CSC-411 Artificial Intelligence

Search Agents

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

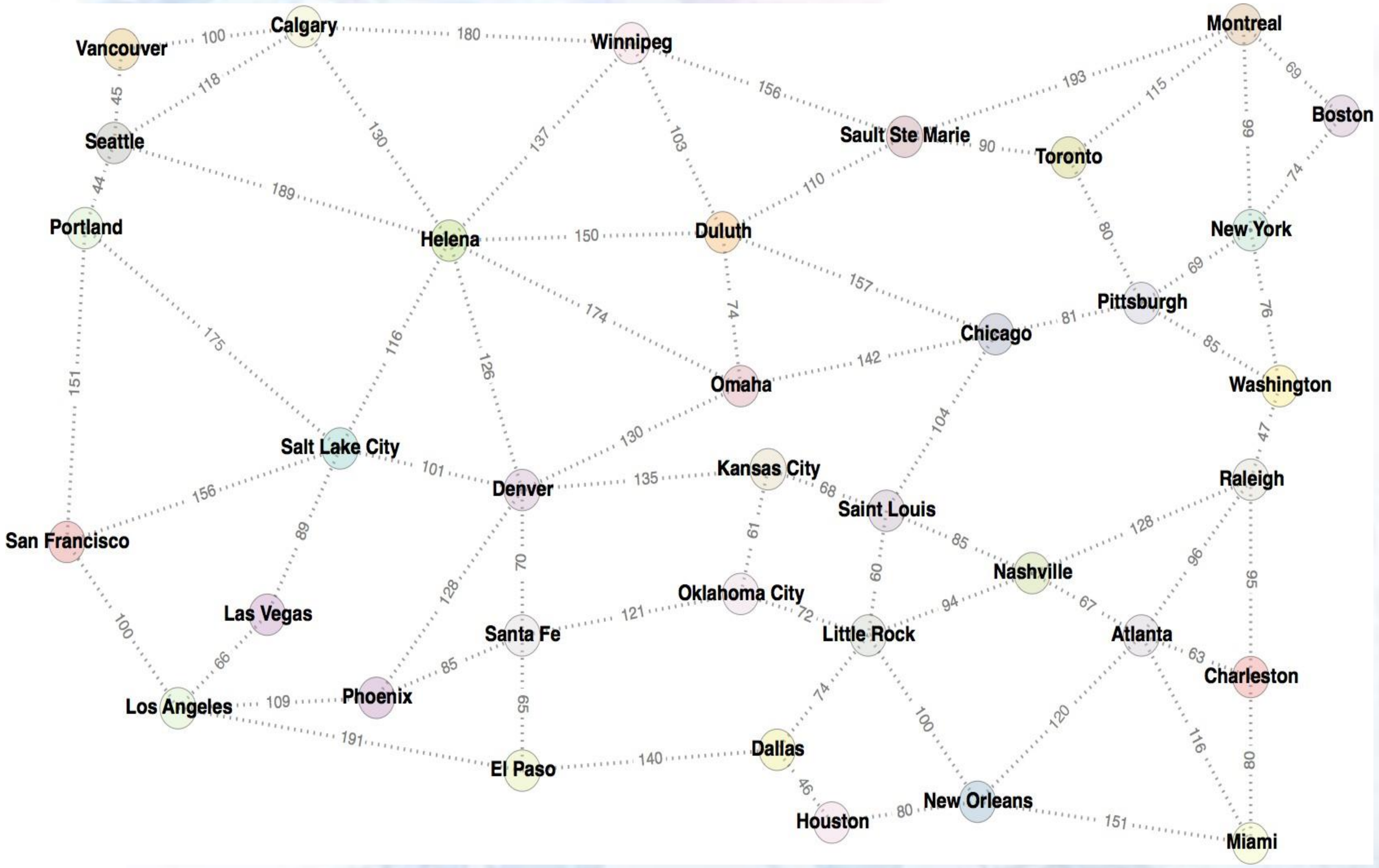
Search Space Regions

- The Search space is divided into three regions:
  - **Explored** (a.k.a. Closed List, Visited Set)
  - **Frontier** (a.k.a. Open List, the Fringe)
  - **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
```

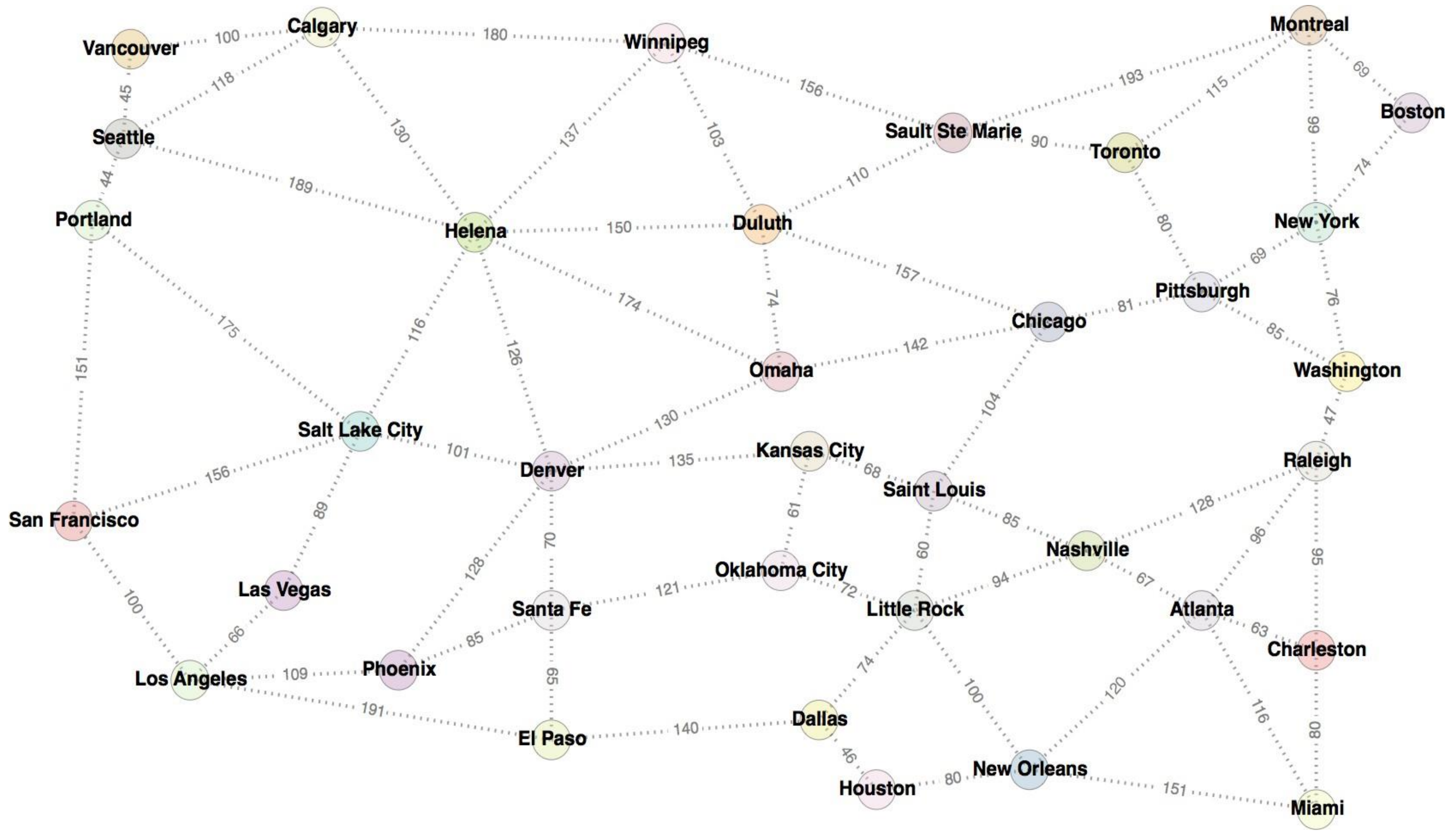


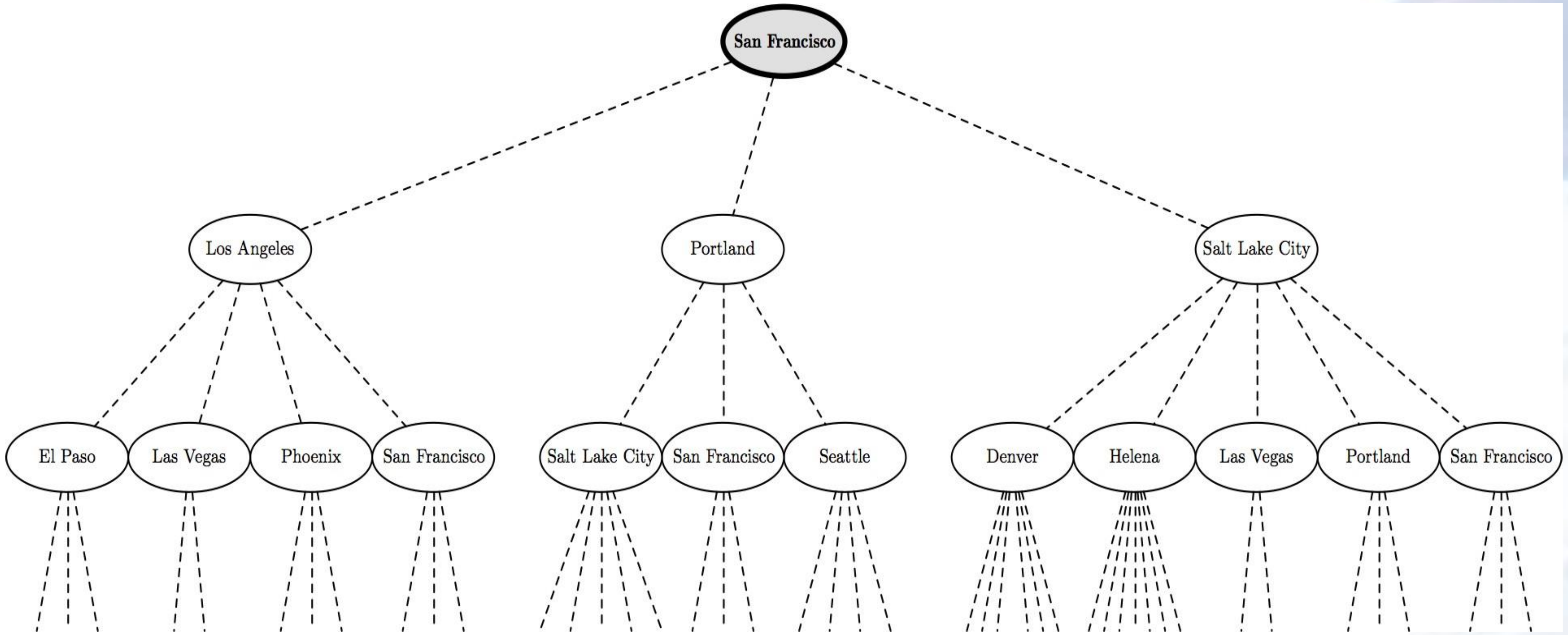
Tree Search Example

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

- In the following we adopt the following color coding: orange nodes are explored, grey nodes are the frontier, white nodes are unexplored, and black nodes are failures.

Tree Search Example





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

initialize frontier **with** initialState

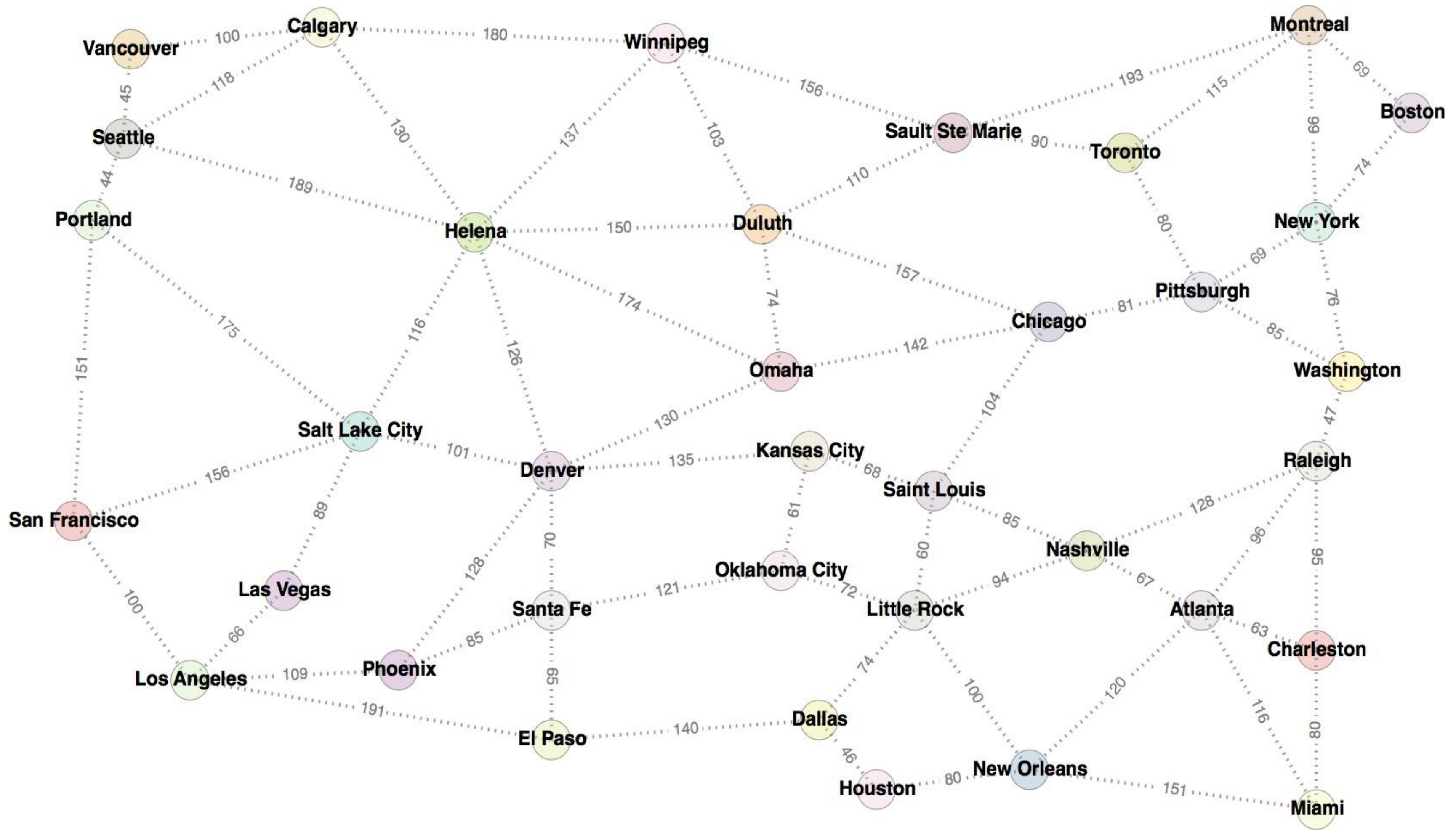
while not frontier.isEmpty():
state = frontier.remove()

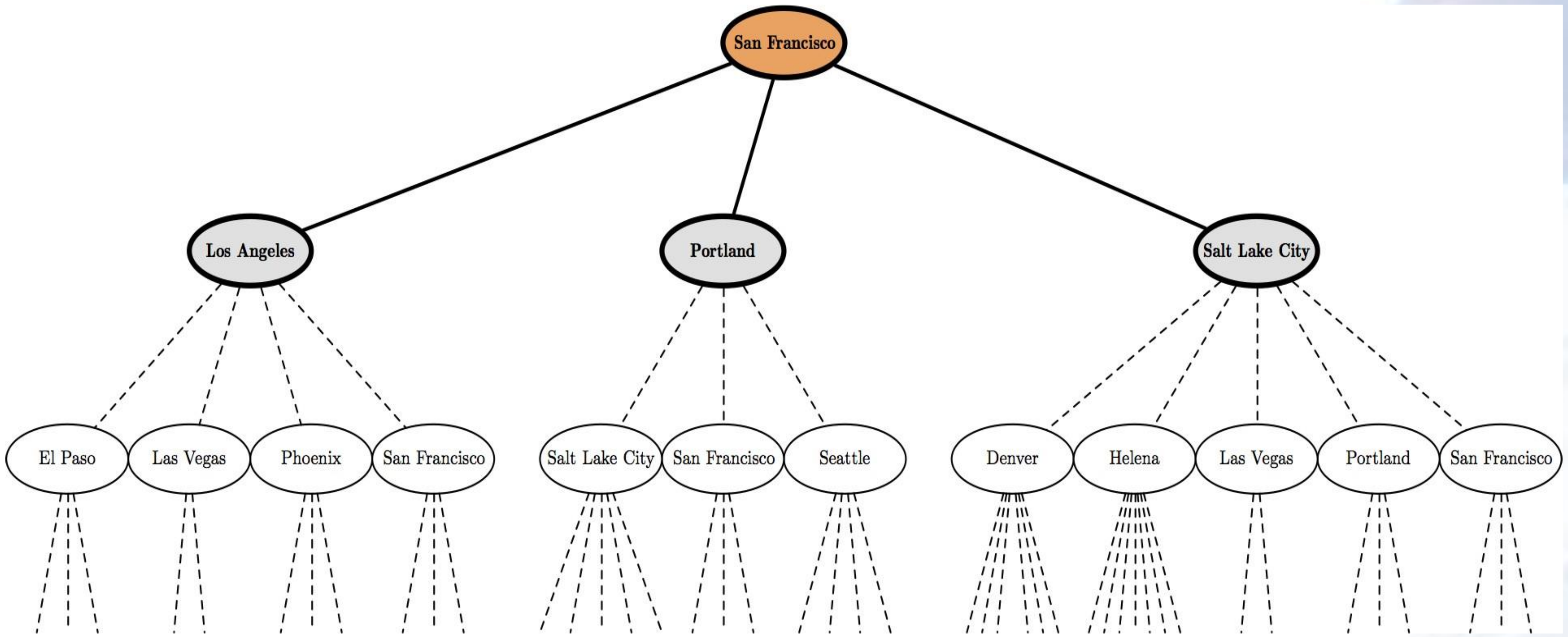
if goalTest(state):
return **SUCCESS(**state**)**

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

return **FAILURE**

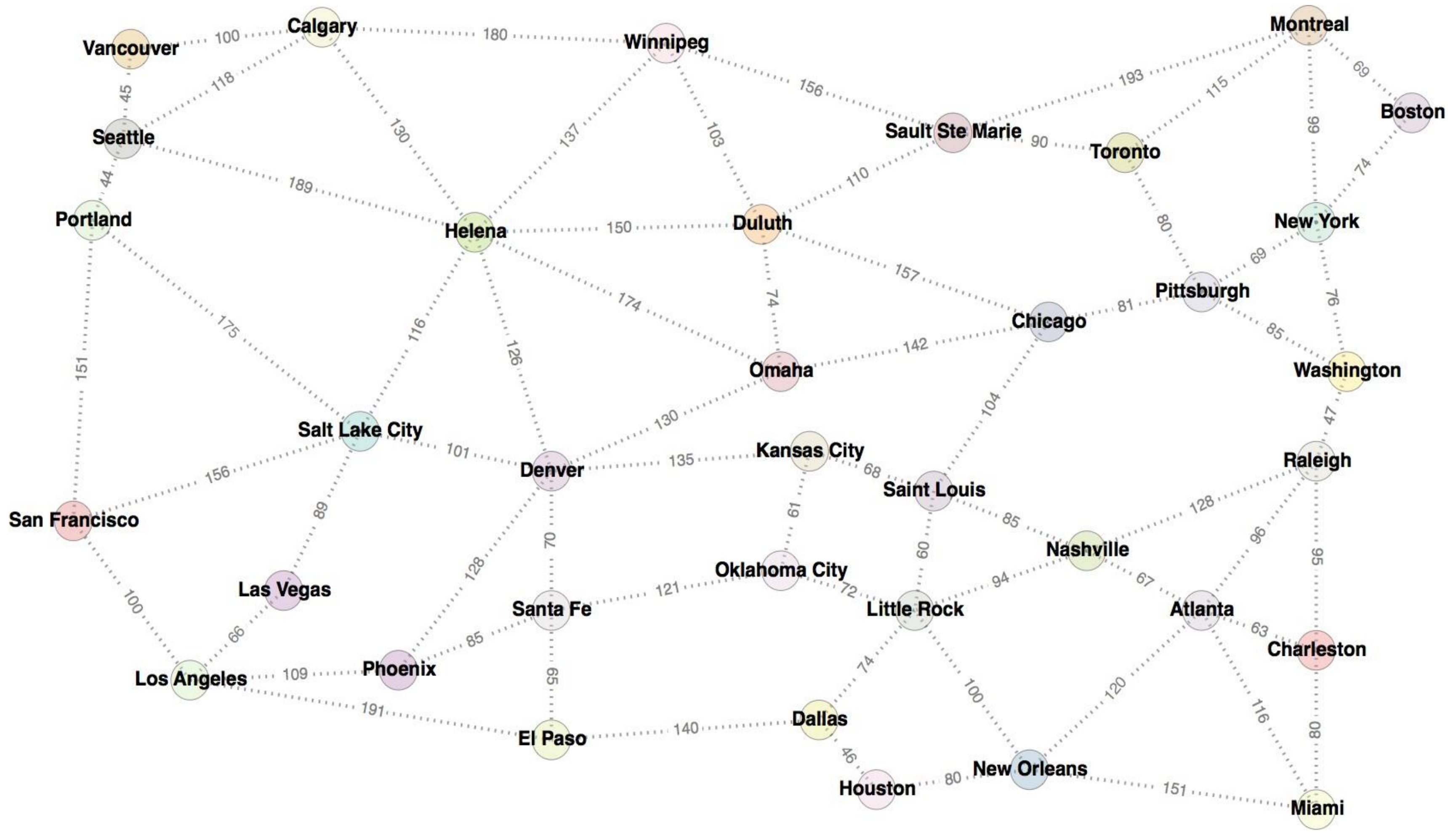
Tree Search Example

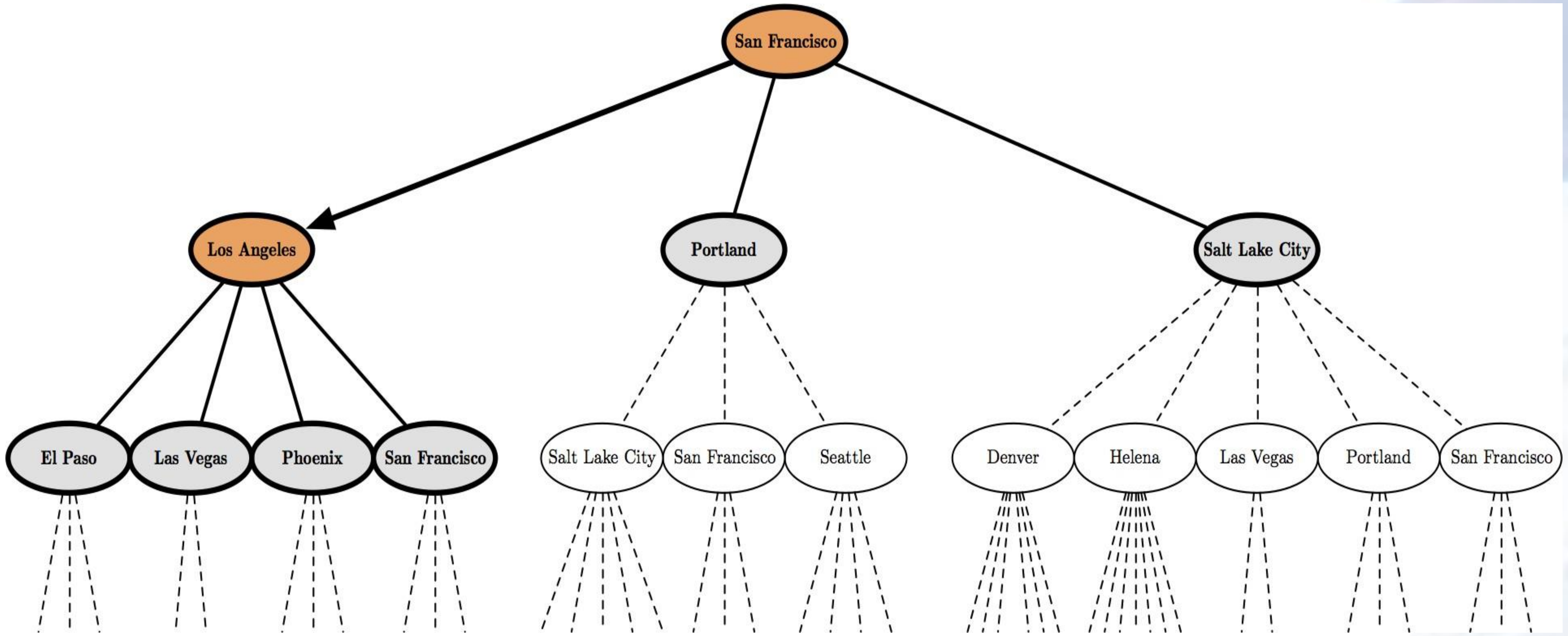





```
function TREE-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    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
```

Tree Search Example





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

    initialize frontier **with** initialState

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

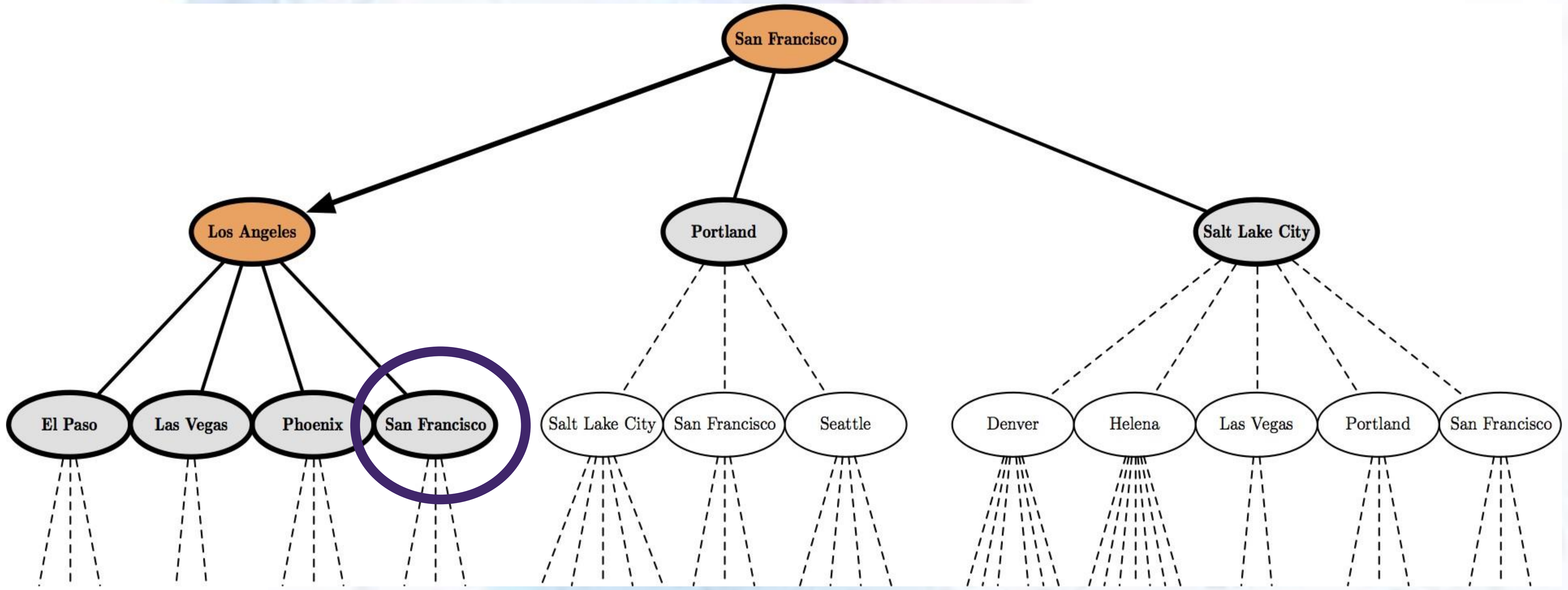
        if goalTest(state):
            return **SUCCESS**(state)

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

    return **FAILURE**

Graph Search

```
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 one exists?
  - Time complexity
    - Number of nodes generated/expanded
  - Space complexity
    - Maximum number of nodes in memory
  - Optimality
    - Does it always find a least-cost solution?

Search Strategies

- Time and space complexity are measured in terms of:
  - b: maximum branching factor of the search tree (actions per state).
  - d: depth of the solution
  - m: maximum depth of the state space (may be ∞) (also noted sometimes D).

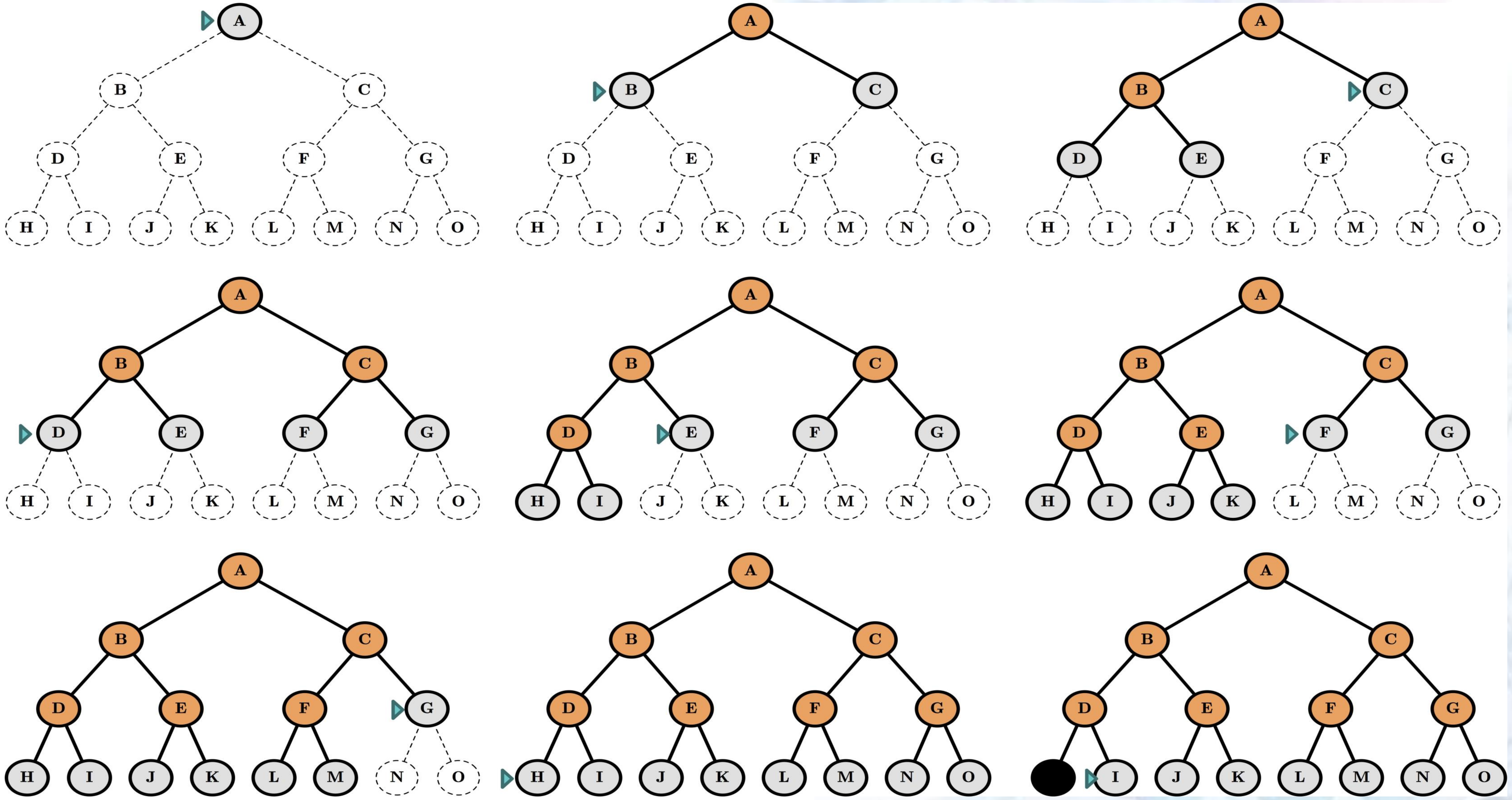
- Two kinds of search: **Uninformed** and **Informed**

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)

- Use no domain knowledge!



Breadth-First Search Algorithm

```
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

- **Complete:** Yes (if b is finite)
- **Time:** 1 + b + b2 + b3 + . . . + bd = O(bd)
- **Space:** $O(b^d)$
- **Note:** If the goal test is applied at expansion rather than generation then O(bd+1)
- **Optimal:** Yes (if cost = 1 per step)
- **Implementation:** Fringe - FIFO (Queue)

BFS Criteria

- **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!**