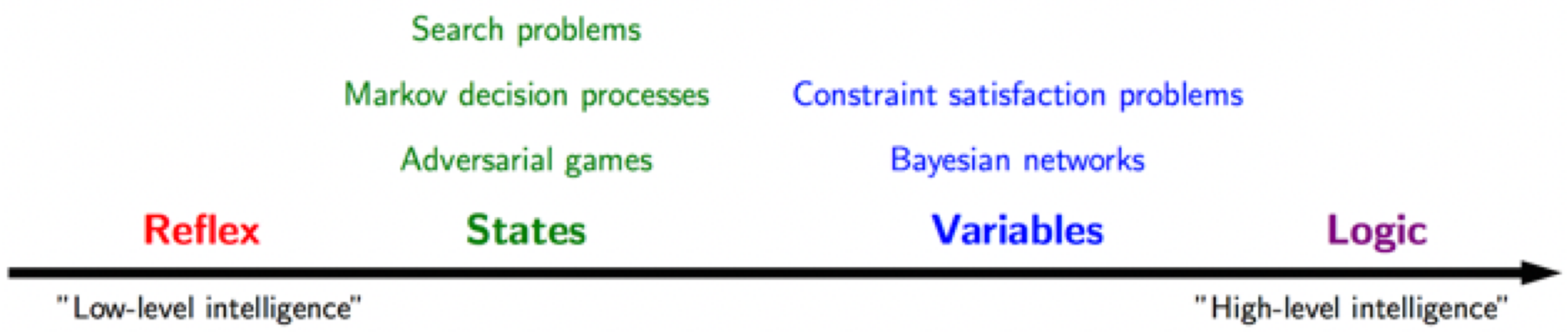
Artificial Intelligence (CS111)

Lecture 3: Uninformed Search

Credit: Ansaf Salleb-Aouissi, and "Artificial Intelligence: A Modern Approach", Stuart Russell and Peter Norvig, and "The Elements of Statistical Learning", Trevor Hastie, Robert Tibshirani, and Jerome Friedman, and "Machine Learning", Tom Mitchell.

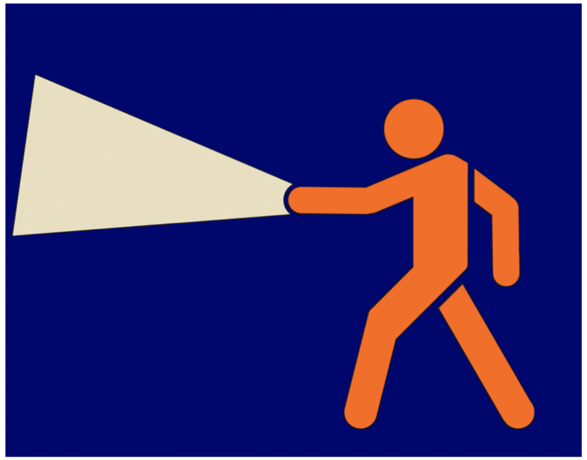
Goal-based agents

- **Reflex agents**: use a mapping from states to actions (lookup).
- **Goal-based agents**: problem solving agents or planning agents.

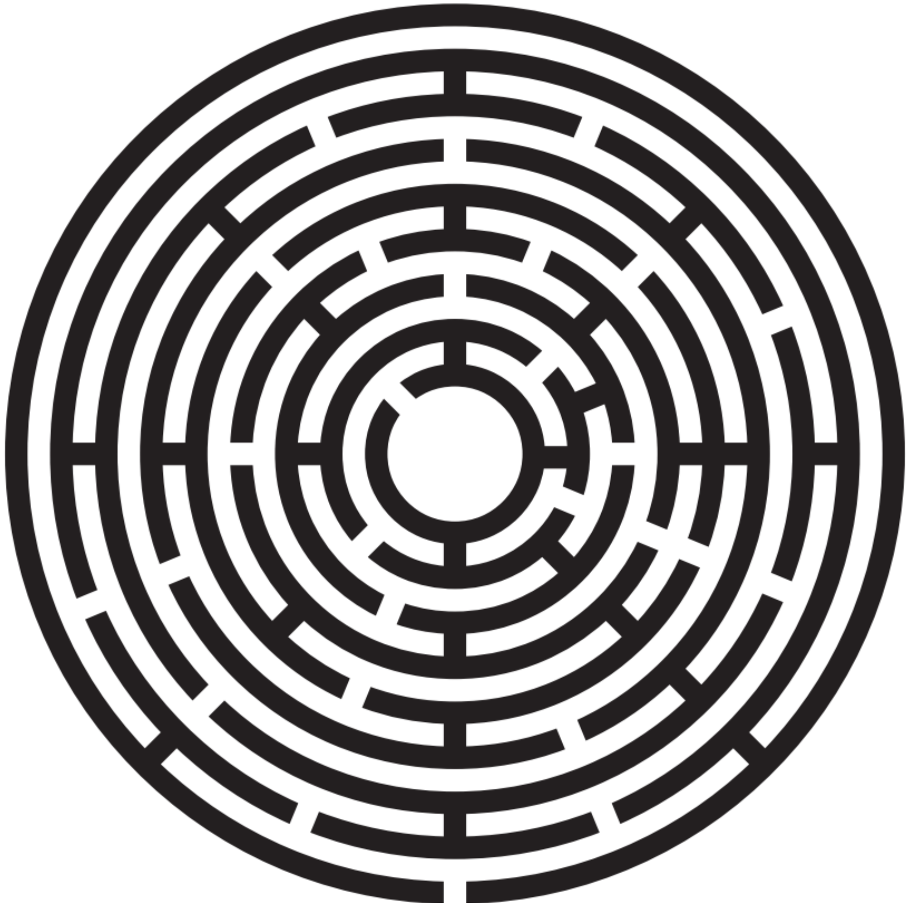


Goal-based agents

- Agents work towards a **goal**.
- Agents consider the impact of **actions** on future **states**, which means that their job is to identify the action or series of actions that lead to the goal.
- Formalized as a **search** through possible solutions.

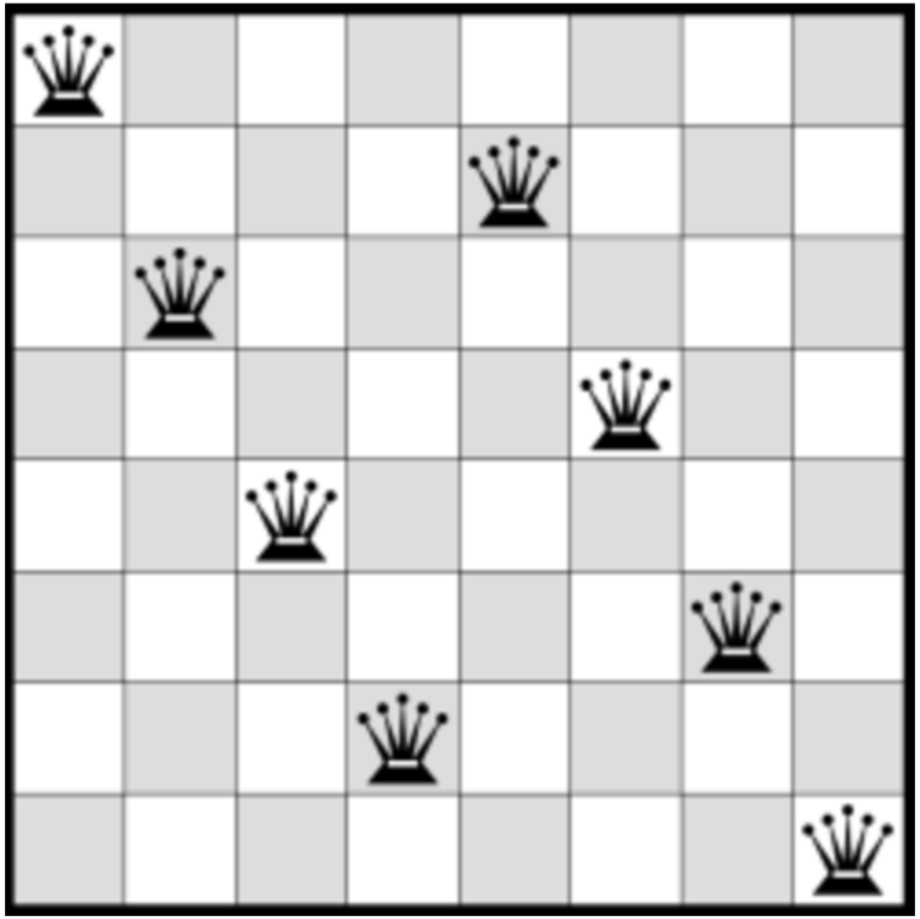


Example: Maze



Example: 8-queen problem

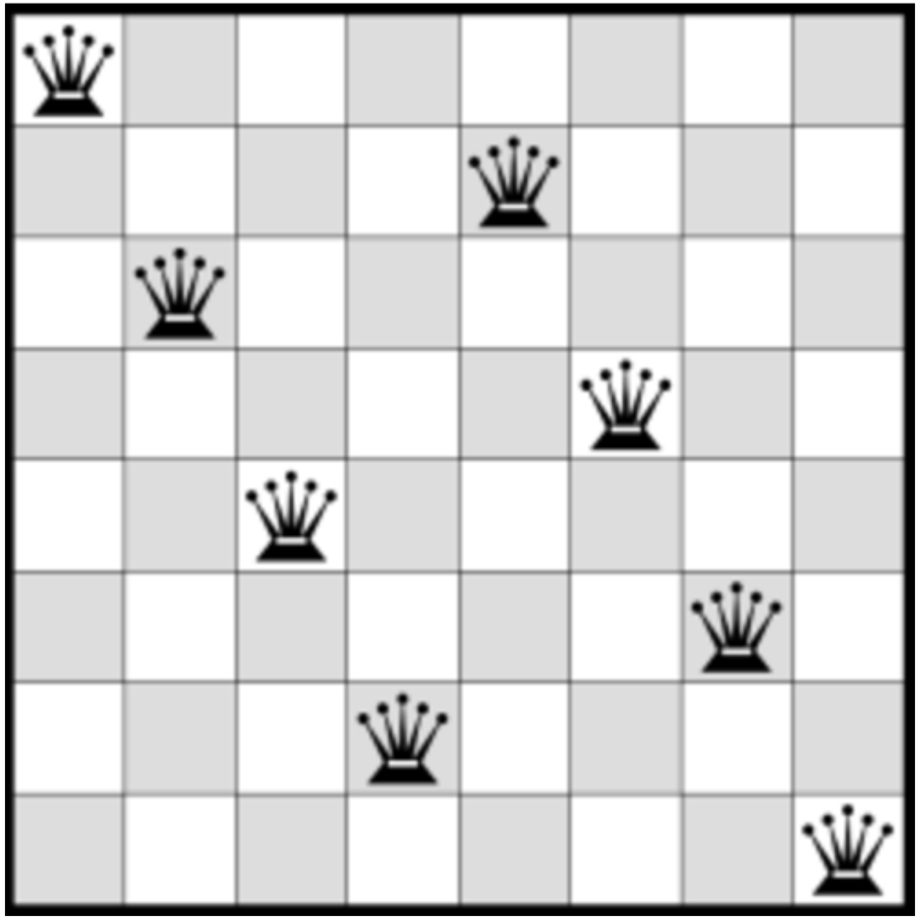
- **Objective**: On a chess board, place 8 queens so that no queen is attacking any other horizontally, vertically or diagonally.



Example: 8-queen problem

Number of possible sequences to investigate:

$$64*63*62*\ldots*57 = 1.8\times 10^{14}$$



Problem solving as search

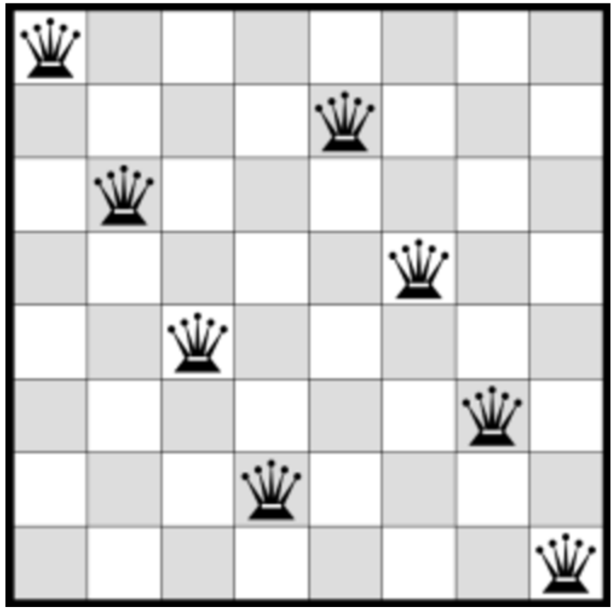
1. **Define the problem through:**
   (a) Goal formulation
   (b) Problem formulation
2. **Solving the problem as a 2-stage process:**
   (a) Search: “mental” or “offline” exploration of several possibilities
   (b) Execute the solution found

Problem formulation

- **Initial state**: the state in which the agent starts.
- **States**: all states reachable from the initial by any sequence of actions. **(State space)**
- **Actions**: possible actions available to the agent. At a state *s*, *Actions*(*s*) returns the set of actions that can be executed in state *s*. **(Action space)**
- **Transition model**: a description of what each action does *Results(s*, *a*).
- **Goal test**: determine if a given state is a goal state.
- **Path cost**: function that assigns a numeric cost to each path w.r.t. performance measure.

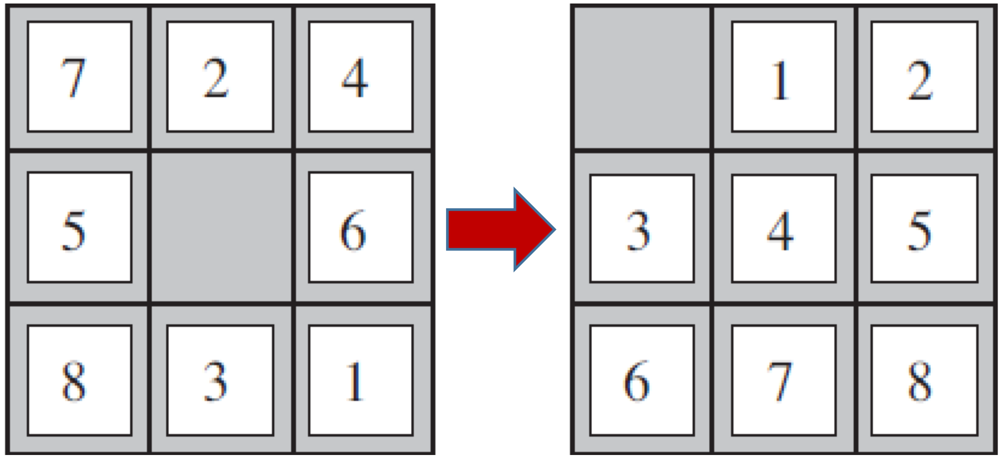
Formulation: 8-queen problem

- **States**: all arrangements of 8 queens on the board.
- **Initial state**: no queen on the board.
- **Actions**: add a queen to any empty square.
- **Transition**: updated board.
- **Goal test**: 8 queens on board without attacked?



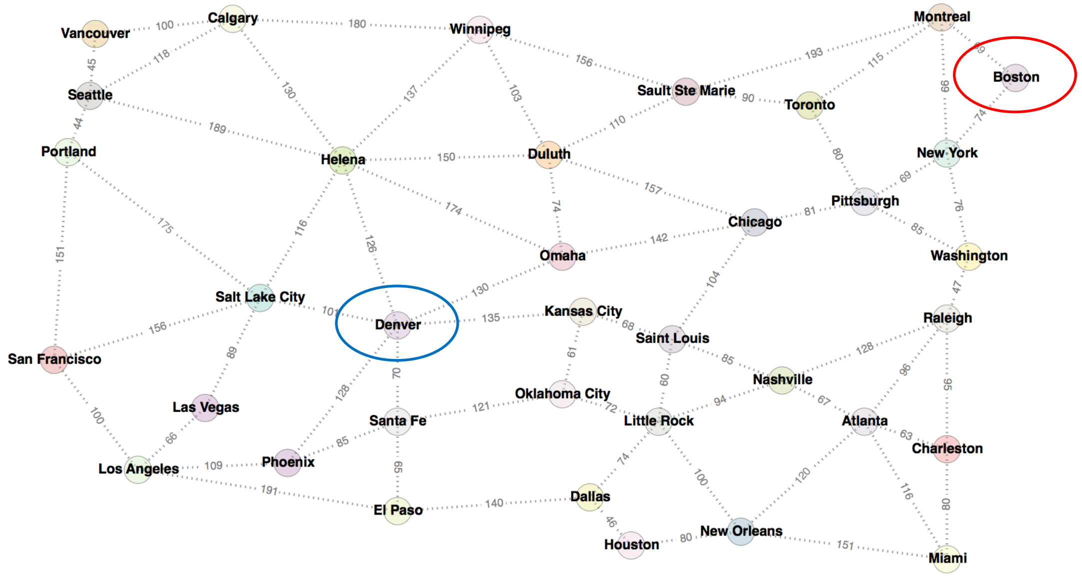
Formulation: 8-puzzle problem

- **States**: any configuration of the 8 tiles on the 3x3 grid
- **Initial state**: any state (e.g., the configuration of the Left).
- **Actions**: move Left, Right, Up or Down.
- **Transition**: Given a state and an action, returns resulting state.
- **Goal state**: the configuration of the Right?
- **Path cost**: #moves.



Formulation: Routing problem

- **States**: In City where City ∈ {New York, San Francisco, Denver,...}.
- **Initial state**: In Boston
- **Actions**: walk to the adjacent city.
- **Transition**: new city.
- **Goal test**: In Denver
- **Path cost**: path length in kilometers.

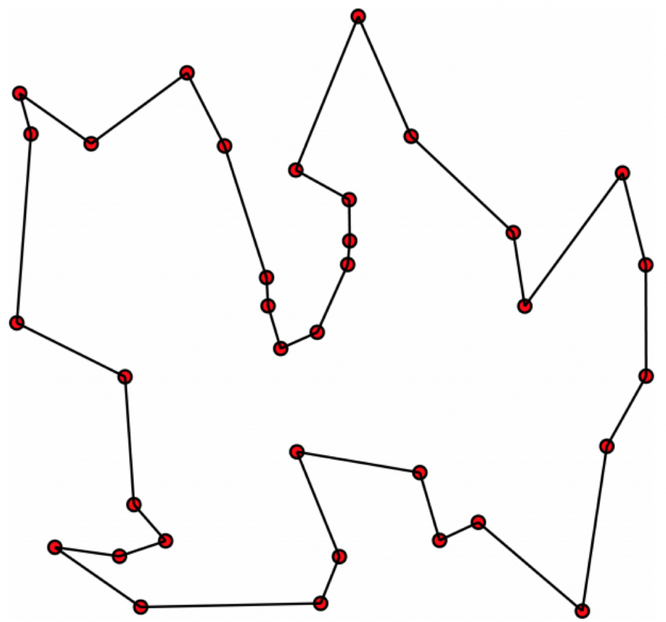


Real-world example: Route finding

- **Route finding problem**: typically our example of map search, where we need to go from location to location using links or transitions. Example of applications include tools for driving directions in websites, in-car systems, etc.

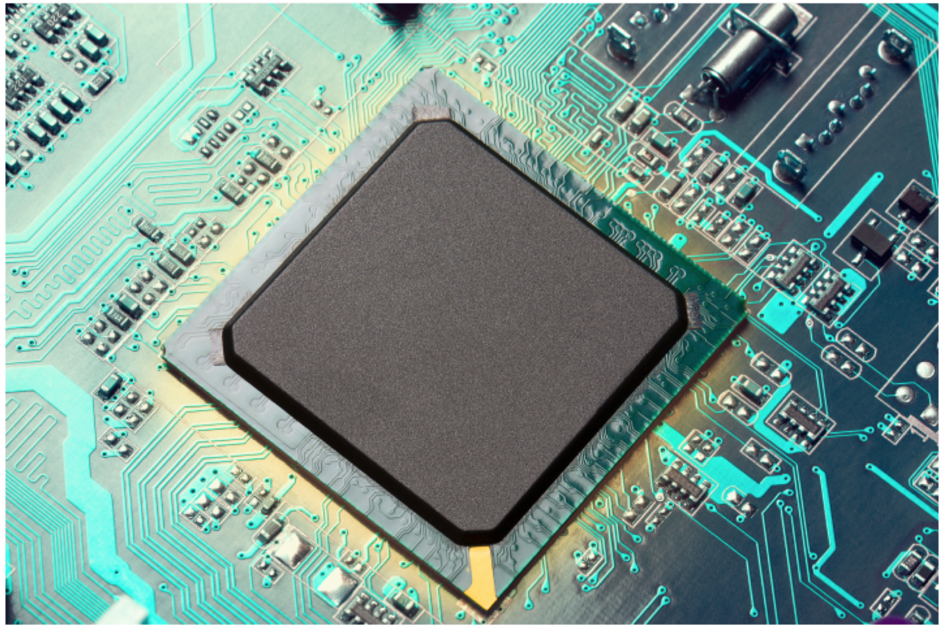
Real-world example: TSP

- **Traveling salesperson problem (TSP)**: Find the shortest tour to visit each city exactly once.



Real-world example: VLSI layout

- **VLSI layout**: position million of components and connections on a chip to minimize area, shorten delays, and they don't overlap and leave space to wiring, which is a complex problem.

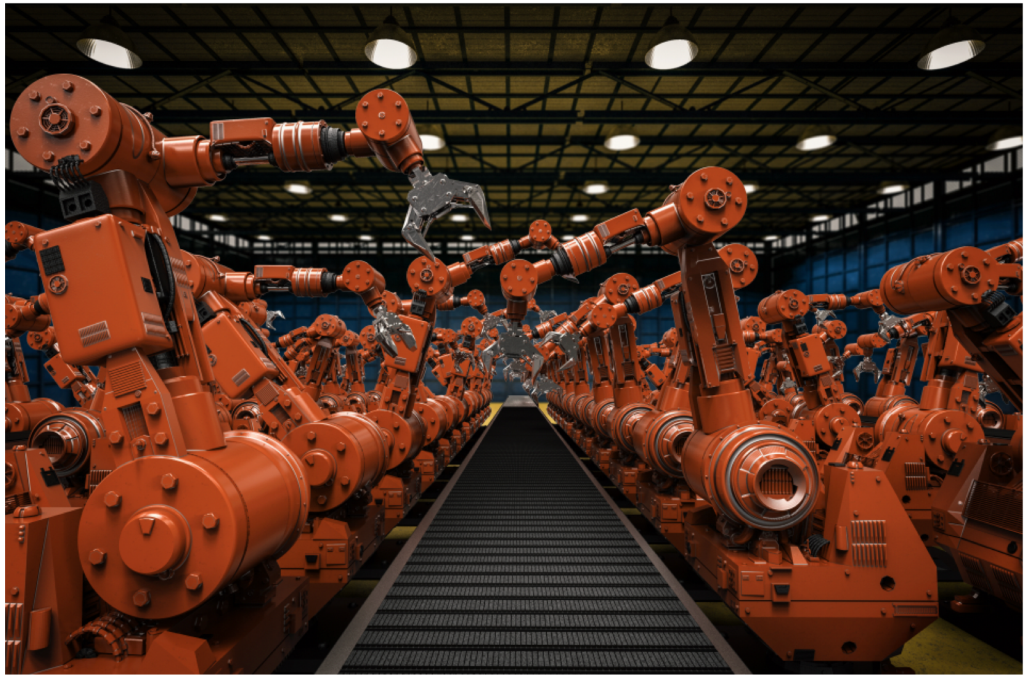


Real-world example: Robot navigation

- **Robot navigation**: Special case of route finding for robots with no specific routes or connections. The robot navigates in 2D or 3D space where the state space and action space are potentially infinite.

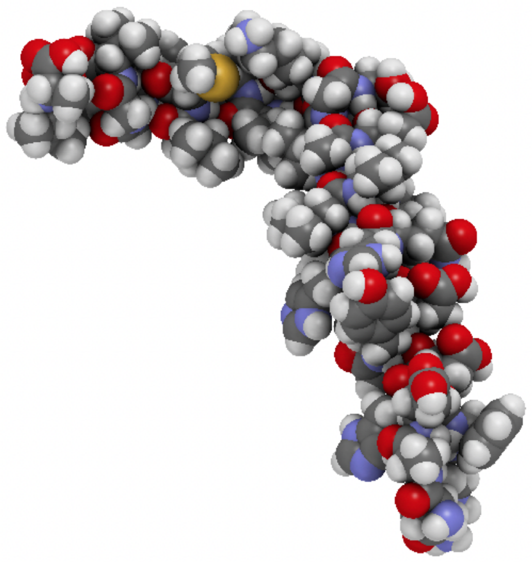
Real-world example: Automatic assembly

- **Automatic assembly sequencing**: find an order to assemble parts of an object, which is in general a difficult and expensive geometric search.



Real-world example: Protein design

- **Protein design**: find a sequence of amino acids that will fold into a 3D protein with the right properties to cure some disease.



State space vs. search space

- **State space**: a *physical* configuration
- **Search space**: an *abstract* configuration represented by a search tree or graph of possible solutions.
- **Search tree models the sequence of actions**.
  - Root: initial state
  - Branches: actions
  - Nodes: results from actions. A node has: parent, children, depth, path cost, associated state in the state space.
- **Expand**: A function that given a node, creates all children nodes

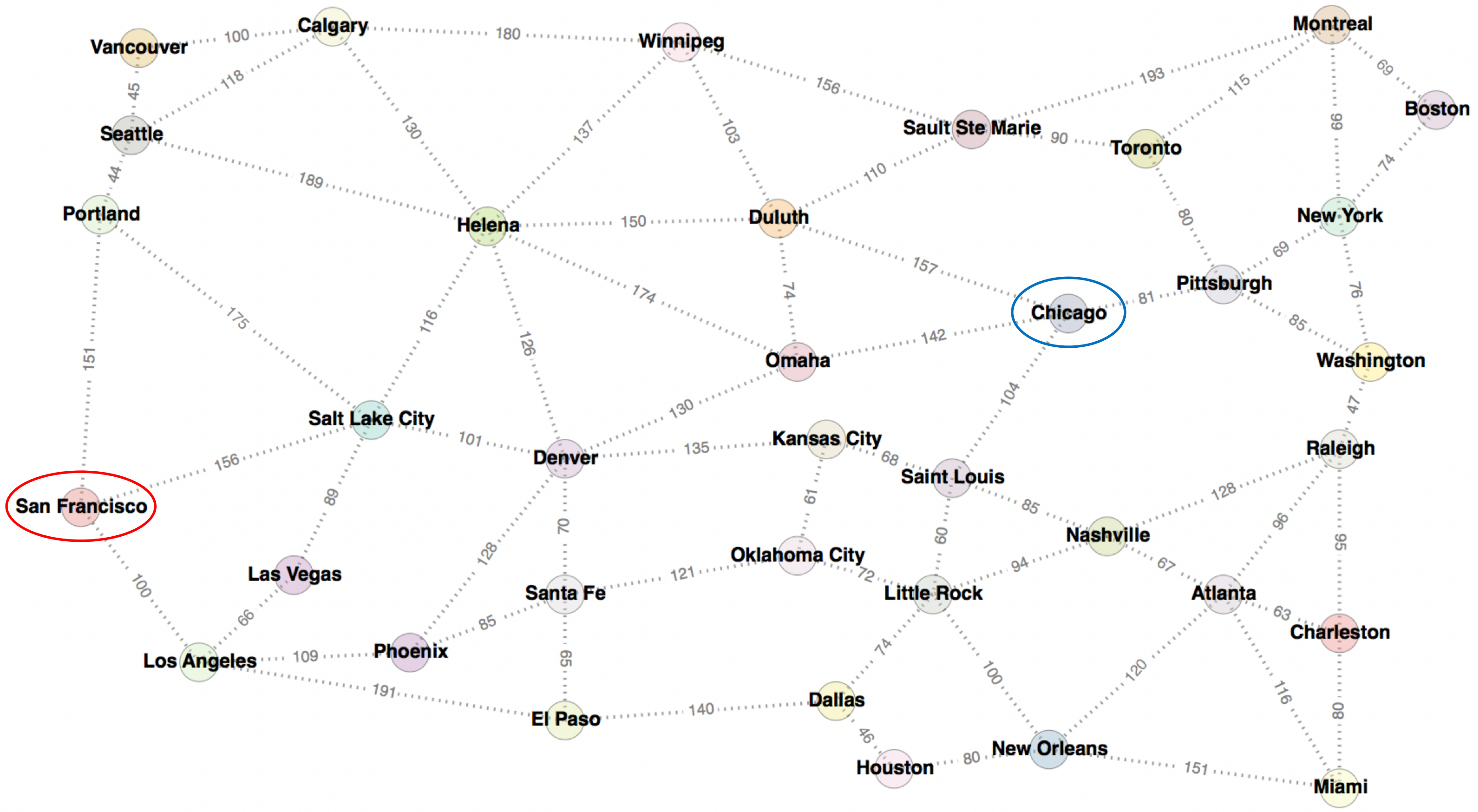
Search Space Regions

- The search space is divided into three regions:
  - 1. **Explored** (a.k.a. Closed List, Visited Set)
  - 2. **Frontier** (a.k.a. Ready list, Open List, the Fringe)
  - 3. **Unexplored**.
- The **essence of search** is moving nodes from regions (3) to (2) to (1), and the essence of search strategy is deciding the order of such moves.

Tree search

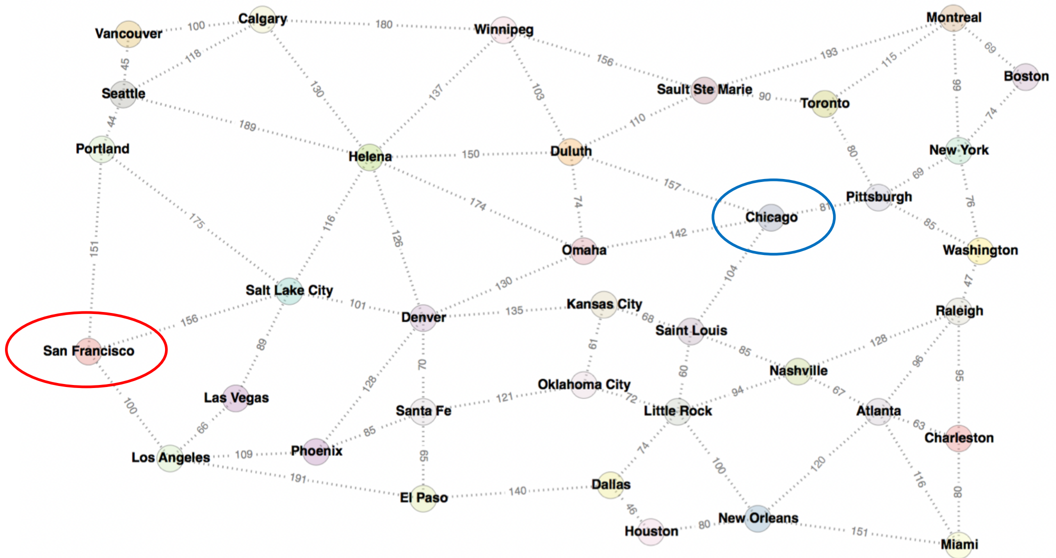
```
function TREE-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    initialize frontier with initialState

    while not frontier.isEmpty():
        state = frontier.remove()

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            frontier.add(neighbor)

    return FAILURE
```

Examples of search agents

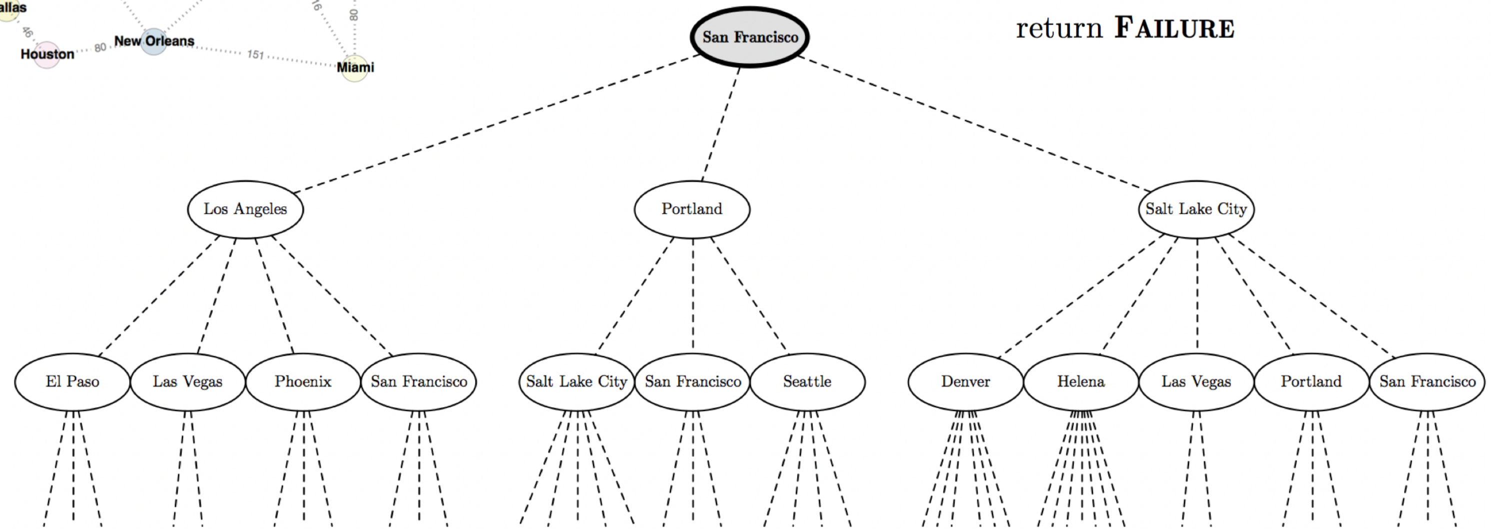


Let’ s show the first steps in growing the search tree to find a route from San Francisco to another city

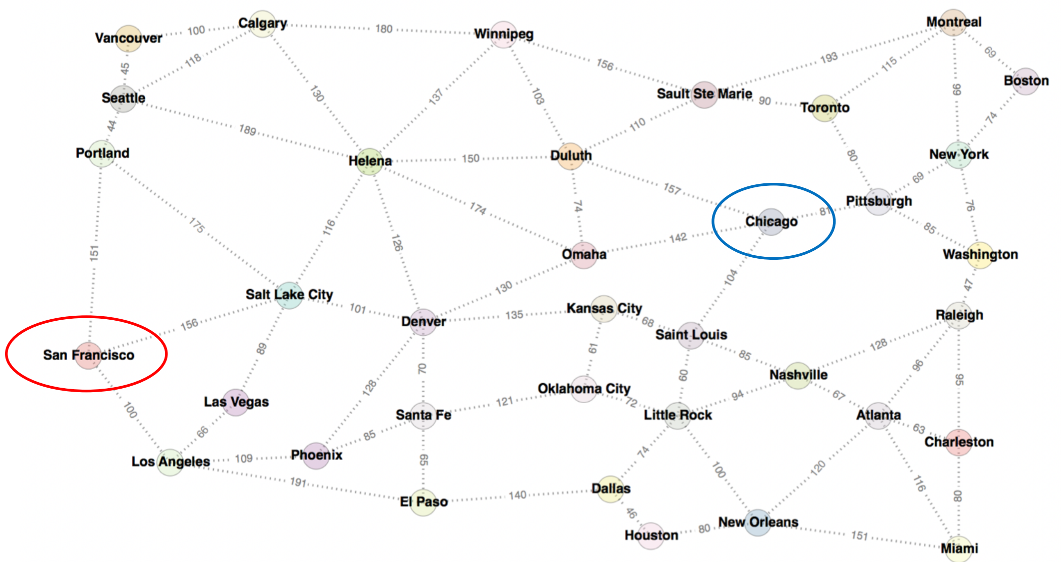
Examples of search agents



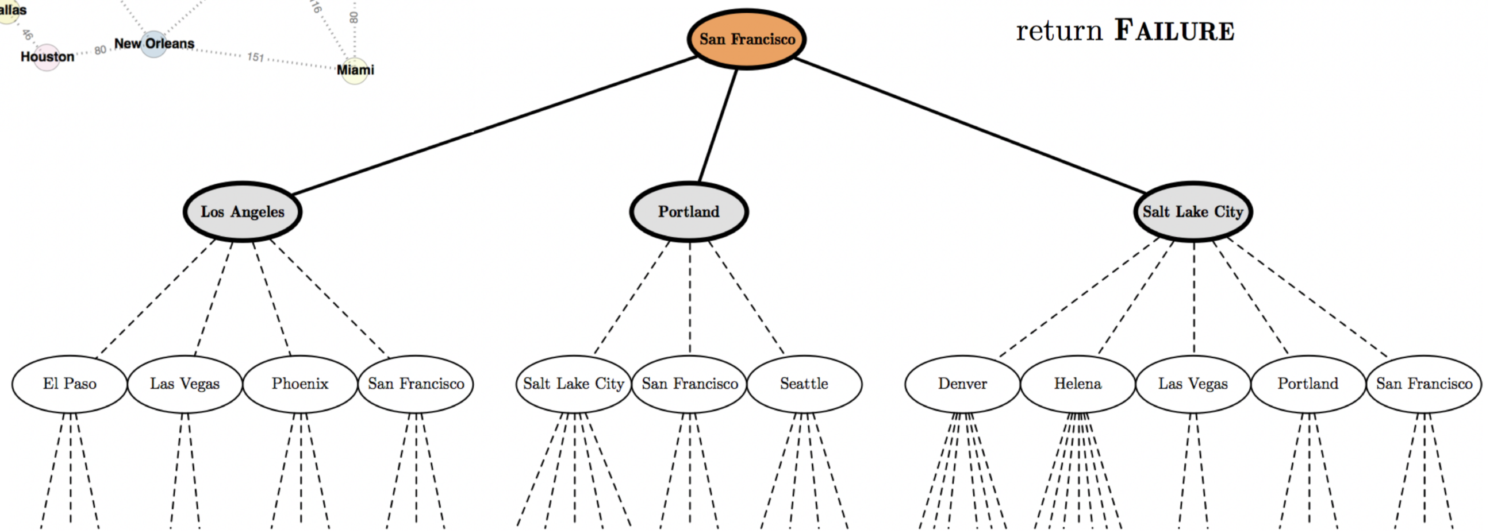

```
function TREE-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    initialize frontier with initialState

    while not frontier.isEmpty():
        state = frontier.remove()

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            frontier.add(neighbor)

    return FAILURE
```



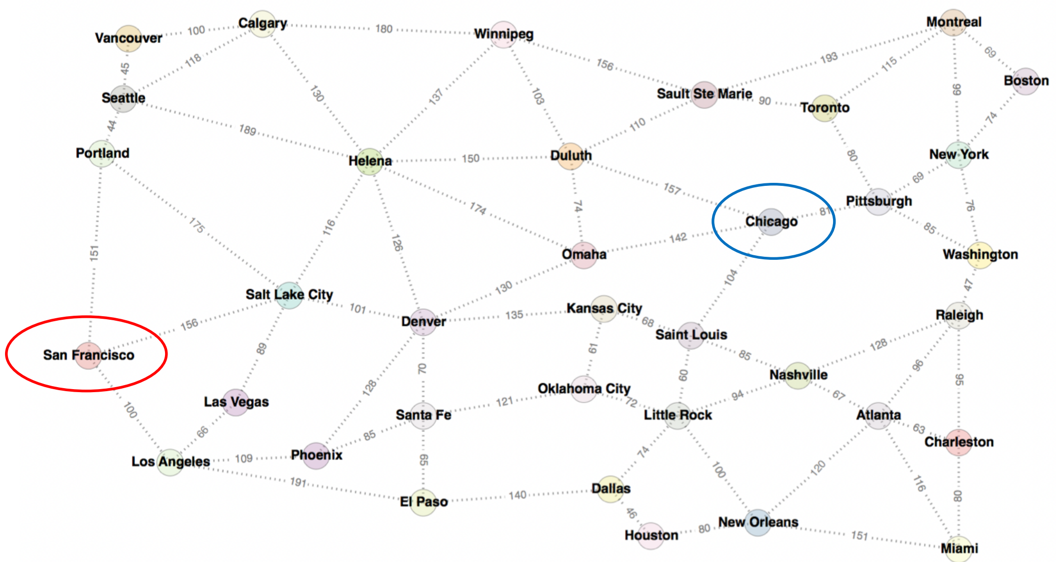
Examples of search agents




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

    initialize frontier with initialState

    while not frontier.isEmpty():
        state = frontier.remove()

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            frontier.add(neighbor)

    return FAILURE
```



Examples of search agents



function TREE-SEARCH(initialState, goalTest)
  returns **SUCCESS** or **FAILURE :**

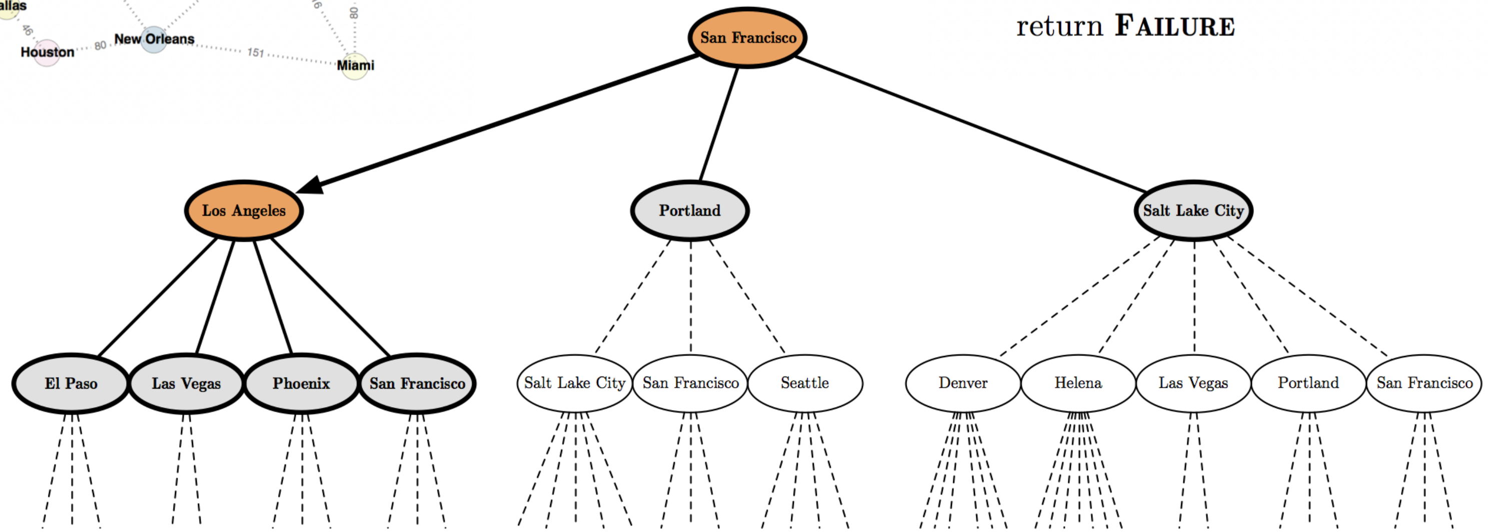
  initialize frontier **with** initialState

  while not frontier.isEmpty():
    state = frontier.remove()

    if goalTest(state):
      return **SUCCESS**(state)

    for neighbor **in** state.neighbors():
      frontier.add(neighbor)

  return **FAILURE**



Graph search

How to handle repeated states?

Graph search

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

    initialize frontier with initialState
    explored = Set.new()

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

        if goalTest(state):
            return SUCCESS(state)

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

    return FAILURE
```

Search strategies

- A strategy is defined by picking the **order of node expansion**
- Strategies are evaluated along the following dimensions:
  - **Completeness**: Does it always find a solution if it exists?
  - **Time complexity**: # nodes generated/expanded.
  - **Space complexity**: maximum # nodes in memory.
  - **Optimality**: Does it always find the least-cost solution?

Search strategies

In particular, time and space complexity are measured regarding:

- $\boldsymbol{b}$ – maximum branching factor of the search tree (actions per state).
- $\boldsymbol{d}$ – depth of the solution.
- $\boldsymbol{m}$ – maximum depth of the state space (may be ∞) (also noted sometimes ***D***).

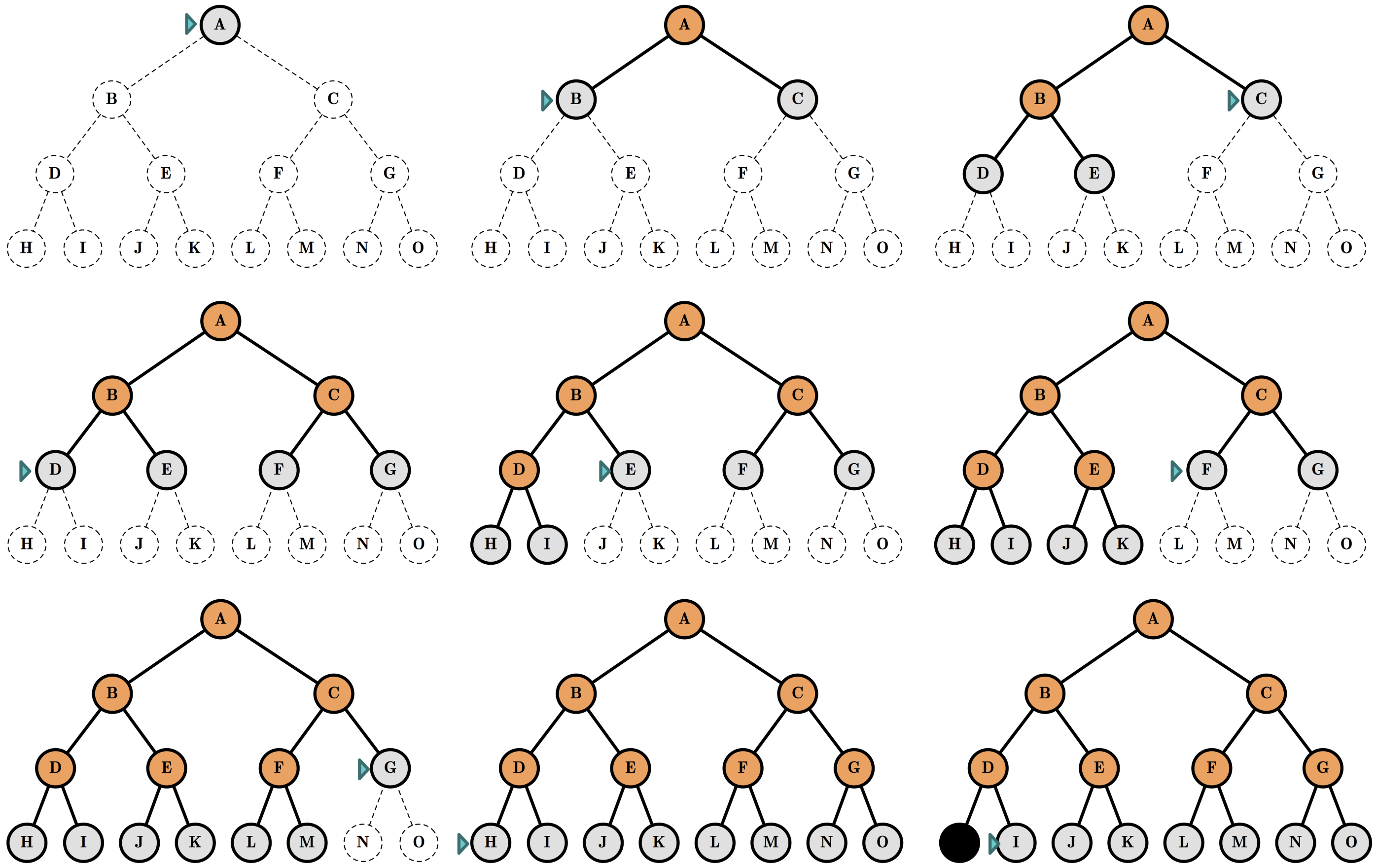
Two kinds of search: **Uninformed** and **Informed**.

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: Pseudo-code

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

    initialize frontier with initialState
    explored = Set.new()

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

        if goalTest(state):
            return SUCCESS(state)

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

    return FAILURE
```

```
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: PF Metrics

- **Complete:** Yes (if b is finite)
- **Time:** $1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$
- **Space:** $O(b^d)$
- **Optimal:** Yes (if cost = 1 per step).
- **Implementation:** frontier: FIFO (Queue)

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

BFS: PF Metrics

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

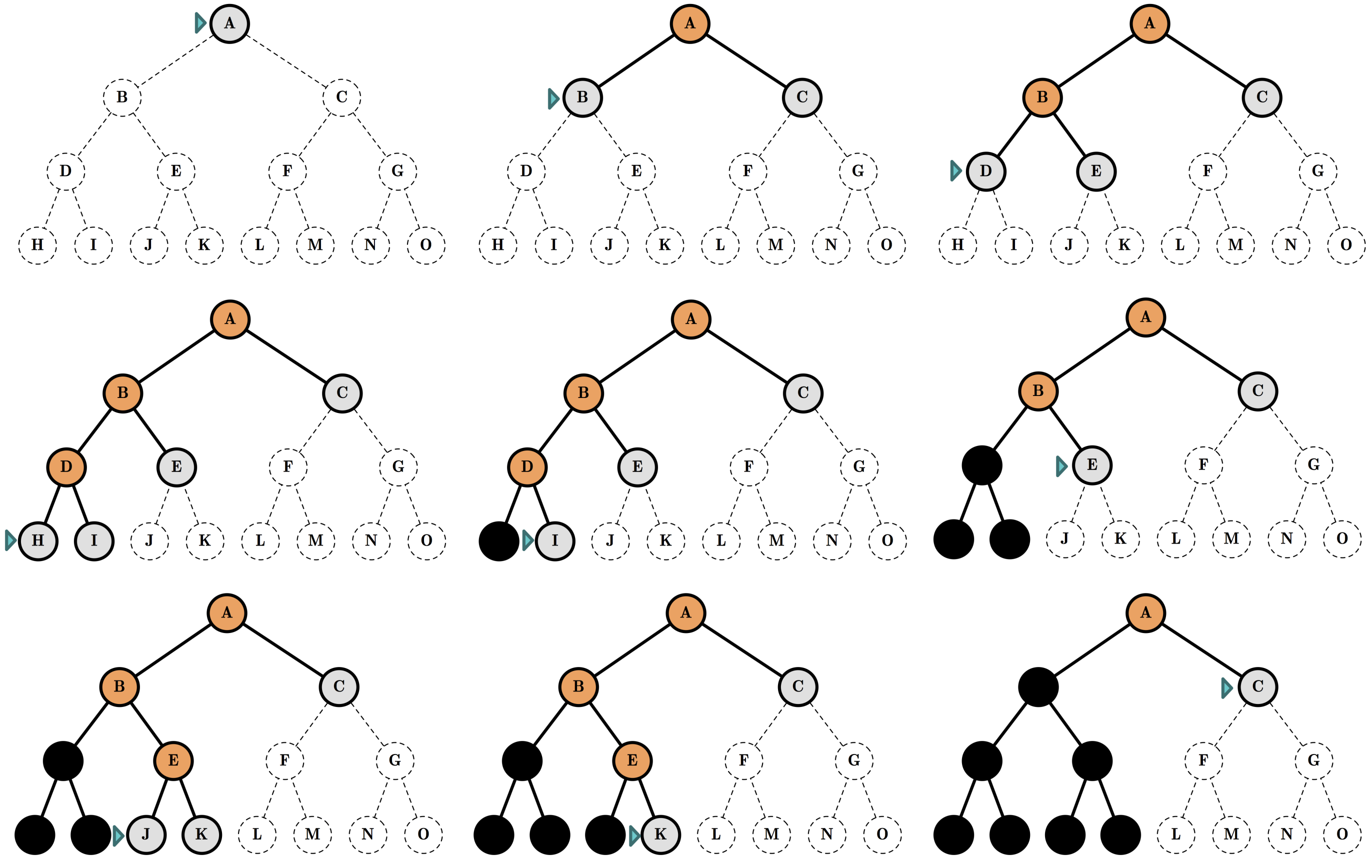
How bad is BFS?

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

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

Depth-first search (DFS)

- DFS: Expand **deepest** first.



Note: Once a node is expanded, it is removed from memory asap all its children are explored.

DFS: Pseudo-code

```
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
```

```
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: PF Metrics

- **Complete:**
  No: fails in infinite-depth spaces, spaces with loops.
  Modify to avoid repeated states along path: complete in finite spaces
- **Time:** $1 + b + b^2 + b^3 + \cdots + b^m = O(b^m)$
  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: PF Metrics

How bad is DFS?

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

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.

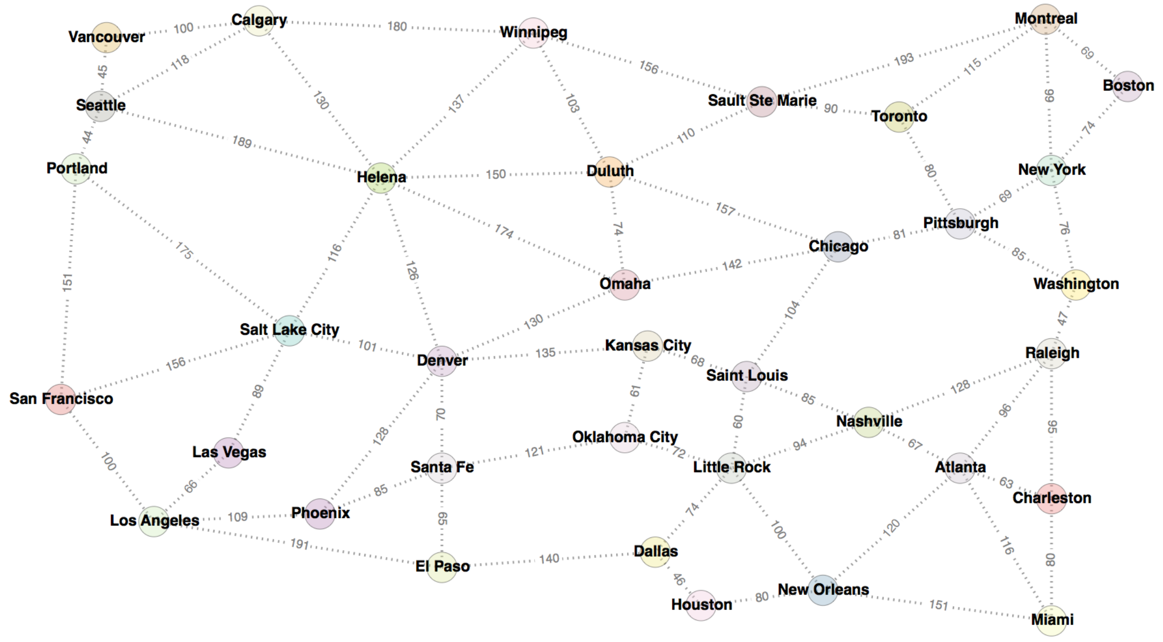
We go down from 10 exabytes in BFS to 156 kilobytes in DFS!

Depth-limited search (DLS)

- DFS with depth limit L (nodes at level L has no successors).
- Select some limit in depth to explore with DFS

Depth-limited search (DLS)

- 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 search (IDS)

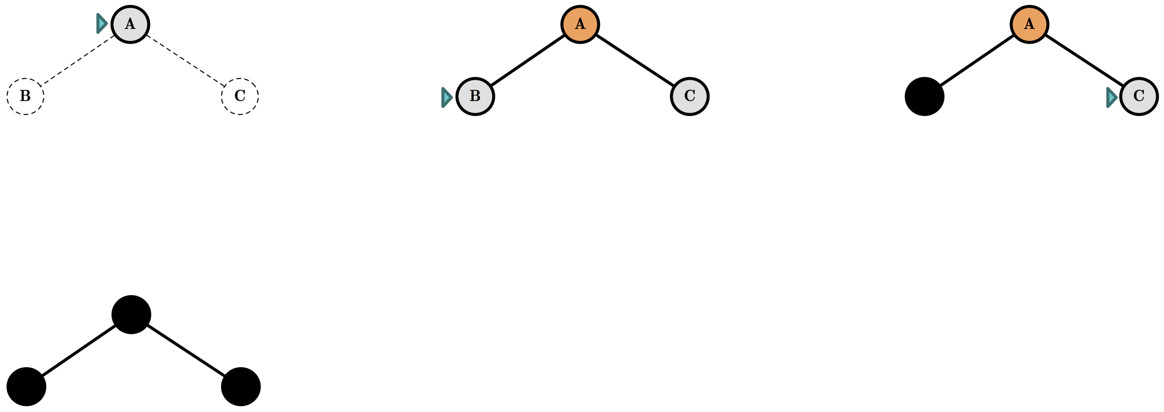
- Combines the benefits of BFS and DFS.
- Idea: Iteratively increase the search limit until the depth of the shallowest solution *d* is reached.
- Applies **DLS with increasing limits**.
- The algorithm will stop if a solution is found or if DLS returns a failure (no solution).
- Because most of the 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 search (IDS)

- Limit = 0

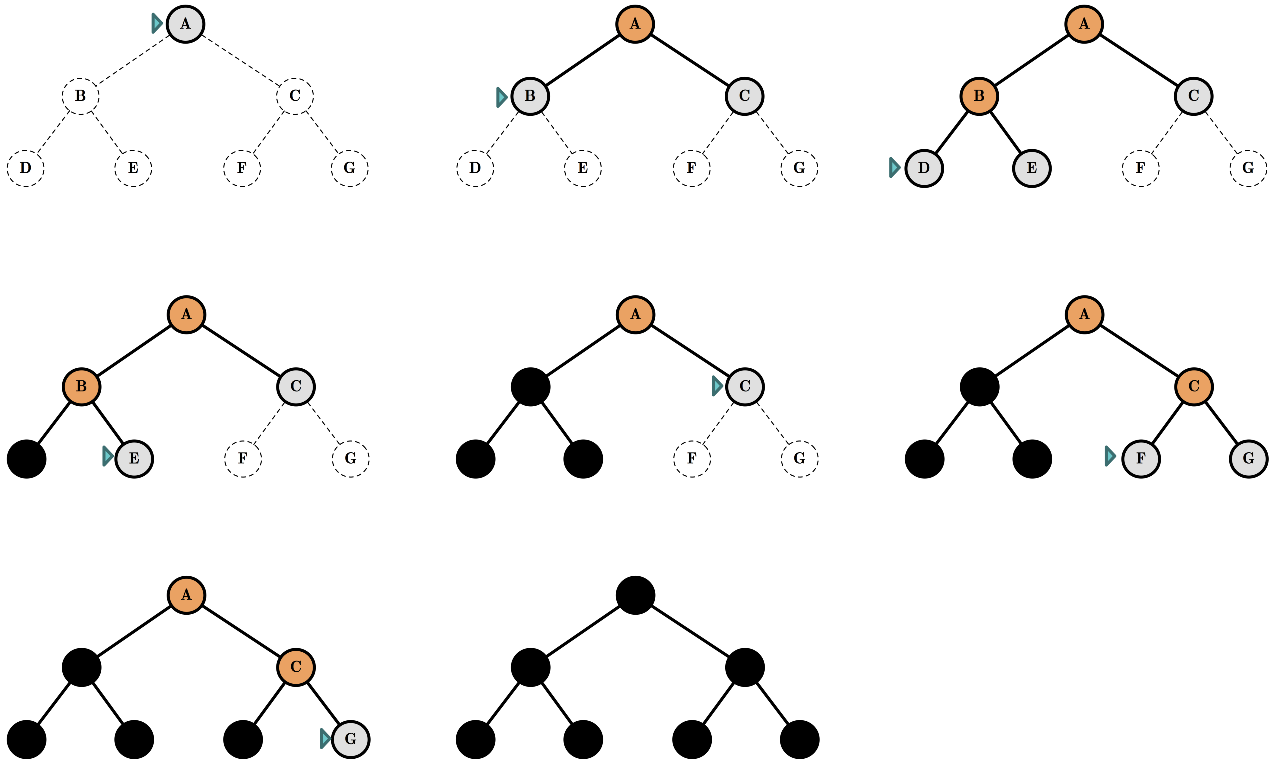
Iterative deepening search (IDS)

- Limit = 1



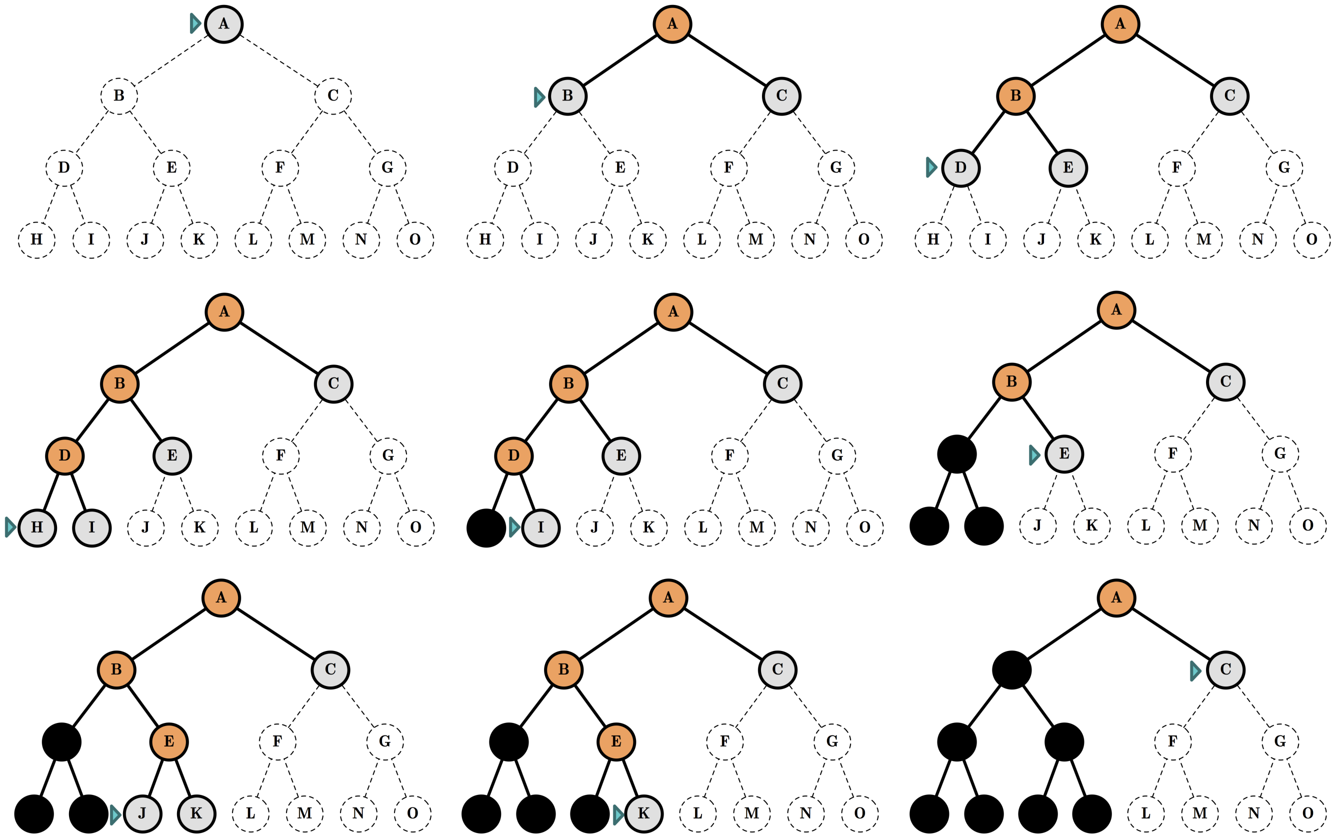
Iterative deepening search (IDS)

- Limit = 2



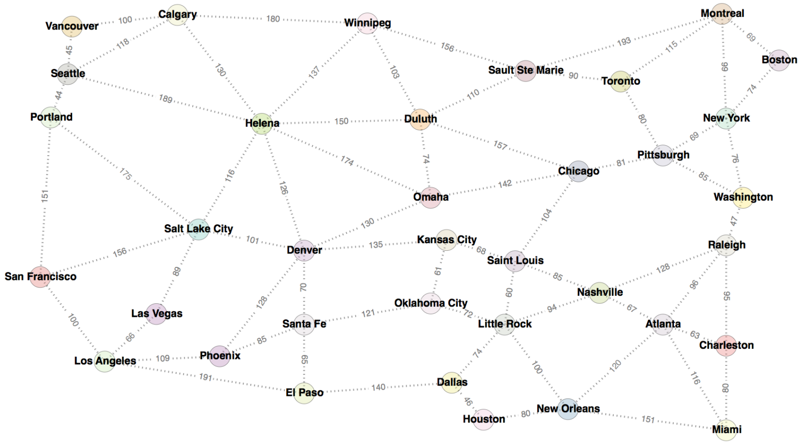
Iterative deepening search (IDS)

- Limit = 3



Uniform-cost search (UCS)

- 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.

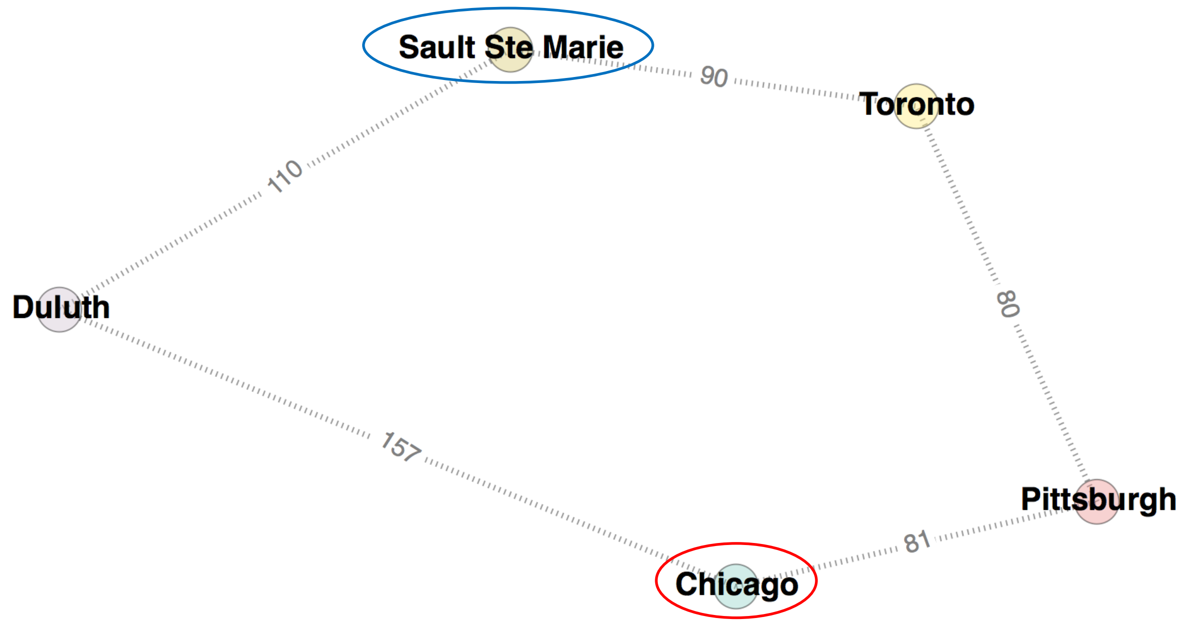


UCS: Pseudo-code

```
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
```

```
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
```

UCS: example



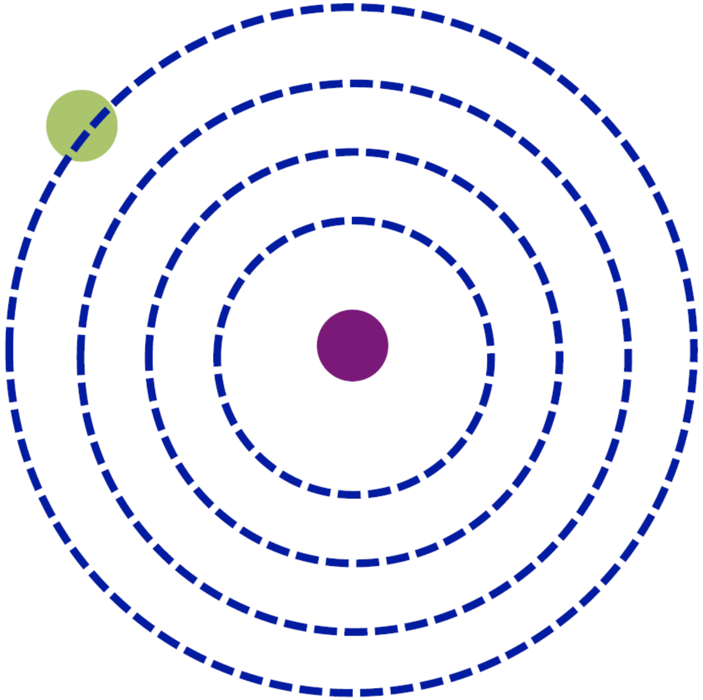
Go from Chicago to Sault Ste Marie. Using BFS, we would find Chicago-Duluth-Sault Ste Marie. However, using UCS, we would find Chicago-Pittsburgh-Toronto-Sault Ste Marie, which is actually the shortest path!

UCS: PF Metrics

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

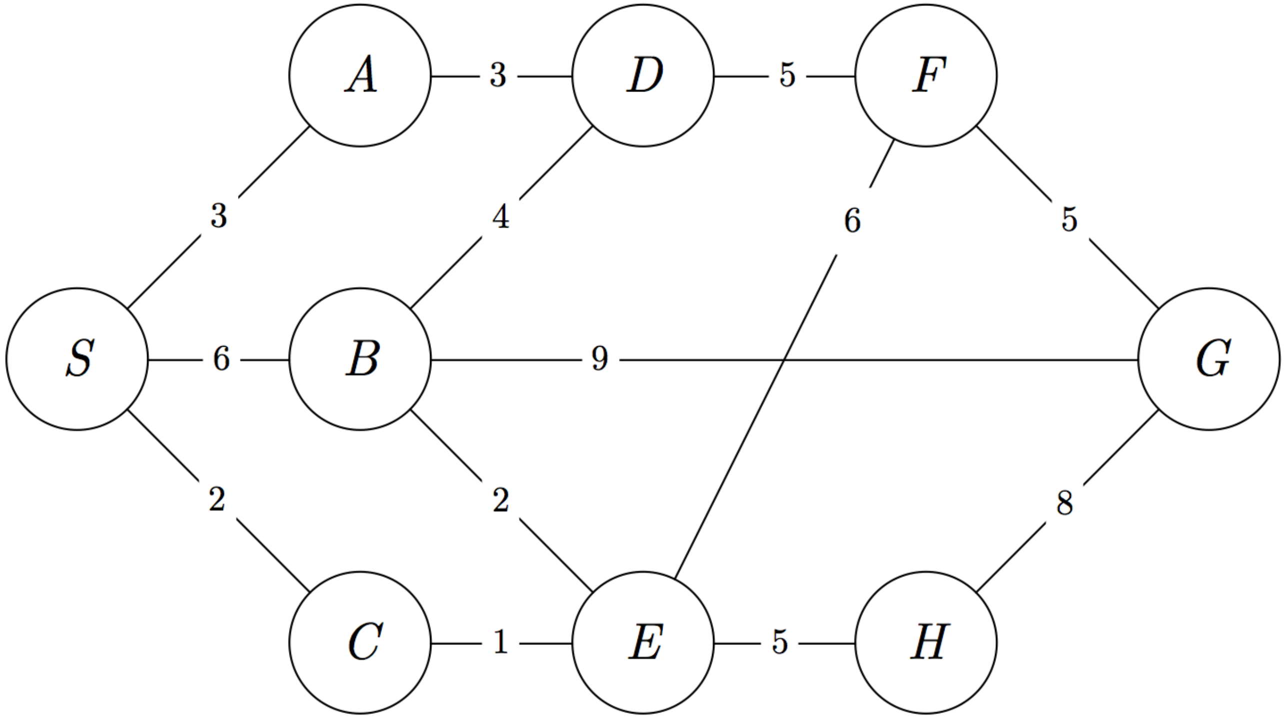
UCS: PF Metrics

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

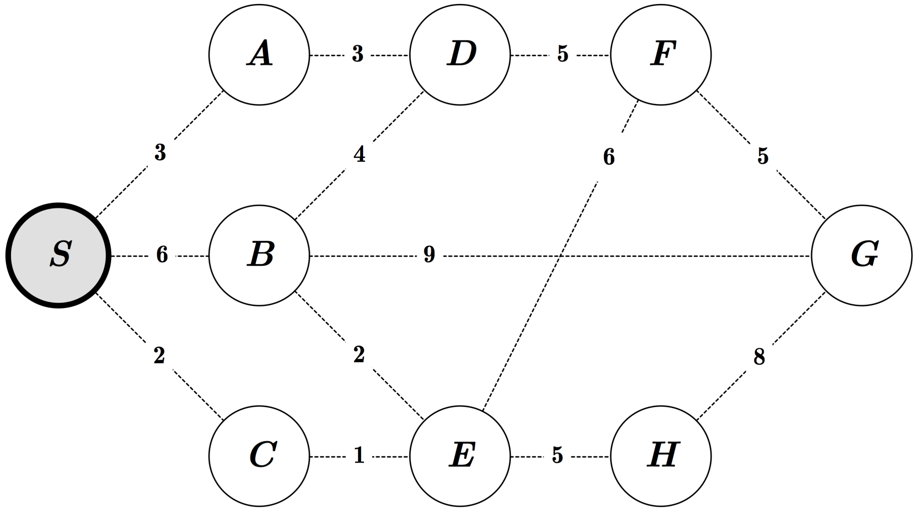


Toy Example

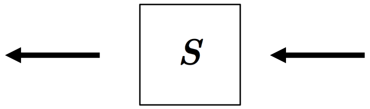
- **Question**: What is the order of visits of the nodes and the path returned by BFS, DFS and UCS?



Toy Example: BFS

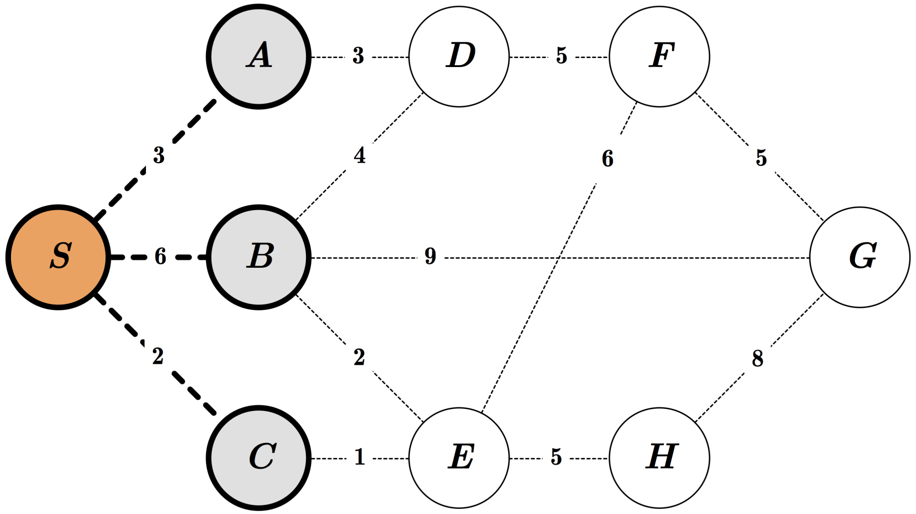


Queue:



Order of Visit:

Toy Example: BFS



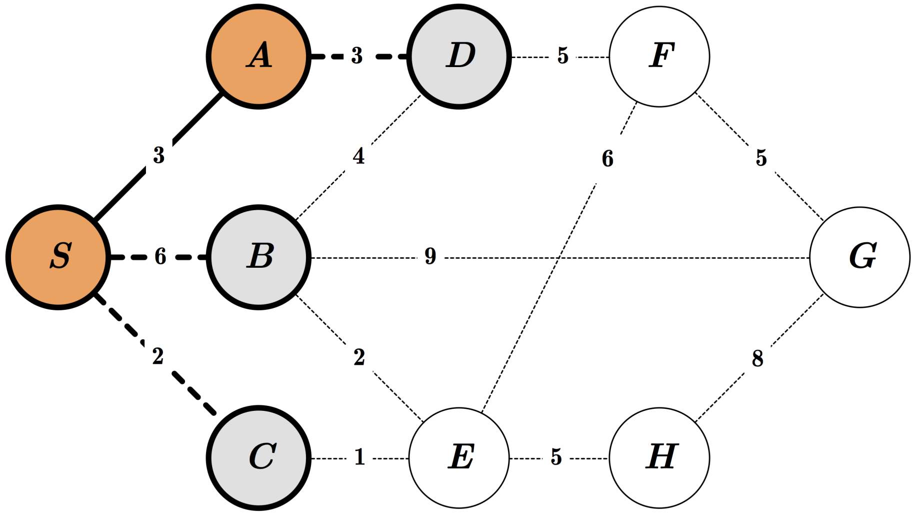
Queue:

S	A	B	C

Order of Visit:

S

Toy Example: BFS



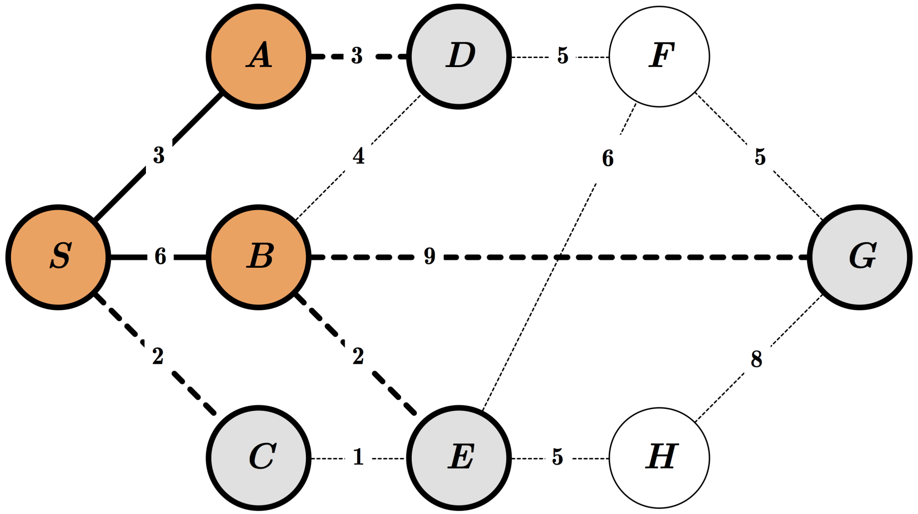
Queue:

S	A	B	C	D

Order of Visit:

S A

Toy Example: BFS



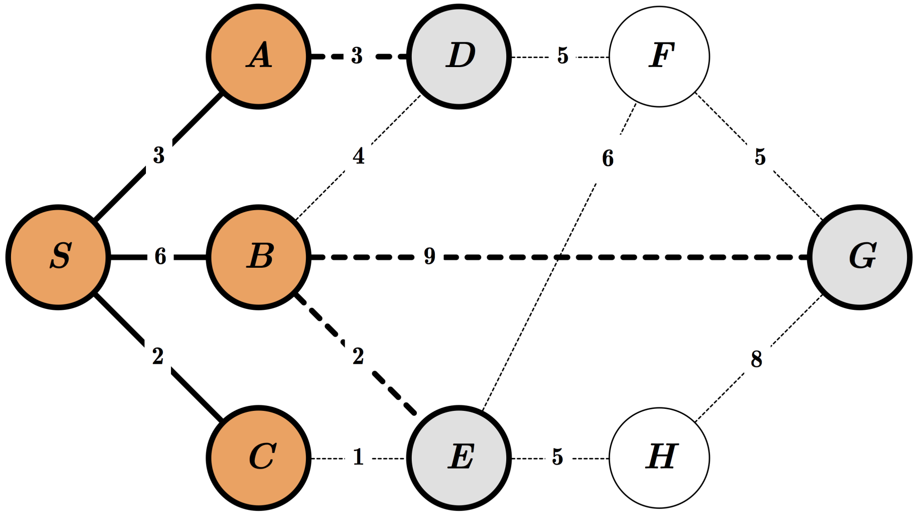
Queue:

S	A	B	C	D	E	G

Order of Visit:

S A B

Toy Example: BFS



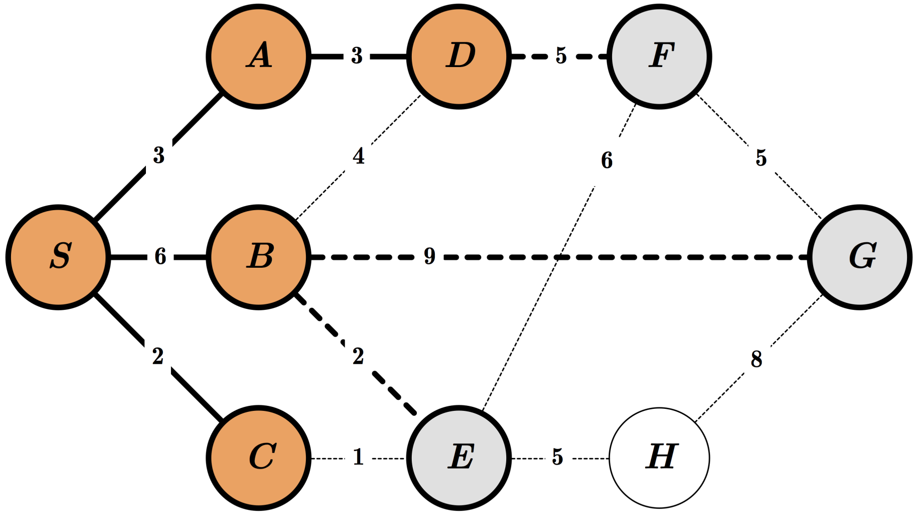
Queue:

S	A	B	C	D	E	G

Order of Visit:

S A B C

Toy Example: BFS



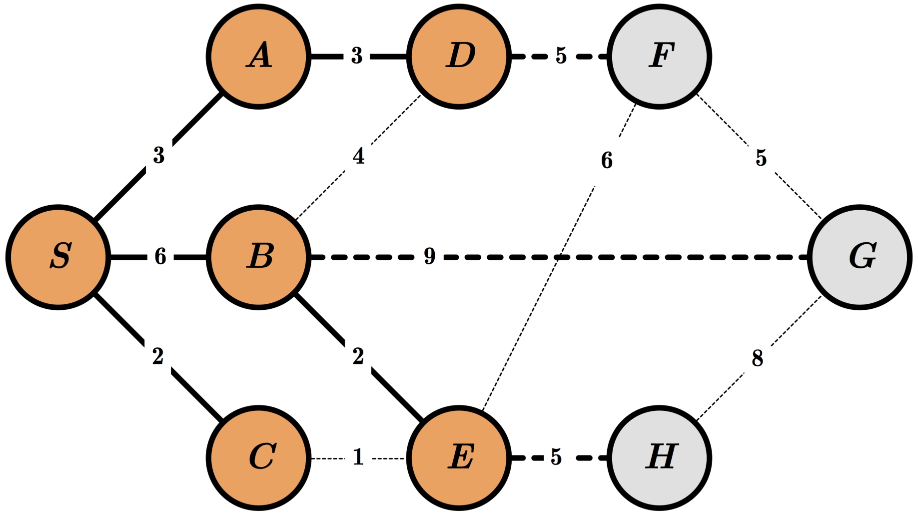
Queue:

S	A	B	C	D	E	G	F

Order of Visit:

S A B C D

Toy Example: BFS



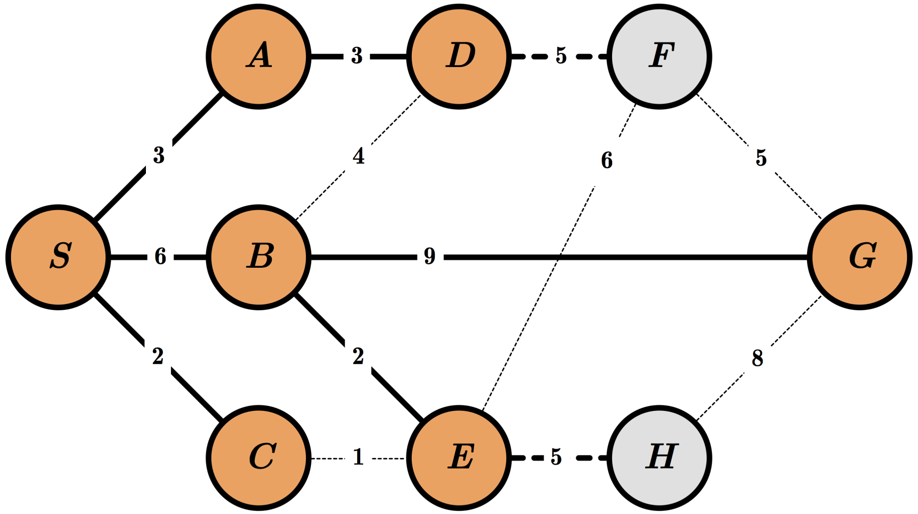
Queue:

S	A	B	C	D	E	G	F	H

Order of Visit:

S A B C D E

Toy Example: BFS



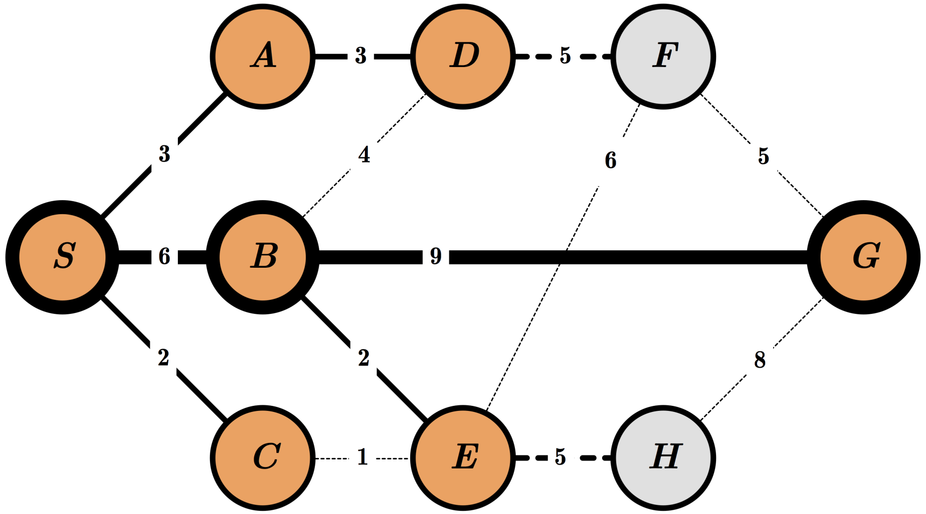
Queue:

S	A	B	C	D	E	G	F	H

Order of Visit:

S A B C D E G

Toy Example: BFS



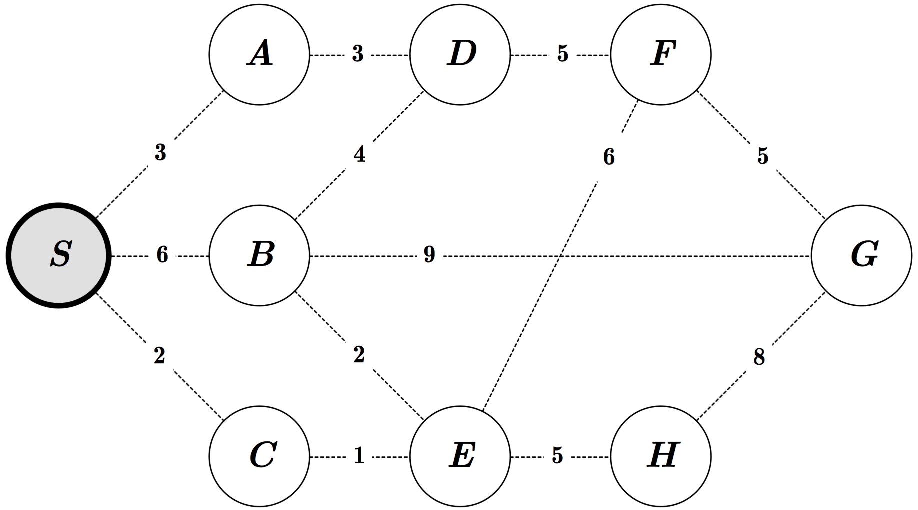
Queue:

S	A	B	C	D	E	G	F	H

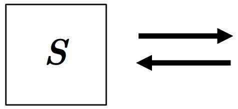
Order of Visit:

S A B C D E G

Toy Example: DFS

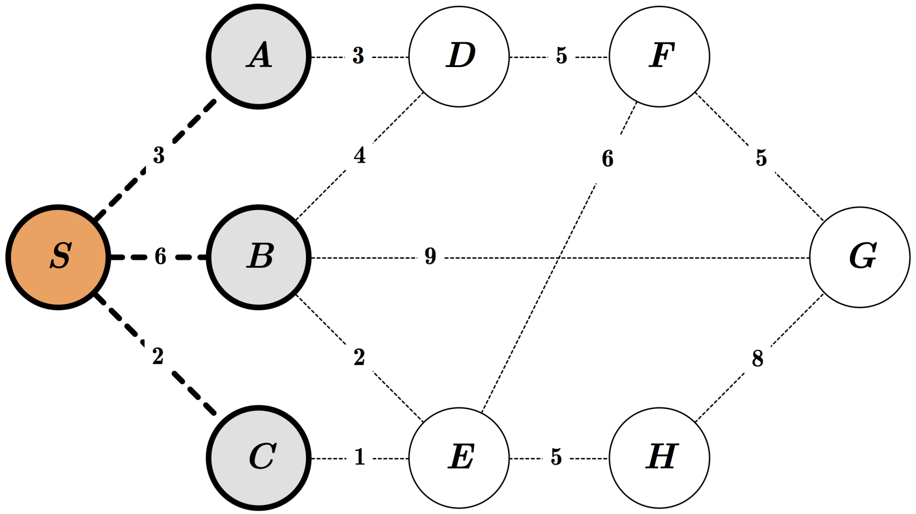


Stack:



Order of Visit:

Toy Example: DFS



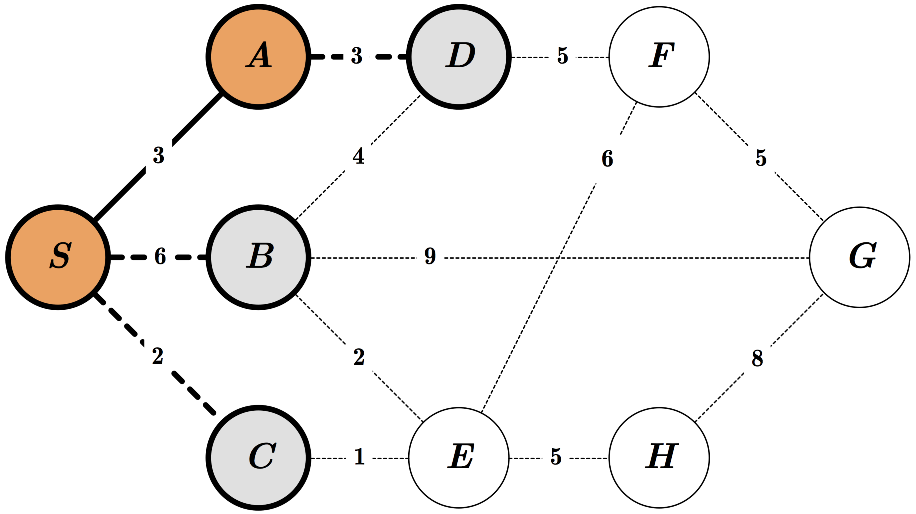
Stack:

S	C	B	A

Order of Visit:

S

Toy Example: DFS



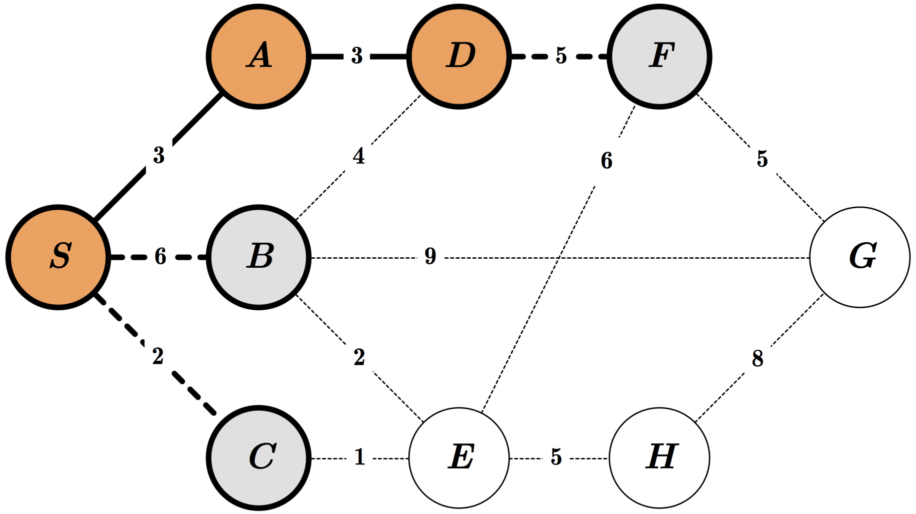
Stack:

S	C	B	A	D

Order of Visit:

S A

Toy Example: DFS



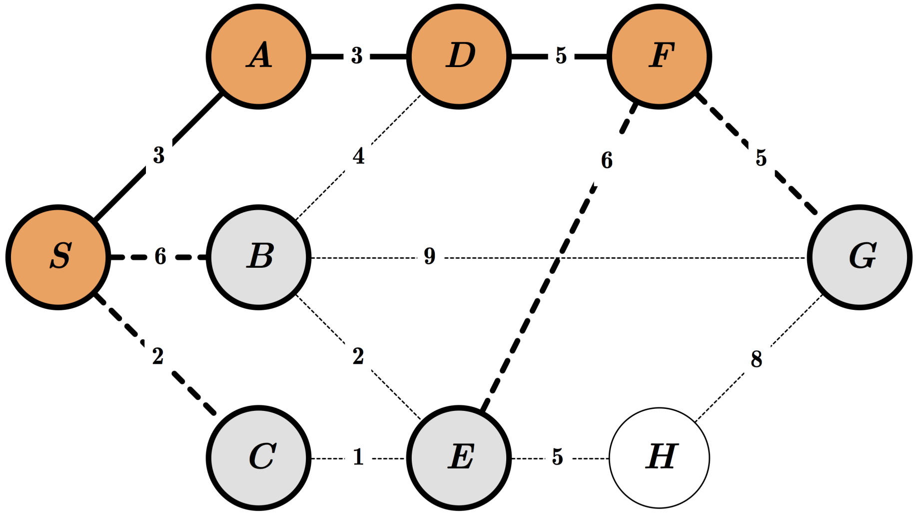
Stack:

S	C	B	A	D	F

Order of Visit:

S A D

Toy Example: DFS



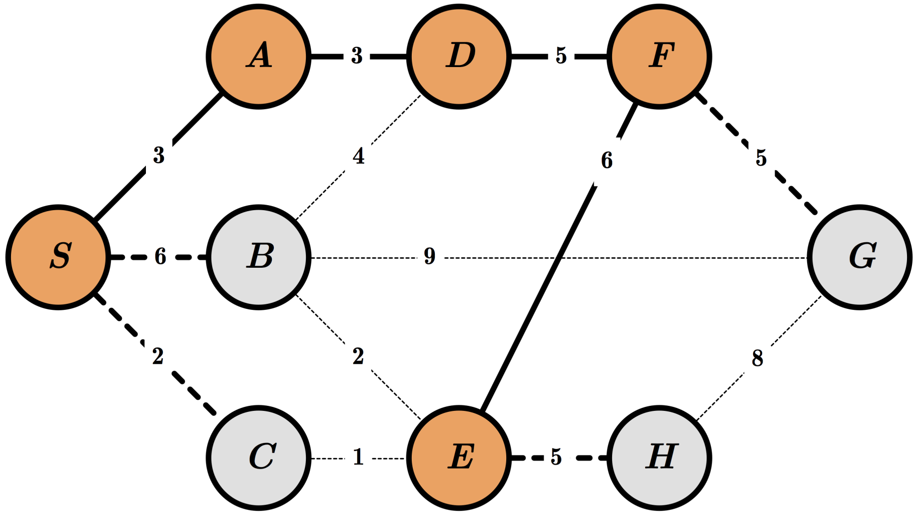
Stack:

S	C	B	A	D	F	G	E

Order of Visit:

S A D F

Toy Example: DFS



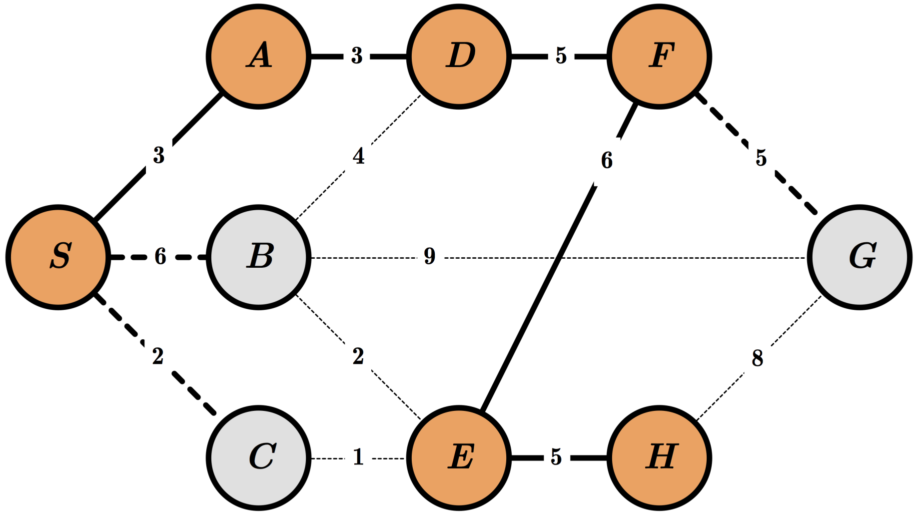
Stack:

S	C	B	A	D	F	G	E	H

Order of Visit:

$S \quad A \quad D \quad F \quad E$

Toy Example: DFS



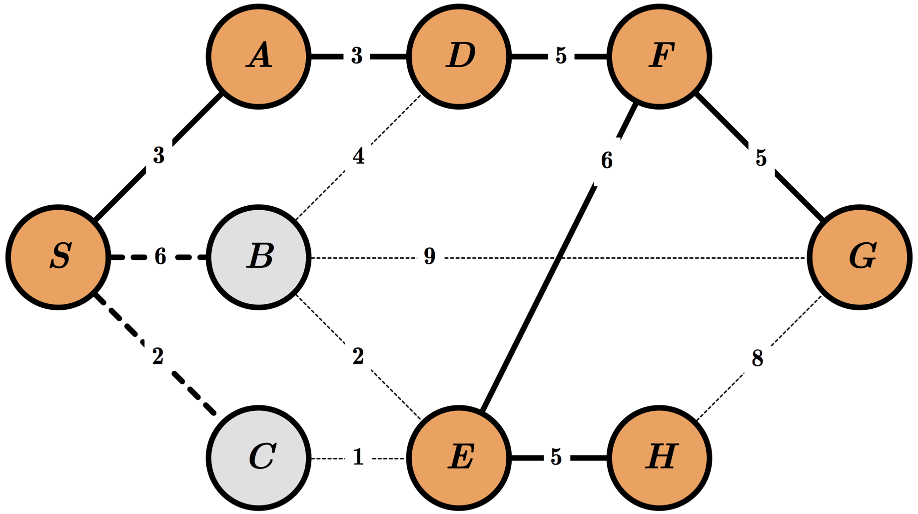
Stack:

S	C	B	A	D	F	G	E	H

Order of Visit:

S A D F E H

Toy Example: DFS



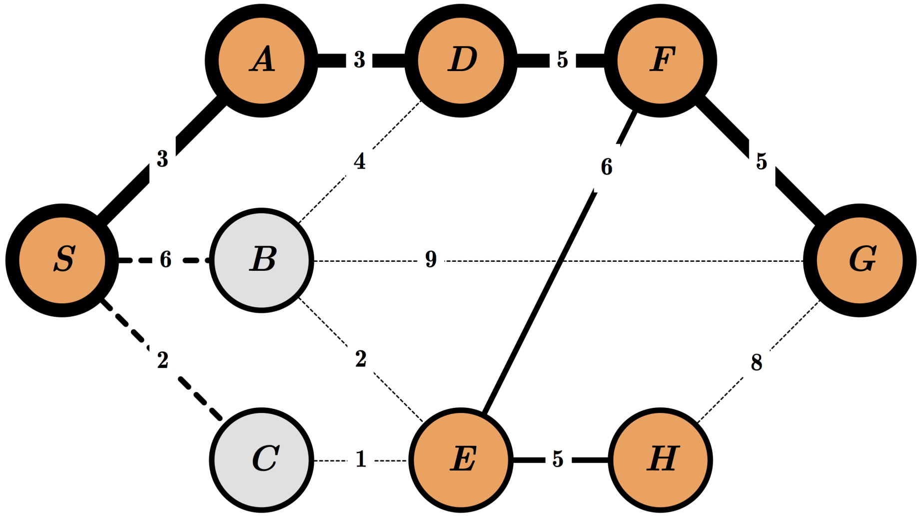
Stack:

S	C	B	A	D	F	G	E	H

Order of Visit:

S A D F E H G

Toy Example: DFS



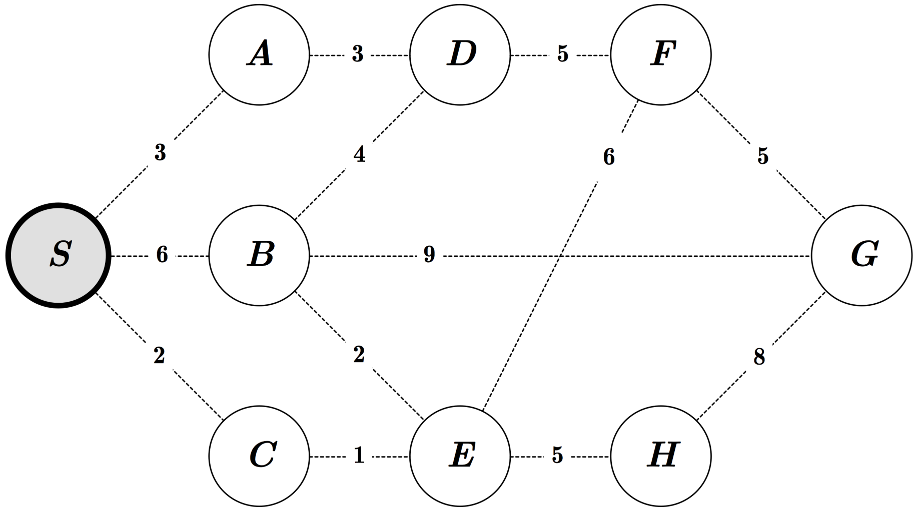
Stack:

S	C	B	A	D	F	G	E	H

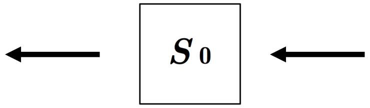
Order of Visit:

S A D F E H G

Toy Example: UCS

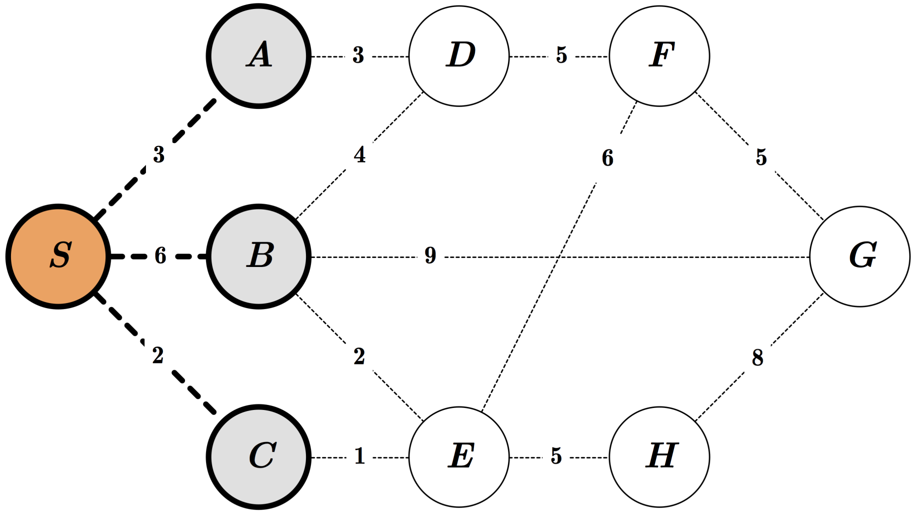


Priority Queue:



Order of Visit:

Toy Example: UCS



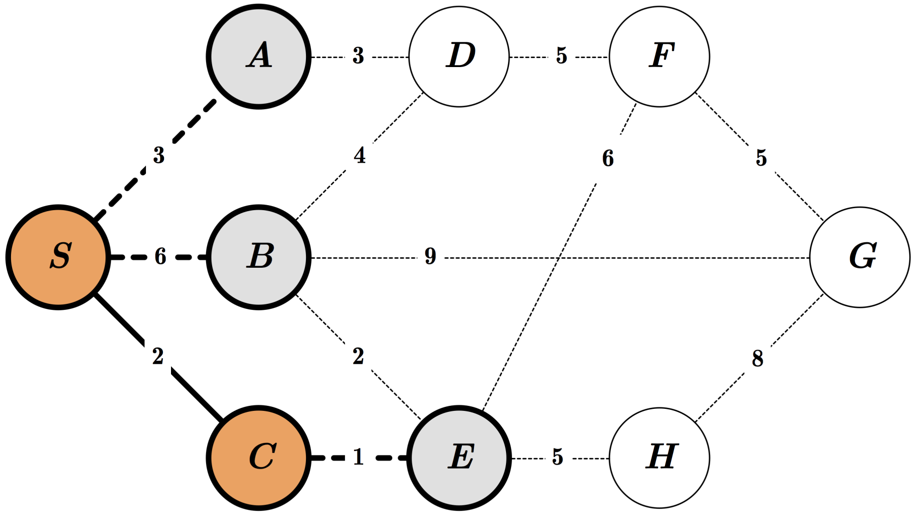
Priority Queue:

S_0	C_2	A_3	B_6

Order of Visit:

S

Toy Example: UCS



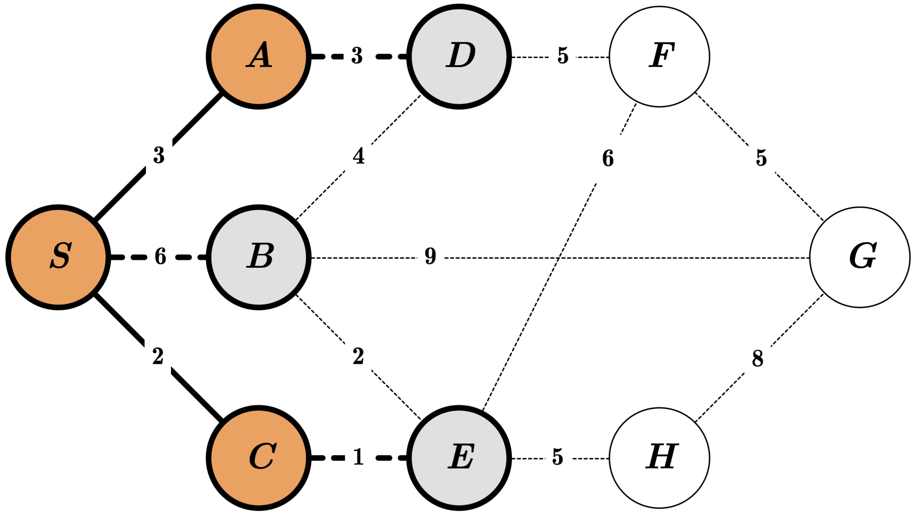
Priority Queue:

S 0	C 2	A 3	E 3	B 6

Order of Visit:

S C

Toy Example: UCS



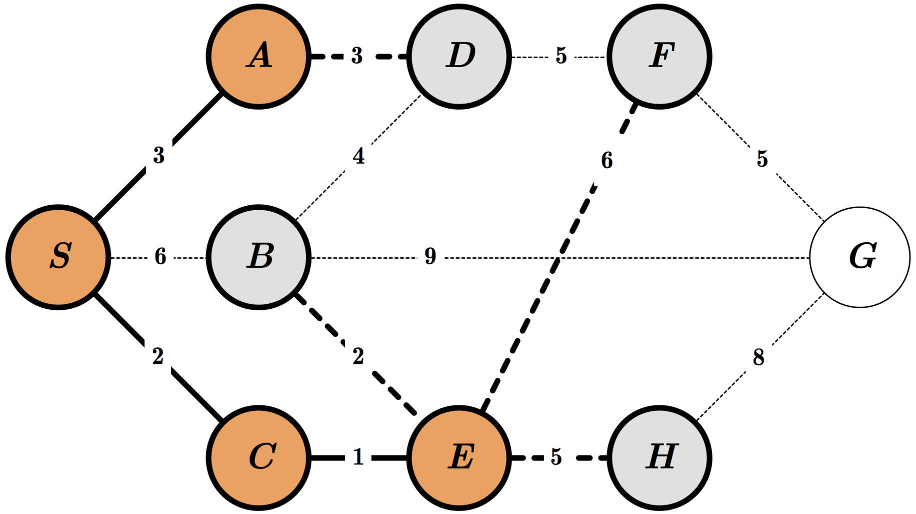
Priority Queue:

S 0	C 2	A 3	E 3	B 6	D 6

Order of Visit:

S C A

Toy Example: UCS



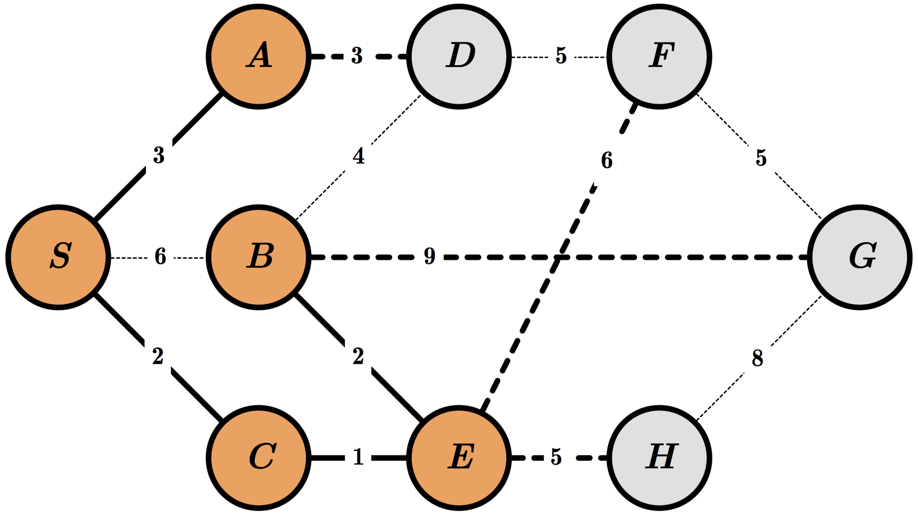
Priority Queue:

S 0	C 2	A 3	E 3	B 5	D 6	H 8	F 9

Order of Visit:

S C A E

Toy Example: UCS



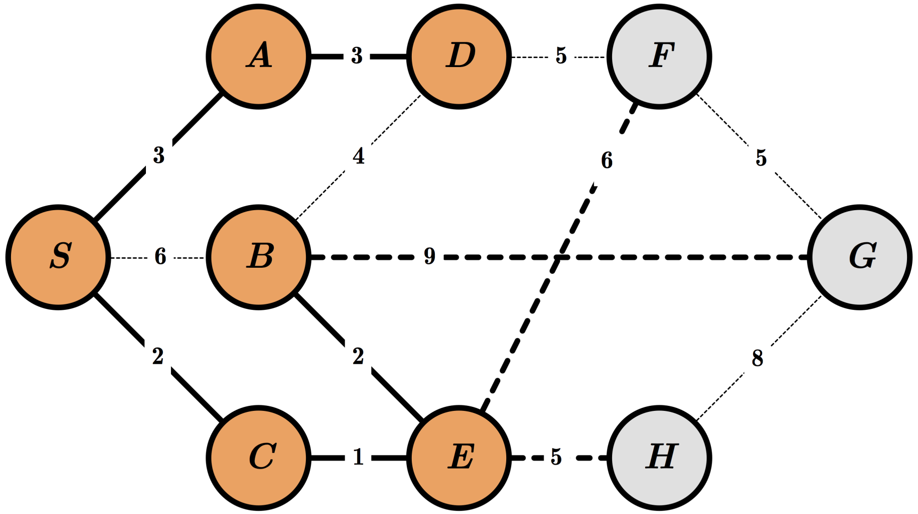
Priority Queue:

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

Toy Example: UCS



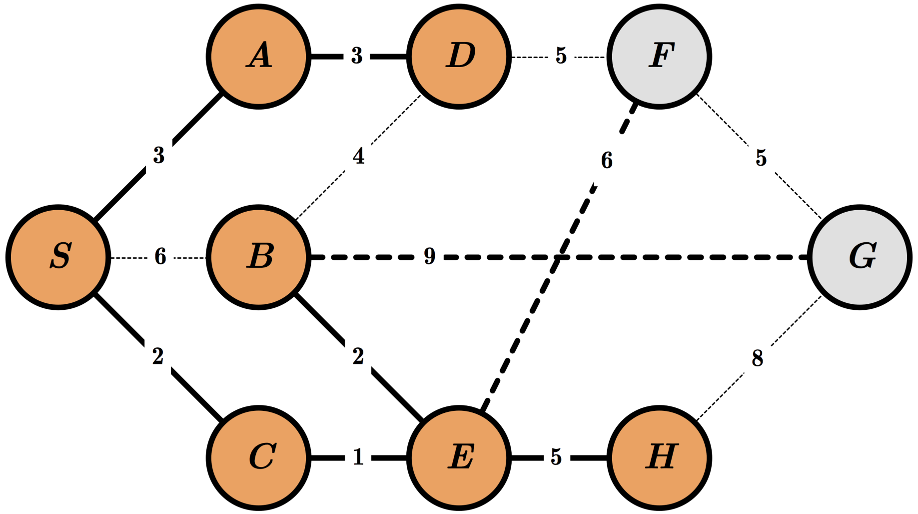
Priority Queue:

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

Toy Example: UCS



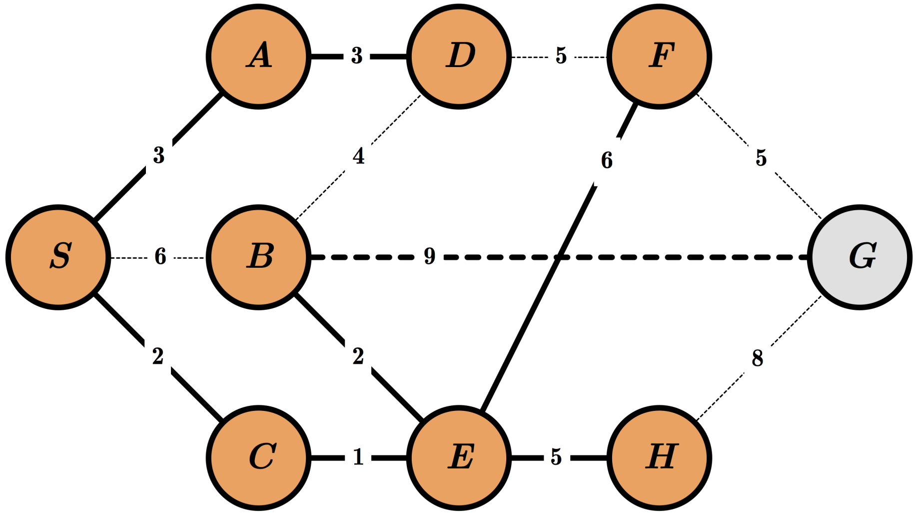
Priority Queue:

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

Toy Example: UCS



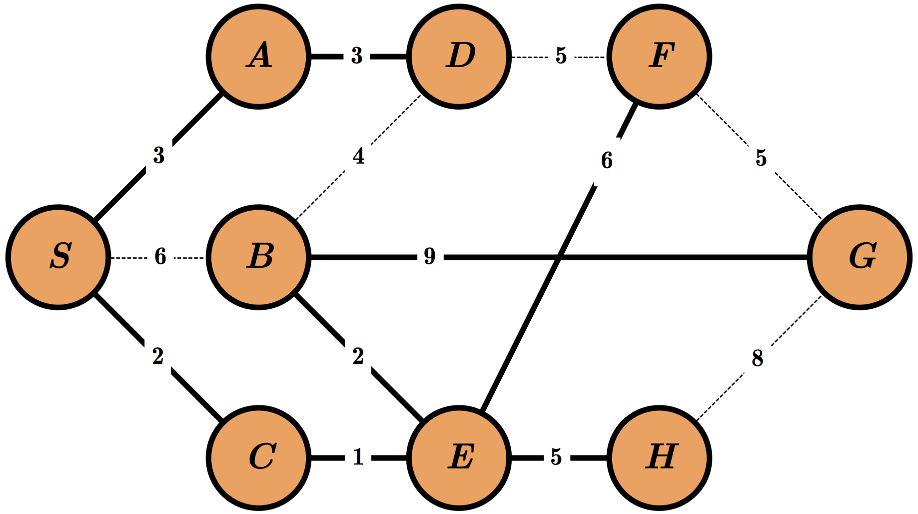
Priority Queue:

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

Toy Example: UCS



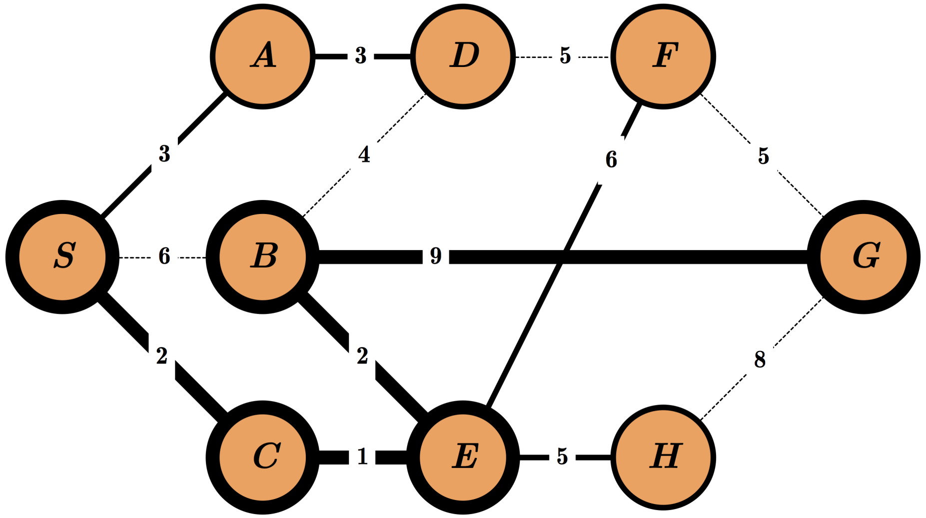
Priority Queue:

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

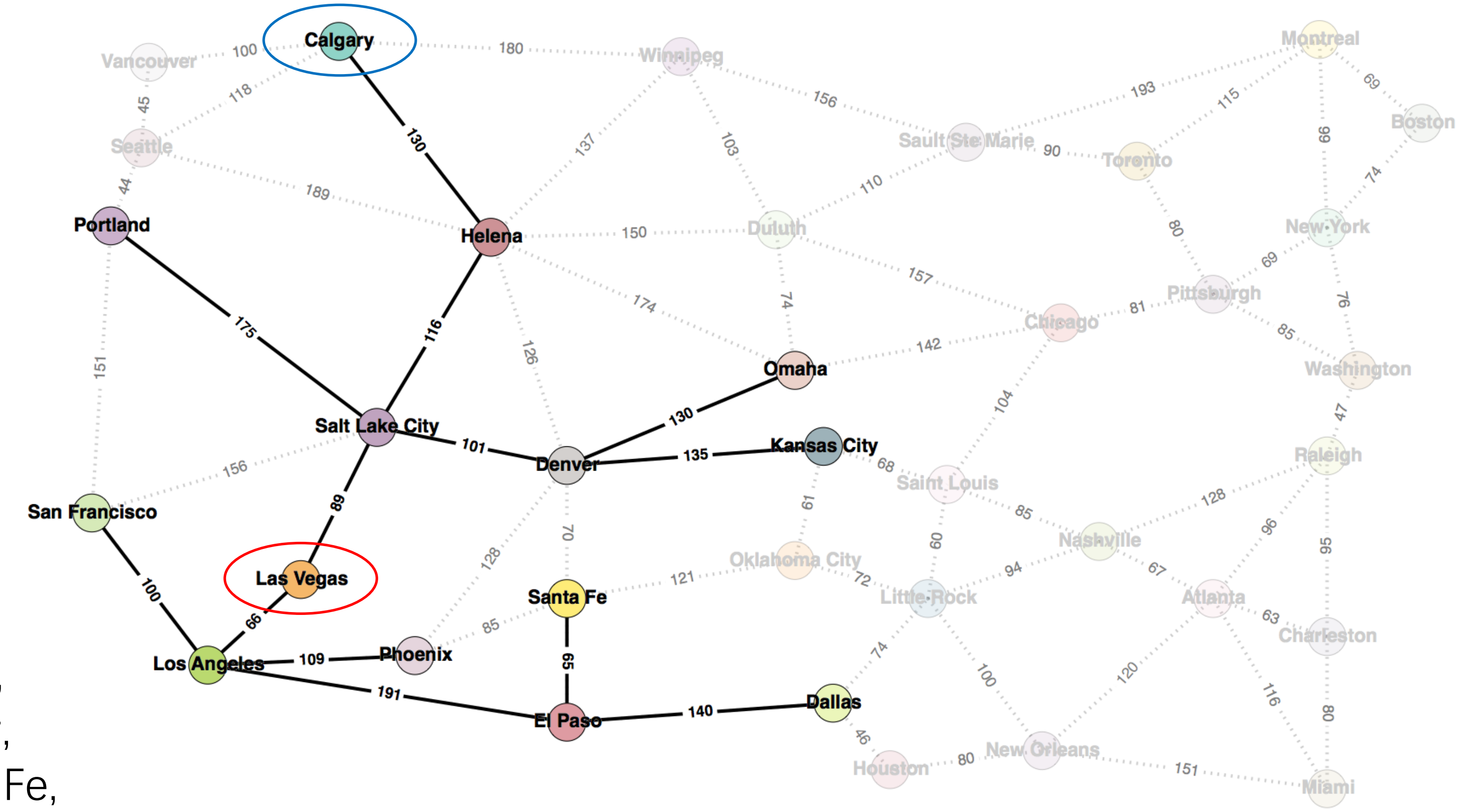
Order of Visit:

S C A E B D H F G

Examples using the map (BFS)

Start: Las Vegas

Goal: Calgary

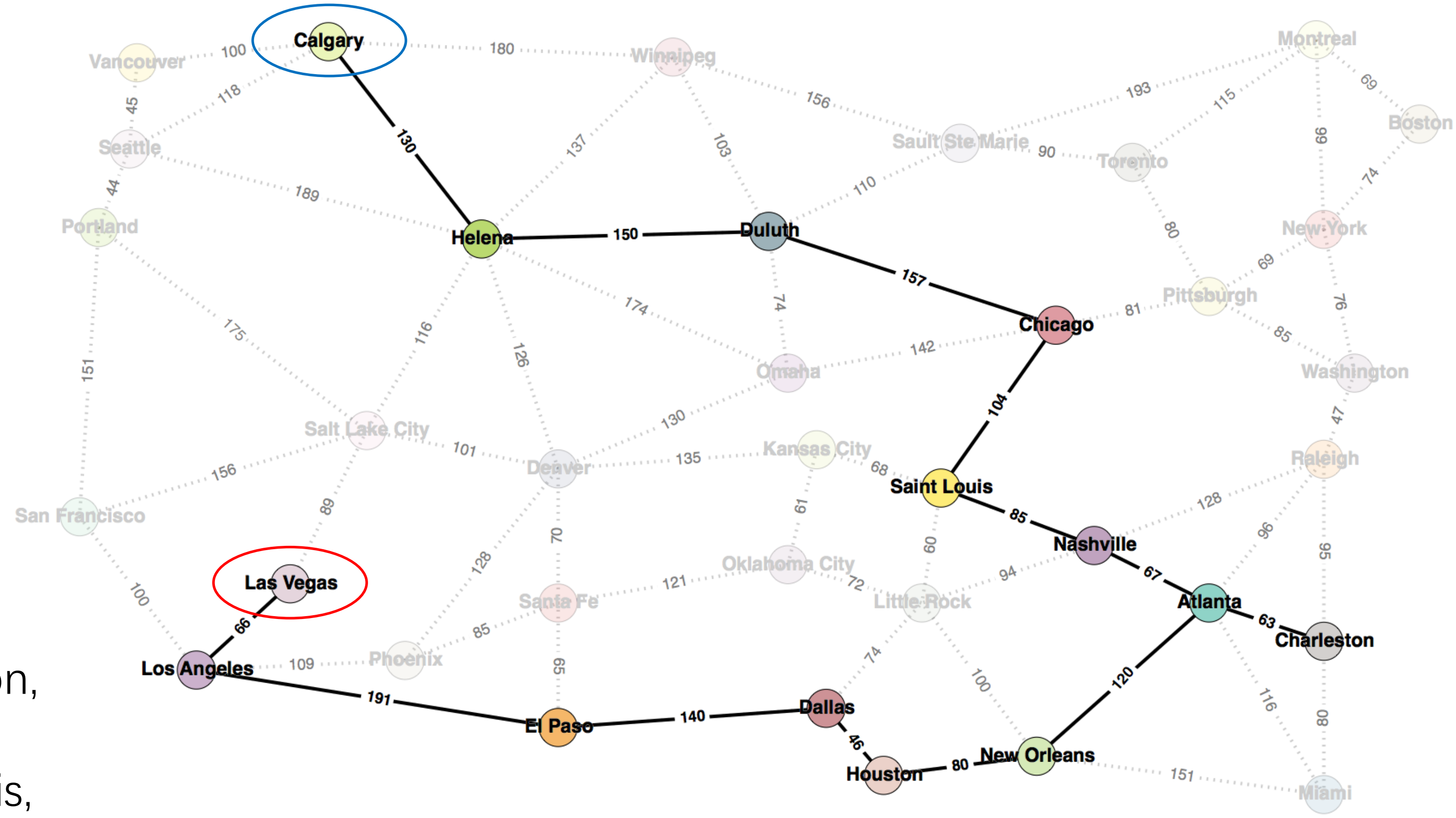


Order of Visit: Las Vegas, Los Angeles, Salt Lake City, El Paso, Phoenix, San Francisco, Denver, Helena, Portland, Dallas, Santa Fe, Kansas City, Omaha, Calgary.

Examples using the map (DFS)

Start: Las Vegas

Goal: Calgary

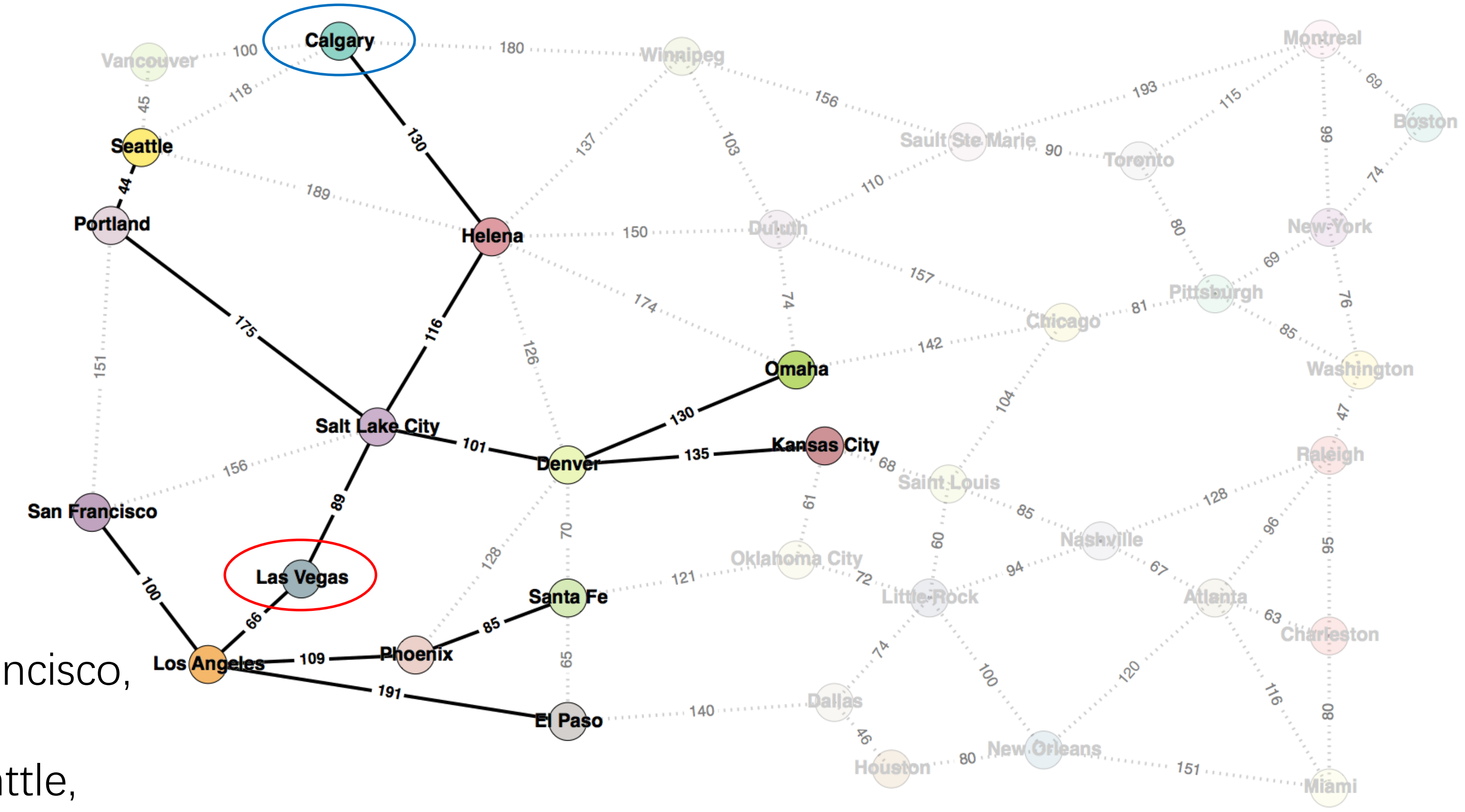


Order of Visit: Las Vegas, Los Angeles, El Paso, Dallas, Houston, New Orleans, Atlanta, Charleston, Nashville, Saint Louis, Chicago, Duluth, Helena, Calgary.

Examples using the map (UCS)

Start: Las Vegas

Goal: Calgary



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.

To be continued