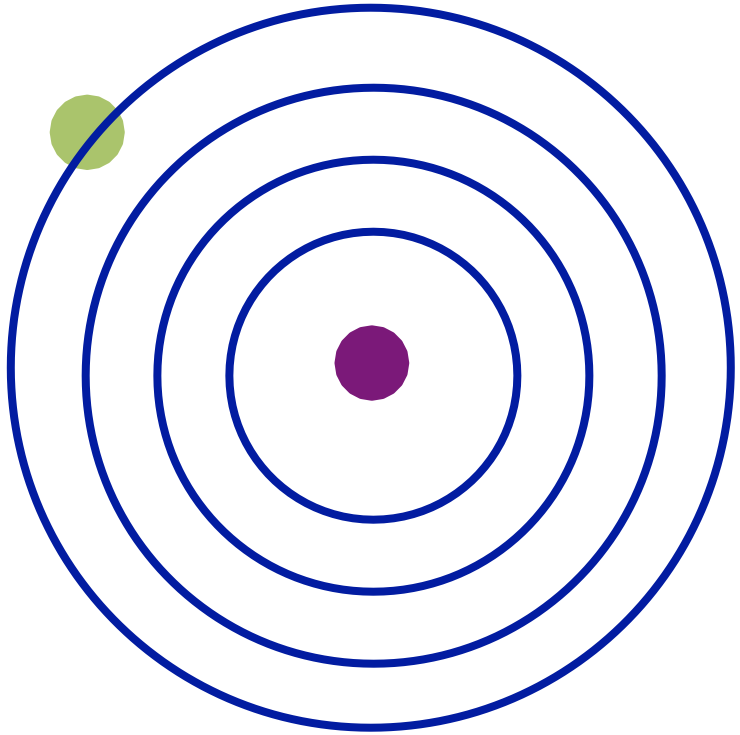

Go from Chicago to Sault Ste Marie. Using BFS, we would find Chicago-Duluth-Sault Ste Marie. However, using UCS, we would find Chicago-Pittsburgh-Toronto-Sault Ste Marie, which is actually the shortest path!

# Uniform-cost search

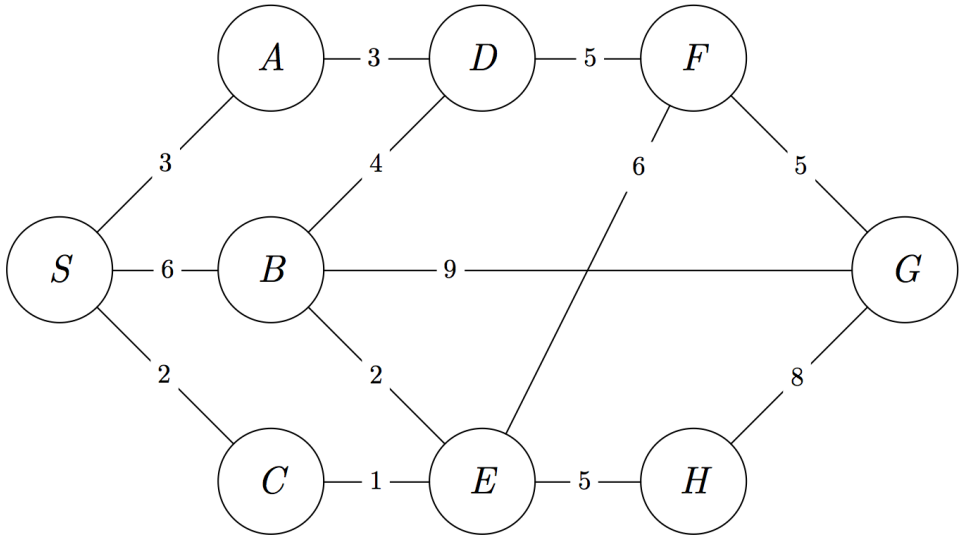
- **Complete** Yes, if solution has a finite cost.
- **Time**
  - Suppose $C^*$: cost of the optimal solution
  - Every action costs at least $s$ (bound on the cost)
  - The effective depth is roughly $C^*/s$ (how deep the *cheapest* solution could be).
  - $O(b^{C^*/s})$
- **Space** # of nodes with $g \leq$ cost of optimal solution, $O(b^{C^*/s})$
- **Optimal** Yes
- **Implementation**: fringe = queue ordered by path cost $g(n)$, lowest first = Heap!

# Uniform-cost search

While complete and optimal, UCS explores the space in every direction because no information is provided about the goal!

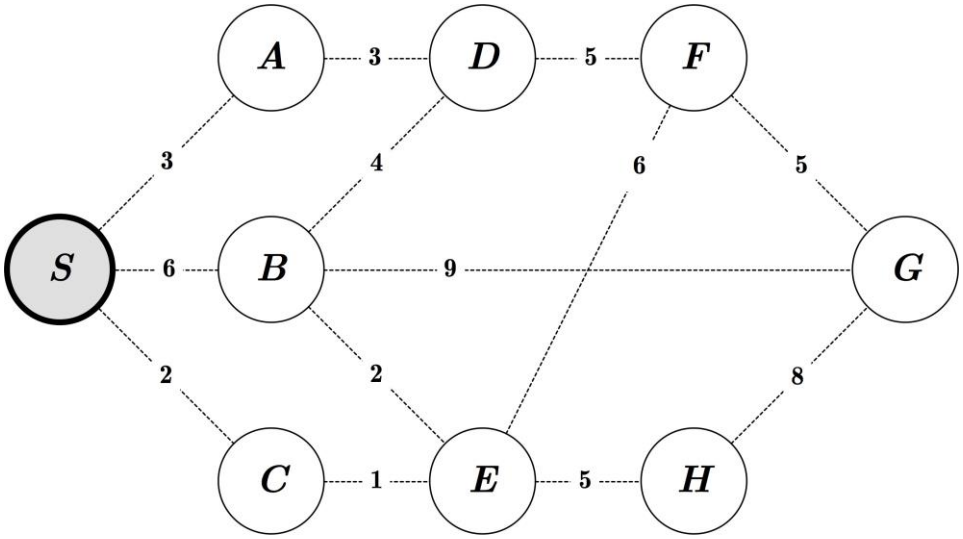


# Exercise




**Question:** What is the **order of visits of the nodes** and the **path** returned by BFS, DFS and UCS?

# Exercise: BFS

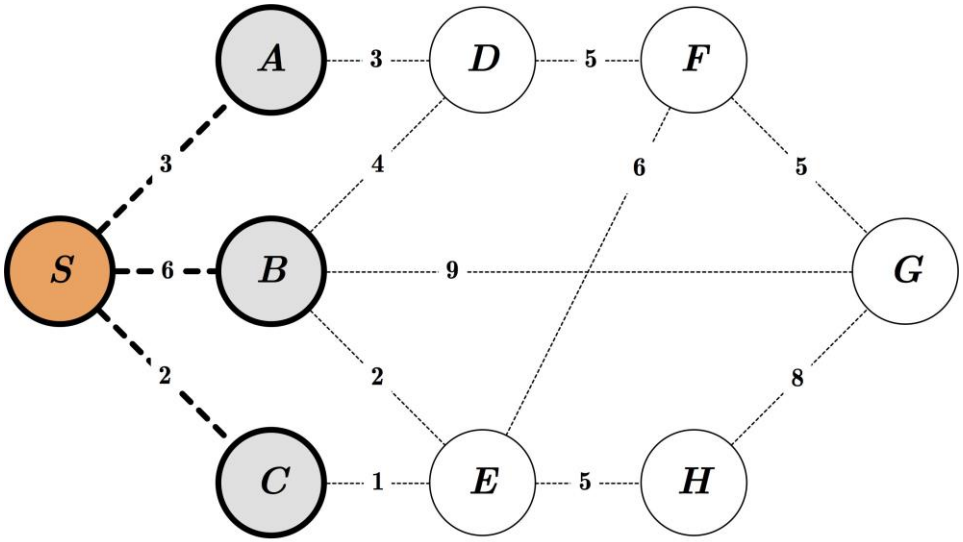



**Queue:**

| |
|---|
| *S* |

**Order of Visit:**

# Exercise: BFS



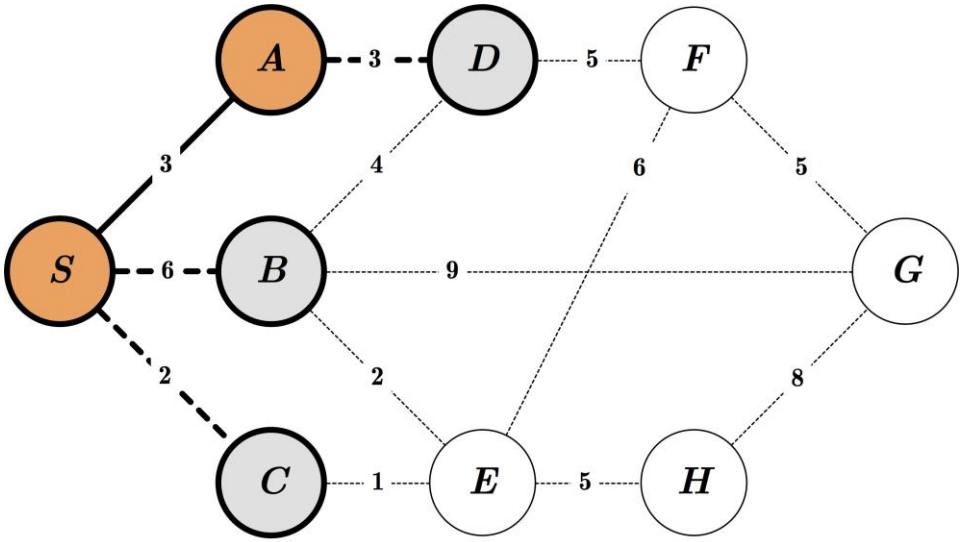

**Queue:**

| | | | |
|---|---|---|---|
| *S* | *A* | *B* | *C* |

**Order of Visit:**

*S*

# Exercise: BFS



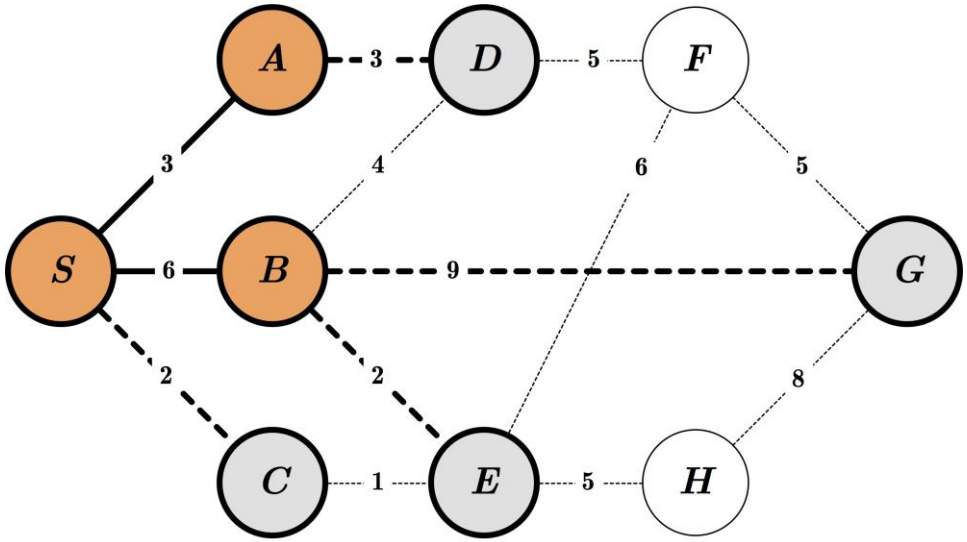

**Queue:**

| | | | | |
|---|---|---|---|---|
| *S* | *A* | *B* | *C* | *D* |

**Order of Visit:**

*S* *A*

# Exercise: BFS



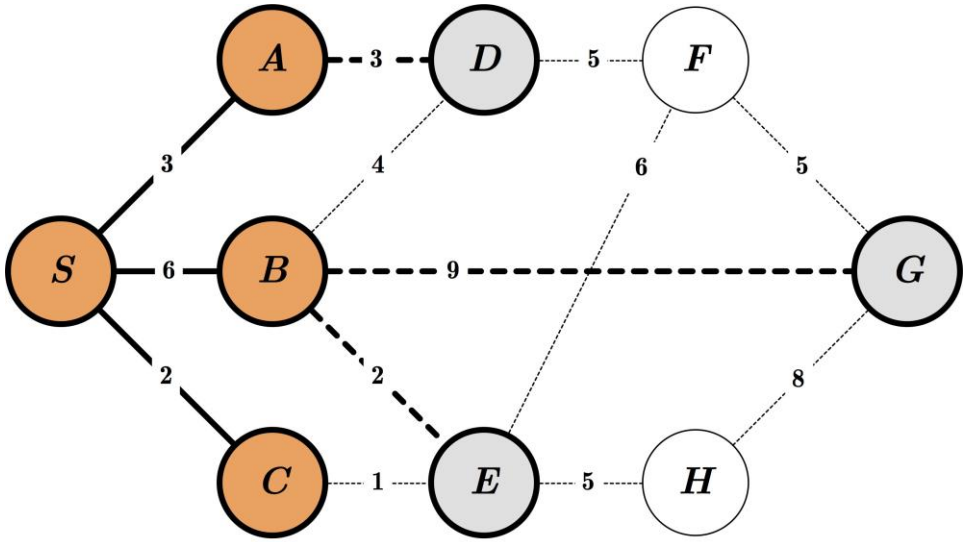

**Queue:**

| | | | | | | |
|---|---|---|---|---|---|---|
| *S* | *A* | *B* | *C* | *D* | *E* | *G* |

**Order of Visit:**

*S* *A* *B*

# Exercise: BFS



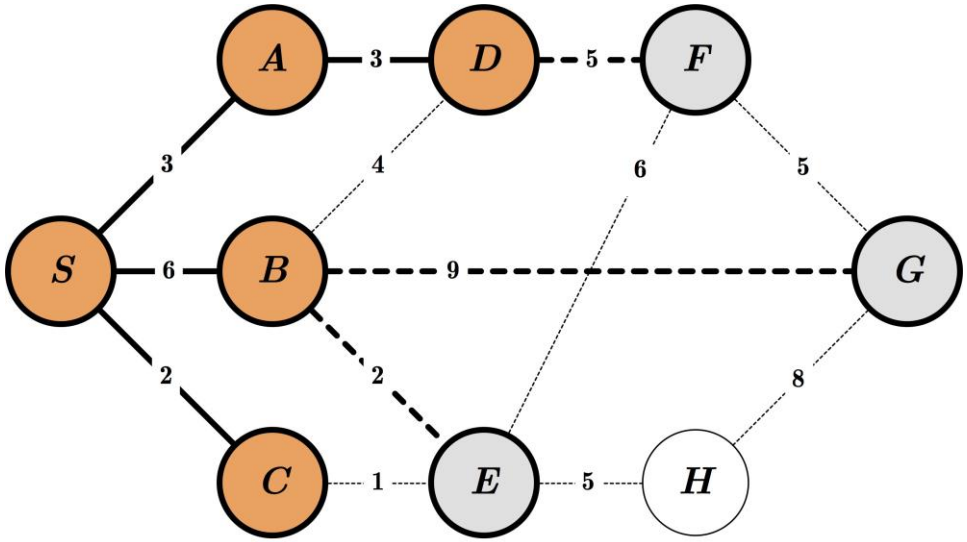

**Queue:**

| | | | | | | |
|---|---|---|---|---|---|---|
| *S* | *A* | *B* | *C* | *D* | *E* | *G* |

**Order of Visit:**

*S* *A* *B* *C*

# Exercise: BFS



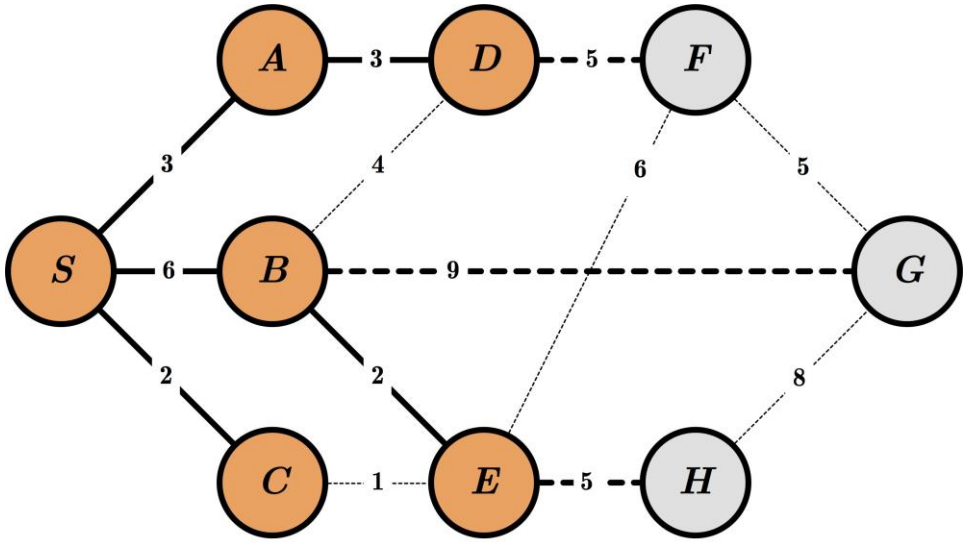

**Queue:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *S* | *A* | *B* | *C* | *D* | *E* | *G* | *F* |

**Order of Visit:**

*S* *A* *B* *C* *D*

# Exercise: BFS



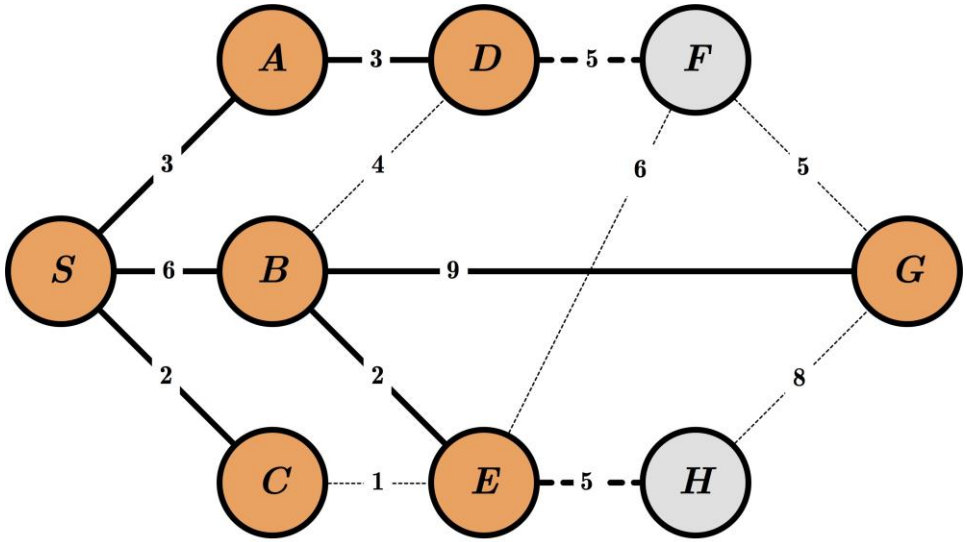

**Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *A* | *B* | *C* | *D* | *E* | *G* | *F* | *H* |

**Order of Visit:**

*S* *A* *B* *C* *D* *E*

# Exercise: BFS



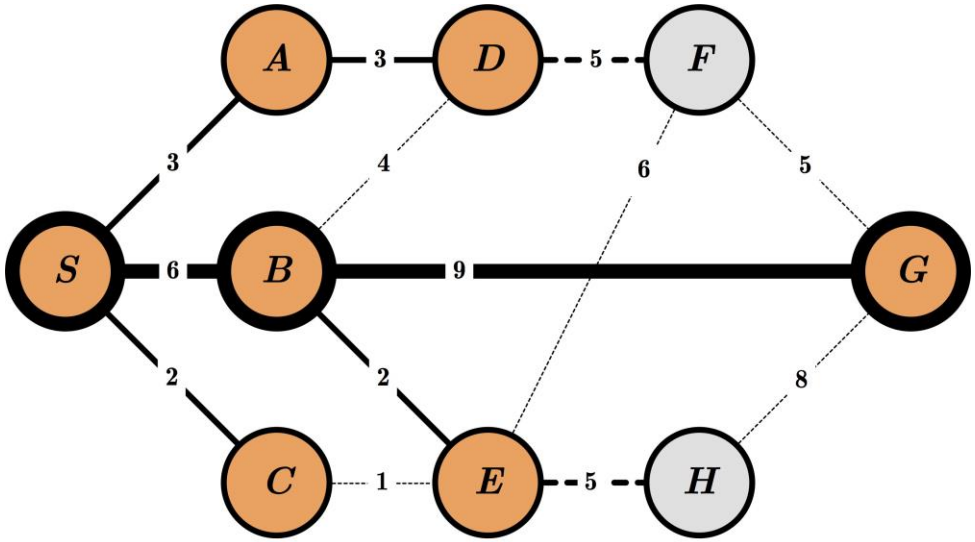

**Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *A* | *B* | *C* | *D* | *E* | *G* | ***F*** | ***H*** |

**Order of Visit:**

***S*** ***A*** ***B*** ***C*** ***D*** ***E*** ***G***

# Exercise: BFS



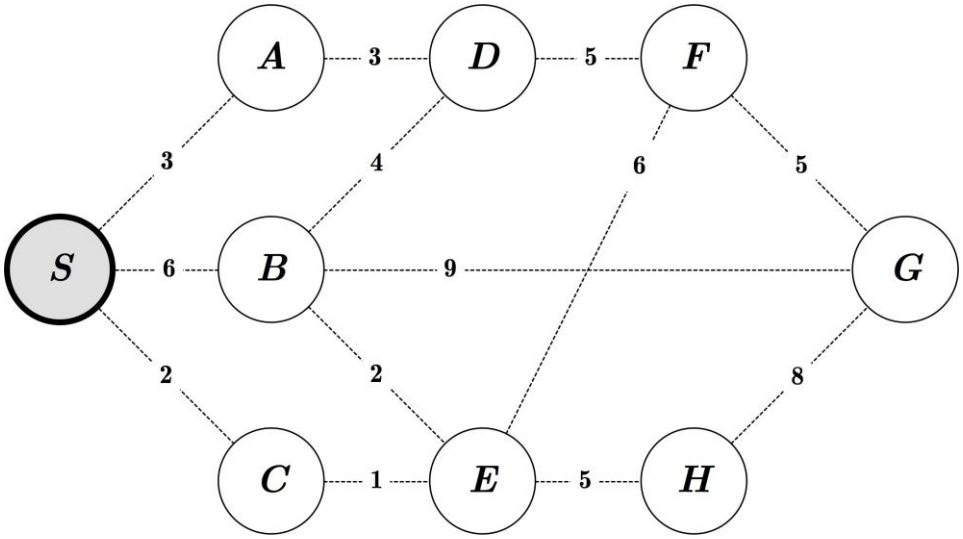

**Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *A* | *B* | *C* | *D* | *E* | *G* | ***F*** | ***H*** |

**Order of Visit:**

***S*** ***A*** ***B*** ***C*** ***D*** ***E*** ***G***

# Exercise: DFS

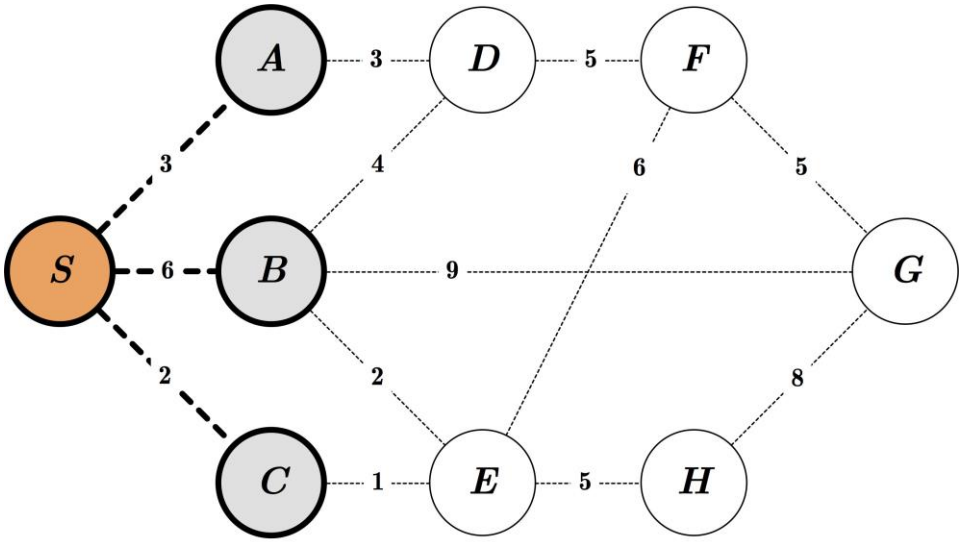



**Stack:**

| |
|---|
| *S* |

**Order of Visit:**

# Exercise: DFS



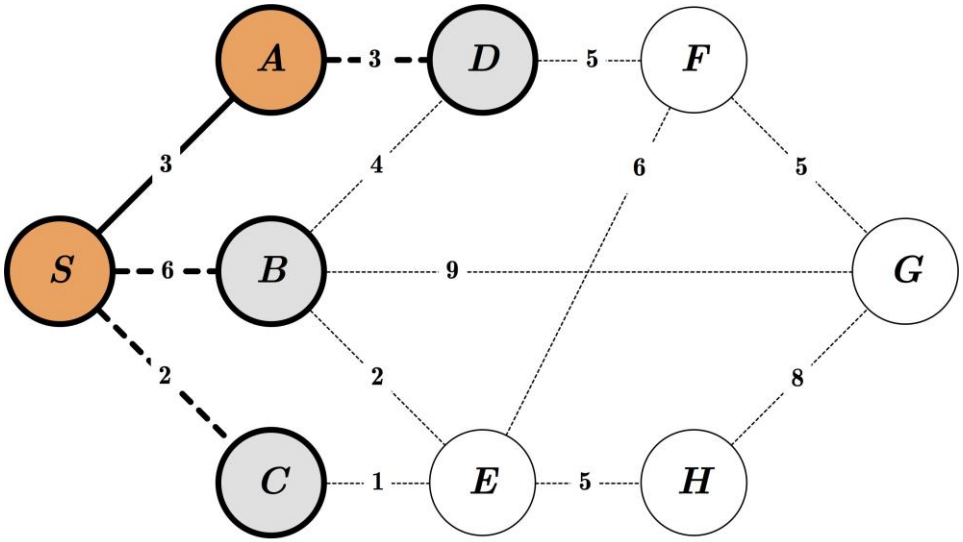

**Stack:**

| | | | |
|---|---|---|---|
| *S* | *C* | *B* | *A* |

**Order of Visit:**

*S*

# Exercise: DFS



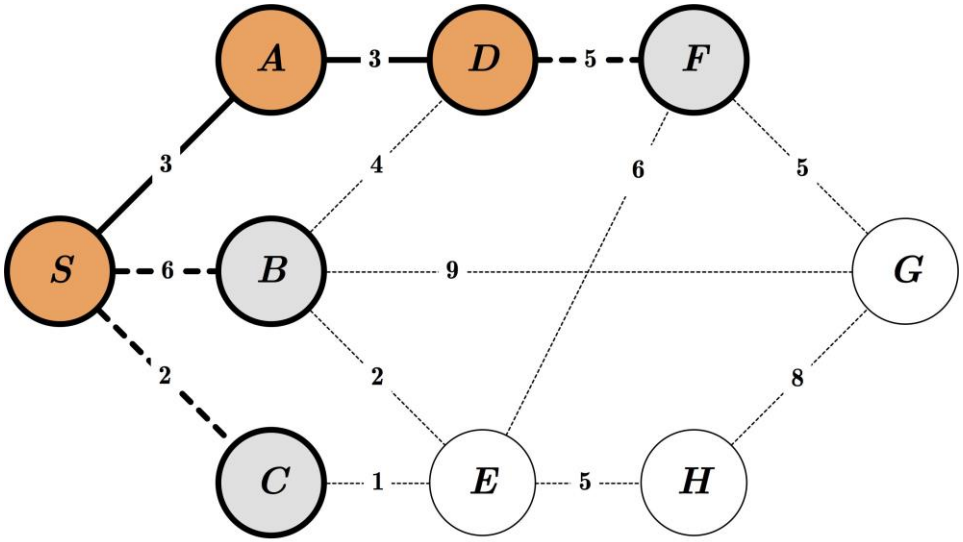

**Stack:**

| | | | | |
|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* |

**Order of Visit:**

*S* *A*

# Exercise: DFS



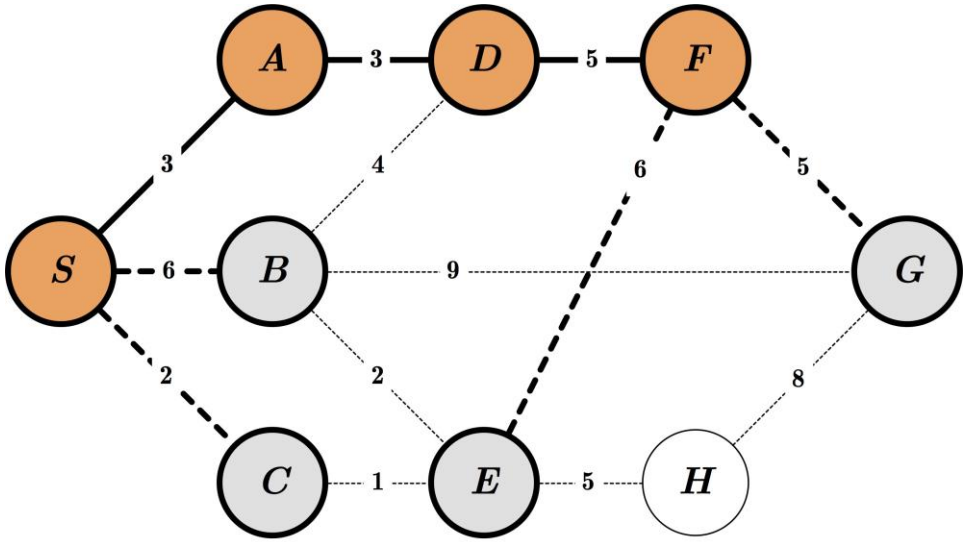

**Stack:**

| | | | | | |
|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* |

**Order of Visit:**

*S* *A* *D*

# Exercise: DFS



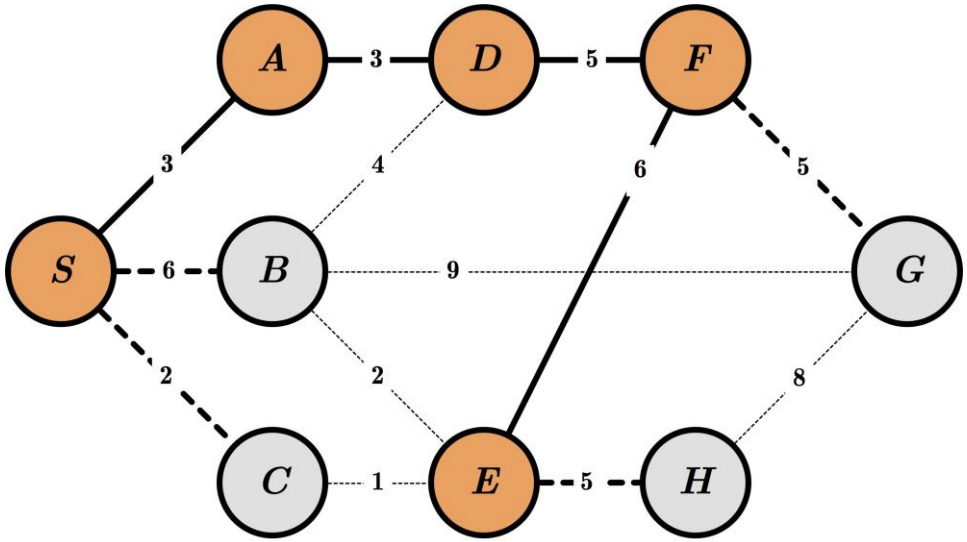

**Stack:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* |

**Order of Visit:**

*S* *A* *D* *F*

# Exercise: DFS



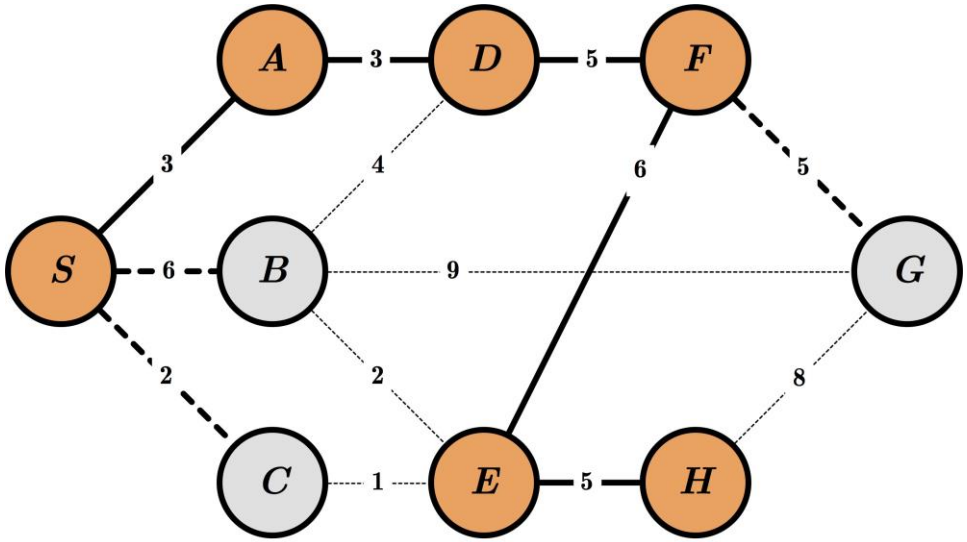

**Stack:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* | *H* |

**Order of Visit:**

*S* *A* *D* *F* *E*

# Exercise: DFS



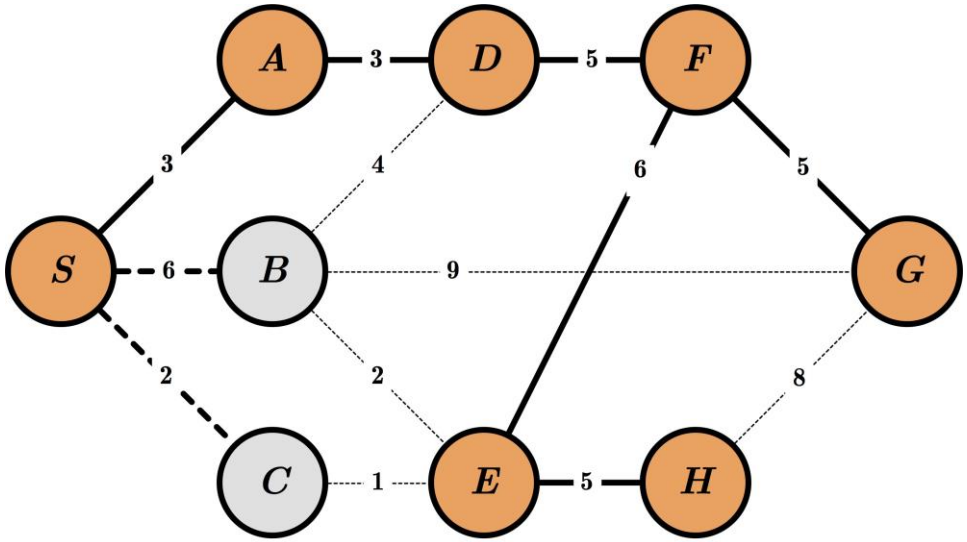

**Stack:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* | *H* |

**Order of Visit:**

*S* *A* *D* *F* *E* *H*

# Exercise: DFS



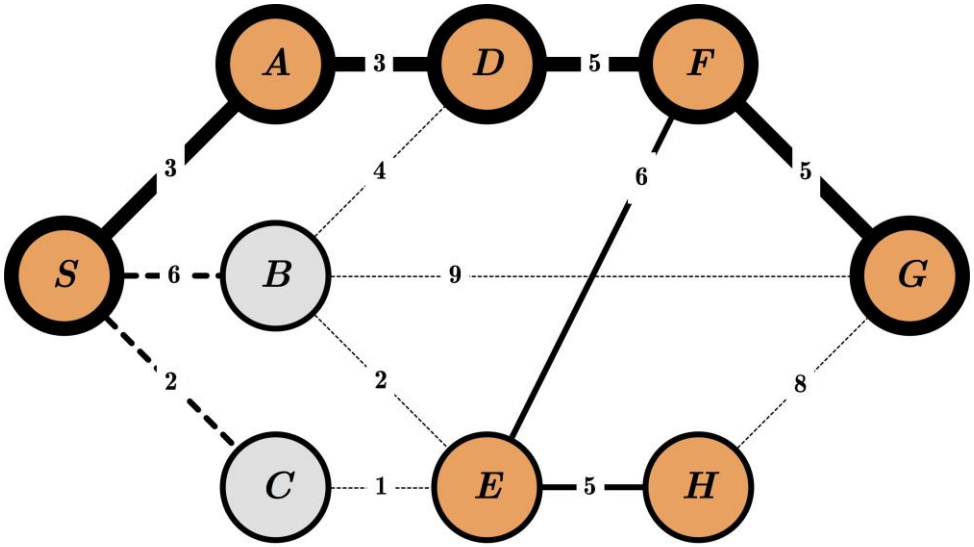

**Stack:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* | *H* |

**Order of Visit:**

*S* *A* *D* *F* *E* *H* *G*

# Exercise: DFS



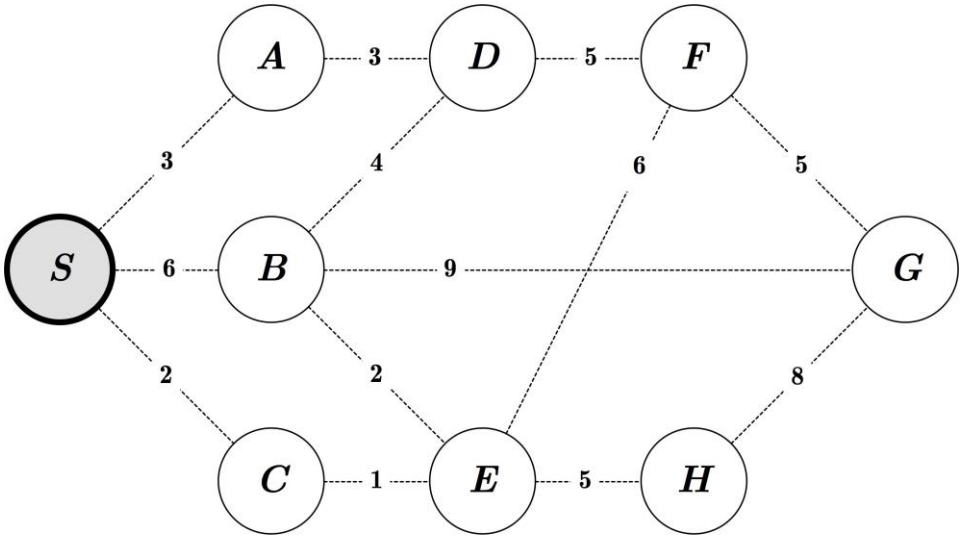

**Stack:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *B* | *A* | *D* | *F* | *G* | *E* | *H* |

**Order of Visit:**

*S* *A* *D* *F* *E* *H* *G*

# Exercise: UCS

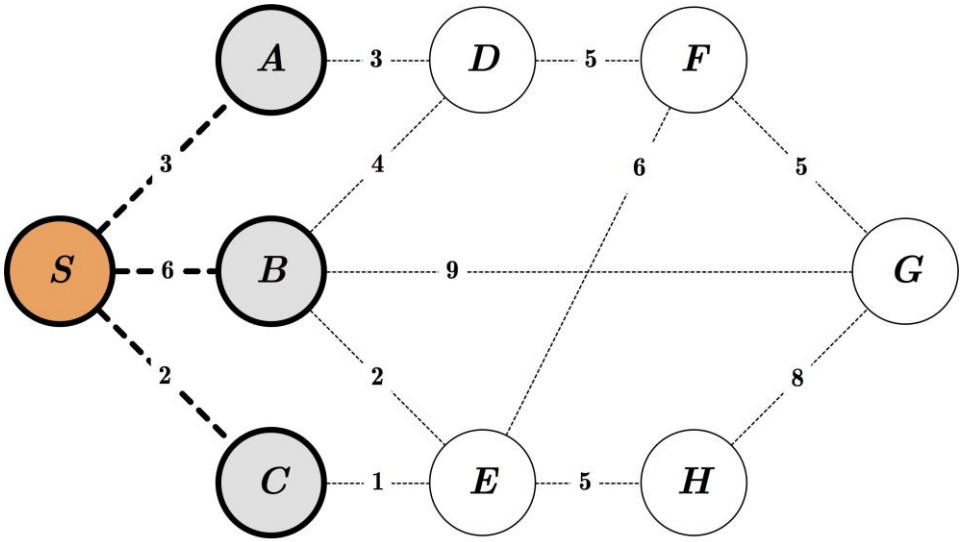



**Priority Queue:**

| $S_0$ |
|---|

**Order of Visit:**

# Exercise: UCS



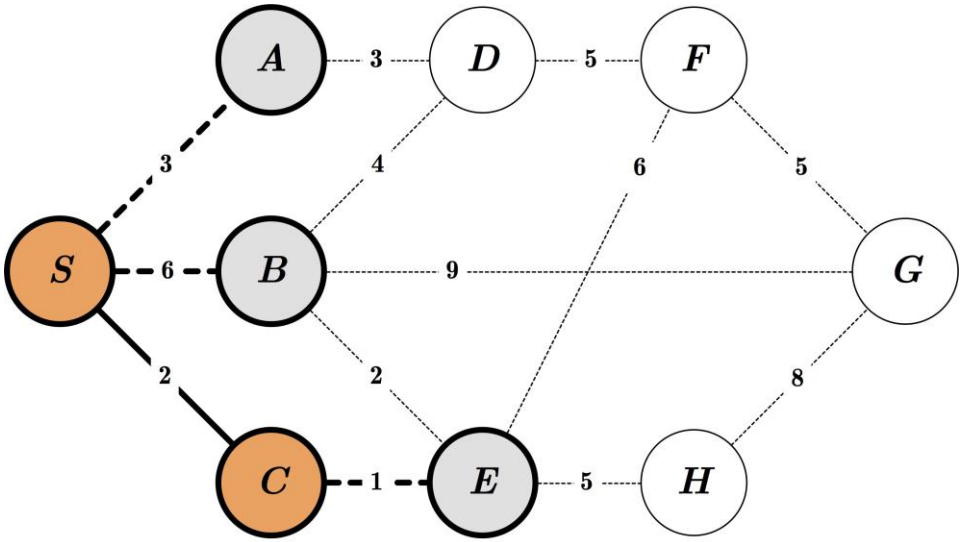

**Priority Queue:**

| | | | |
|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $B_6$ |

**Order of Visit:**

$S$

# Exercise: UCS



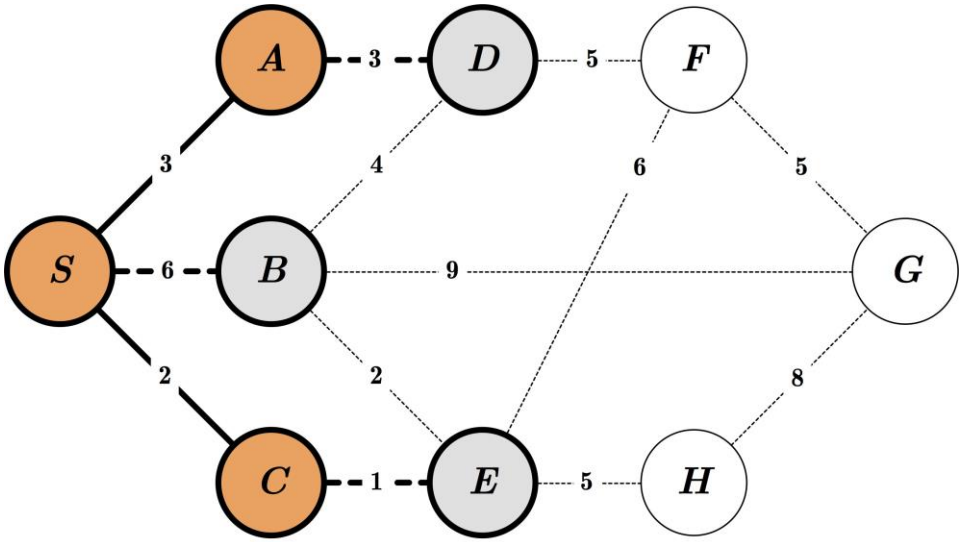

**Priority Queue:**

| | | | | |
|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_6$ |

**Order of Visit:**

$S$ $C$

# Exercise: UCS



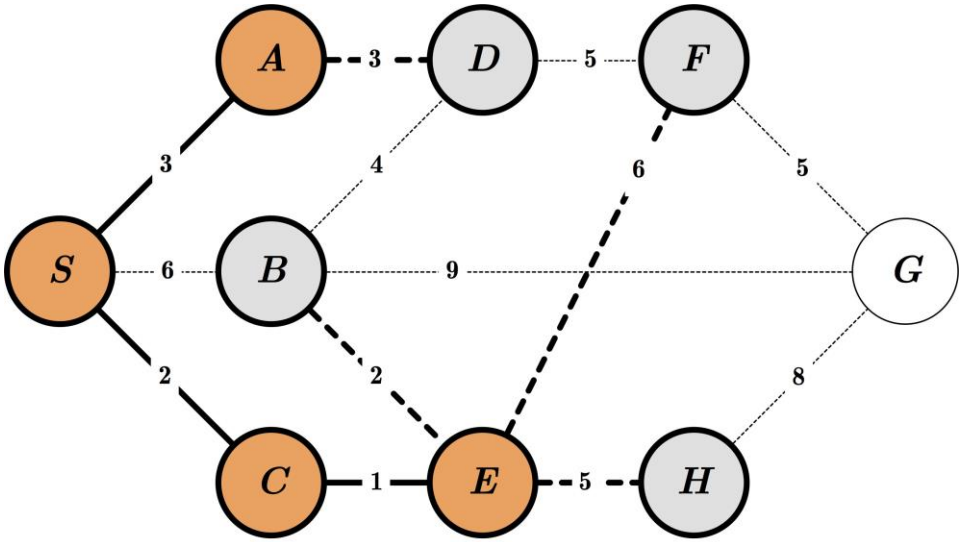

**Priority Queue:**

| | | | | | |
|---|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_6$ | $D_6$ |

**Order of Visit:**

$S$ $C$ $A$

# Exercise: UCS



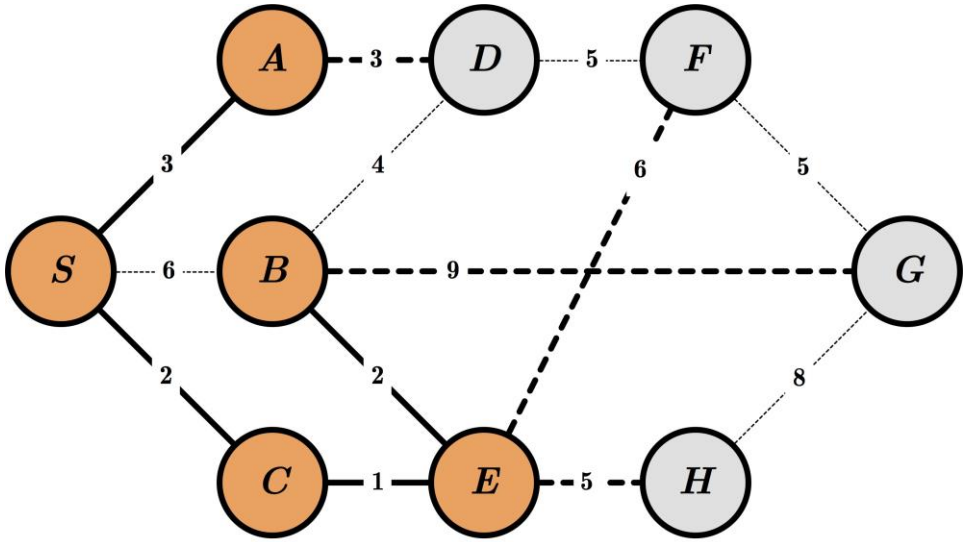

**Priority Queue:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_5$ | $D_6$ | $H_8$ | $F_9$ |

**Order of Visit:**

*S* *C* *A* *E*

# Exercise: UCS



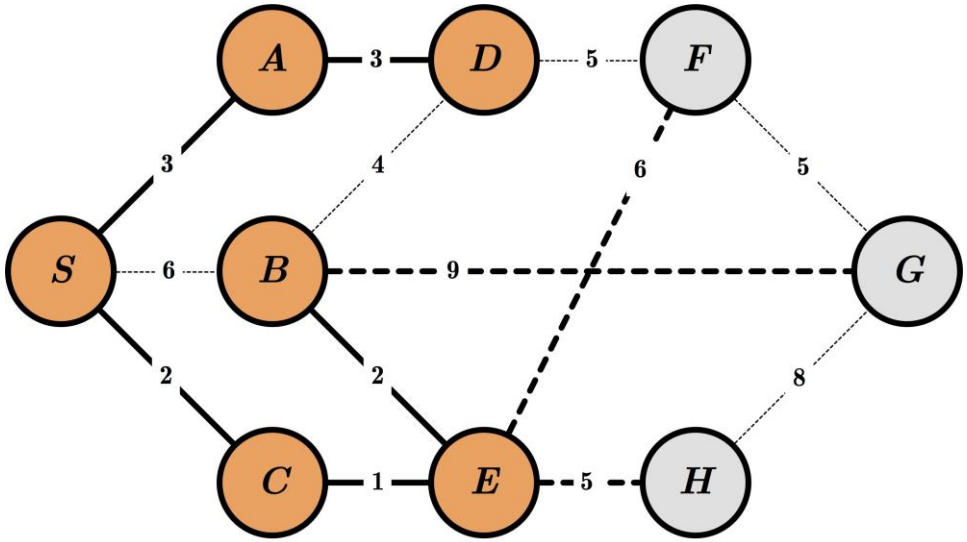

**Priority Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_5$ | $D_6$ | $H_8$ | $F_9$ | $G_{14}$ |

**Order of Visit:**

$S$ $C$ $A$ $E$ $B$

# Exercise: UCS



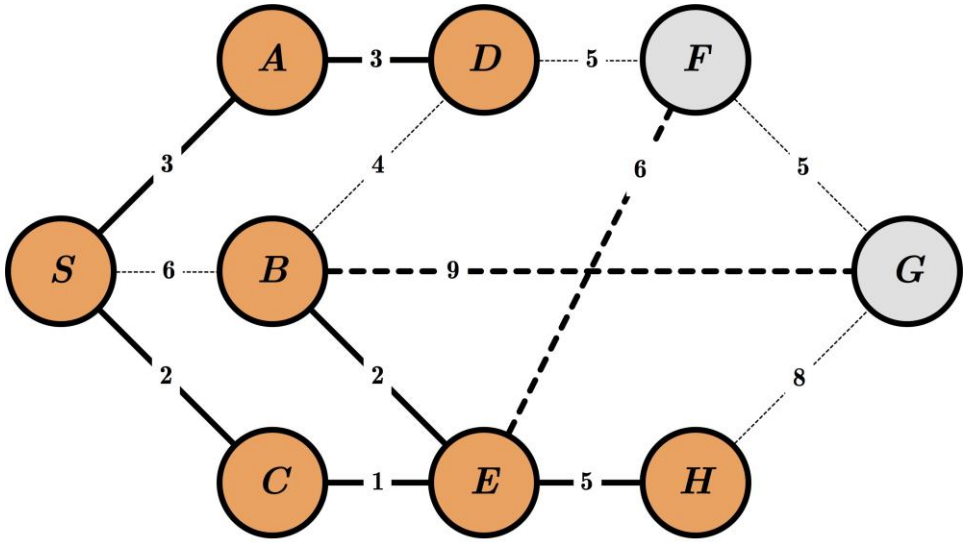

**Priority Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_5$ | $D_6$ | $H_8$ | $F_9$ | $G_{14}$ |

**Order of Visit:**

*S* *C* *A* *E* *B* *D*

# Exercise: UCS



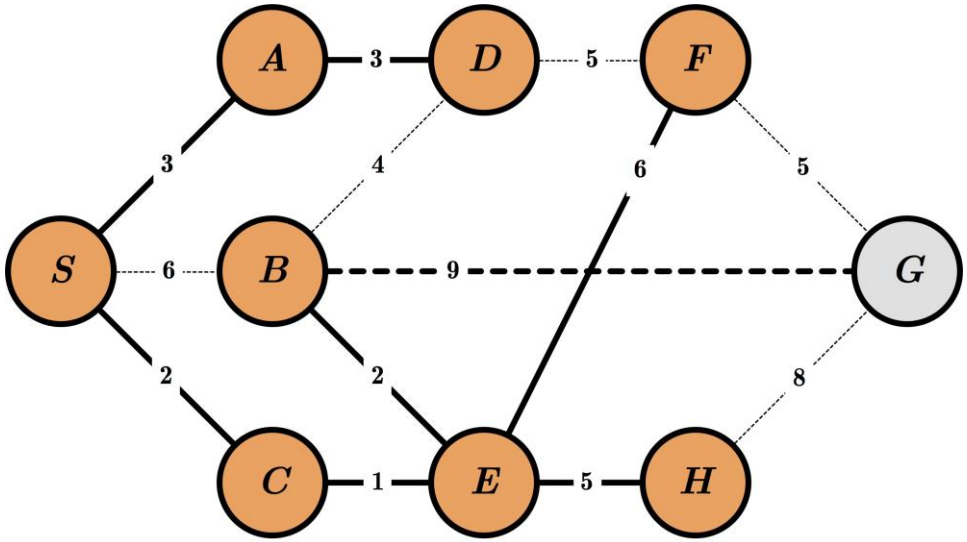

**Priority Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_5$ | $D_6$ | $H_8$ | $F_9$ | $G_{14}$ |

**Order of Visit:**

*S* *C* *A* *E* *B* *D* *H*

# Exercise: UCS



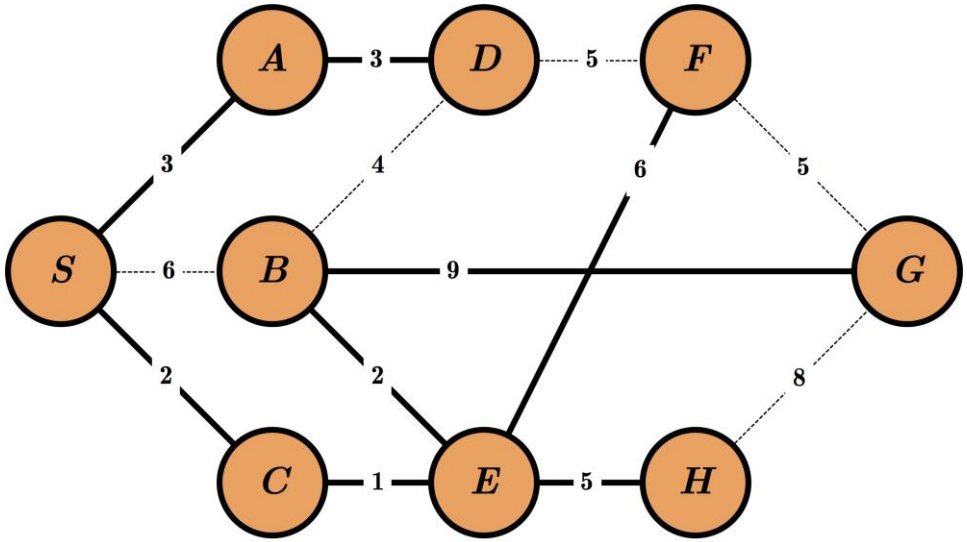

**Priority Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_5$ | $D_6$ | $H_8$ | $F_9$ | $G_{14}$ |

**Order of Visit:**

*S* *C* *A* *E* *B* *D* *H* *F*

# Exercise: UCS



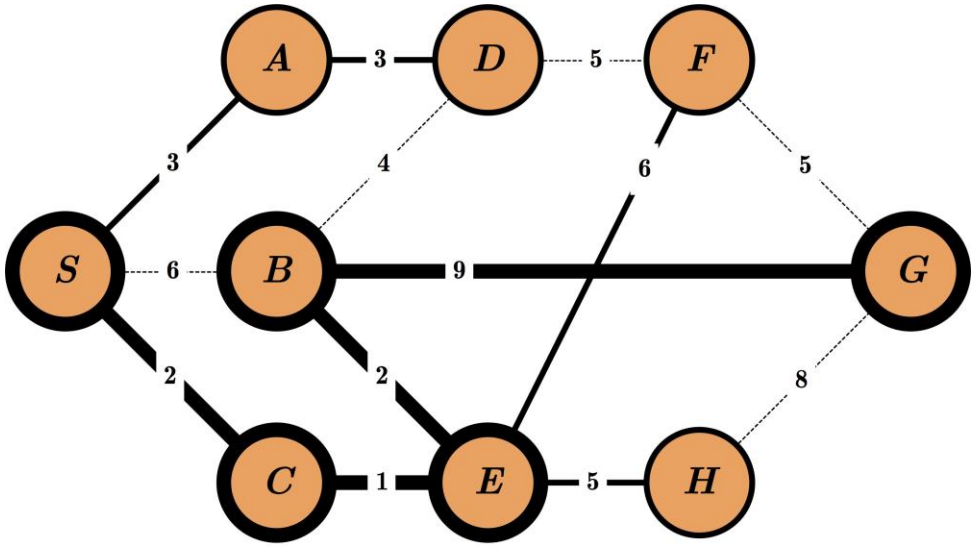

**Priority Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_5$ | $D_6$ | $H_8$ | $F_9$ | $G_{14}$ |

**Order of Visit:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *S* | *C* | *A* | *E* | *B* | *D* | *H* | *F* | *G* |

# Exercise: UCS




**Priority Queue:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_0$ | $C_2$ | $A_3$ | $E_3$ | $B_5$ | $D_6$ | $H_8$ | $F_9$ | $G_{14}$ |

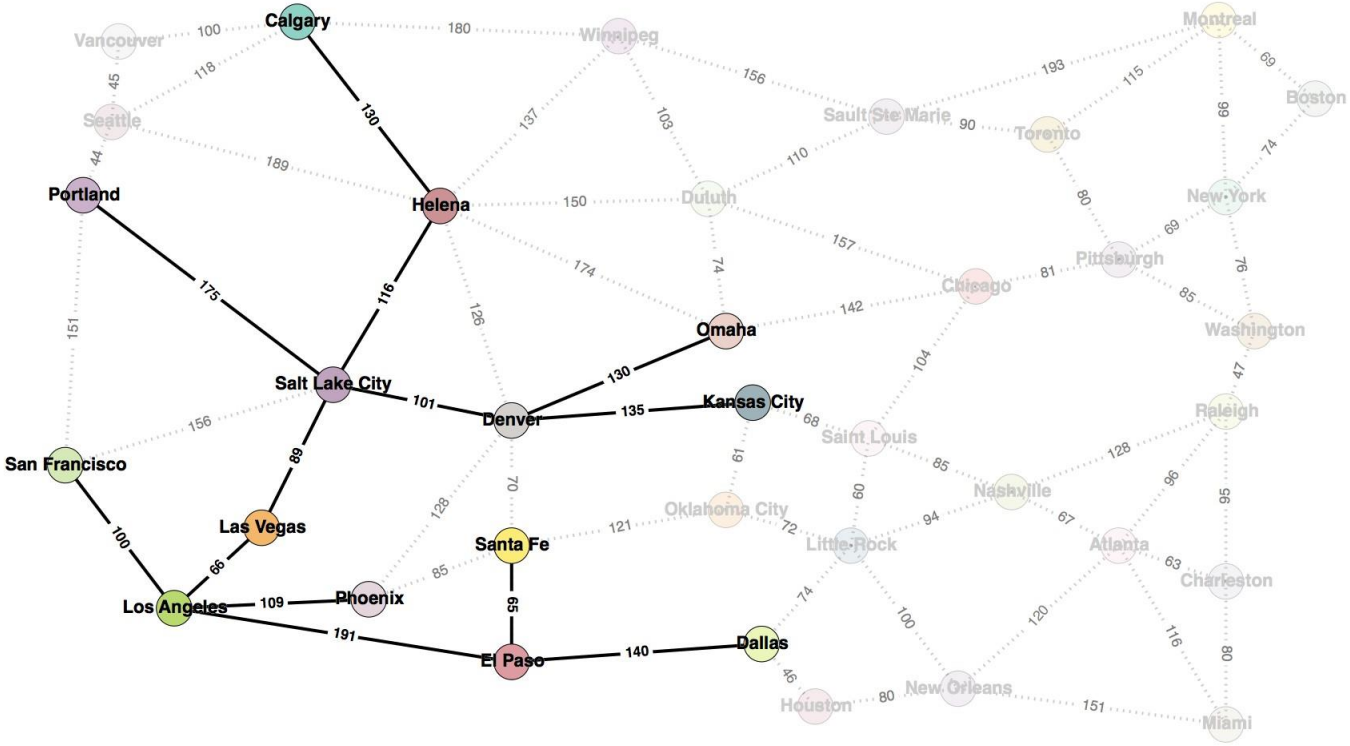
**Order of Visit:**

*S* *C* *A* *E* *B* *D* *H* *F* *G*

# Examples using the map

**Start: Las Vegas**

**Goal: Calgary**



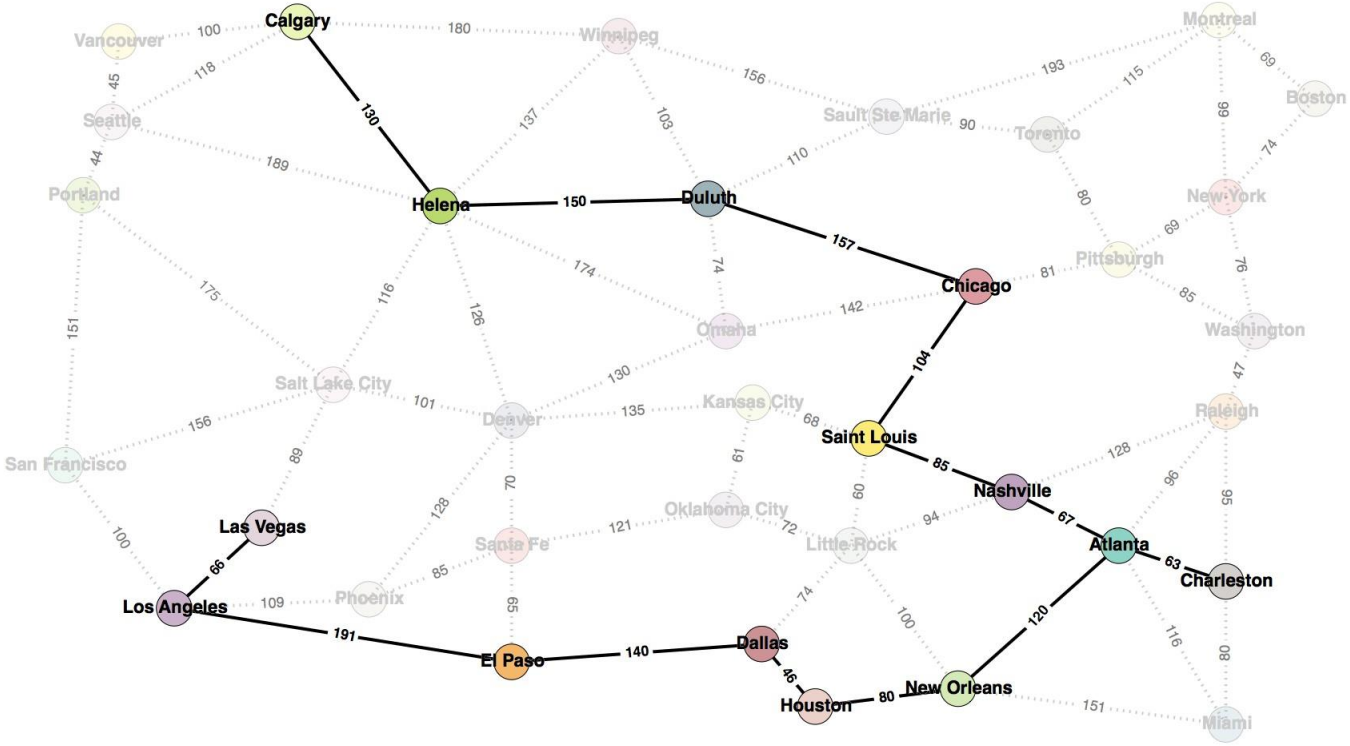

BFS

Order of Visit: Las Vegas, Los Angeles, Salt Lake City, El Paso, Phoenix, San Francisco, Denver, Helena, Portland, Dallas, Santa Fe, Kansas City, Omaha, Calgary.

# Examples using the map

**Start: Las Vegas**

**Goal: Calgary**



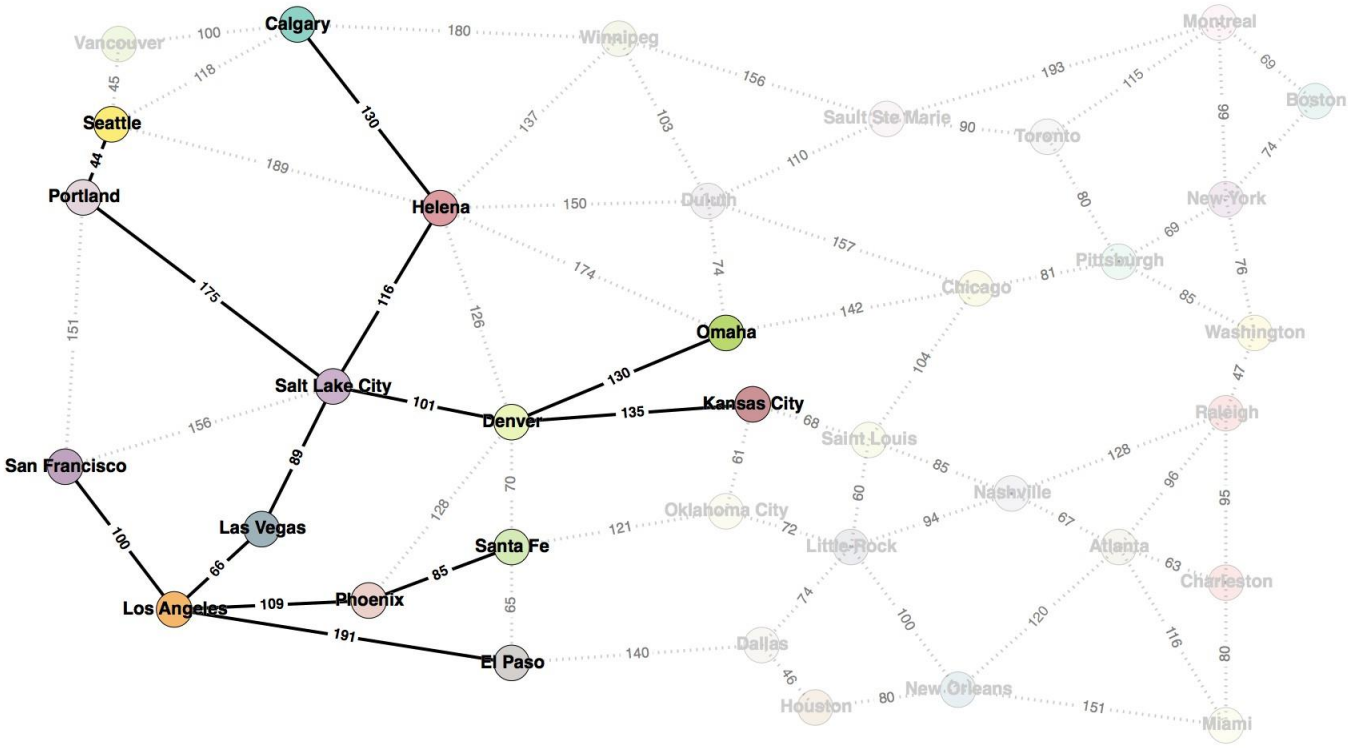

DFS

Order of Visit: Las Vegas, Los Angeles, El Paso, Dallas, Houston, New Orleans, Atlanta, Charleston, Nashville, Saint Louis, Chicago, Duluth, Helena, Calgary.

# Examples using the map

**Start: Las Vegas**

**Goal: Calgary**




UCS

Order of Visit: Las Vegas, Los Angeles, Salt Lake City, San Francisco, Phoenix, Denver, Helena, El Paso, Santa Fe, Portland, Seattle, Omaha, Kansas City, Calgary.

# Credit

- Artificial Intelligence, A Modern Approach. Stuart Russell and Peter Norvig. Third Edition. Pearson Education.