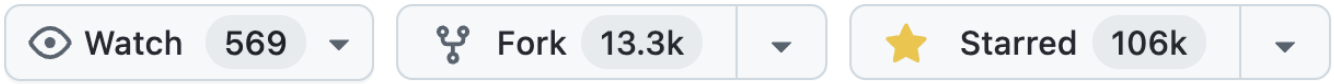
Algorithm Techniques

Graph Theory - Basic Knowledge

Xian Su

Recommended Reference Material:

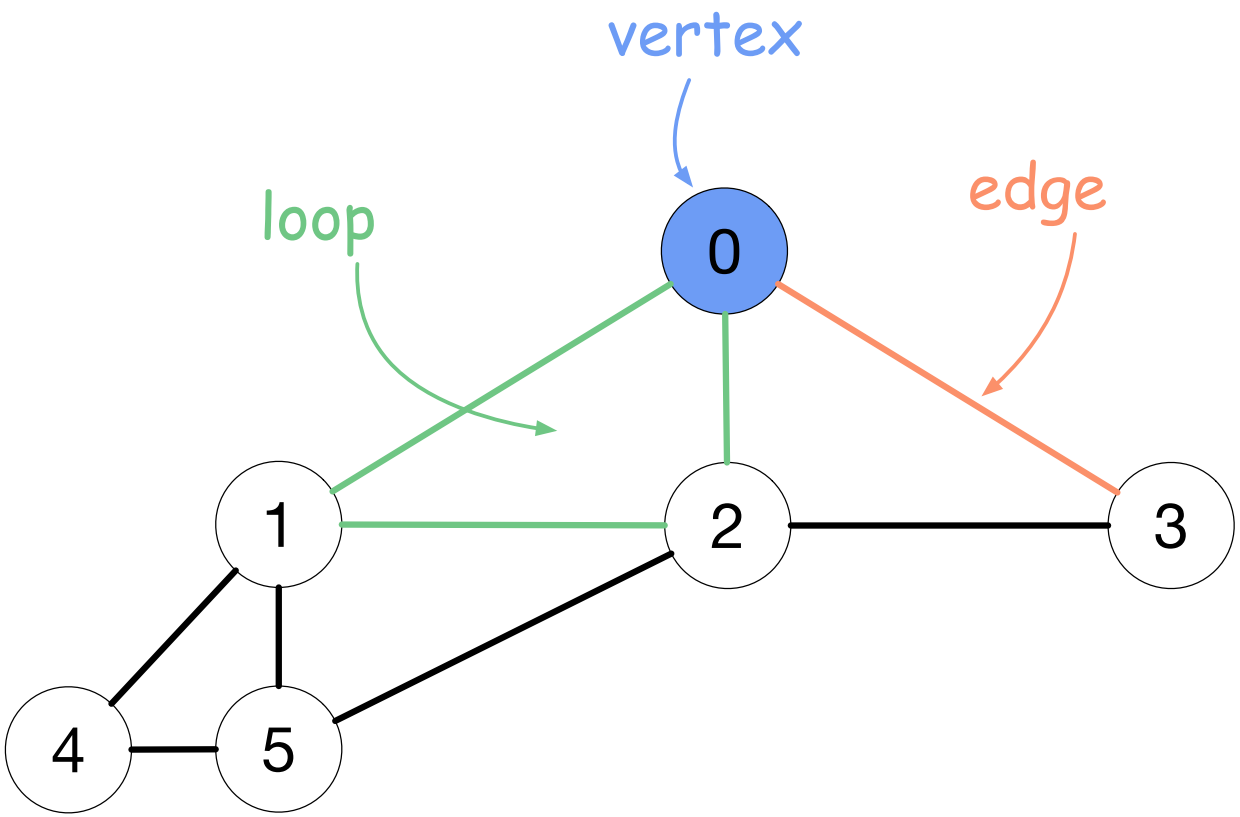
- Github Open Source Project: Hello Algo
- Github Repo: krahets/hello-algo



- Animated Illustrations
- Support More than 10 Programming Languages
- Programming, Data Structure, Algorithm, and Algorithm Techniques

Graphs and Trees

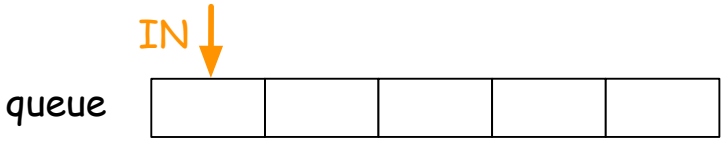
- A graph is a finite set of vertices that are connected by edges.
- A loop may exist in a graph where an edge leads back to the original vertex.



Necessary Recap: Queue & Stack

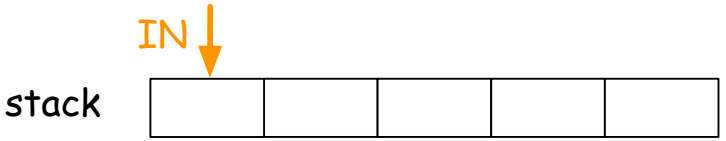
- Queue

a sequence {4, 2, 5, 3, 1}



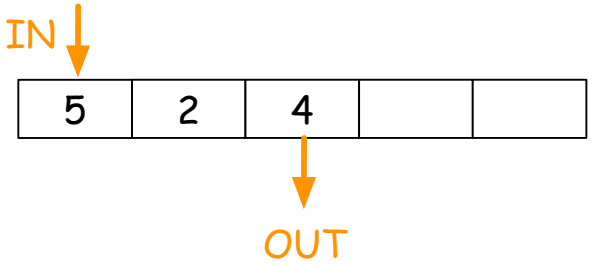
- Stack

a sequence {4, 2, 5, 3, 1}

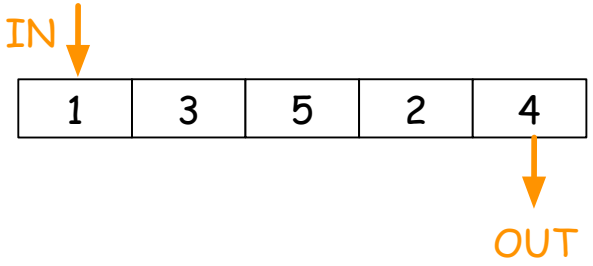


First IN First OUT

a sequence {4, 2, 5, 3, 1}

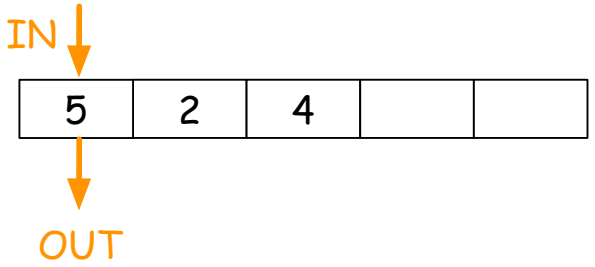


a sequence {}

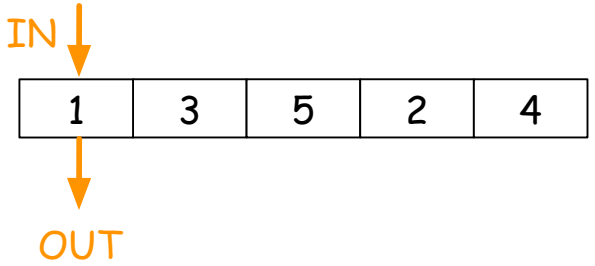


First IN Last OUT

a sequence {4, 2, 5, 3, 1}



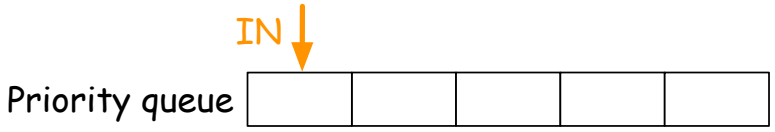
a sequence {}



Necessary Recap: Priority Queue

- The priority queue (PQ) is an abstract concept (a queue where elements are dequeued based on priority).

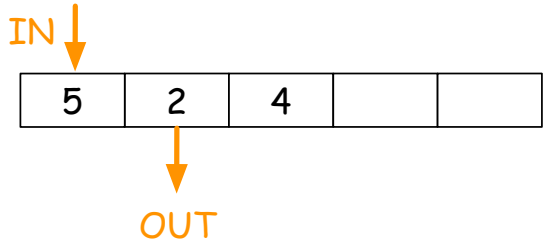
a sequence {4, 2, 5, 3, 1}



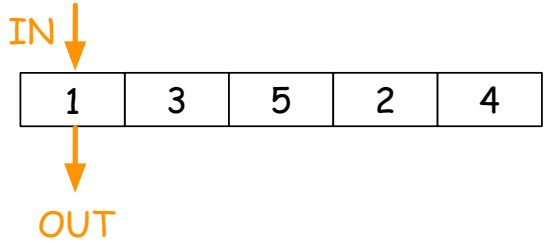
First IN PRIORITY OUT

e.g., priority = minimum value

a sequence {4, 2, 5, 3, 1}



a sequence {}



Example 1: Vertex Traversal in an Undirected Graph

Given an undirected graph $G = (V, E)$, where:

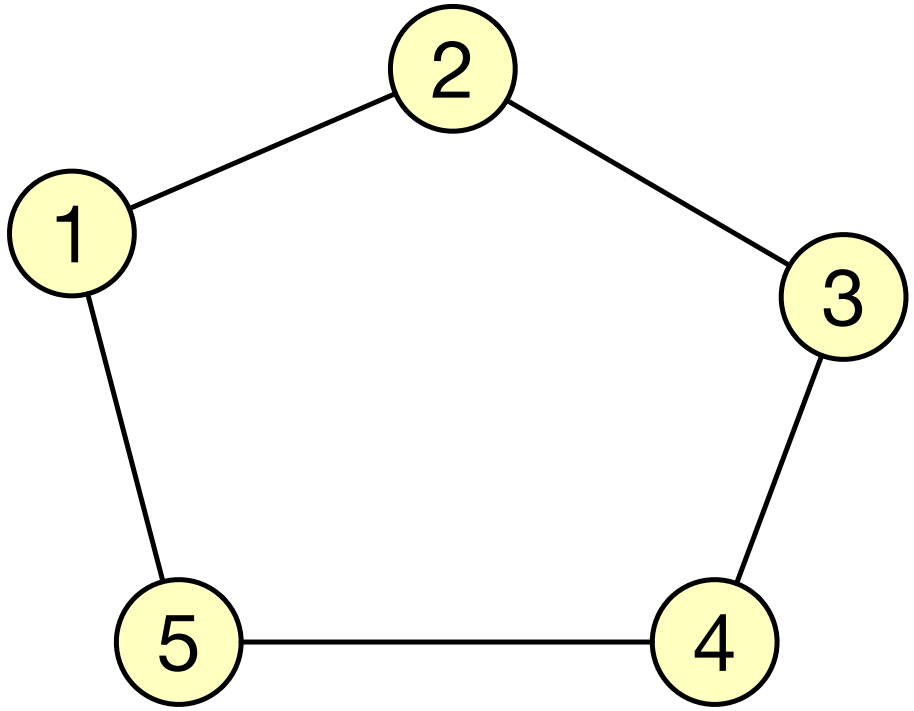
- $V = \{1,2,3,4,5\}$
- $E = \{(1,2), (1,5), (2,3), (3,4), (4,5)\}$

1. Use **Depth-First Search (DFS)** starting from vertex 1 and output the order of visited vertices.
2. Use **Breadth-First Search (BFS)** starting from vertex 1 and output the order of visited vertices.

Example 1: Vertex Traversal in an Undirected Graph

Given an undirected graph $G = (V, E)$, where:

- $V = \{1,2,3,4,5\}$
- $E = \{(1,2), (1,5), (2,3), (3,4), (4,5)\}$



1. Use **Depth-First Search (DFS)** starting from vertex 1 and output the order of visited vertices.
2. Use **Breadth-First Search (BFS)** starting from vertex 1 and output the order of visited vertices.

Example 2: Path Search in a Directed Graph

Given a directed graph $G = (V, E)$, where:

- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, C), (B, D), (C, D), (D, E)\}$

1. Use **BFS** starting from vertex A to find all paths to vertex E.

2. Use **DFS** starting from vertex A to list the sequence of all visited vertices during the traversal.

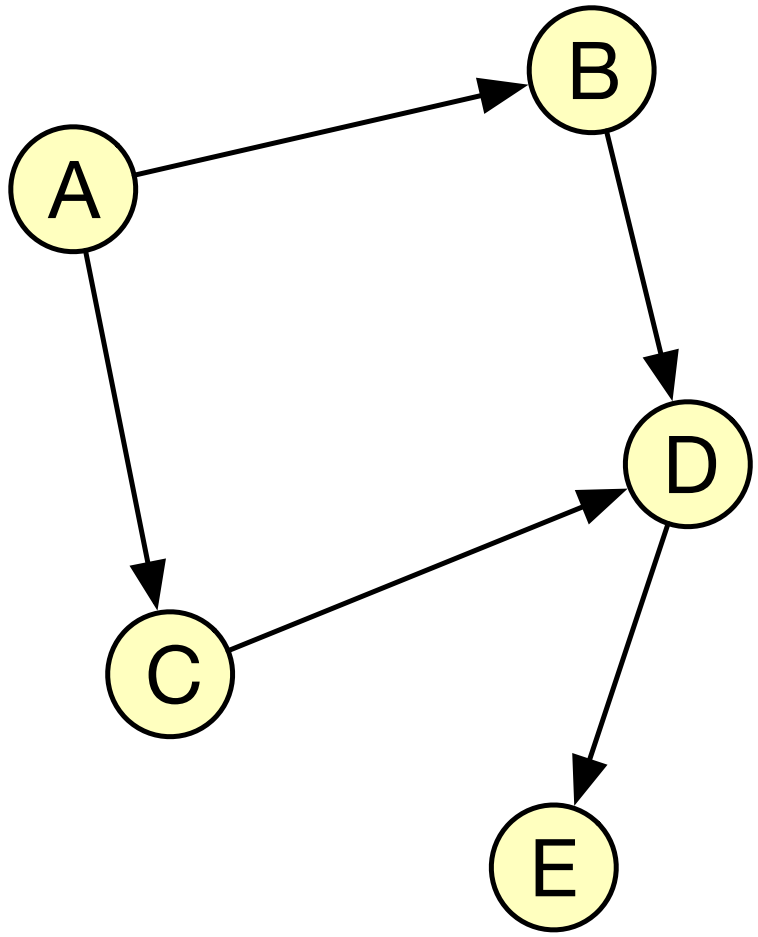
Example 2: Path Search in a Directed Graph

Given a directed graph $G = (V, E)$, where:

- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, C), (B, D), (C, D), (D, E)\}$

1. Use **BFS** starting from vertex A to find all paths to vertex E.

2. Use **DFS** starting from vertex A to list the sequence of all visited vertices during the traversal.

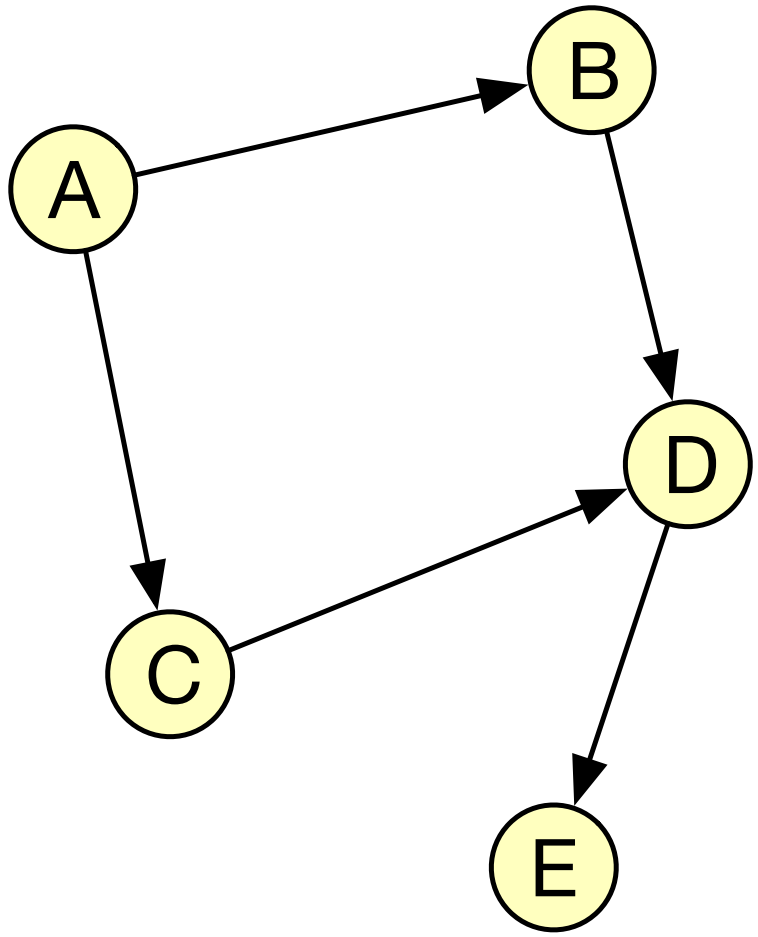


Example 2: Path Search in a Directed Graph

Given a directed graph $G = (V, E)$, where:

- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, C), (B, D), (C, D), (D, E)\}$

1. Use **BFS** starting from vertex A to find all paths to vertex E.

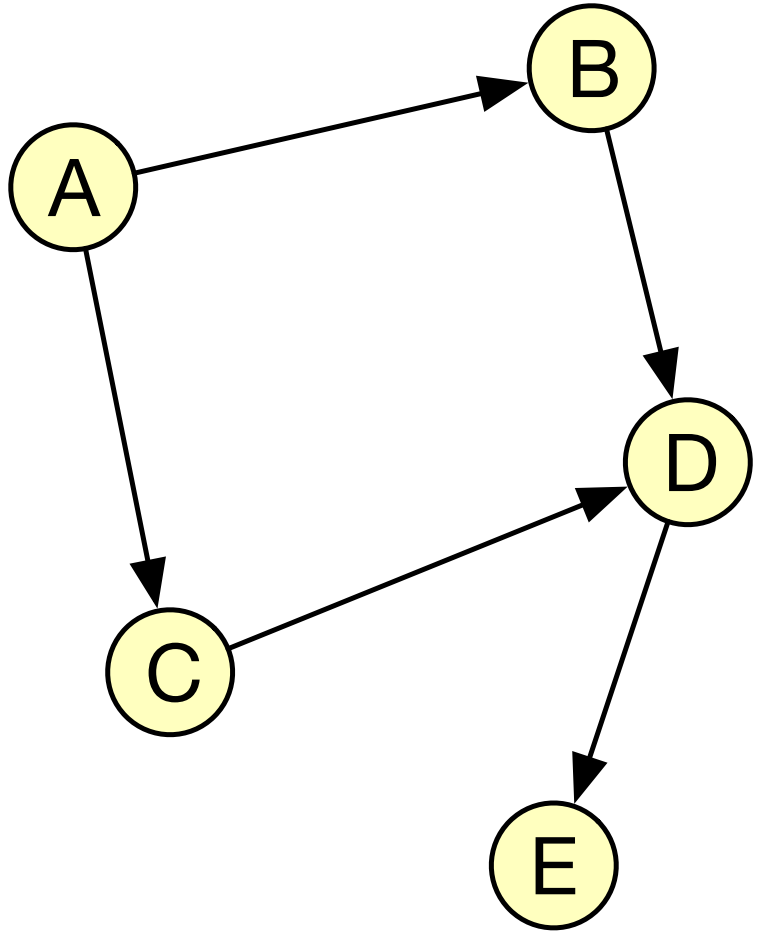


Example 2: Path Search in a Directed Graph

Given a directed graph $G = (V, E)$, where:

- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, C), (B, D), (C, D), (D, E)\}$

2. Use **DFS** starting from vertex A to list the sequence of all visited vertices during the traversal.



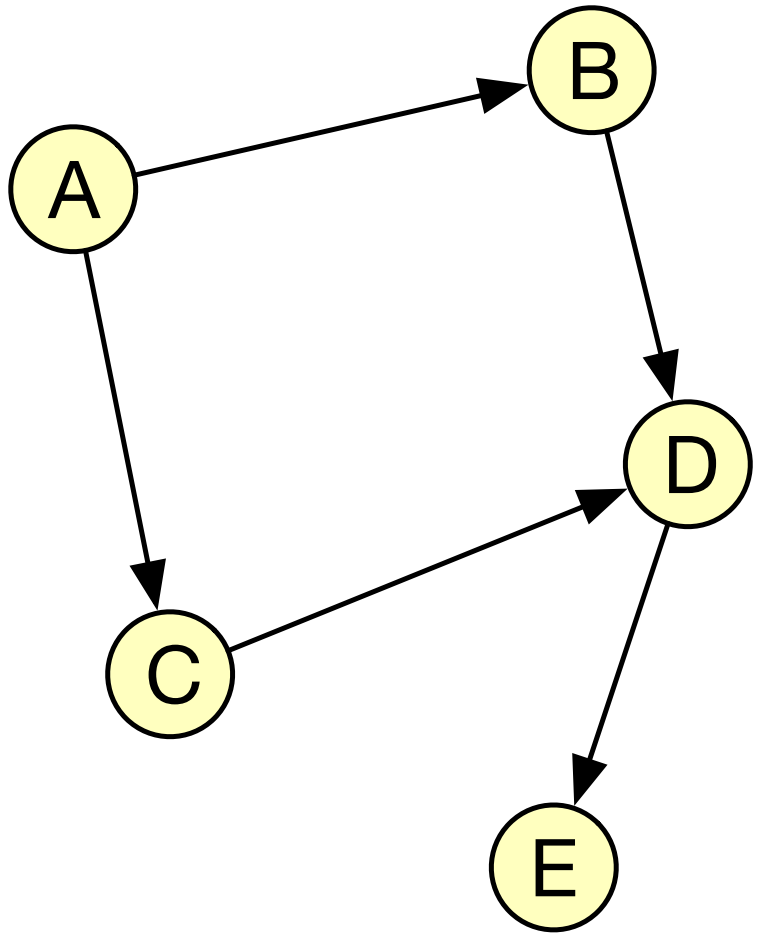

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = []
    visited.append(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

graph = {1: [2, 5], 2: [1, 3], 3: [2, 4], 4: [3, 5], 5: [1, 4]}
print(dfs(graph, 1))
```

Example 2: Path Search in a Directed Graph

Given a directed graph $G = (V, E)$, where:

- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, C), (B, D), (C, D), (D, E)\}$

1. Use **BFS** starting from vertex A to find all paths to vertex E.



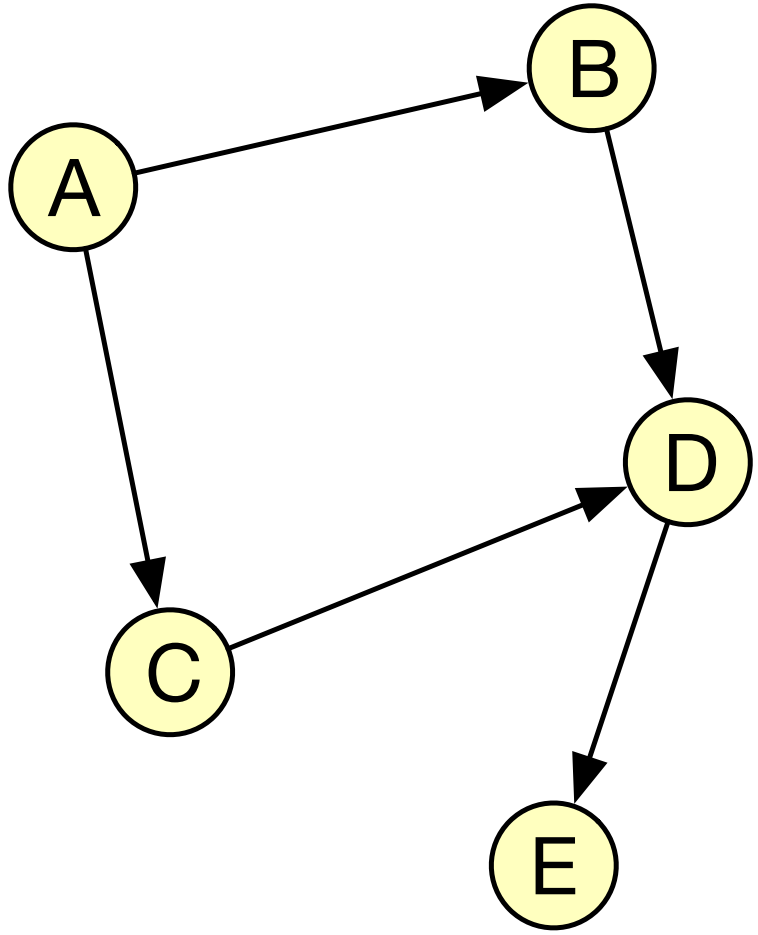

```
def bfs_all_paths(graph, start, target):
    queue = [[start]]
    paths = []
    while queue:
        path = queue.pop(0)
        node = path[-1]
        if node == target:
            paths.append(path)
        for neighbor in graph[node]:
            if neighbor not in path:
                queue.append(path + [neighbor])
    return paths

graph = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': ['E'], 'E': []}
print(bfs_all_paths(graph, 'A', 'E'))
```

Example 2: Path Search in a Directed Graph

Given a directed graph $G = (V, E)$, where:

- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, C), (B, D), (C, D), (D, E)\}$

2. Use **DFS** starting from vertex A to list the sequence of all visited vertices during the traversal.




```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = []
    visited.append(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

graph = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': ['E'], 'E': []}
print(dfs(graph, 'A'))
```

Example 3: Shortest Pathfinding

You are given a 2D grid of size 5×5. Each cell in the grid represents a location, and you can either move up, down, left, or right between adjacent cells, provided that the cell is passable (0 and 1 represent road and obstacle, respectively).

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Your task is to find the shortest path from a given start cell (green) to a goal cell (red) while avoiding obstacles. Specifically, we use the following grid to represent the grid and (x, y) to represent the location.

Example 3: Shortest Pathfinding

You are given a 2D grid of size 5×5. Each cell in the grid represents a location, and you can either move up, down, left, or right between adjacent cells, provided that the cell is passable (0 and 1 represent road and obstacle, respectively).

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Your task is to find the shortest path from a given start cell (green) to a goal cell (red) while avoiding obstacles. Specifically, we use the following grid to represent the grid and (x, y) to represent the location.

Required to use graph algorithm, but where is the graph?

Example 3: Shortest Pathfinding

You are given a 2D grid of size 5×5. Each cell in the grid represents a location, and you can either move up, down, left, or right between adjacent cells, provided that the cell is passable (0 and 1 represent road and obstacle, respectively).

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Required to use graph algorithm, but where is the graph?

- Vertex:
  - state/position
- Edge:
  - move between positions

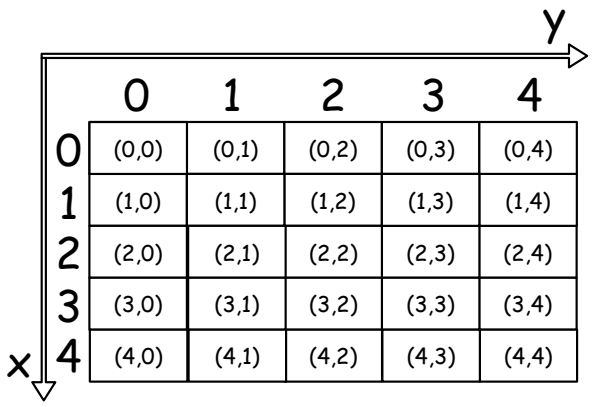
Example 3: Shortest Pathfinding

You are given a 2D grid of size 5×5. Each cell in the grid represents a location, and you can either move up, down, left, or right between adjacent cells, provided that the cell is passable (0 and 1 represent road and obstacle, respectively).

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Required to use graph algorithm, but where is the graph?

- Vertex:
  - state/position
- Edge:
  - move between positions



$$\Longleftrightarrow \quad G = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}_{5\times 5}$$

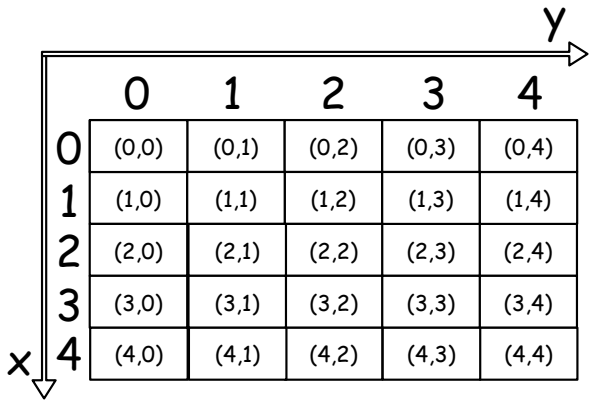
Example 3: Shortest Pathfinding

You are given a 2D grid of size 5×5. Each cell in the grid represents a location, and you can either move up, down, left, or right between adjacent cells, provided that the cell is passable (0 and 1 represent road and obstacle, respectively).

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Required to use graph algorithm, but where is the graph?

- Vertex:
  - States/positions: (x, y)
  - Root node ???
  - Goal state ???
- Edge:
  - move between positions: ???



$$G = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}_{5\times5}$$

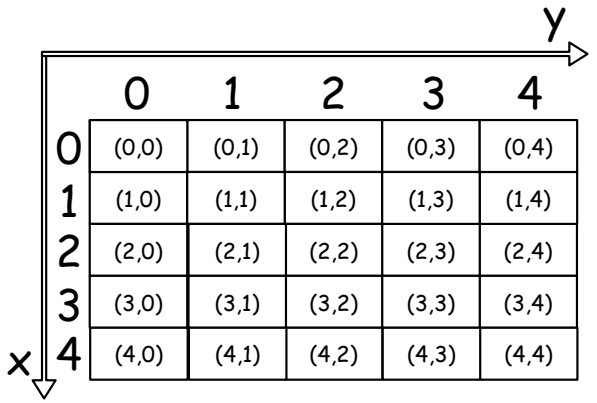
Example 3: Shortest Pathfinding

You are given a 2D grid of size 5×5. Each cell in the grid represents a location, and you can either move up, down, left, or right between adjacent cells, provided that the cell is passable (0 and 1 represent road and obstacle, respectively).

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Required to use graph algorithm, but where is the graph?

- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions: ???



$$\Longleftrightarrow \quad G = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}_{5\times5}$$

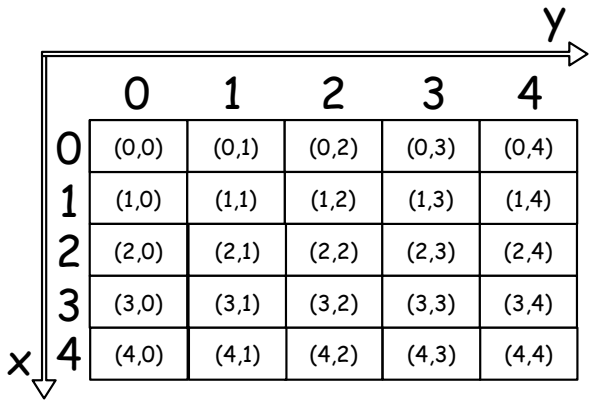
Example 3: Shortest Pathfinding

You are given a 2D grid of size 5×5. Each cell in the grid represents a location, and you can either move up, down, left, or right between adjacent cells, provided that the cell is passable (0 and 1 represent road and obstacle, respectively).

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Required to use graph algorithm, but where is the graph?

- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions: Up, down, left, right --> (-1, 0), (1, 0), (0, -1), (0, 1)



$$G = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}_{5\times5}$$

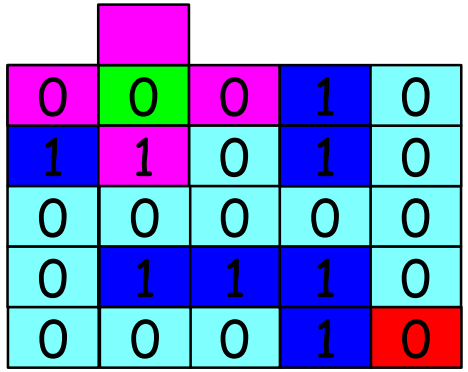
Example 3: Shortest Pathfinding

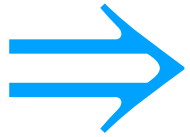
You are given a 2D grid of size 5×5. Each cell in the grid represents a location, and you can either move up, down, left, or right between adjacent cells, provided that the cell is passable (0 and 1 represent road and obstacle, respectively).

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Can we move arbitrarily?

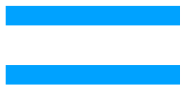
- Edge: move between positions: Up, down, left, right --> (-1, 0), (1, 0), (0, -1), (0, 1)

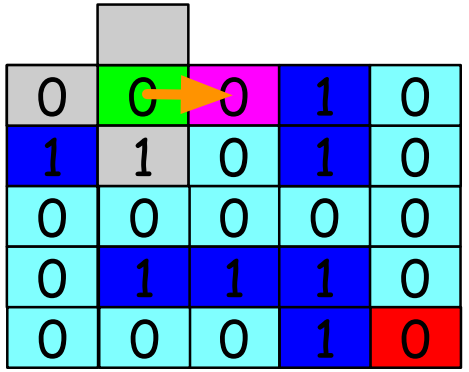




Only one option:

$(0,1) \rightarrow (0,2)$

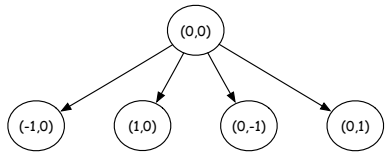




Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

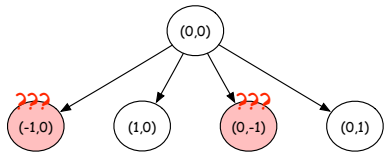


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

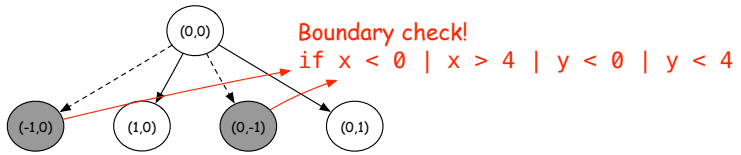


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

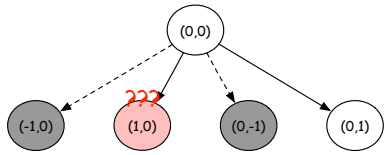


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

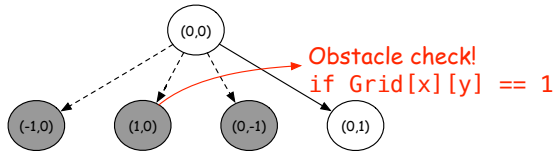


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

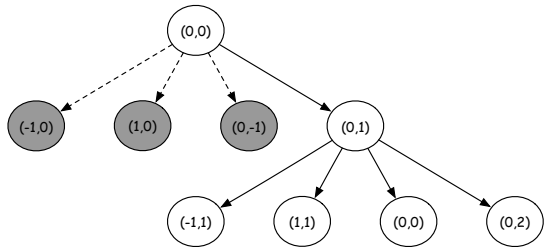


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

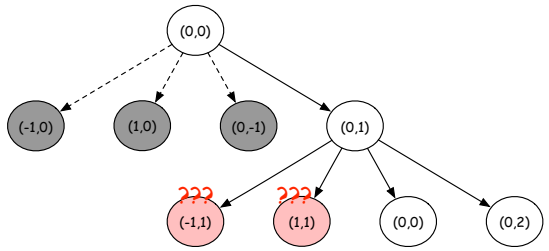


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

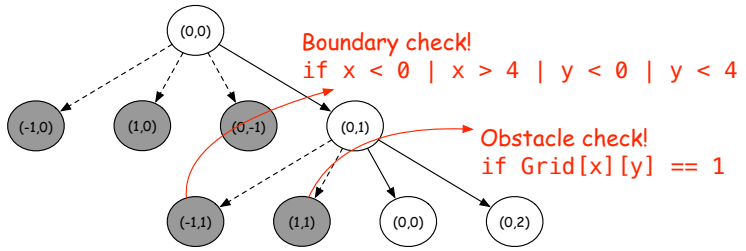


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

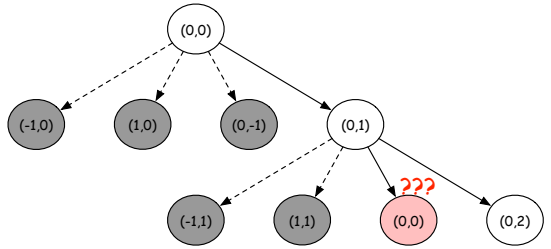


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

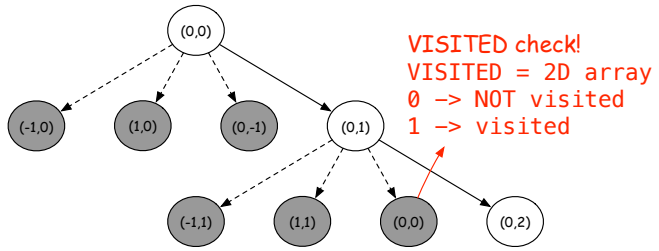


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

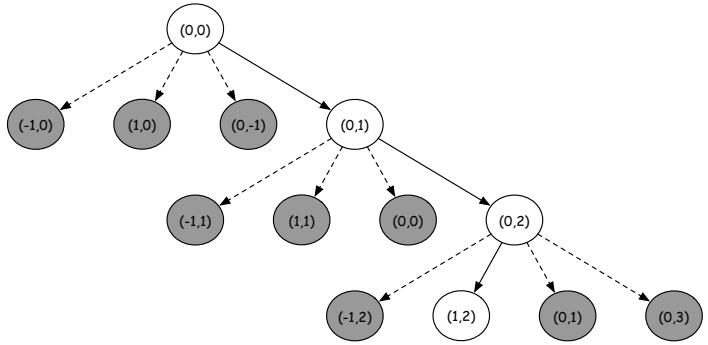


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

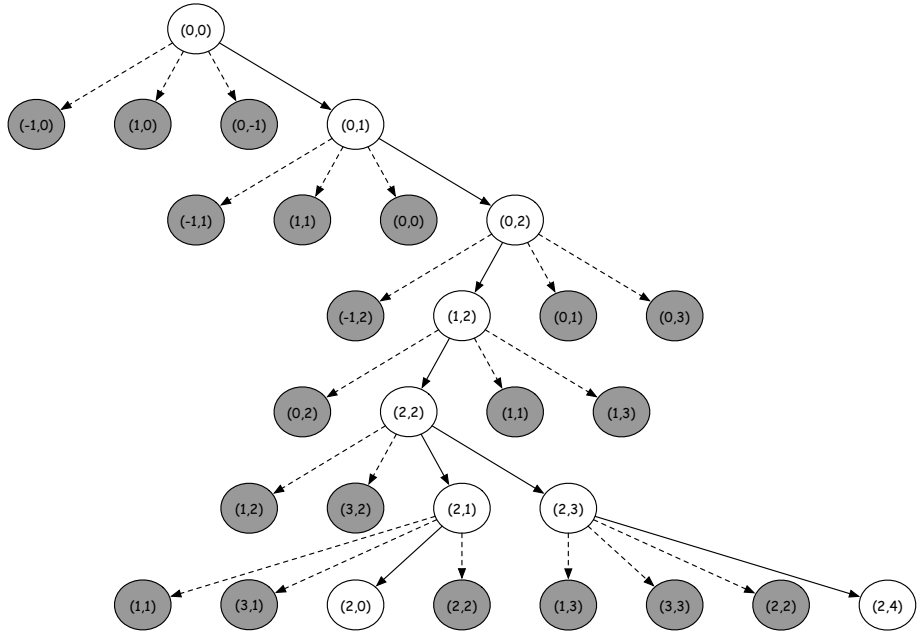


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

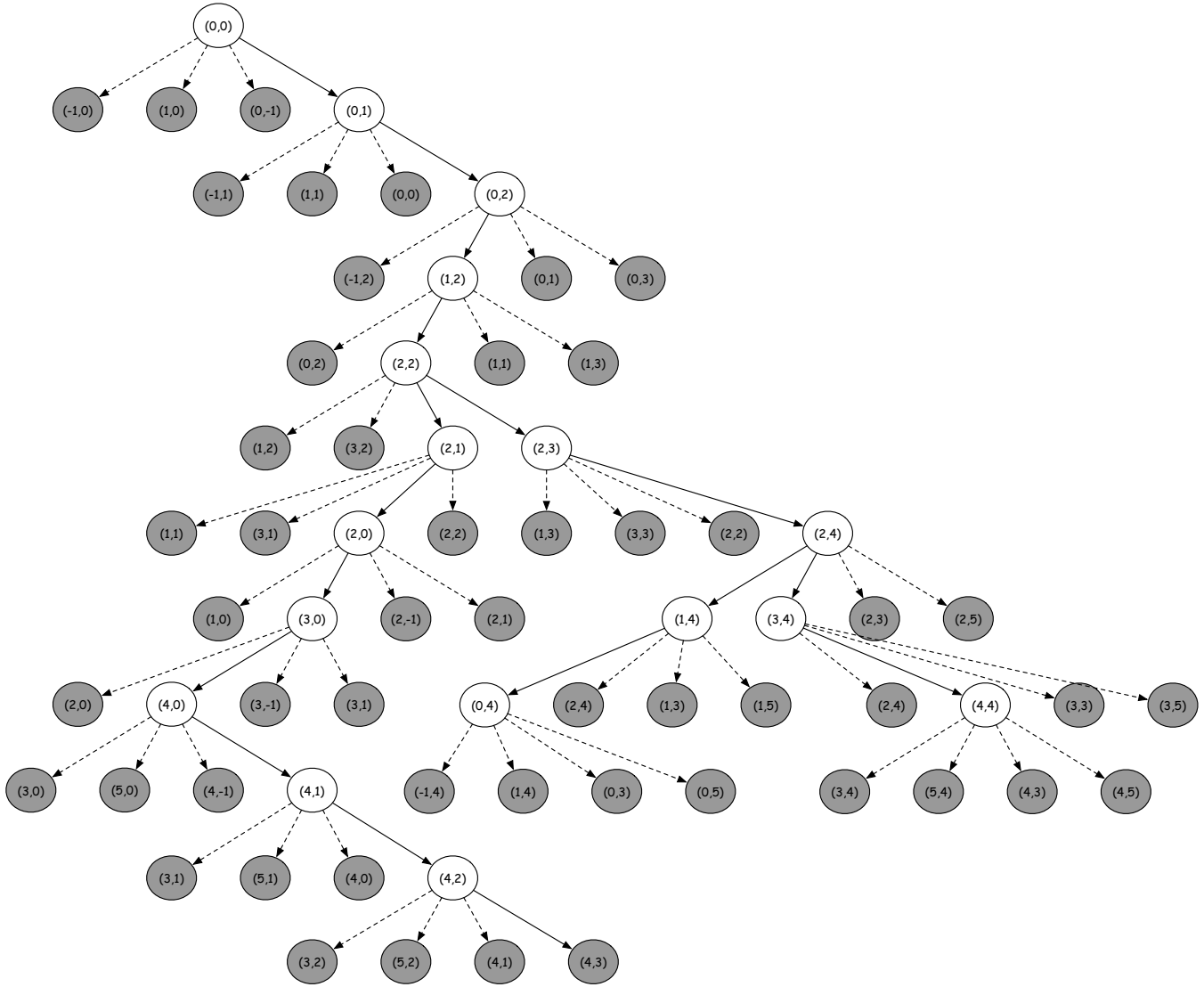


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

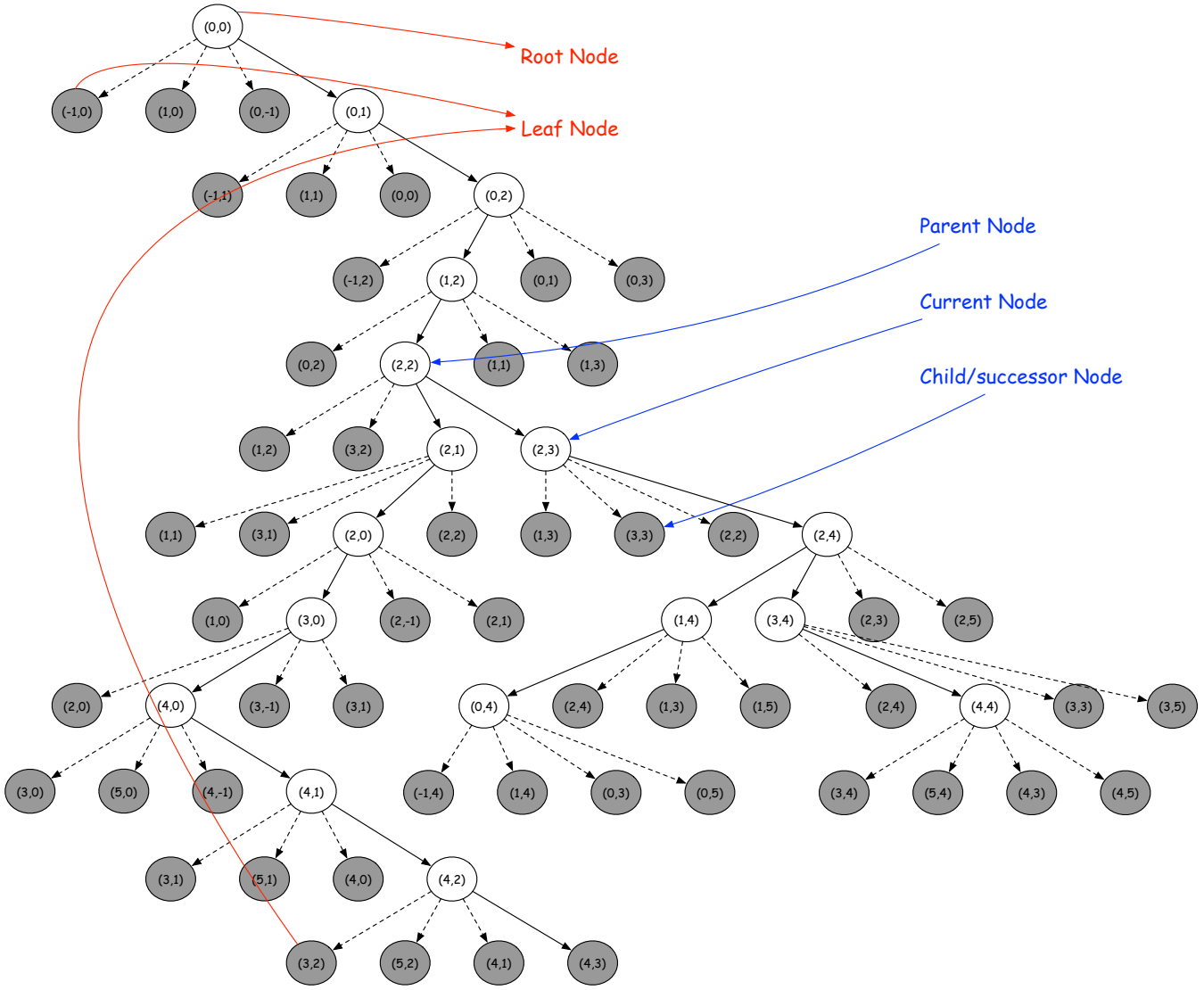


- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0



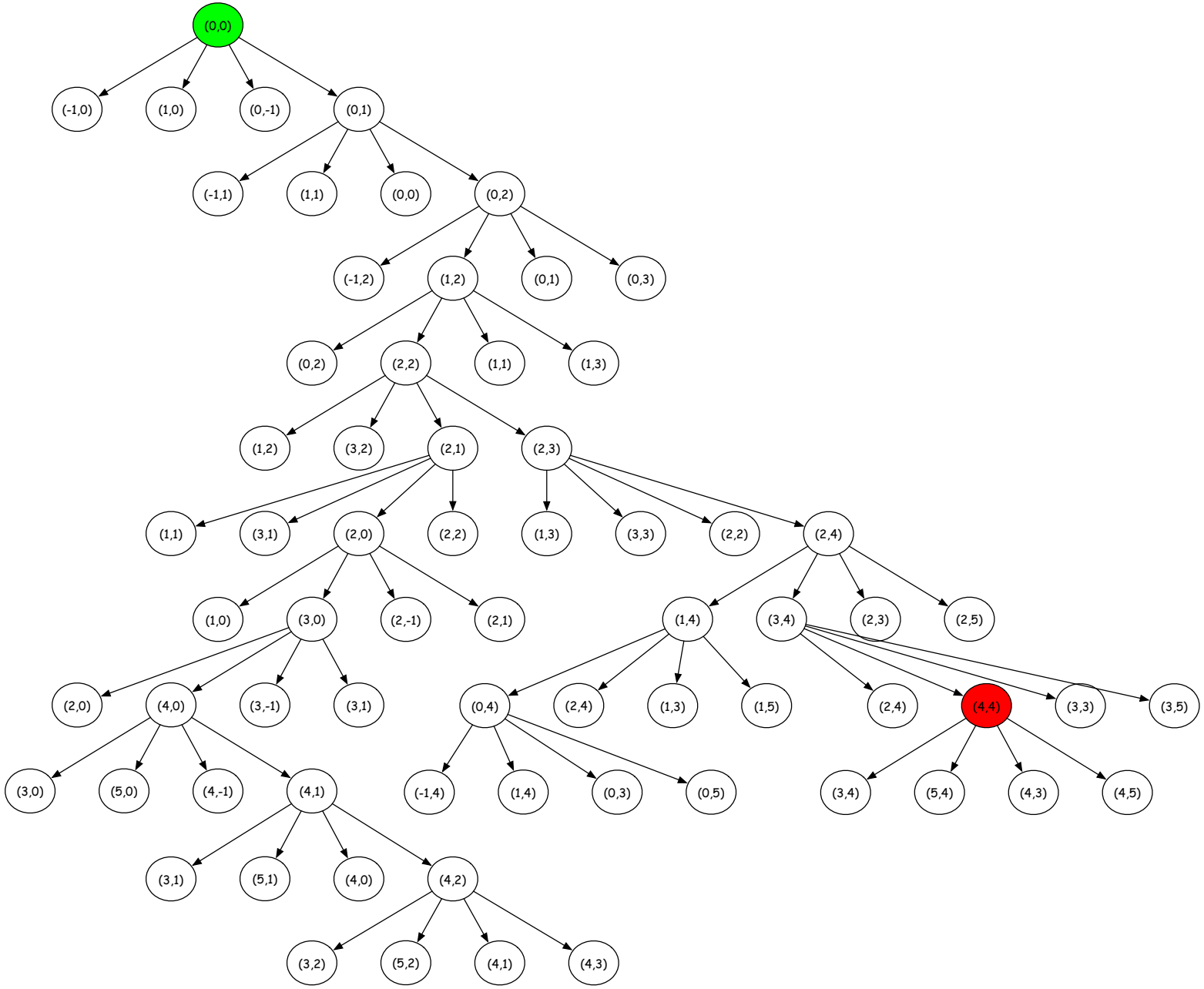
- Representing the state/search space is the first step toward solving the problem.
- The state space comprises all possible states and can be represented as a graph.

Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.

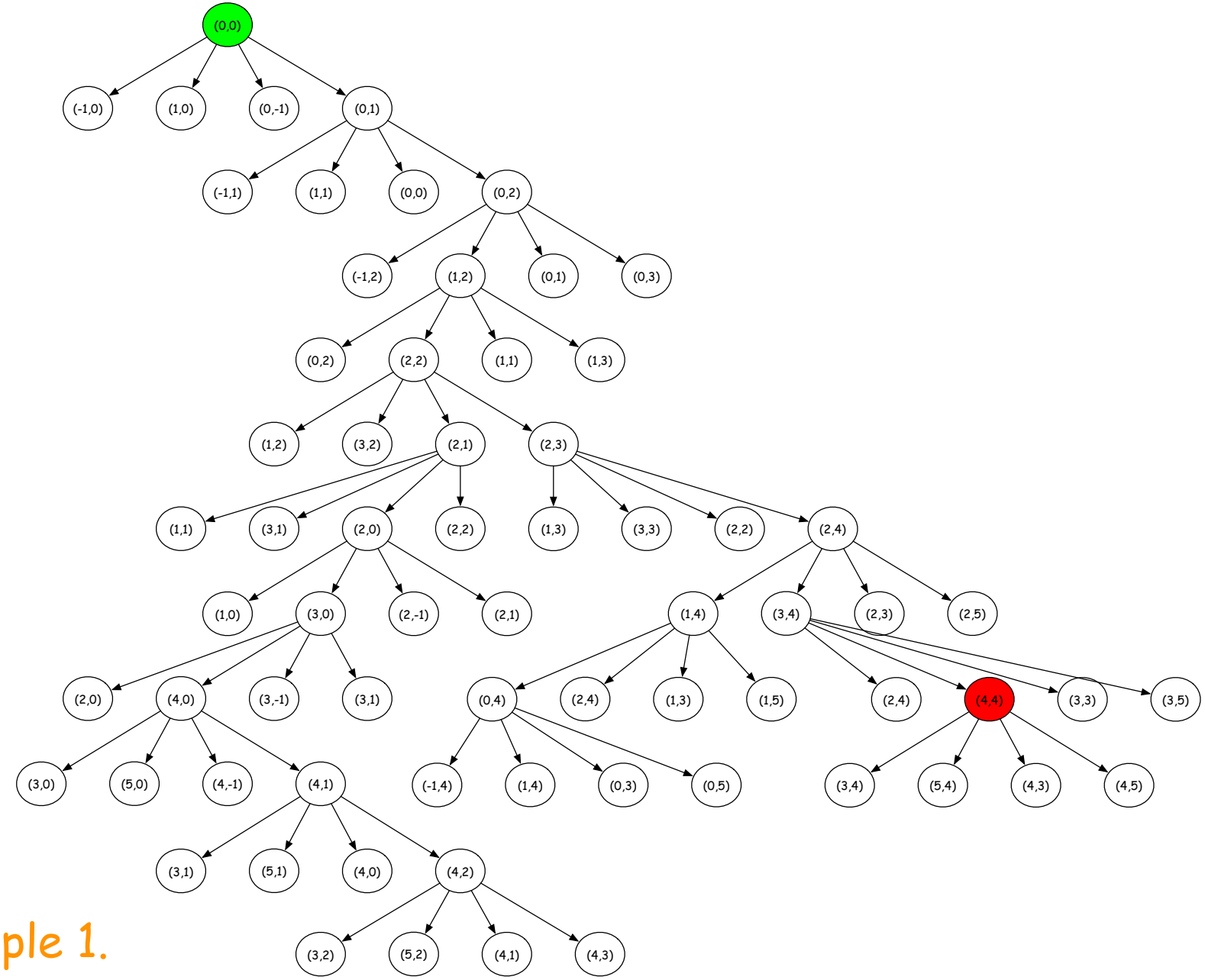


Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.

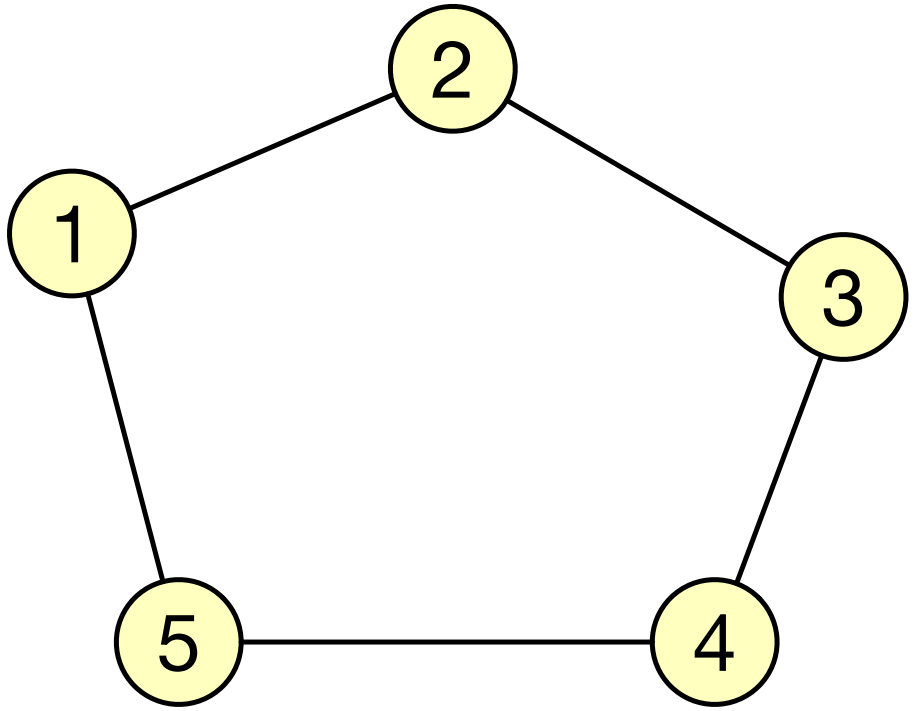


Basically, it's the same as Example 1.

Example 1: Vertex Traversal in an Undirected Graph

Given an undirected graph $G = (V, E)$, where:

- $V = \{1,2,3,4,5\}$
- $E = \{(1,2), (1,5), (2,3), (3,4), (4,5)\}$



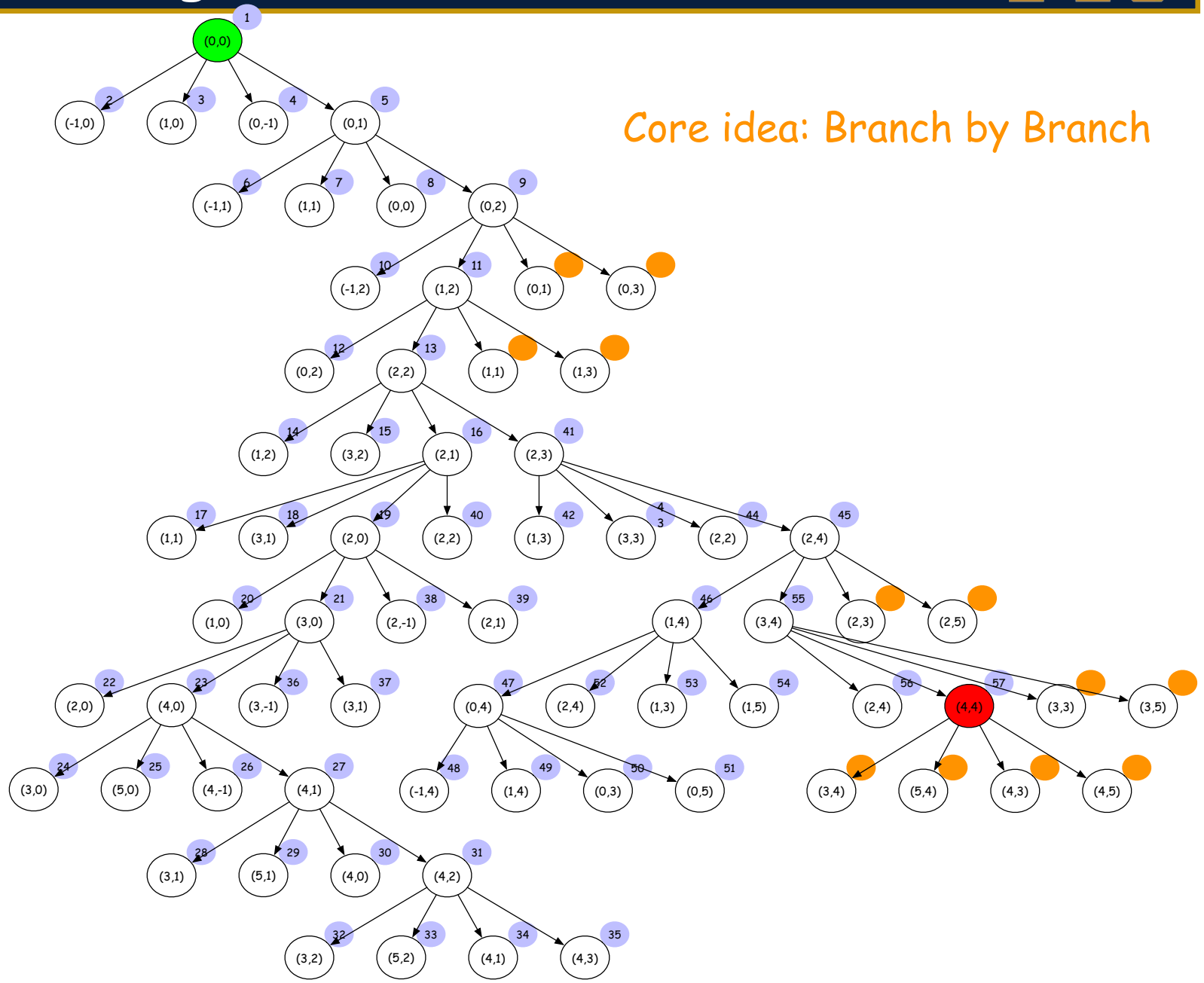
1. Use **Depth-First Search (DFS)** starting from vertex 1 and output the order of visited vertices.
2. Use **Breadth-First Search (BFS)** starting from vertex 1 and output the order of visited vertices.

Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.



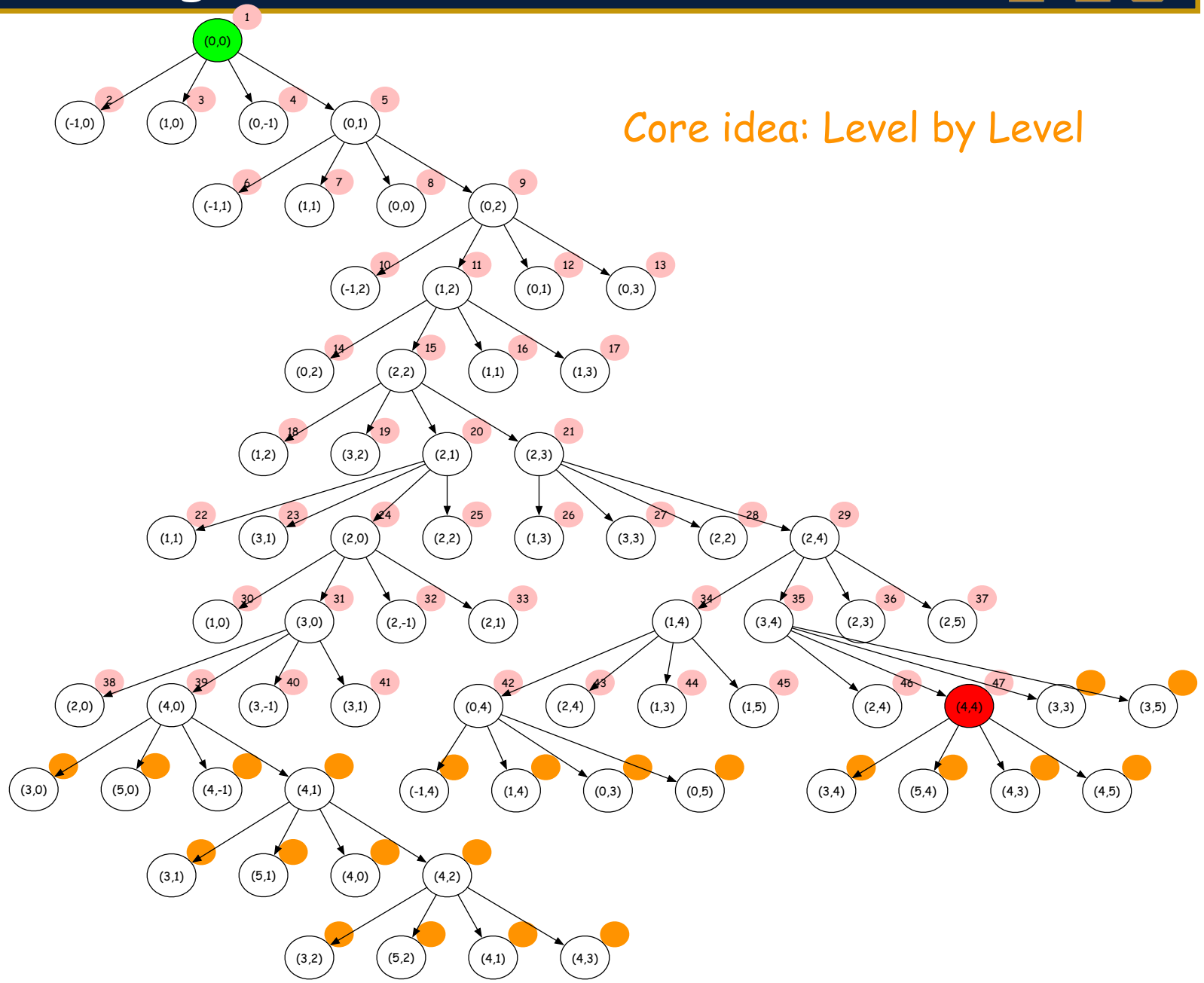
Core idea: Branch by Branch

Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.

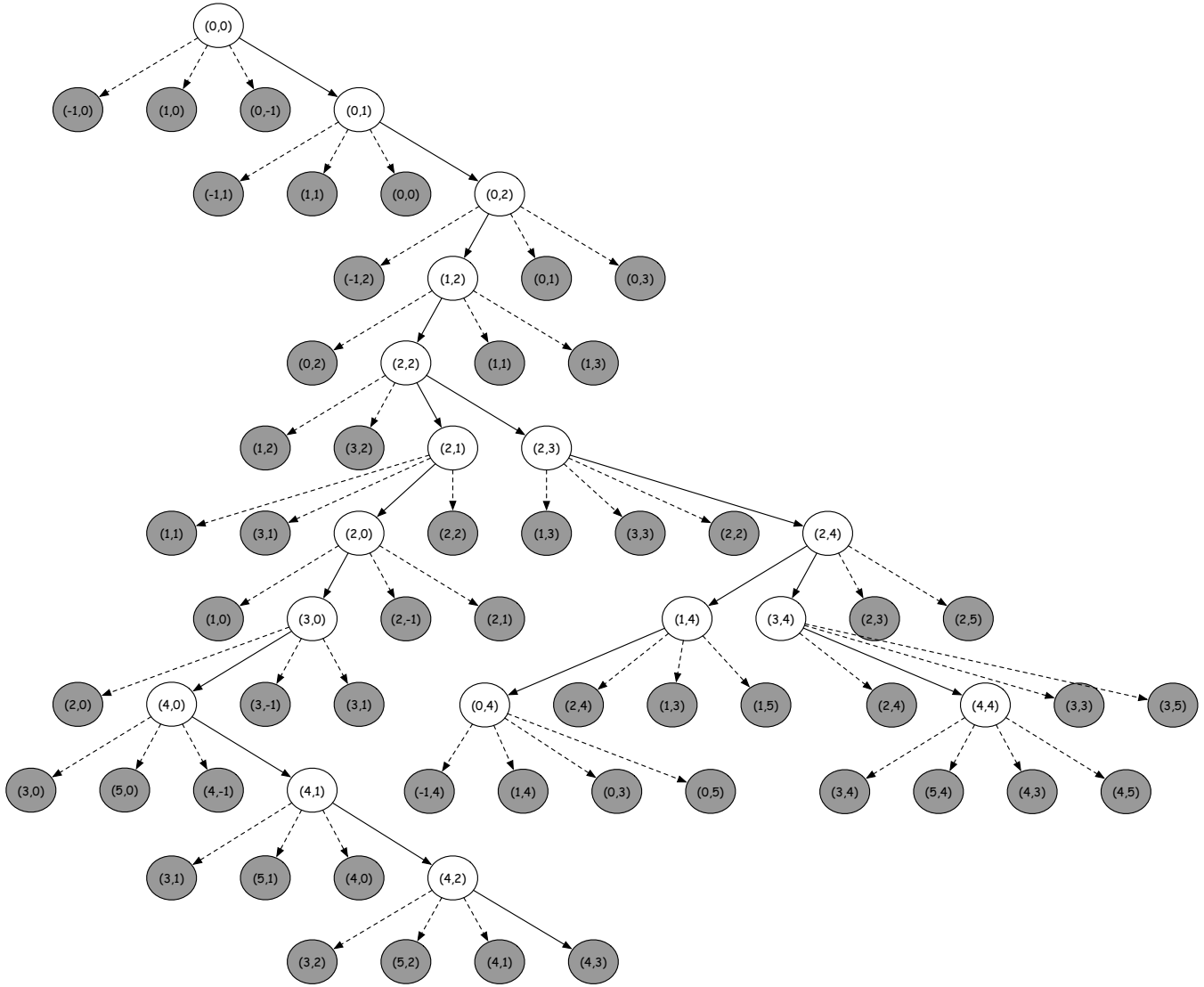


Core idea: Level by Level

Example 3: Shortest Pathfinding

- Given the following Grid, where 0 and 1 represent road and obstacle, respectively, try to find the shortest path from the green block to the red block.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0



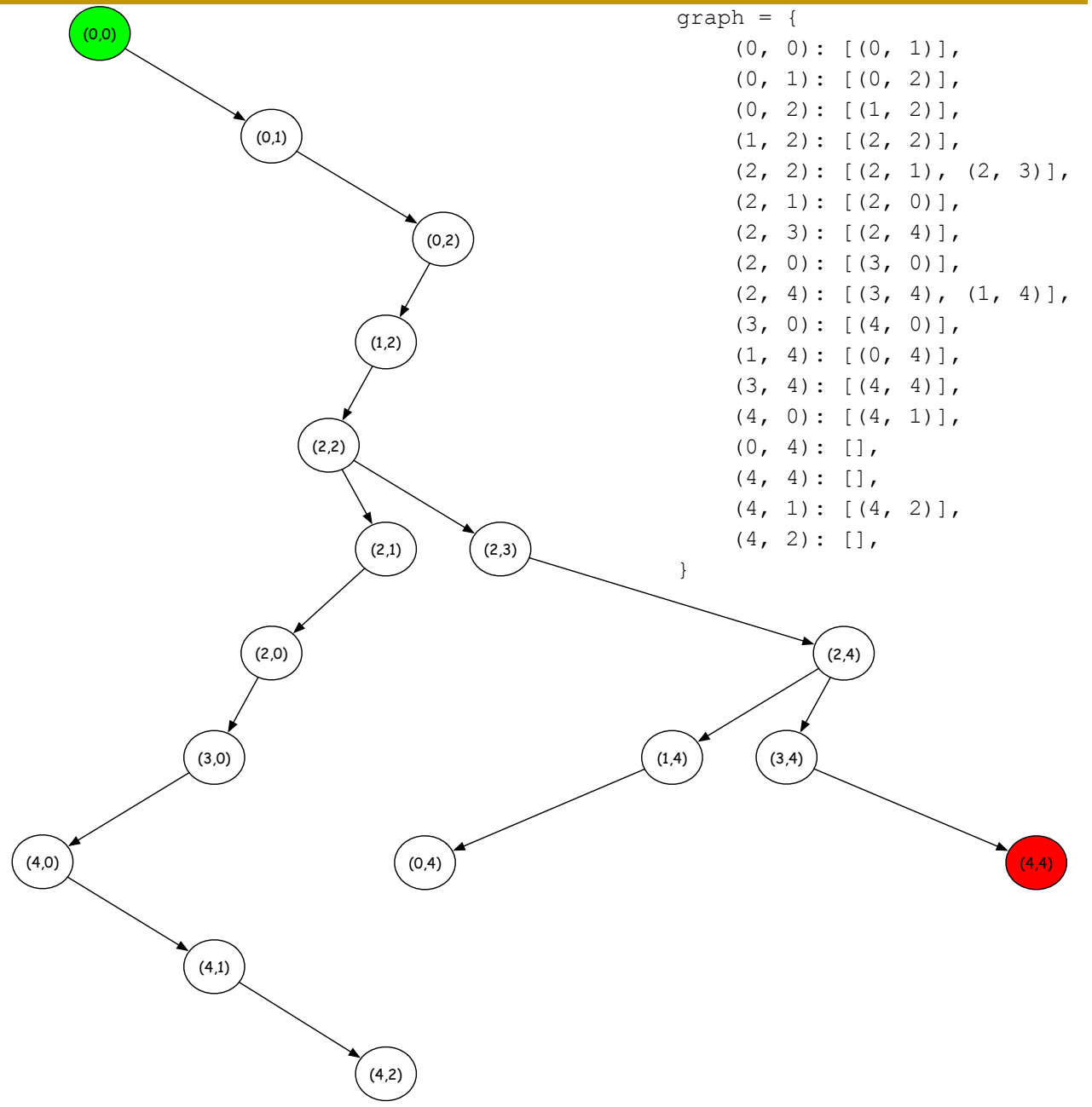
- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)

Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.



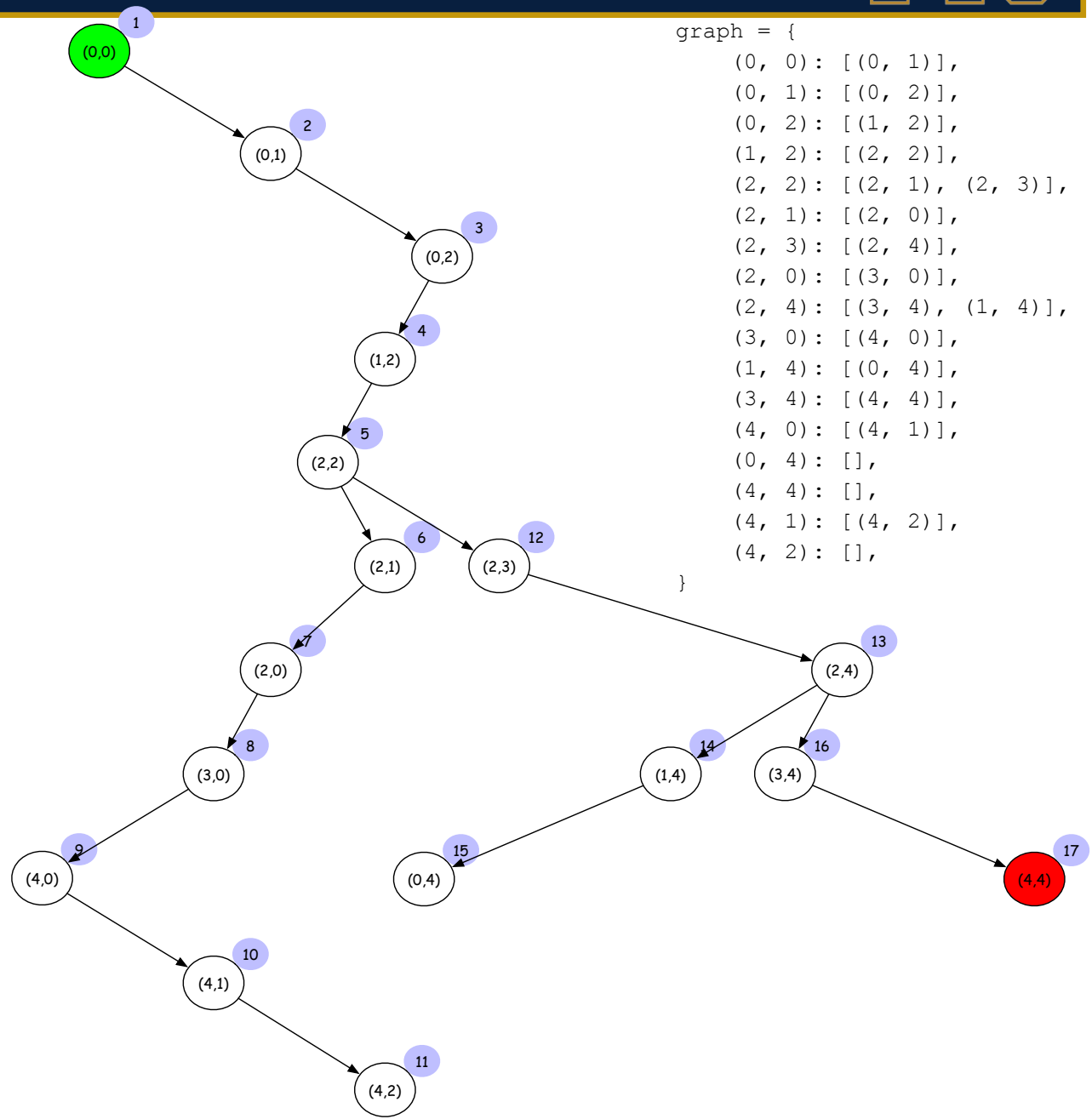

```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.



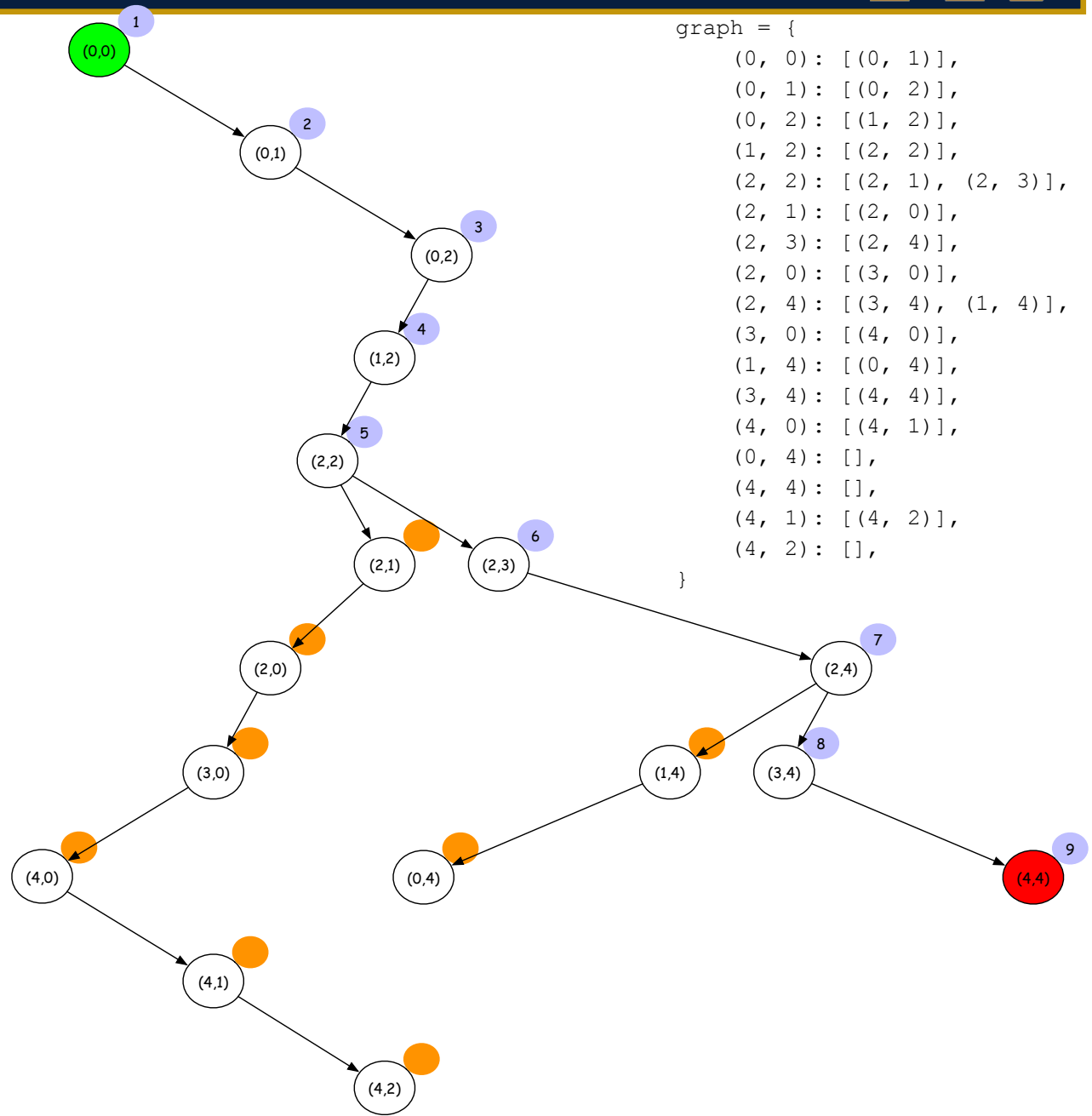

```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

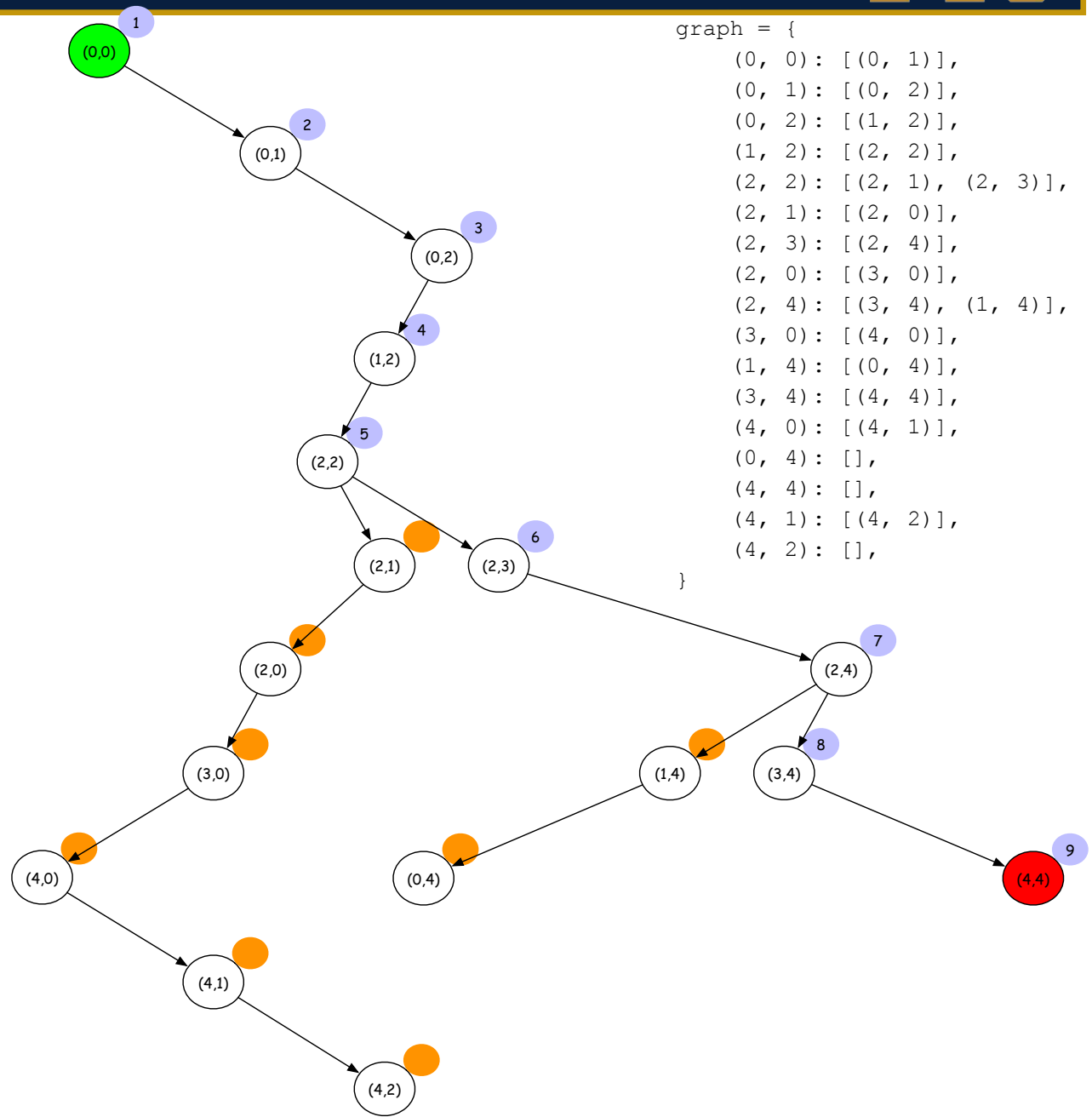
2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.




```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

Example 3: Shortest Pathfinding

```
def dfs(graph, start, goal, path=None,
    visited=None):
    if path is None:
        path = [start]
    if visited is None:
        visited = set()

    if start == goal:
        return path

    visited.add(start)

    for neighbor in graph.get(start, []):
        if neighbor not in visited:
            result = dfs(graph, neighbor,
      goal, path + [neighbor], visited)
            if result:
                return result
    return None
```

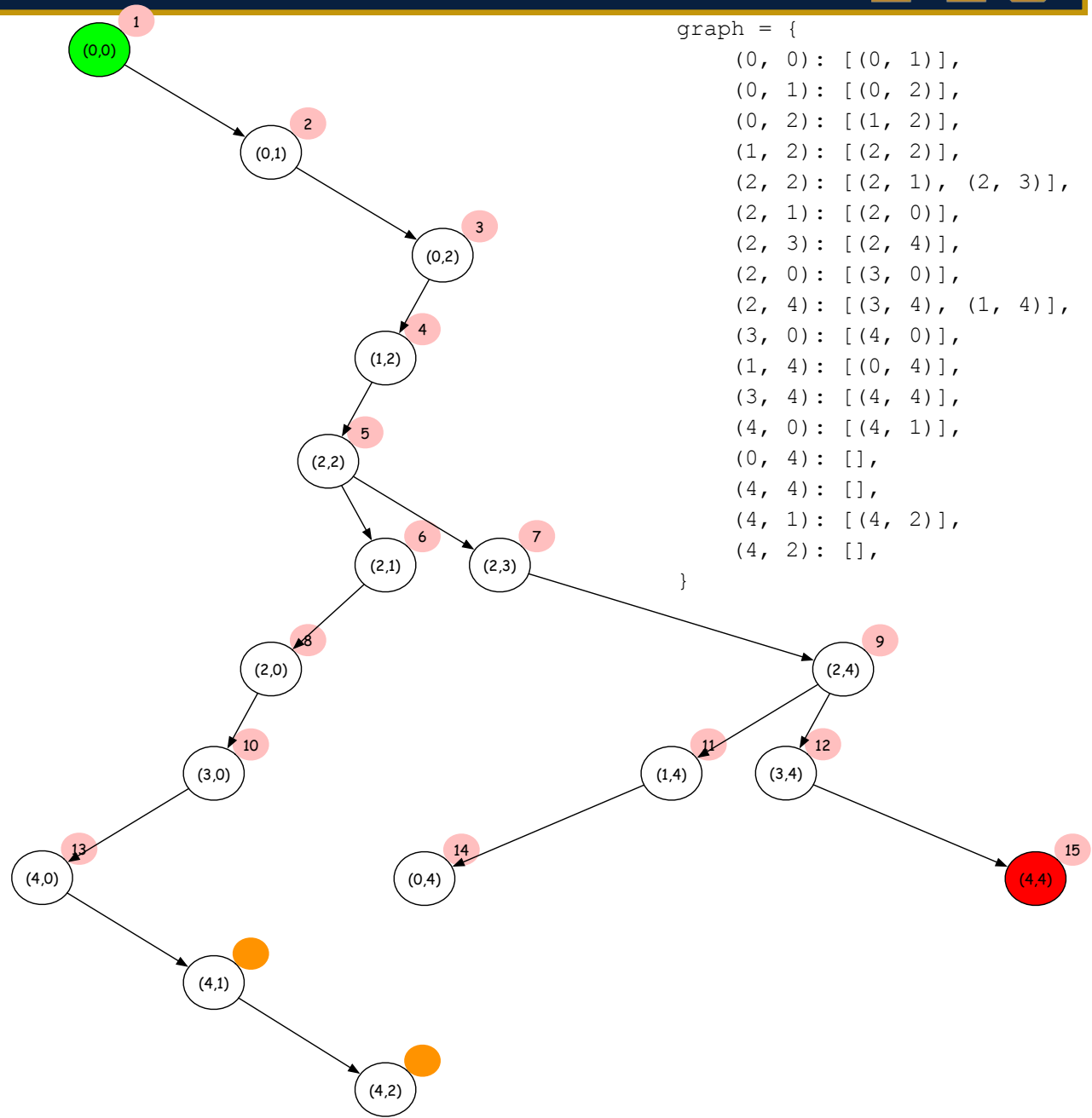


Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.

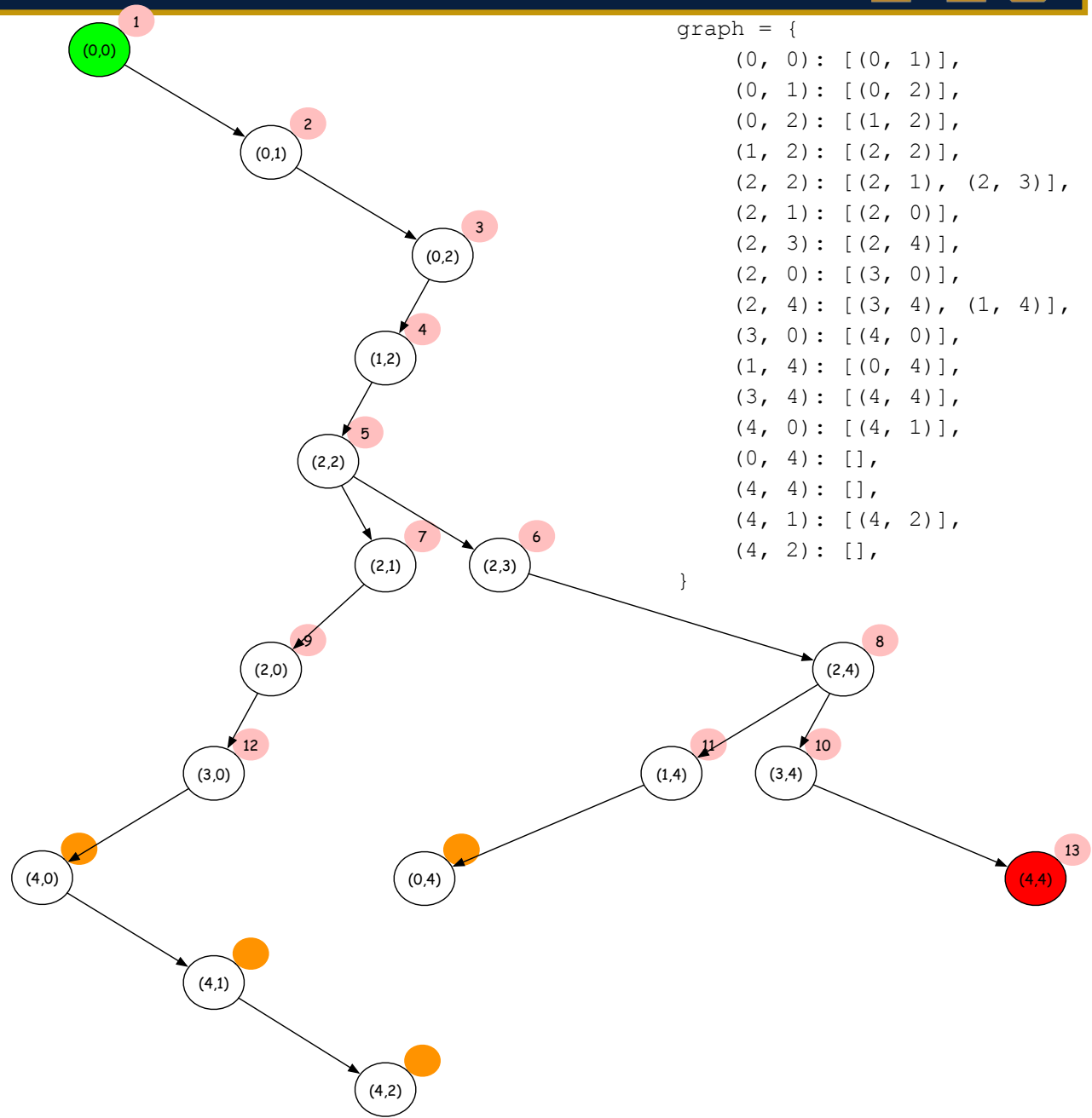


Example 3: Shortest Pathfinding

Given an directed graph $G = (V, E)$:

1. Use **Depth-First Search (DFS)** starting from vertex $(0,0)$ and output the order of visited vertices.

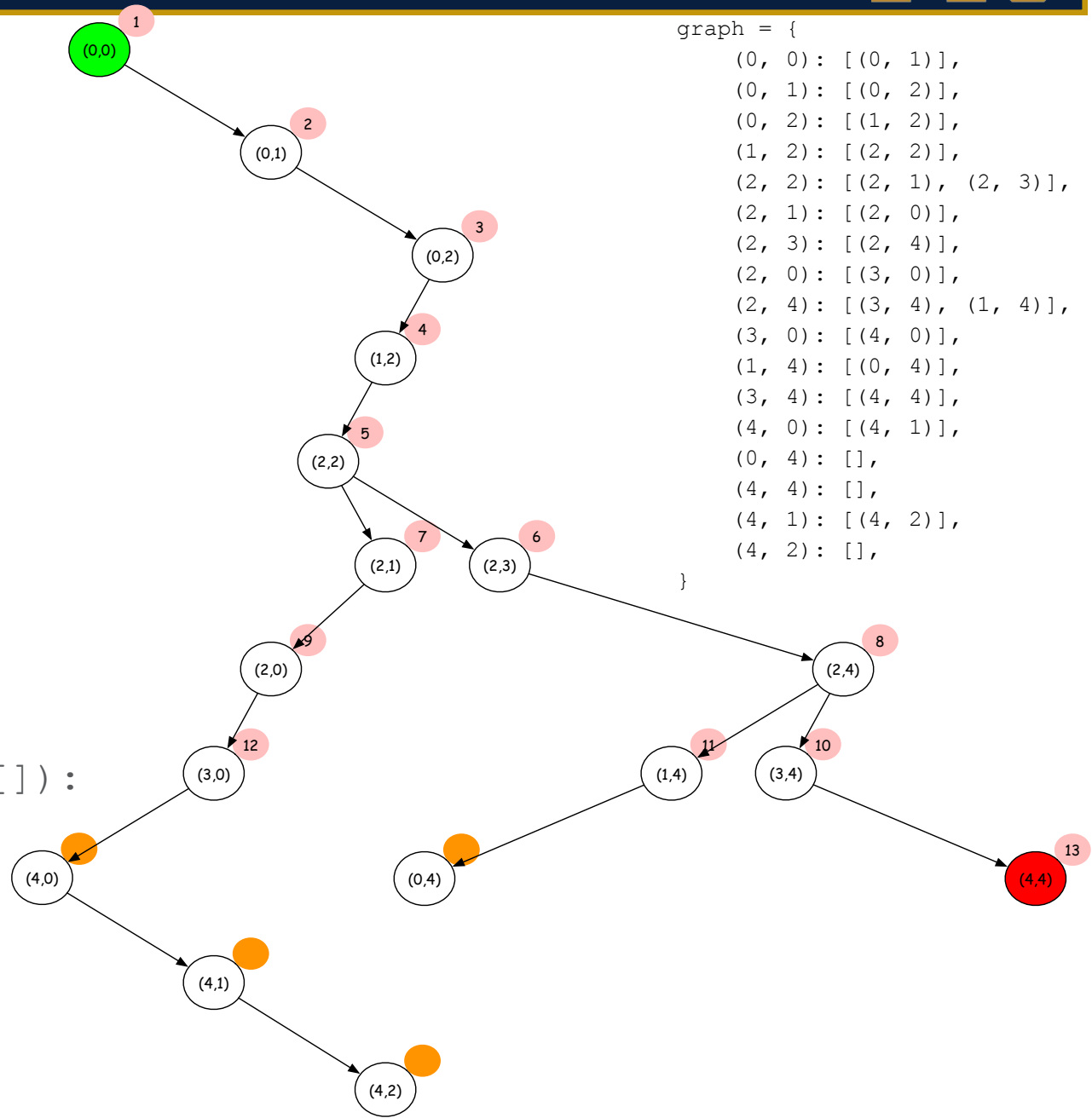
2. Use **Breadth-First Search (BFS)** starting from vertex $(4,4)$ and output the order of visited vertices.




```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

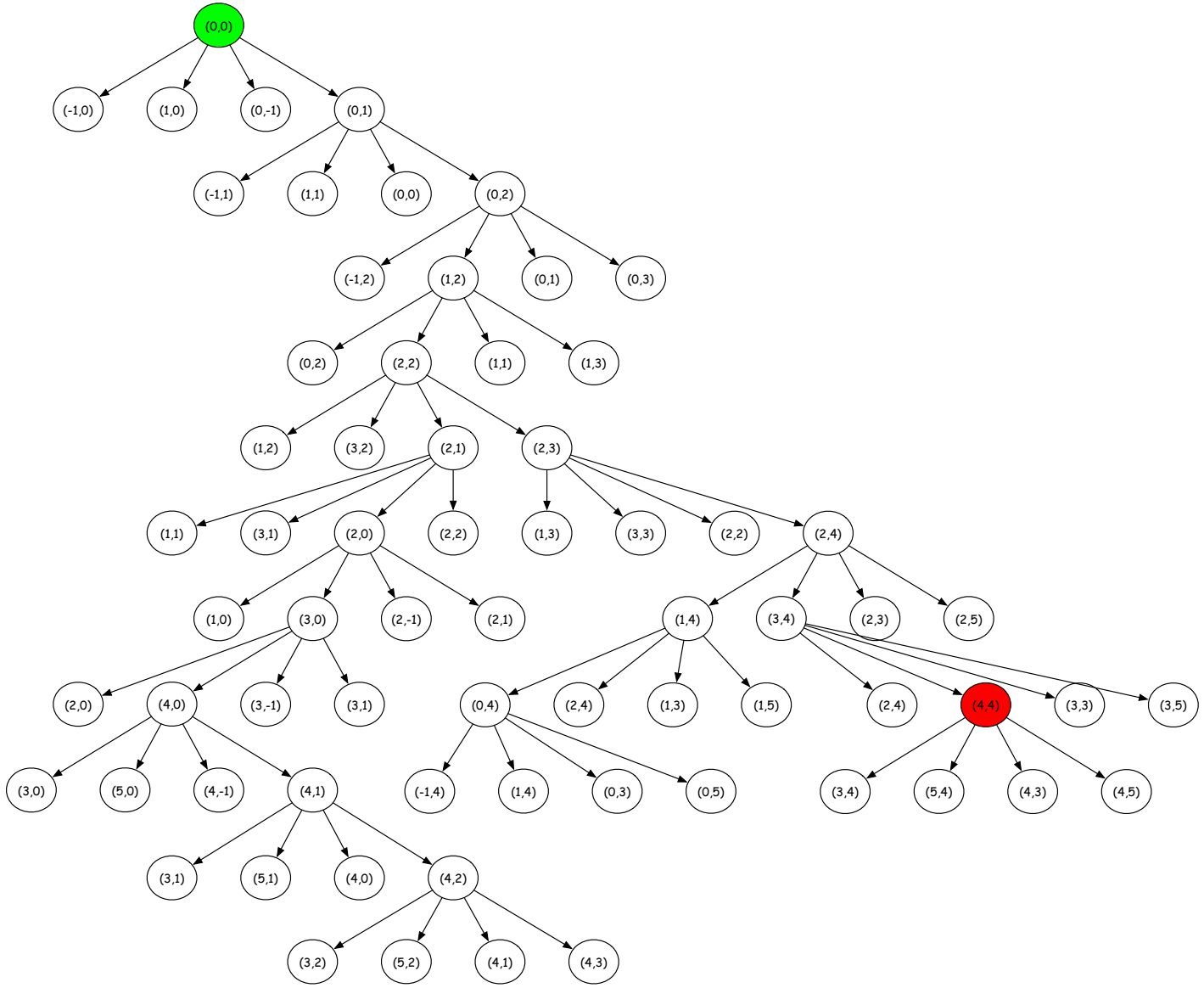
Example 3: Shortest Pathfinding

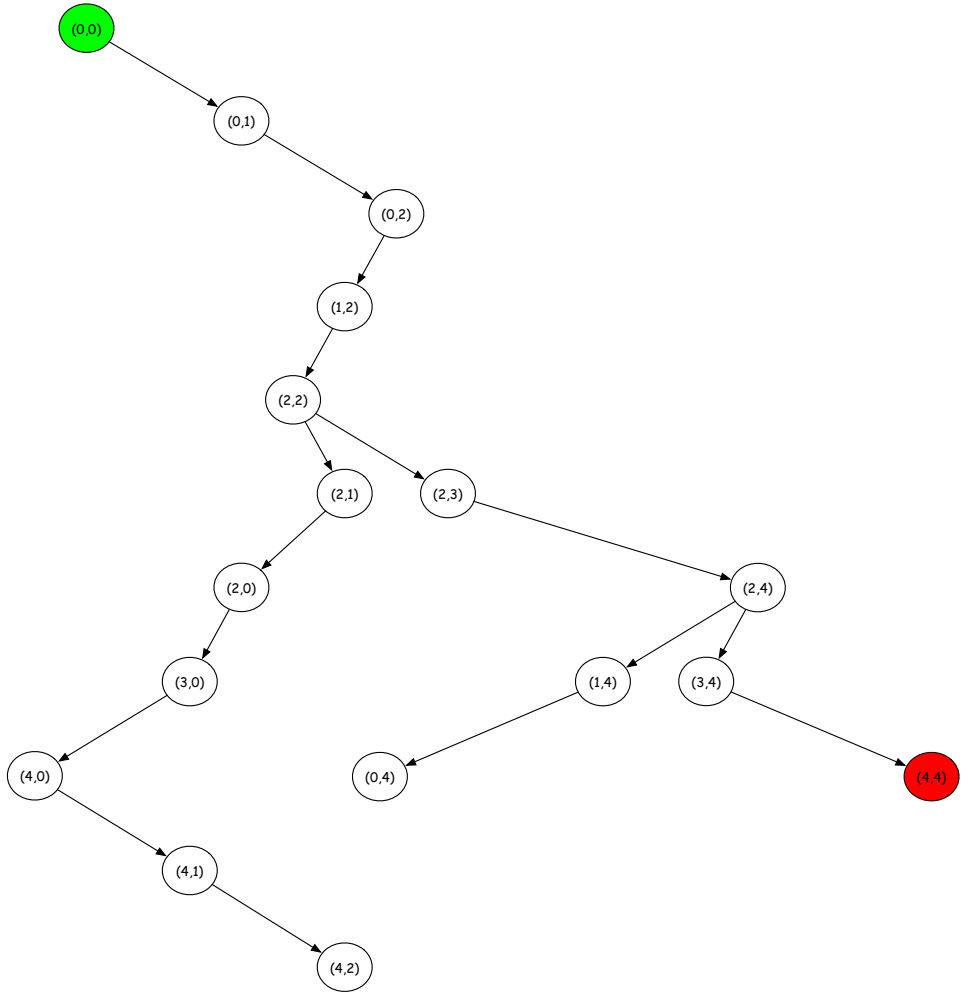
```python
from collections import deque

def bfs(graph, start, goal):
    queue = deque([[start]])
    visited = set()

    while queue:
        path = queue.popleft()
        node = path[-1]

        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in graph.get(node, []):
                new_path = path + [neighbor]
                queue.append(new_path)
    return None
```



Example 3: Shortest Pathfinding

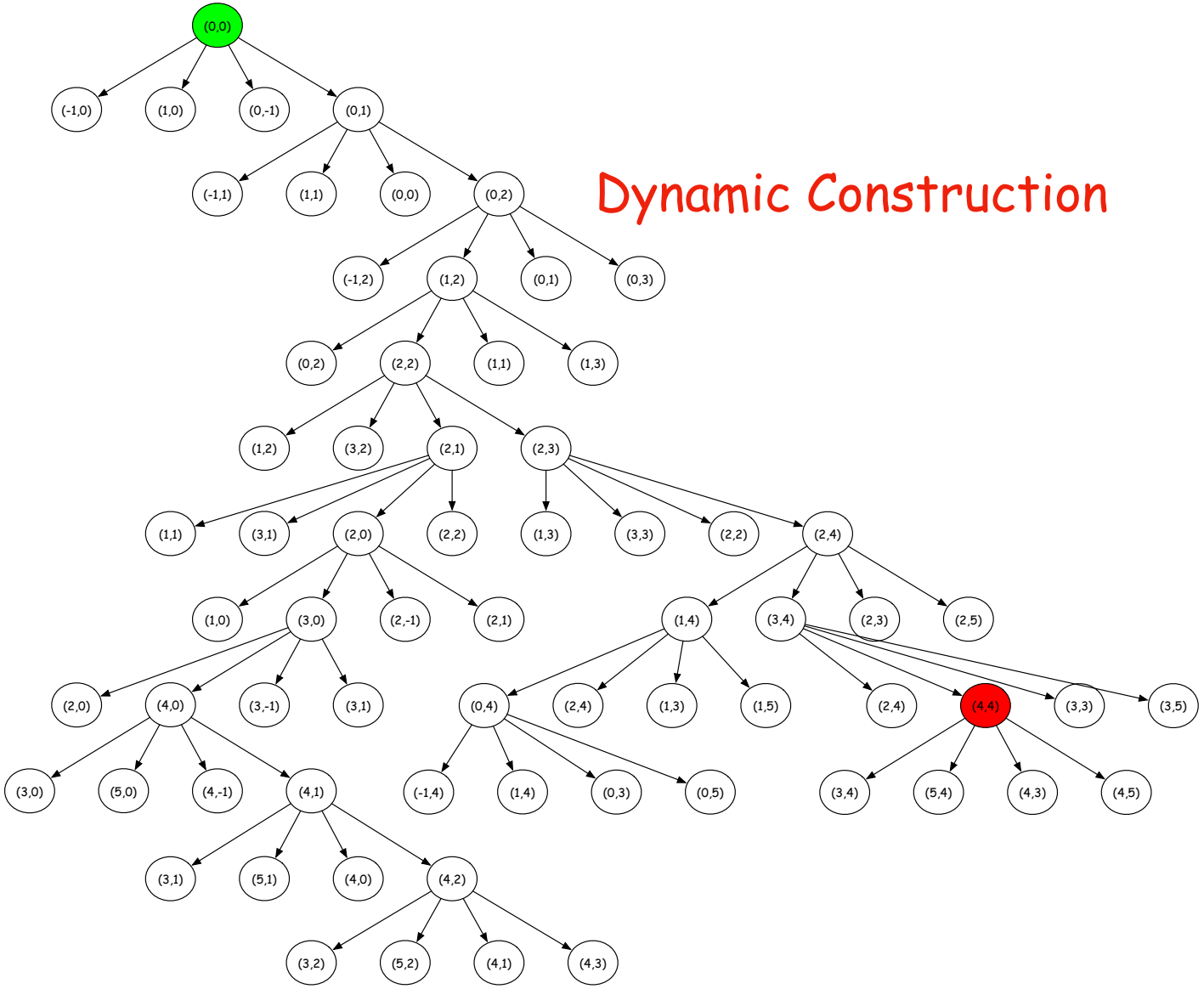
1. What's the difference between the following two graph/tree? Why?



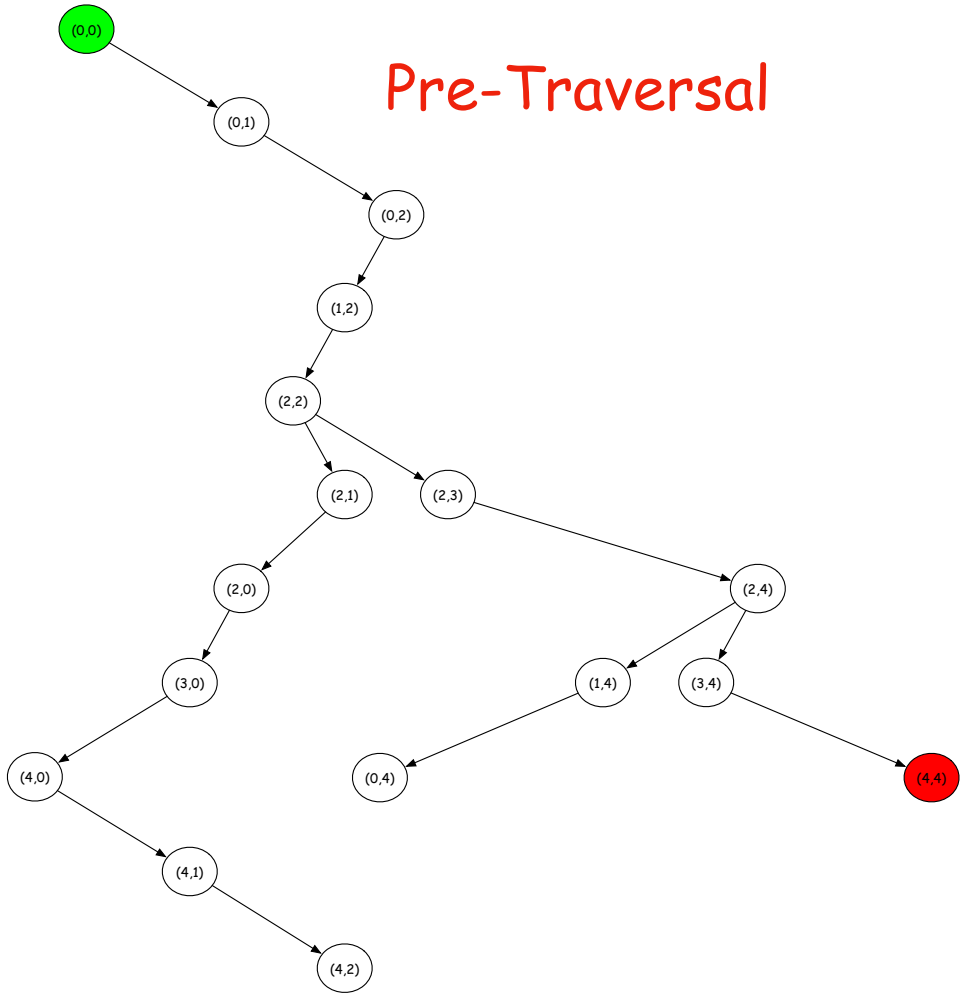


Example 3: Shortest Pathfinding

1. What's the difference between the following two graph/tree? Why?



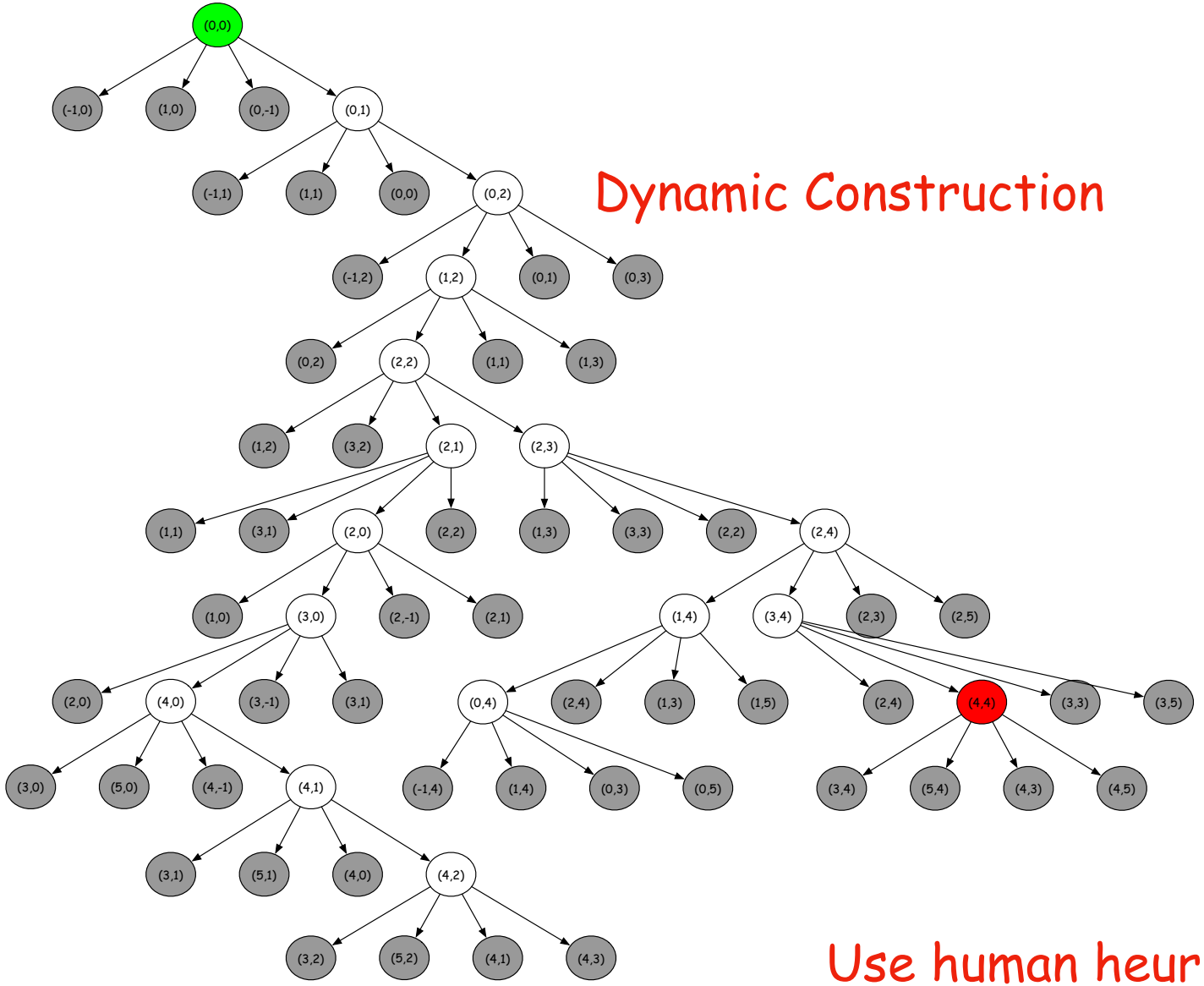
Dynamic Construction



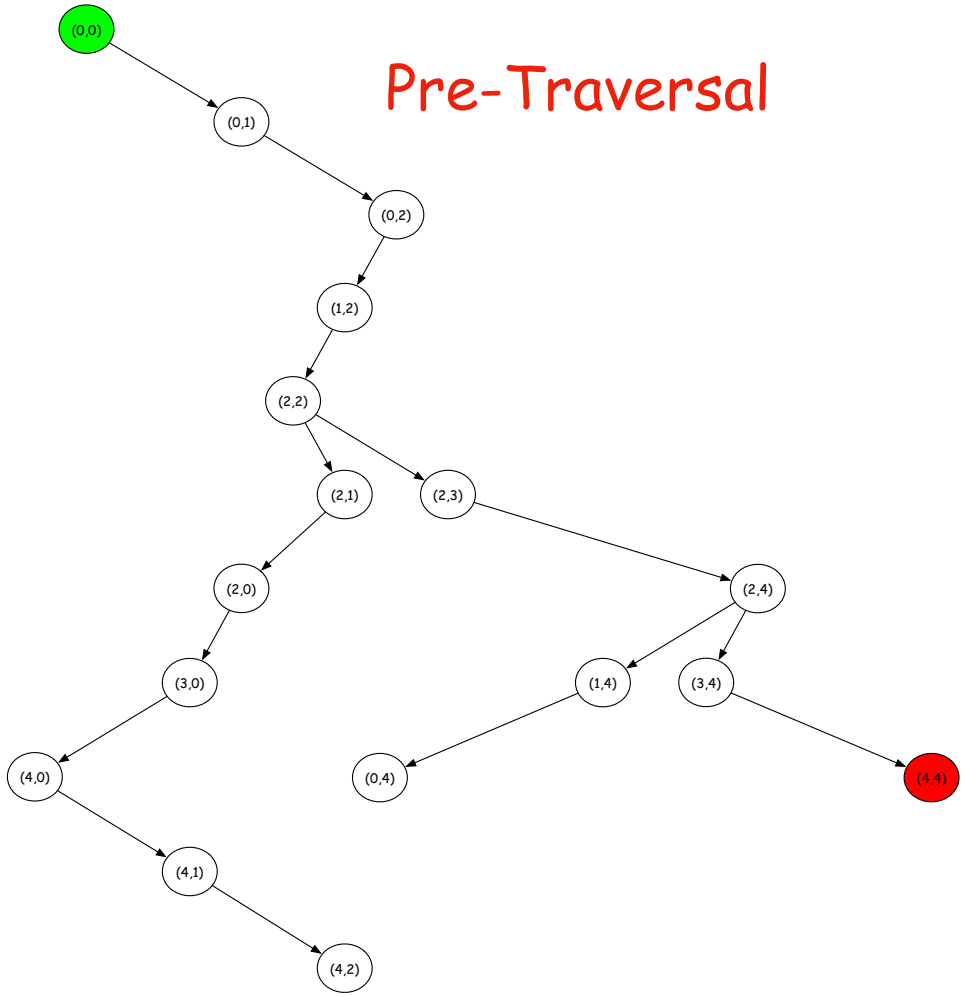
Pre-Traversal

Example 3: Shortest Pathfinding

1. What's the difference between the following two graph/tree? Why?



Dynamic Construction



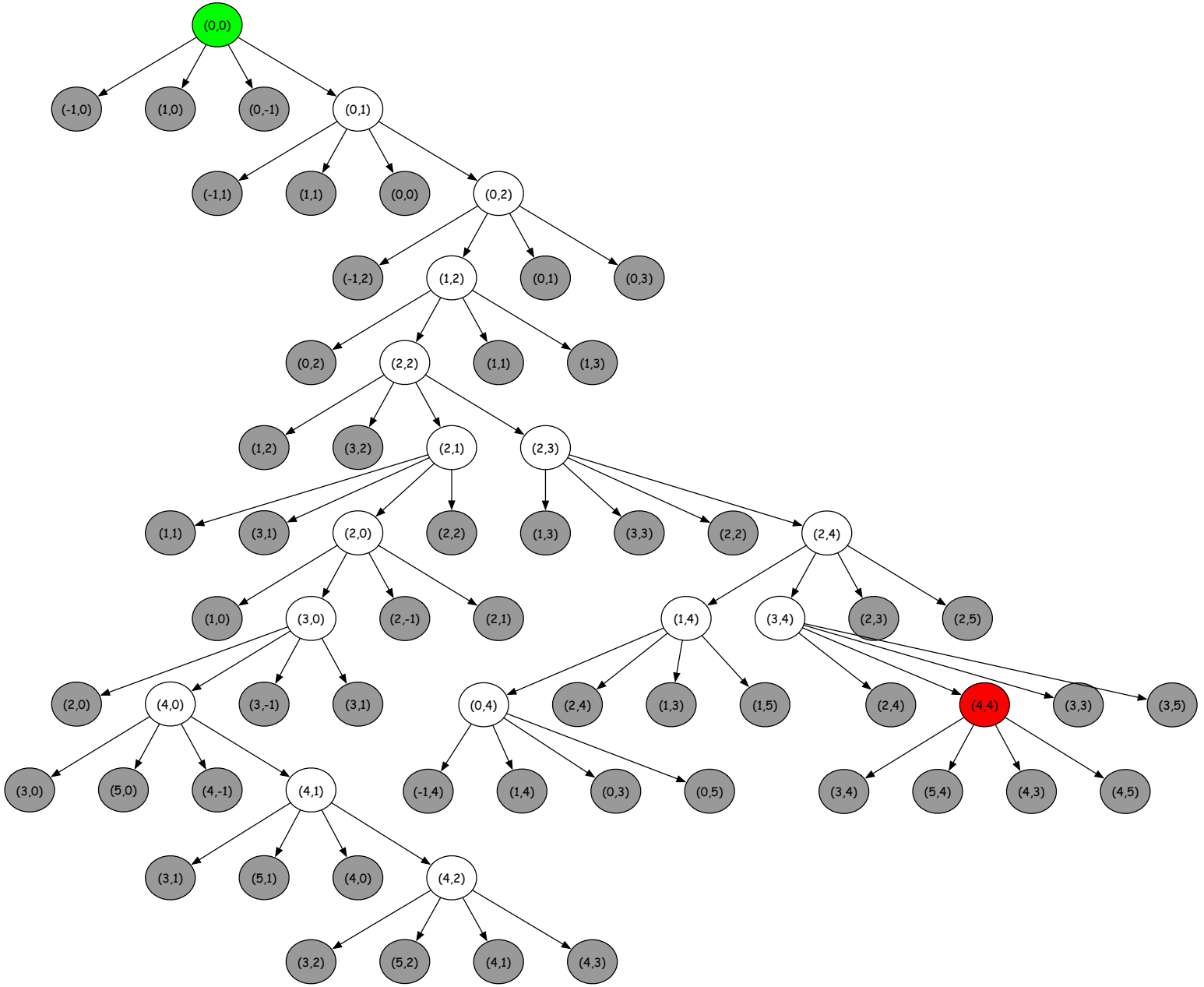
Pre-Traversal

Use human heuristics to remove the leaf nodes.

Example 3: Shortest Pathfinding

In practice, we could only follow the logic of the graph/tree structure instead of constructing the graph/tree explicitly.

In short, the "graph/tree" is a virtual logical structure only.



Example 3: Shortest Pathfinding

Pseudocode of BFS

```
BFS_Shortest_Path(Grid, Start, Goal):
    Initialize a queue Q
    Initialize a 2D array 'visited' with False values
    Add (Start_x, Start_y, 0) to Q  // (x, y, steps)
    Mark Start as visited

    while Q is not empty:
        (x, y, steps) = Q.dequeue()

        // If goal is reached, return steps
        if (x, y) == Goal:
            return steps

        // Explore neighbors in all directions (up, down, left, right)
        for each (dx, dy) in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_x = x + dx
            new_y = y + dy

            // Check if the move is valid (within grid bounds, not visited, not blocked)
            if is_valid(new_x, new_y) and not visited[new_x][new_y]:
                Mark (new_x, new_y) as visited
                Add (new_x, new_y, steps + 1) to Q

    // If the goal cannot be reached, return -1
    return -1
```

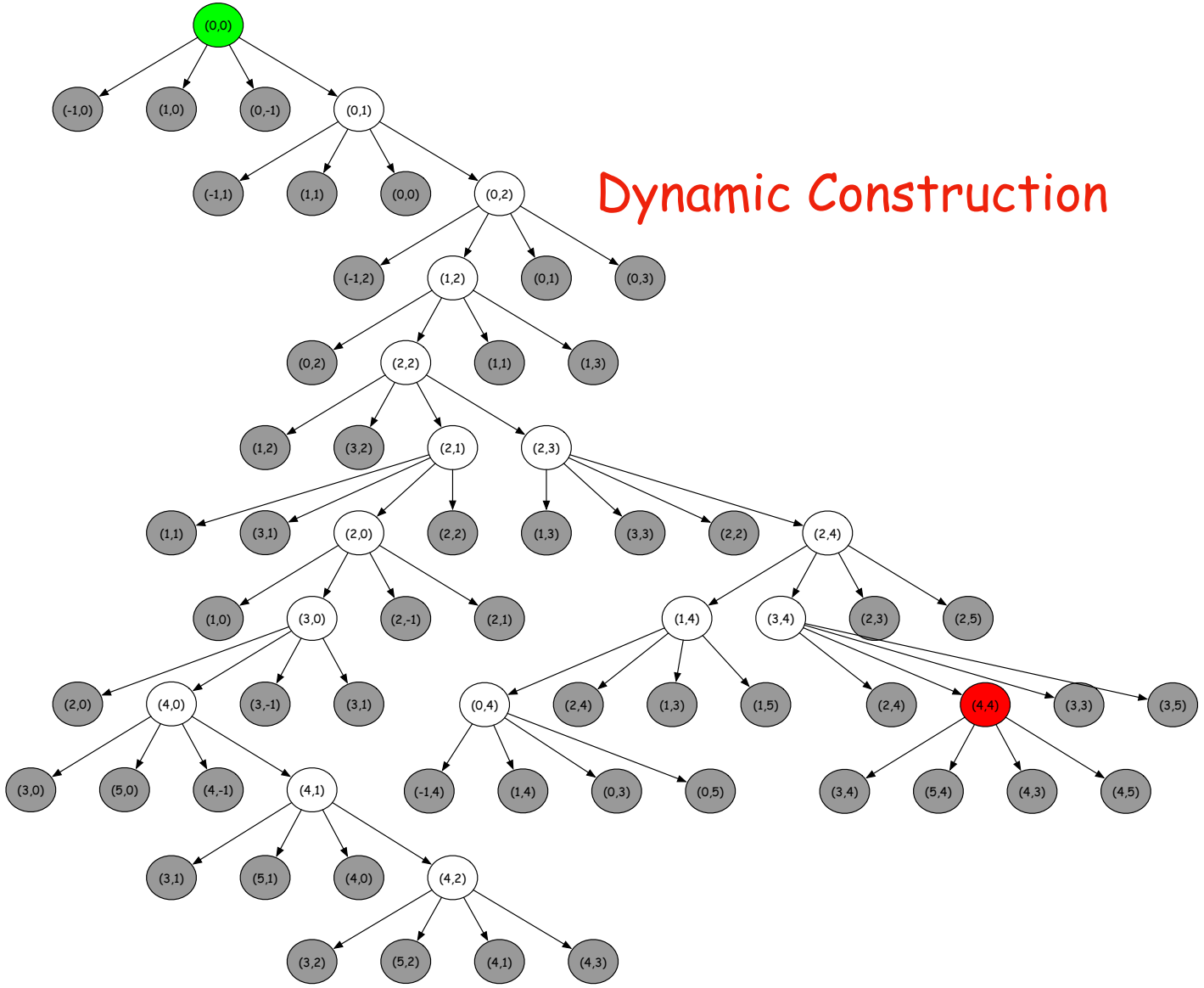
Example 3: Shortest Pathfinding

Pseudocode of DFS

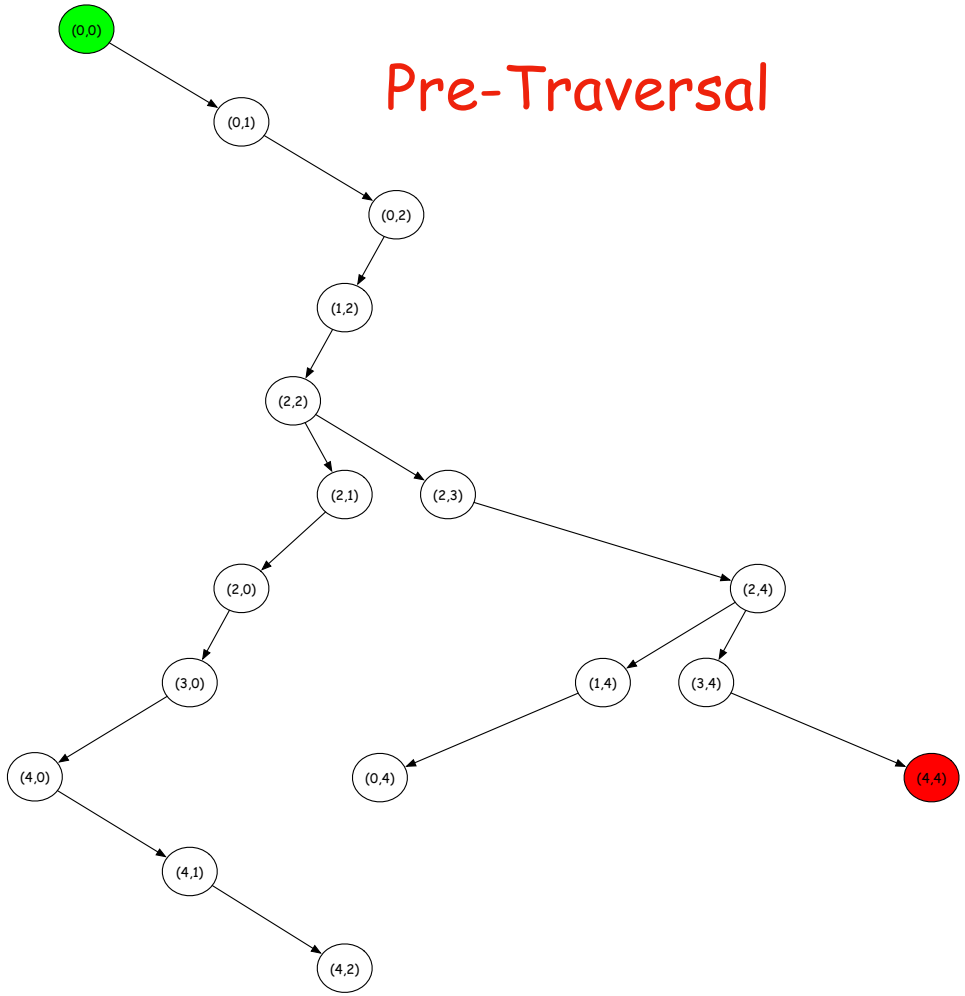
```
DFS_Shortest_Path_Stack(Grid, Start, Goal):
    Initialize a stack S
    Initialize a 2D array 'visited' with False values
    Add (Start_x, Start_y, 0) to S  // (x, y, steps)
    Set min_steps = infinity
    Set found = False

    while S is not empty:
        (x, y, steps) = S.pop()

        // If goal is reached, update min_steps
        if (x, y) == Goal:
            min_steps = min(min_steps, steps)
            found = True
            continue

        // Mark the current cell as visited
        Mark (x, y) as visited

        // Explore neighbors in all directions (up, down, left, right)
        for each (dx, dy) in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_x = x + dx
            new_y = y + dy

            // Check if the move is valid (within grid bounds, not visited, not blocked)
            if is_valid(new_x, new_y) and not visited[new_x][new_y]:
                Add (new_x, new_y, steps + 1) to S

    // If the goal was reached, return min_steps
    if found:
        return min_steps
    else:
         // use -1 to denote there is no solution
        return -1
```

Example 3: Shortest Pathfinding

1. What's the difference between the following two graph/tree? Why?



Dynamic Construction



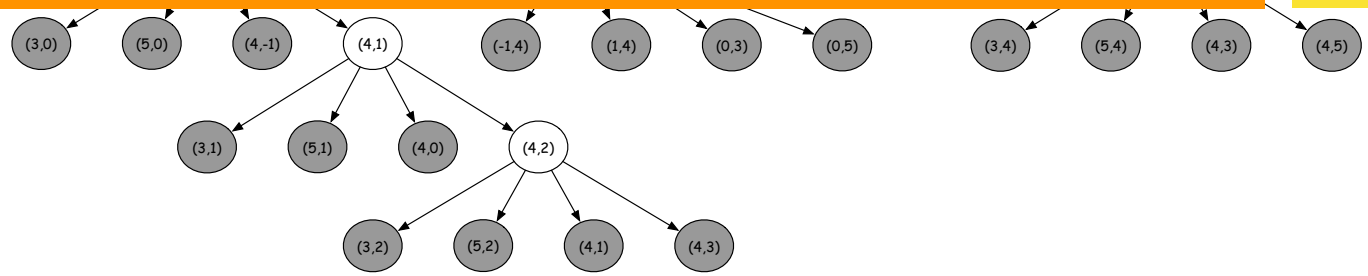
Pre-Traversal

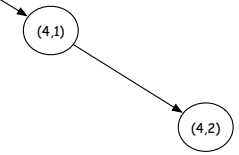
Example 3: Shortest Pathfinding

1. What's the difference between the following two graph/tree? Why?

```
BFS_Shortest_Path(Grid, Start, Goal):
    Initialize a queue Q
    Initialize a 2D array 'visited' with False values
    Add (Start_x, Start_y, 0) to Q  // (x, y, steps)
    Mark Start as visited

    while Q is not empty:
        (x, y, steps) = Q.dequeue()

       if (x, y) == Goal:
            return steps

       for each (dx, dy) in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_x = x + dx
            new_y = y + dy

           if is_valid(new_x, new_y) and not visited[new_x][new_y]:
                Mark (new_x, new_y) as visited
                Add (new_x, new_y, steps + 1) to Q

    return -1
```

```
from collections import deque

def bfs(graph, start, goal):
    queue = deque([[start]])
    visited = set()

    while queue:
        path = queue.popleft()
        node = path[-1]

        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in graph.get(node, []):
                new_path = path + [neighbor]
                queue.append(new_path)
    return None
```



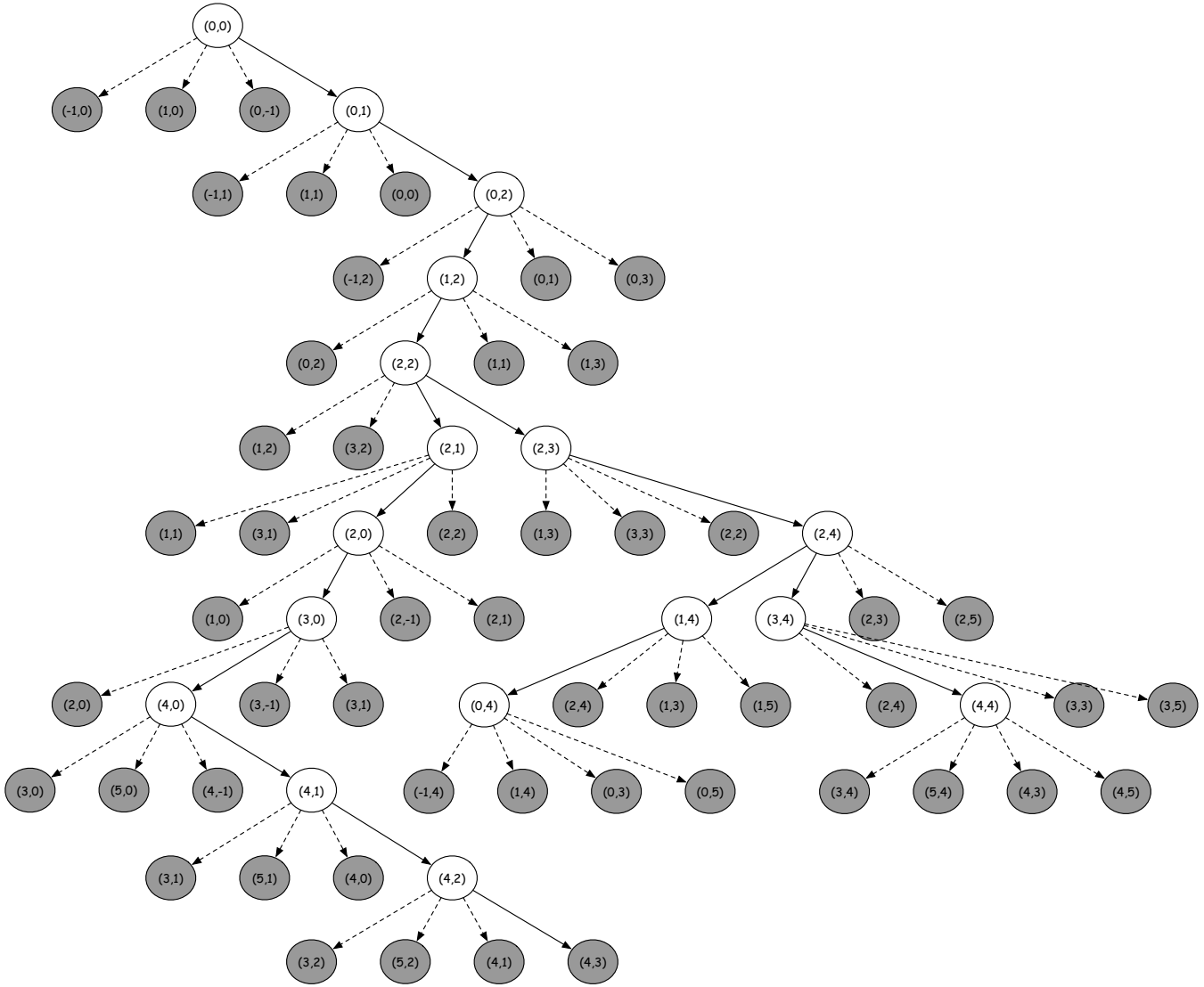


Example 3: Shortest Pathfinding

Do we have other possible graph/tree structure for this problem?

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

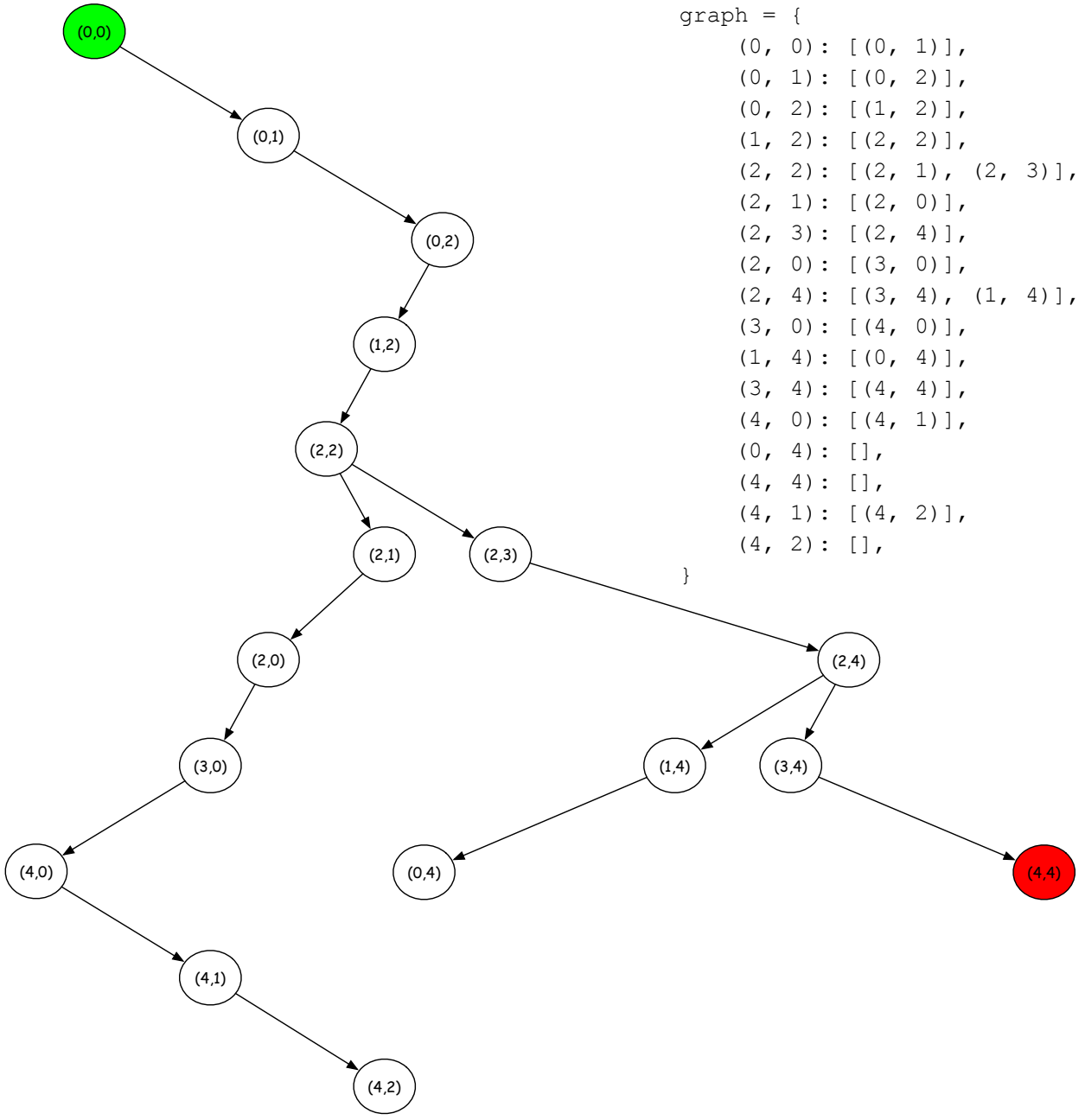
- Vertex:
  - States/positions: (x, y)
  - Root node --> initial state --> (0, 0)
  - Goal state --> (4, 4)
- Edge:
  - move between positions:
  - Up, down, left, right
  - (-1, 0), (1, 0), (0, -1), (0, 1)



Example 3: Shortest Pathfinding

For easy to show the logic, let's start from the constructed simplified tree.

We have tried 4 traversal orders:



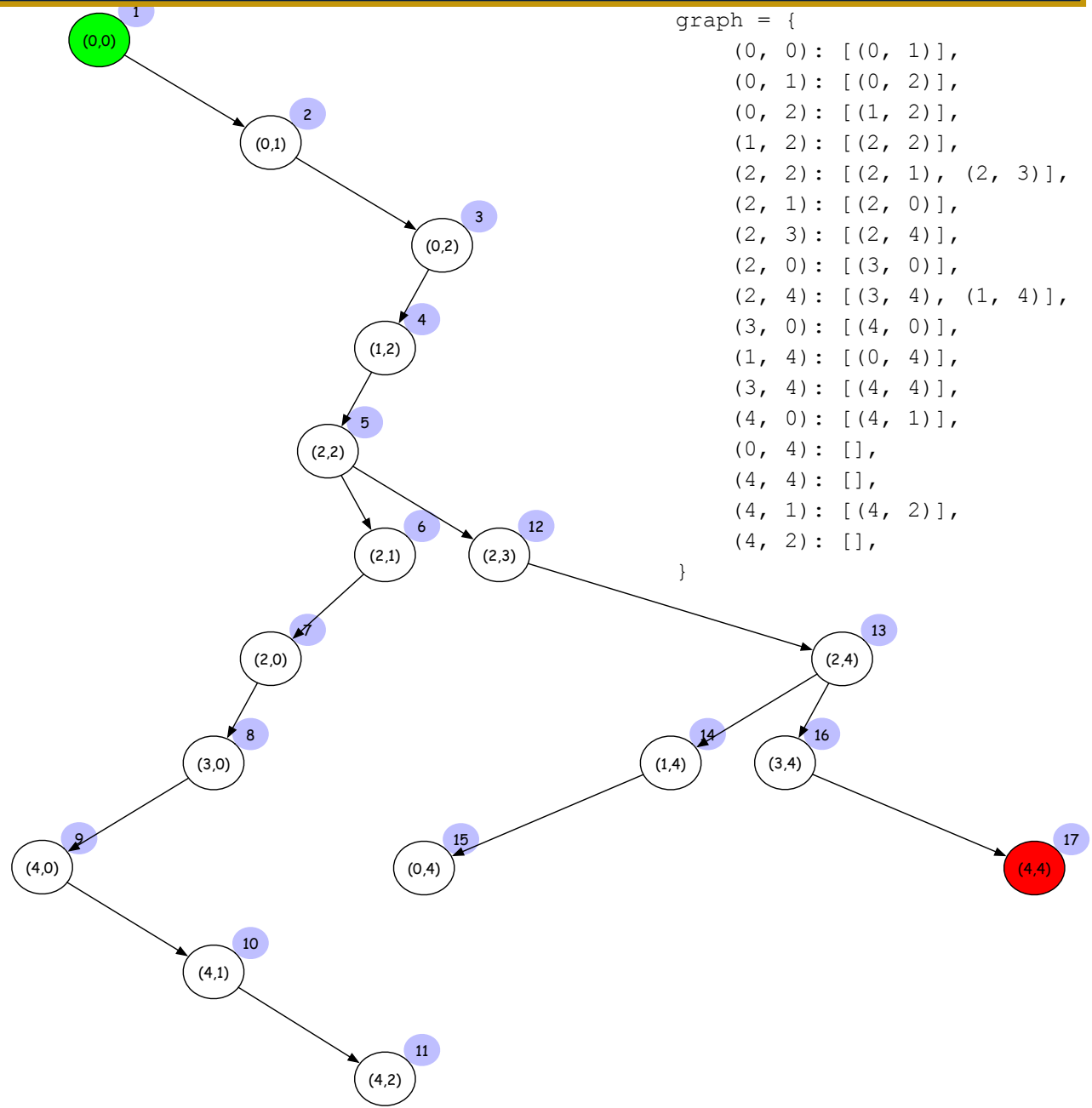

```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

Example 3: Shortest Pathfinding

For easy to show the logic, let's start from the constructed simplified tree.

We have tried 4 traversal orders:

1. DFS (Push Right into the Stack first)
   - 17 steps



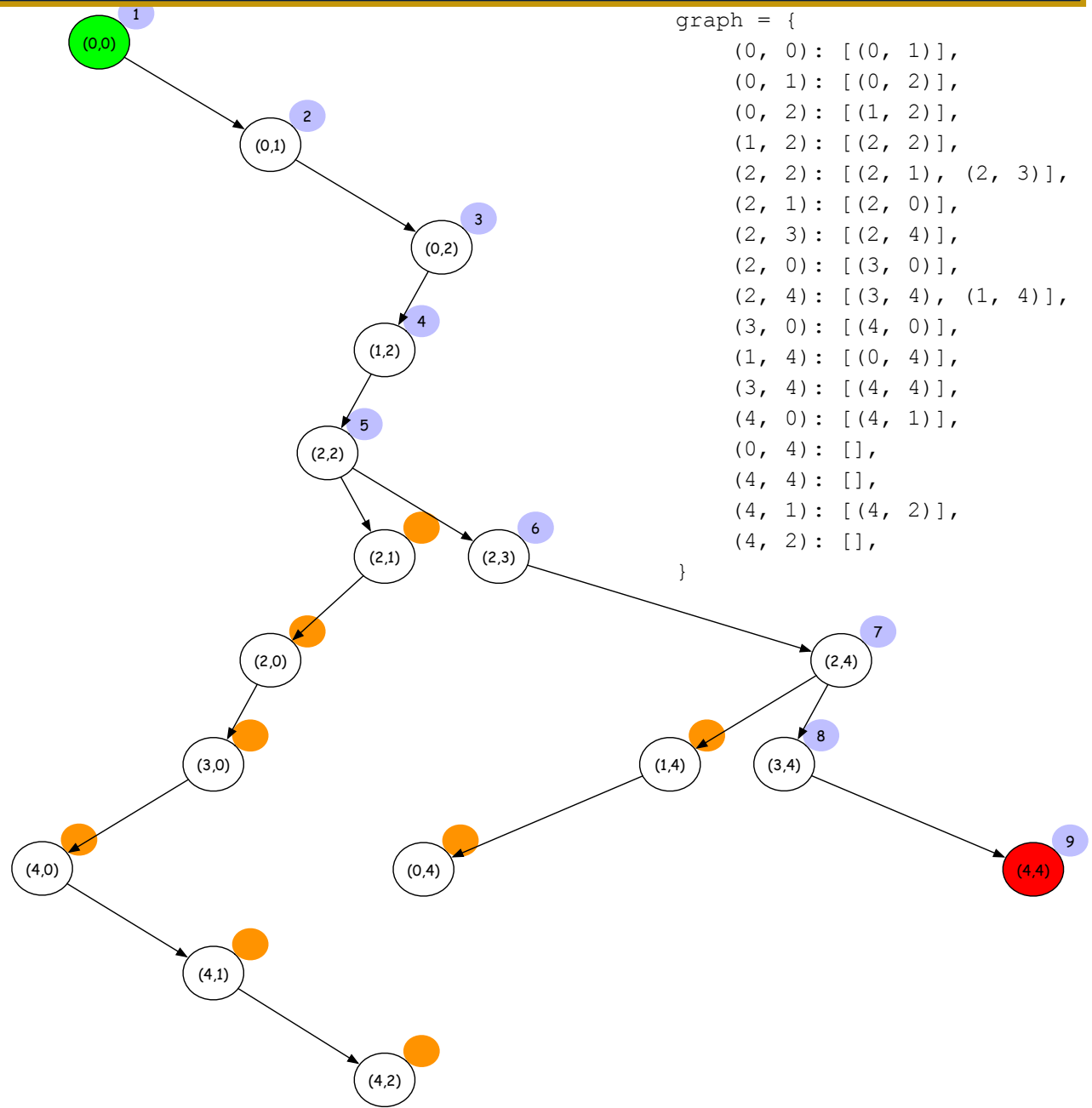

```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

Example 3: Shortest Pathfinding

For easy to show the logic, let's start from the constructed simplified tree.

We have tried 4 traversal orders:

1. DFS (Push Right into the Stack first)
   - 17 steps
2. DFS (Push Left into the Stack first)
   - 9 steps



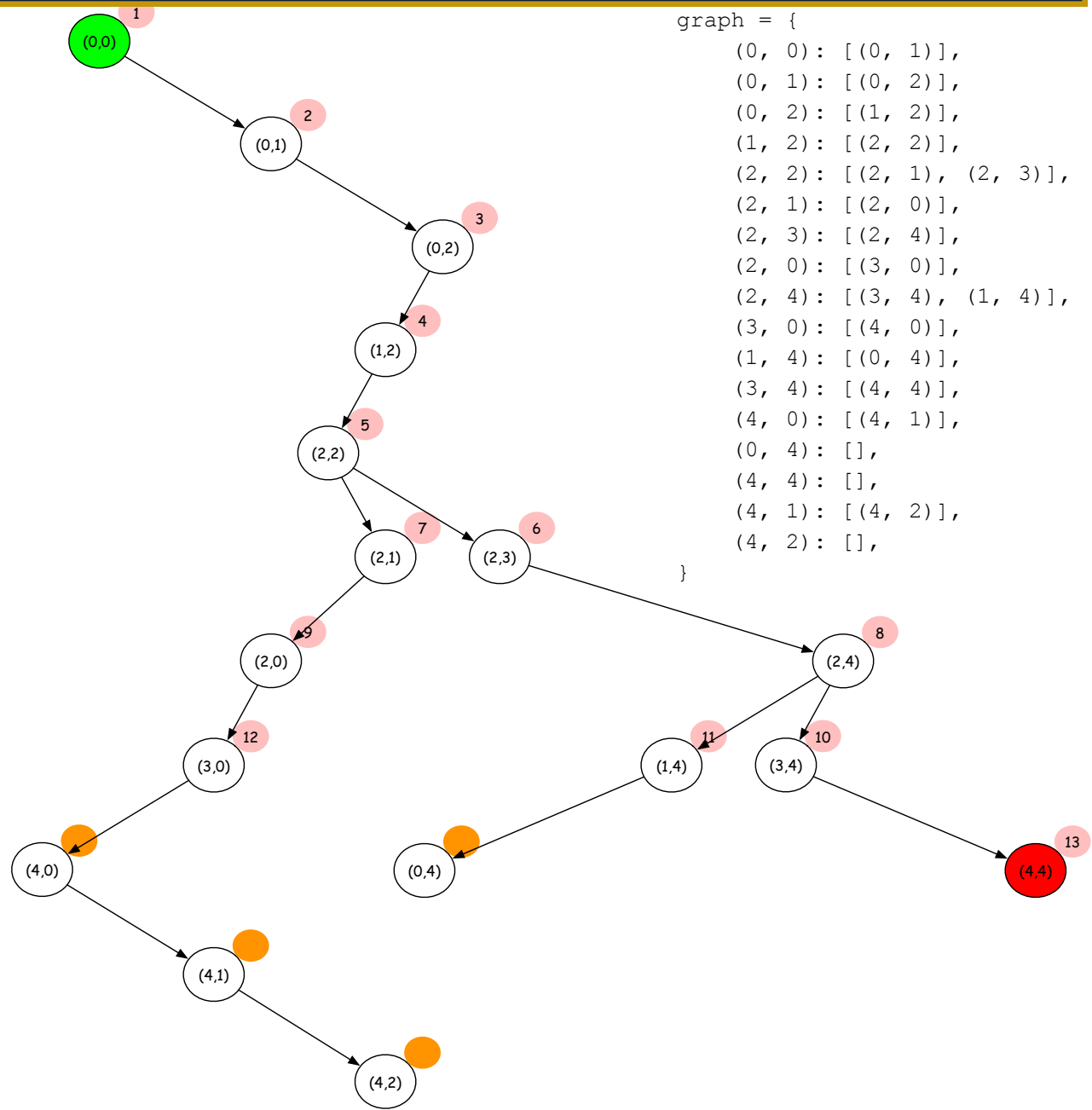

```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

Example 3: Shortest Pathfinding

For easy to show the logic, let's start from the constructed simplified tree.

We have tried 4 traversal orders:

1. DFS (Push Right into the Stack first)
   - 17 steps
2. DFS (Push Left into the Stack first)
   - 9 steps
3. BFS (Push Right into the Queue first)
   - 13 steps



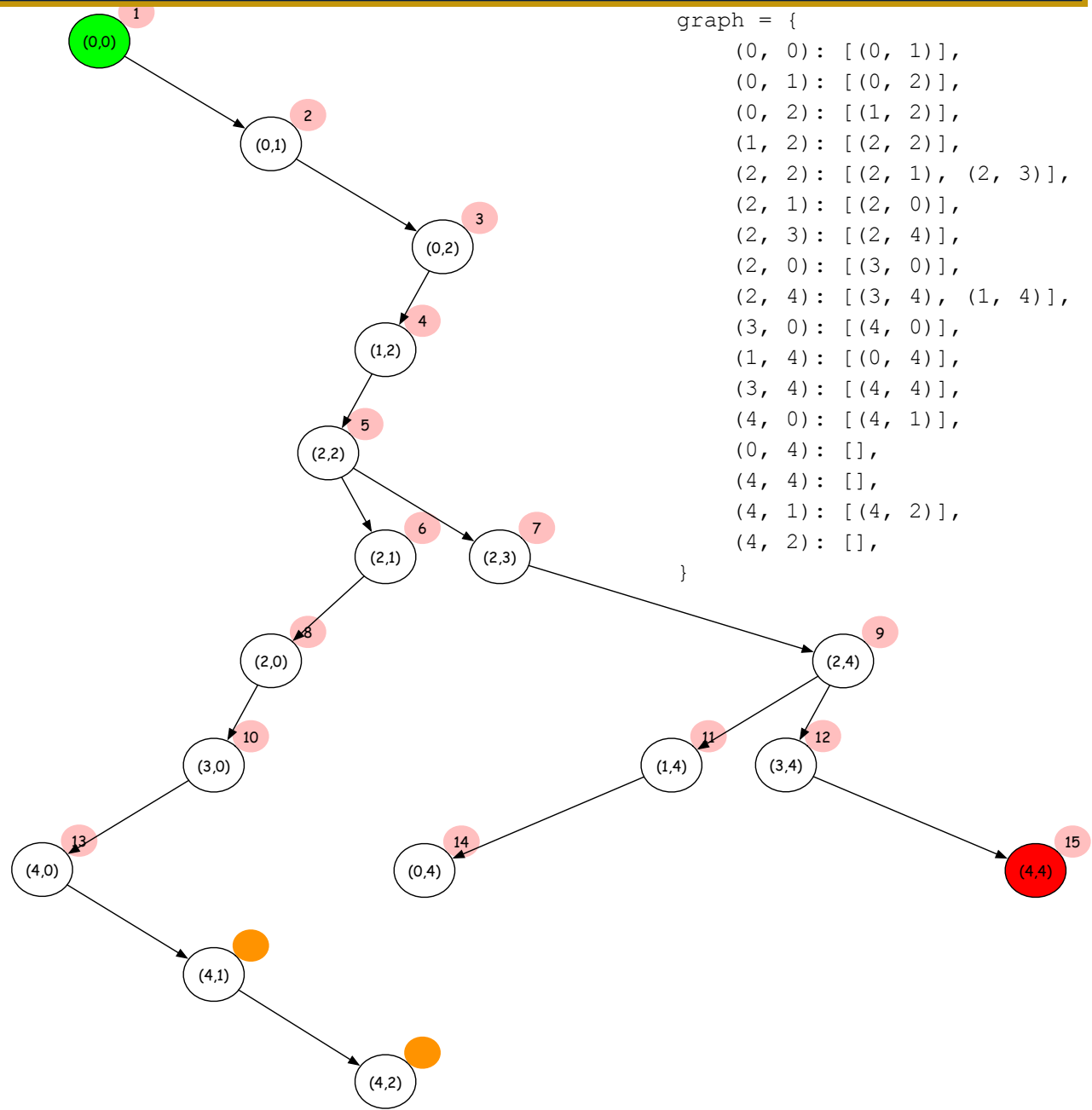

```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

Example 3: Shortest Pathfinding

For easy to show the logic, let's start from the constructed simplified tree.

We have tried 4 traversal orders:

1. DFS (Push Right into the Stack first)
   - 17 steps
2. DFS (Push Left into the Stack first)
   - 9 steps
3. BFS (Push Right into the Queue first)
   - 13 steps
4. BFS (Push Left into the Queue first)
   - 15 steps

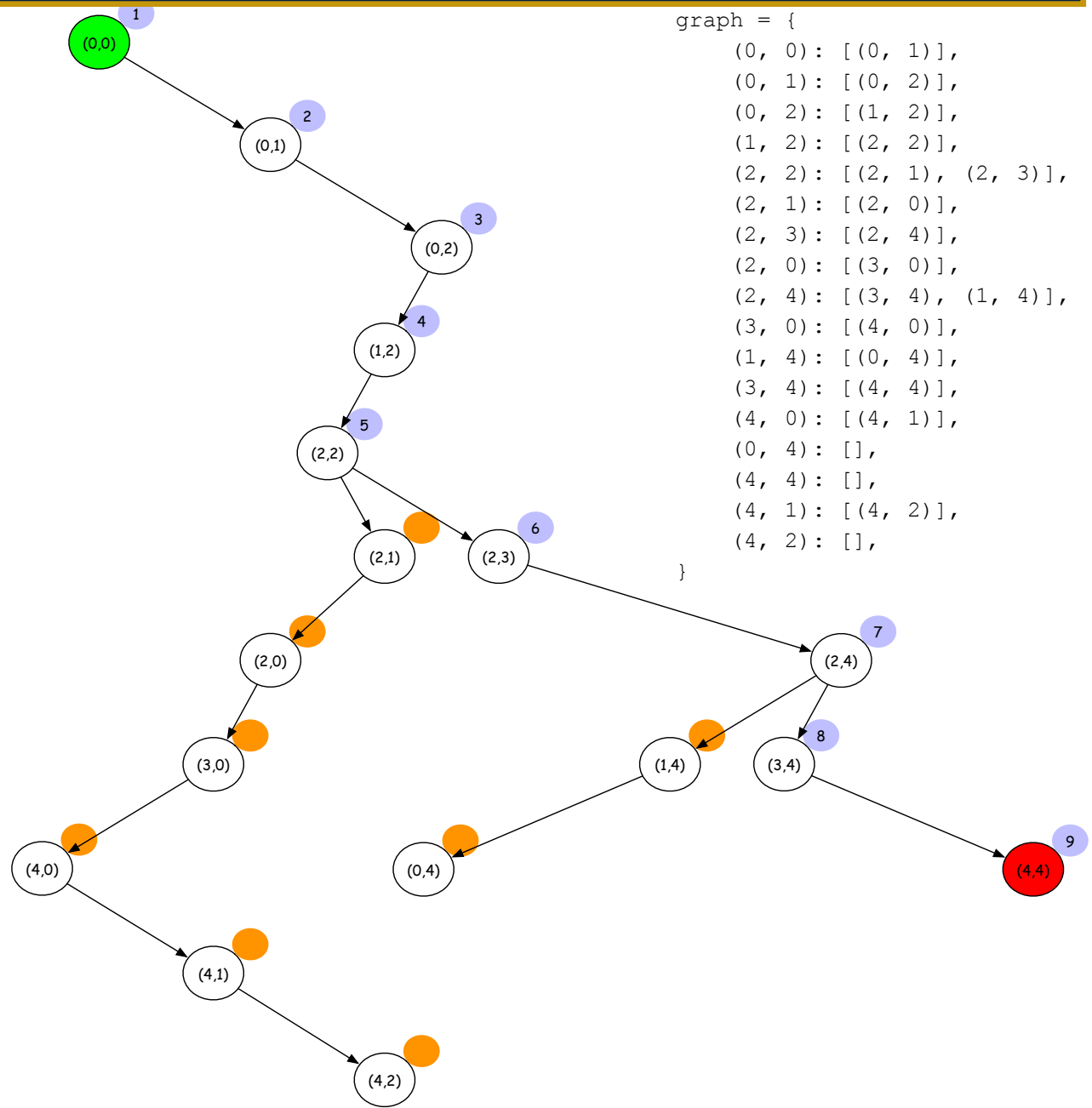


Example 3: Shortest Pathfinding

For easy to show the logic, let's start from the constructed simplified tree.

We have tried 4 traversal orders:

1. DFS (Push Right into the Stack first)
   - 17 steps
2. DFS (Push Left into the Stack first)
   - 9 steps
3. BFS (Push Right into the Queue first)
   - 13 steps
4. BFS (Push Left into the Queue first)
   - 15 steps




```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

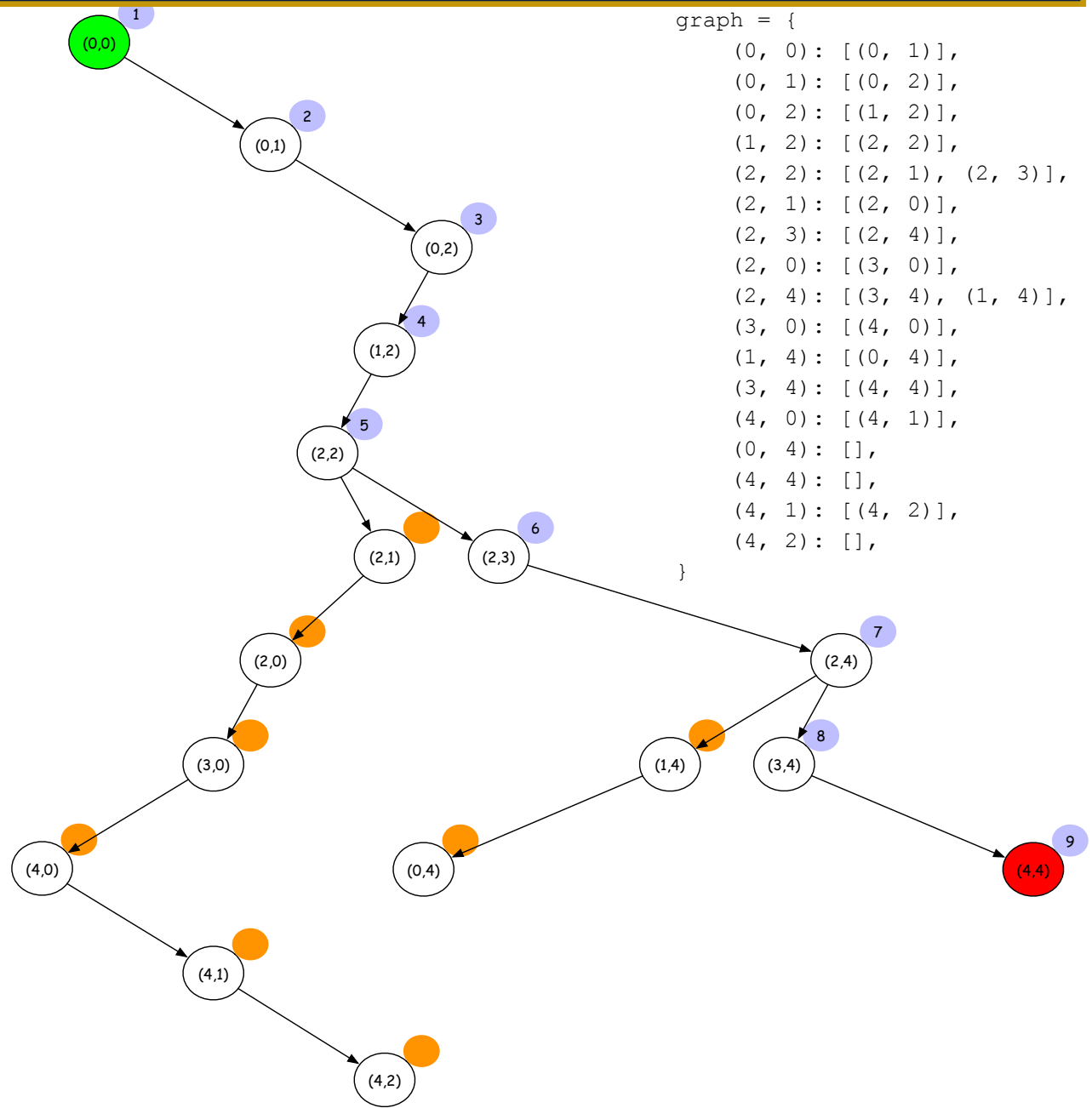
Example 3: Shortest Pathfinding

For easy to show the logic, let's start from the constructed simplified tree.

We have tried 4 traversal orders:

1. DFS (Push Right into the Stack first)
   - 17 steps
2. DFS (Push Left into the Stack first)
   - 9 steps
3. BFS (Push Right into the Queue first)
   - 13 steps
4. BFS (Push Left into the Queue first)
   - 15 steps

Why we save the running time (with the same time complexity)? Can we gunarantee to reproduce the best case?




```
graph = {
    (0, 0): [(0, 1)],
    (0, 1): [(0, 2)],
    (0, 2): [(1, 2)],
    (1, 2): [(2, 2)],
    (2, 2): [(2, 1), (2, 3)],
    (2, 1): [(2, 0)],
    (2, 3): [(2, 4)],
    (2, 0): [(3, 0)],
    (2, 4): [(3, 4), (1, 4)],
    (3, 0): [(4, 0)],
    (1, 4): [(0, 4)],
    (3, 4): [(4, 4)],
    (4, 0): [(4, 1)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2)],
    (4, 2): [],
}
```

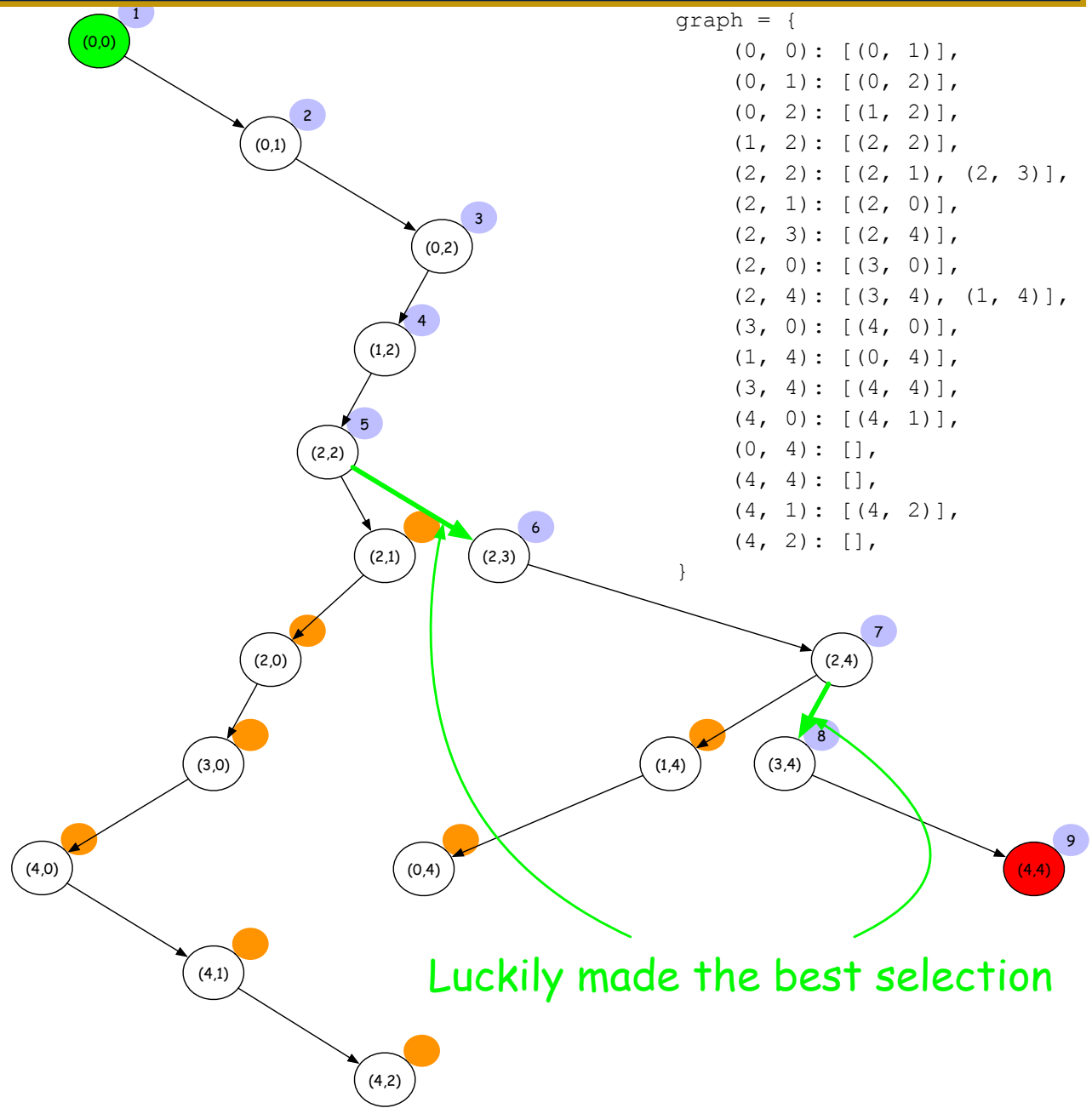
Example 3: Shortest Pathfinding

For easy to show the logic, let's start from the constructed simplified tree.

We have tried 4 traversal orders:

1. DFS (Push Right into the Stack first)
   - 17 steps
2. DFS (Push Left into the Stack first)
   - 9 steps
3. BFS (Push Right into the Queue first)
   - 13 steps
4. BFS (Push Left into the Queue first)
   - 15 steps

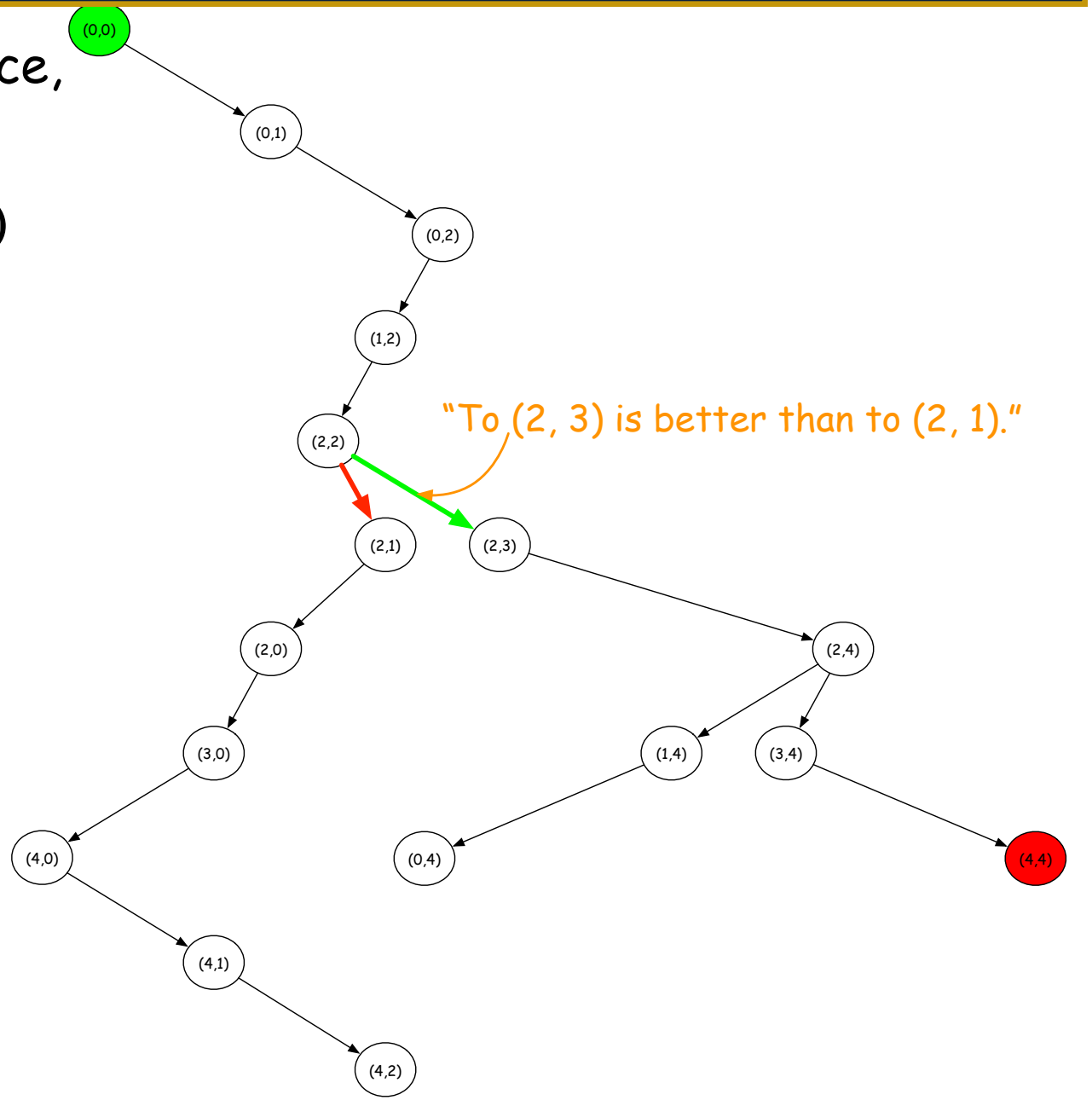
Why we save the running time (with the same time complexity)? Can we gunarantee to reproduce the best case?



Example 3: Shortest Pathfinding

During the search, we may need some guidance, e.g.,

$((2,2),(2,3))$ is better than $((2,2),(2,1))$

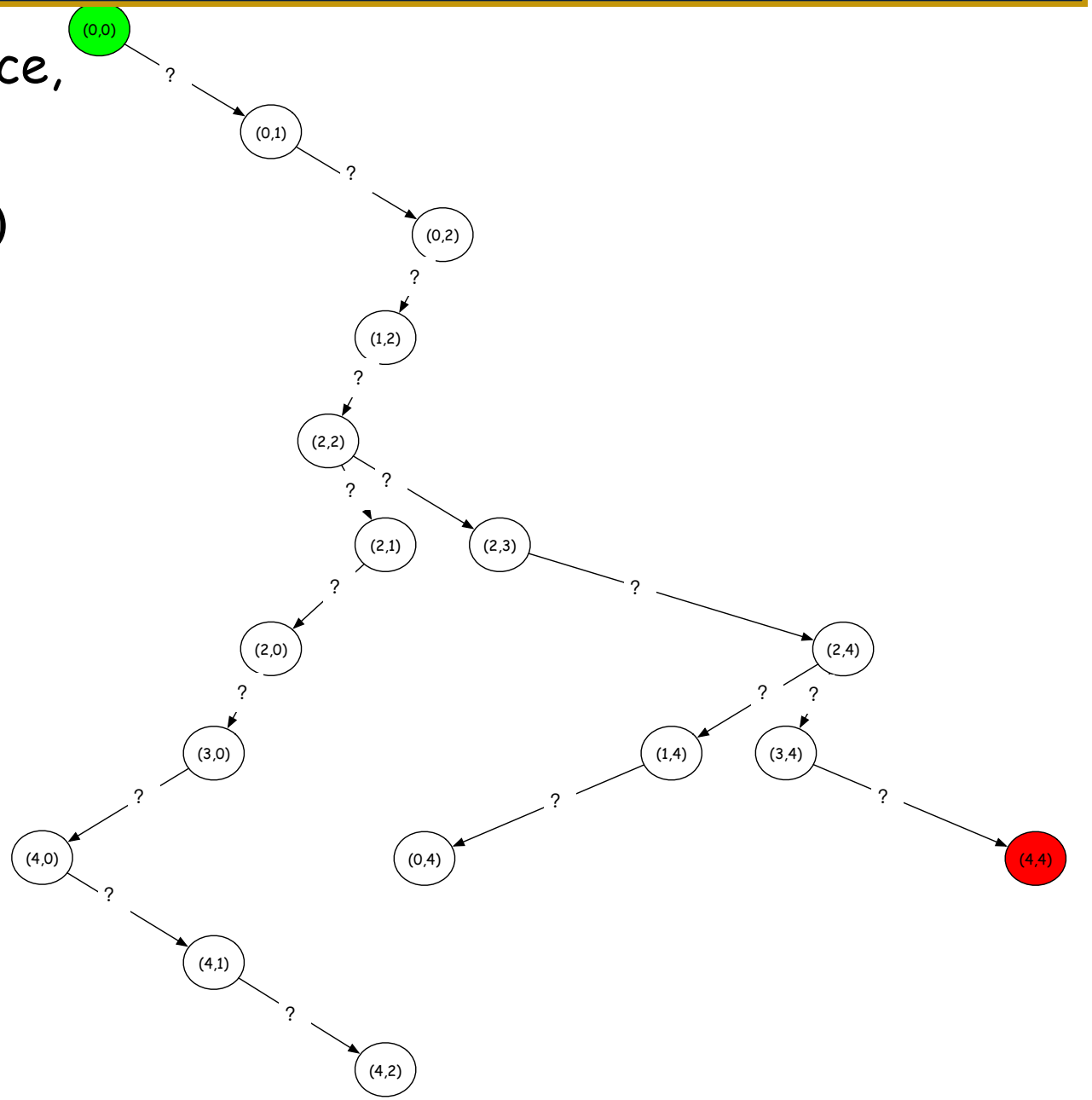


Example 3: Shortest Pathfinding

During the search, we may need some guidance, e.g.,

$((2,2),(2,3))$ is better than $((2,2),(2,1))$

We may need one WEIGHTED graph.



Example 3: Shortest Pathfinding

During the search, we may need some guidance, e.g.,

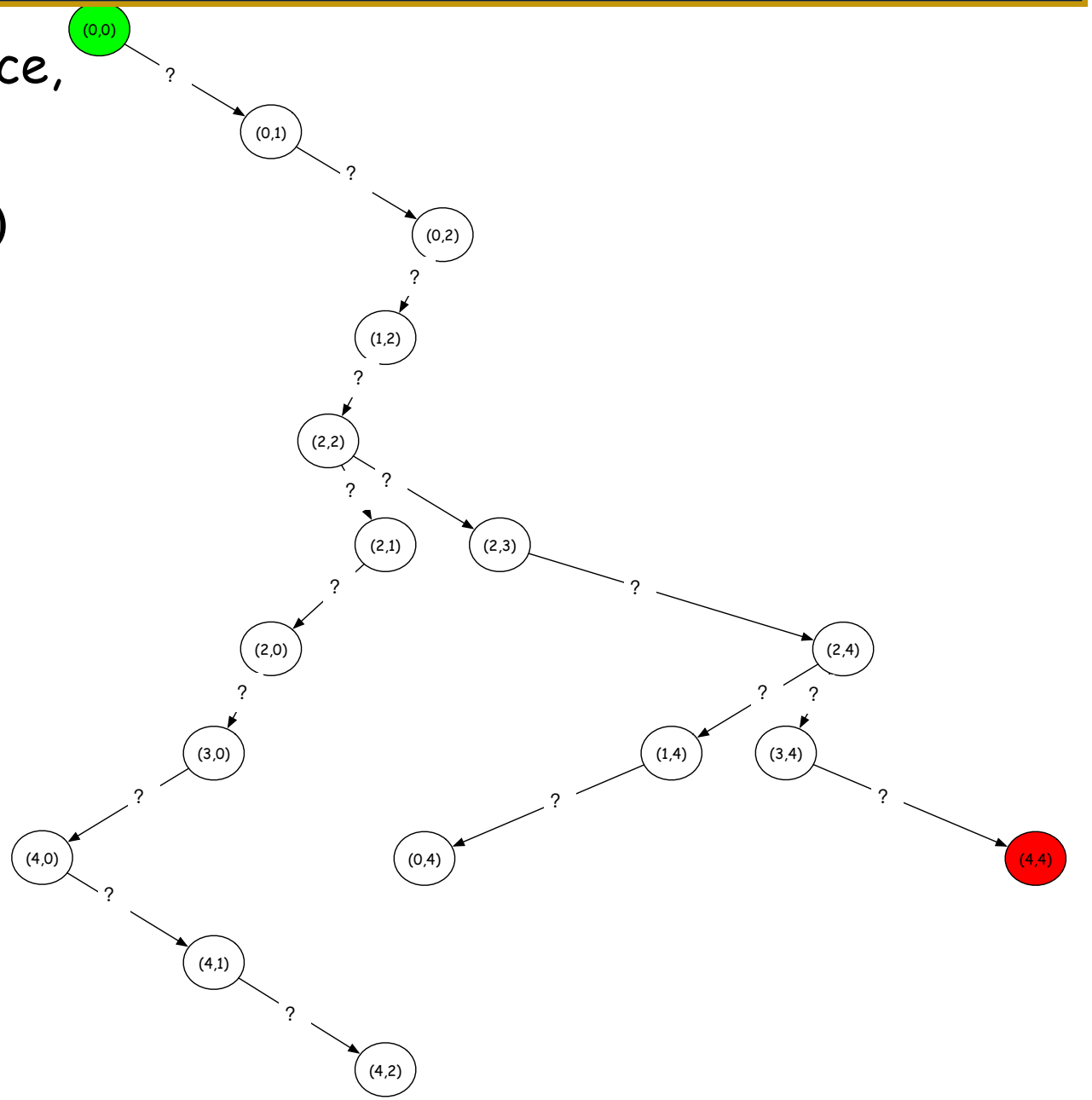
$((2,2),(2,3))$ is better than $((2,2),(2,1))$

We may need one WEIGHTED graph.

To get the proper weight/guidance, we need to find some useful knowledge/heuristic from the specific problem.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

Any intuition?



Example 3: Shortest Pathfinding

During the search, we may need some guidance, e.g.,

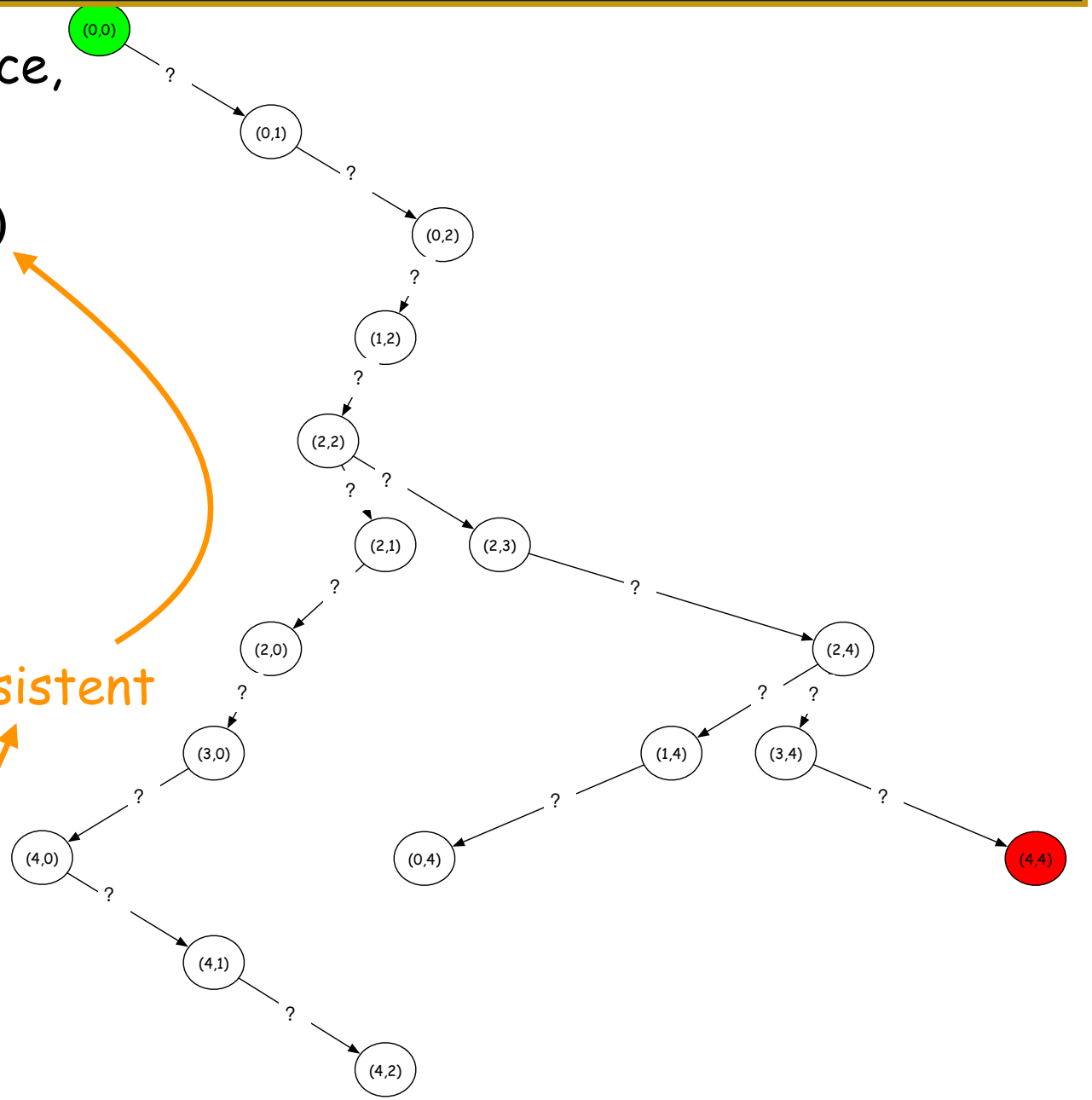
$((2,2),(2,3))$ is better than $((2,2),(2,1))$

We may need one WEIGHTED graph.

To get the proper weight/guidance, we need to find some useful knowledge/heuristic from the specific problem.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

consistent



Any intuition?

For $(2,2)$, $(2,3)$ should be better than $(2,1)$.

Example 3: Shortest Pathfinding

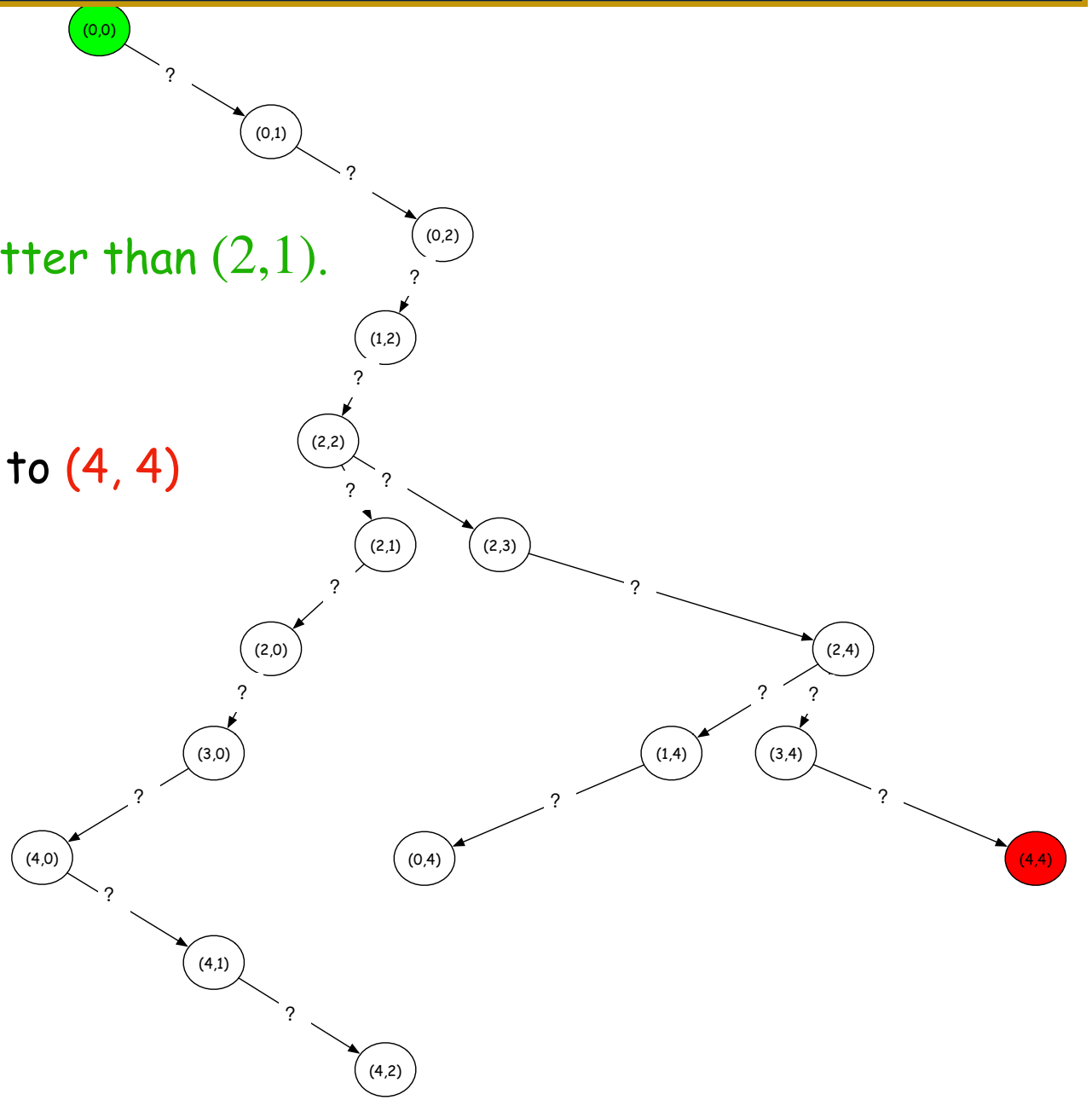
0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0

For (2,2), (2,3) should be better than (2,1).

We has this intuition, as the distance from (2, 3) to (4, 4) is shorter than the distance from (2, 1) to (4, 4).

Manhattan distance

- A way to measure the distance between two points in a grid where you can only move along the grid lines, either horizontally or vertically.
- Formula:
  - If you have two points (x_1, y_1) and (x_2, y_2) on a grid, the Manhattan distance d is
  - $d = |x_2 - x_1| + |y_2 - y_1|$



Example 3: Shortest Pathfinding

0	0	0	1	0
1	1	0	1	0
0	C	0	C	0
0	1	1	1	0
0	0	0	1	0

For (2,2), (2,3) should be better than (2,1).

We has this intuition, as the distance from (2, 3) to (4, 4) is shorter than the distance from (2, 1) to (4, 4).

Manhattan distance

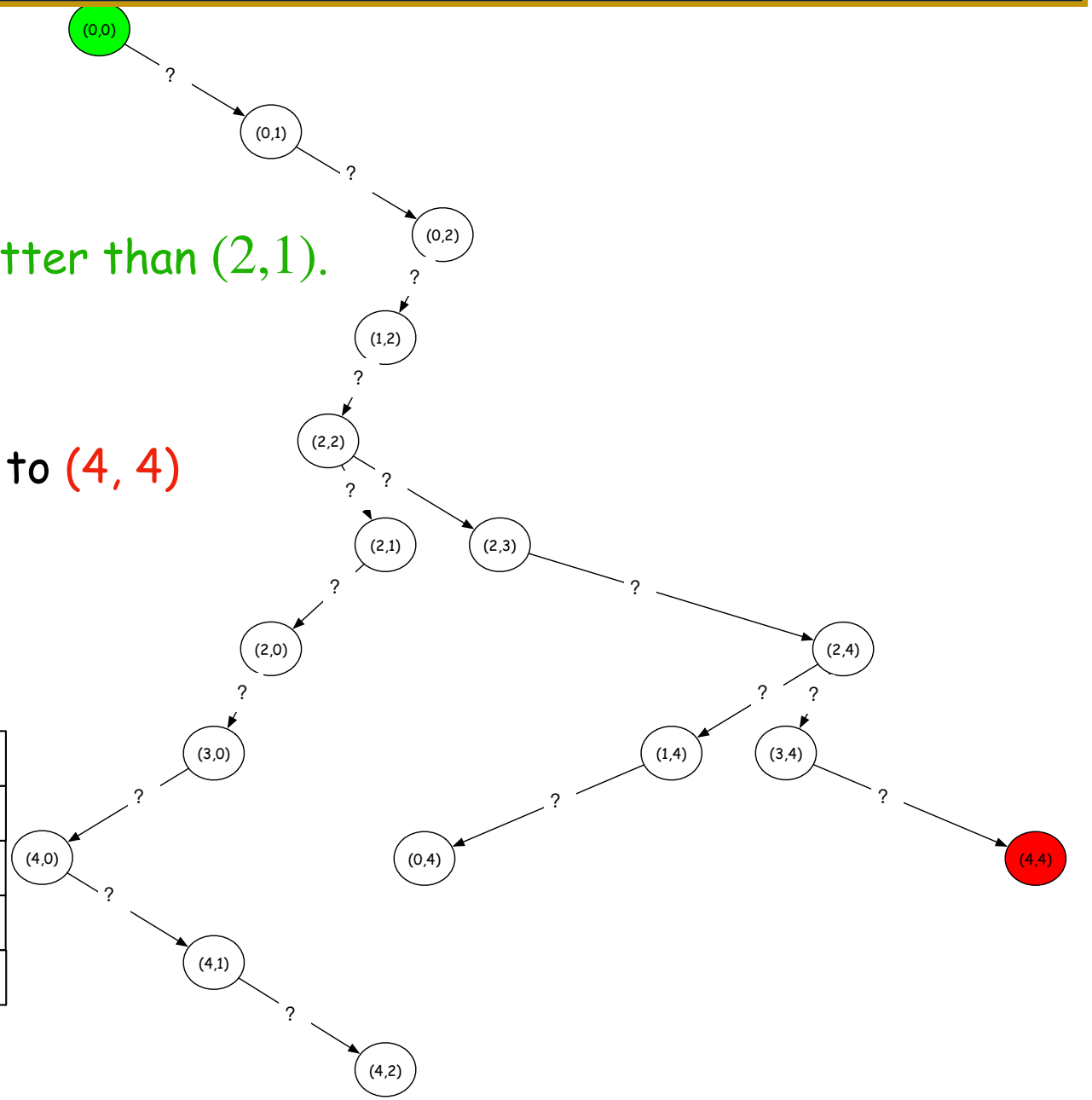
$$d = |x_2 - x_1| + |y_2 - y_1|$$

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

$$d_1 = |3 - 0| + |1 - 0| = 4$$

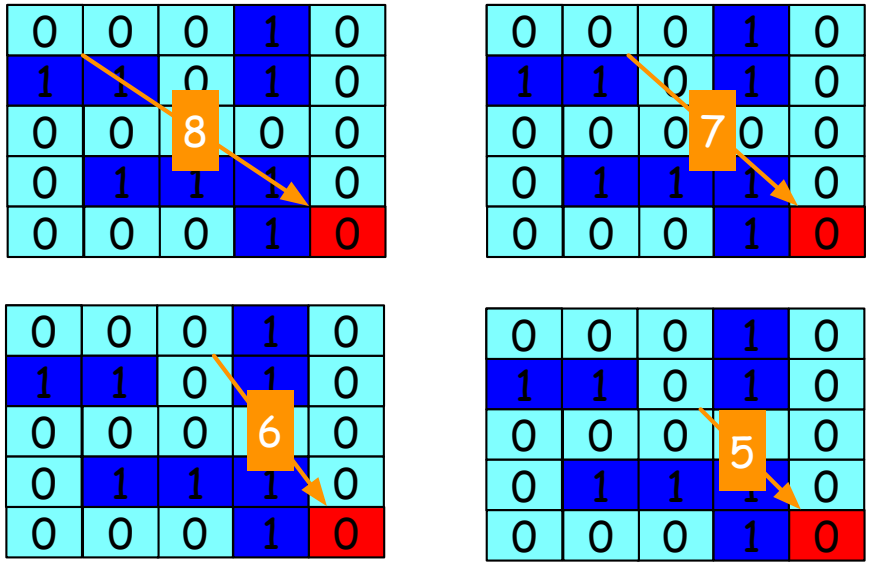
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

$$d_2 = |4 - 1| + |3 - 2| = 4$$



Example 3: Shortest Pathfinding

Calculate the distance for all positions as the weight.



Heuristic function $h(n)$ can return An rational estimate of the cost from node n to the goal.

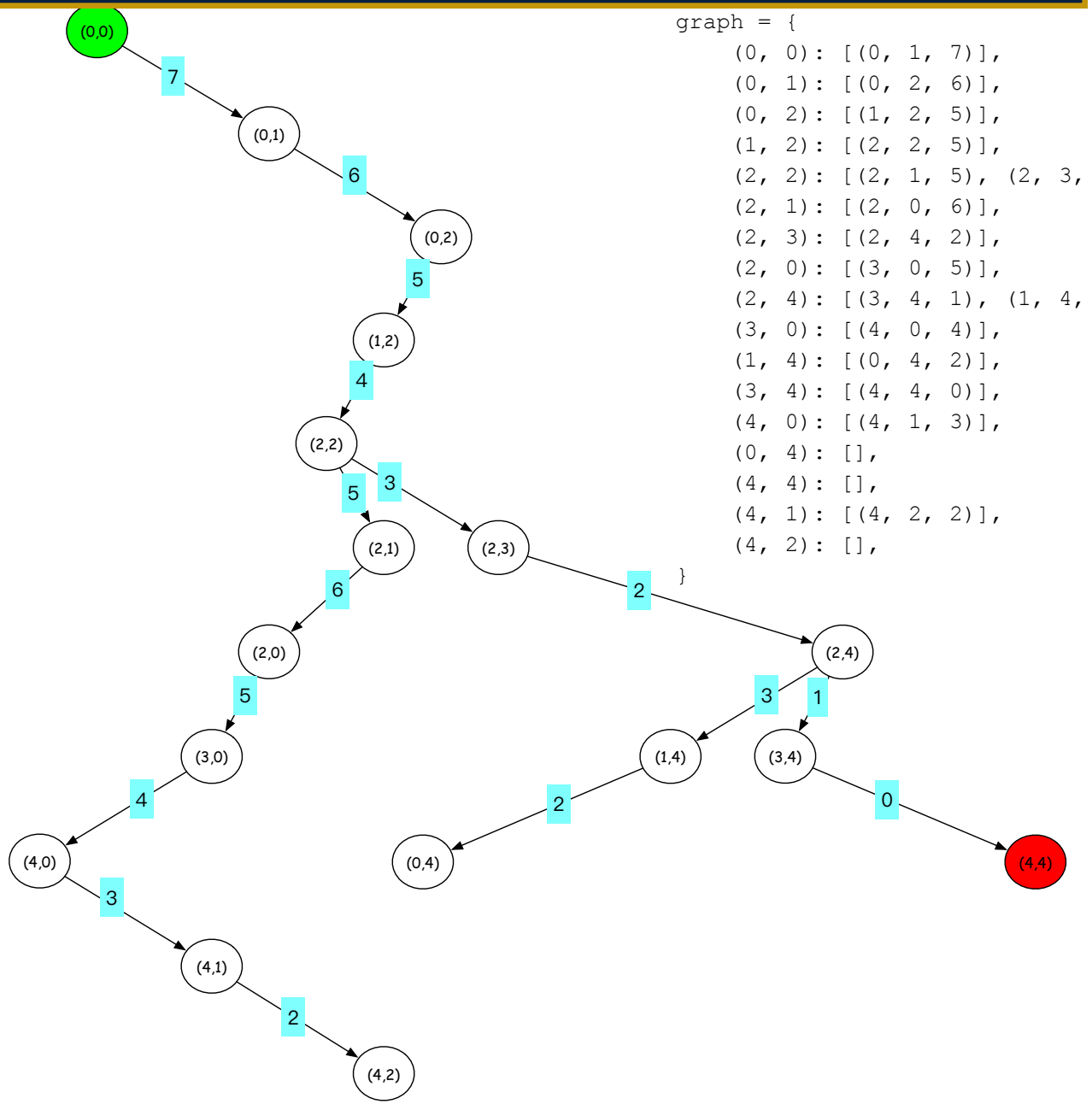
In this example, manhattan distance is $h(n)$.

```
def manhattan_distance(curent_node, target):
    x1, y1 = curent_node
    x2, y2 = target

    return abs(x1 - x2) + abs(y1 - y2)
```

Example 3: Shortest Pathfinding

Greedy Best-First Search:

- In given weighted graph, just pick the best local option every time.




```
graph = {
    (0, 0): [(0, 1, 7)],
    (0, 1): [(0, 2, 6)],
    (0, 2): [(1, 2, 5)],
    (1, 2): [(2, 2, 5)],
    (2, 2): [(2, 1, 5), (2, 3,
    (2, 1): [(2, 0, 6)],
    (2, 3): [(2, 4, 2)],
    (2, 0): [(3, 0, 5)],
    (2, 4): [(3, 4, 1), (1, 4,
    (3, 0): [(4, 0, 4)],
    (1, 4): [(0, 4, 2)],
    (3, 4): [(4, 4, 0)],
    (4, 0): [(4, 1, 3)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2, 2)],
    (4, 2): [],
}
```

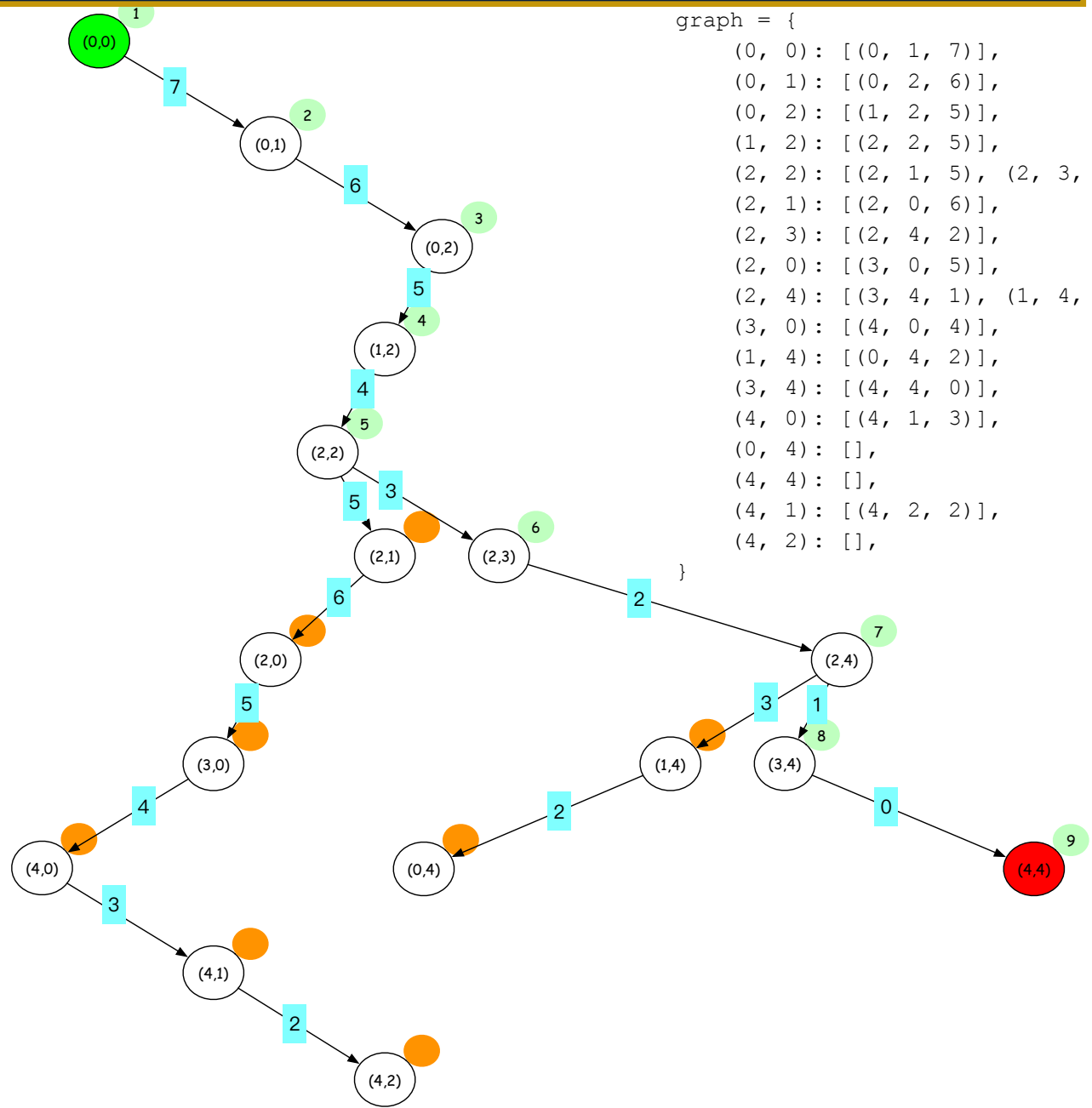
Example 3: Shortest Pathfinding

Greedy Best-First Search:

- In given weighted graph, just pick the best local option every time.

We use the Manhattan distance as the estimation to guide the greedy best-first search.

- Sometimes, it works well and find the best solution (lucky).




```
graph = {
    (0, 0): [(0, 1, 7)],
    (0, 1): [(0, 2, 6)],
    (0, 2): [(1, 2, 5)],
    (1, 2): [(2, 2, 5)],
    (2, 2): [(2, 1, 5), (2, 3,
    (2, 1): [(2, 0, 6)],
    (2, 3): [(2, 4, 2)],
    (2, 0): [(3, 0, 5)],
    (2, 4): [(3, 4, 1), (1, 4,
    (3, 0): [(4, 0, 4)],
    (1, 4): [(0, 4, 2)],
    (3, 4): [(4, 4, 0)],
    (4, 0): [(4, 1, 3)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2, 2)],
    (4, 2): [],
}
```

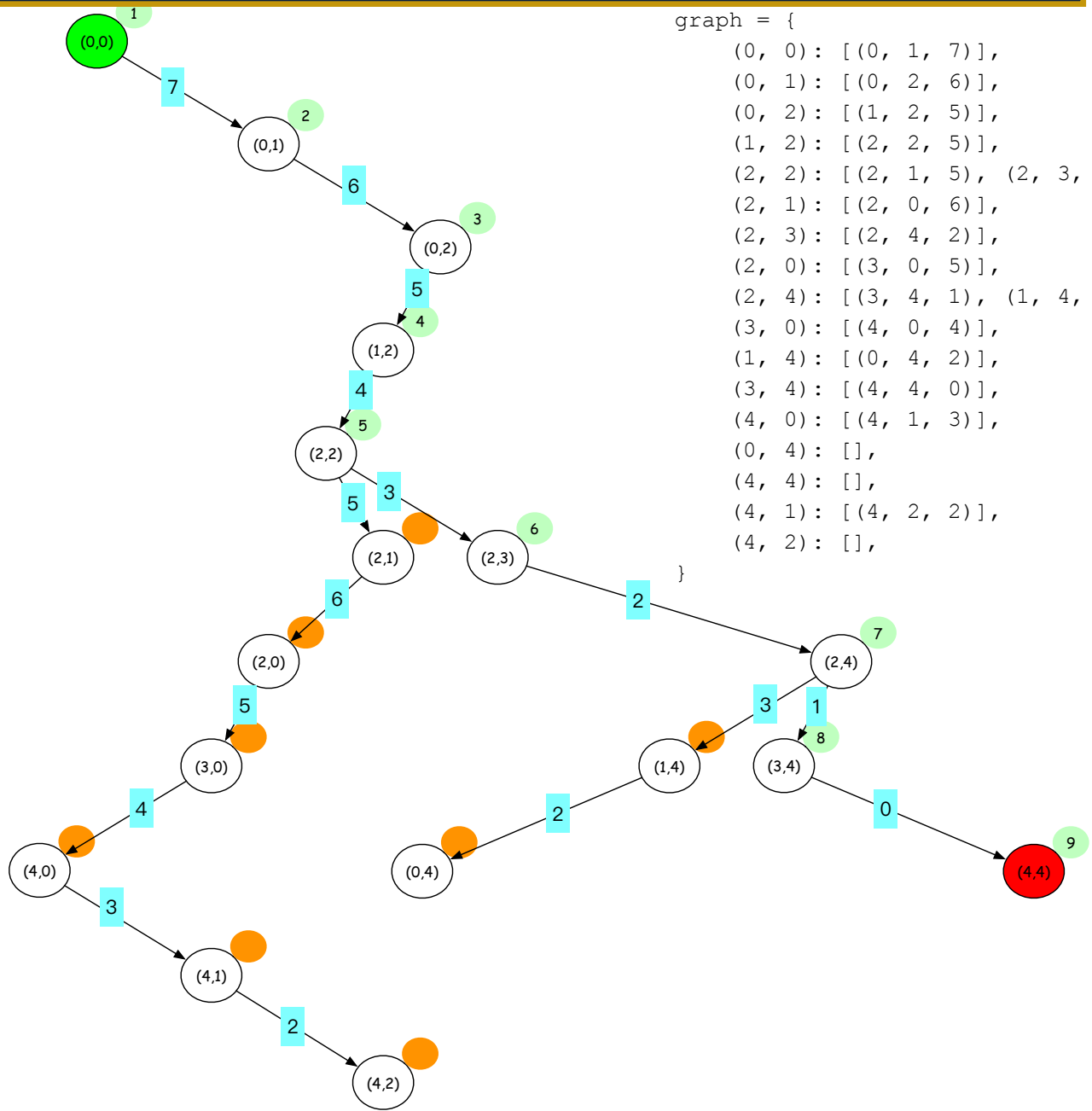
Example 3: Shortest Pathfinding

Greedy Best-First Search:

- In given weighted graph, just pick the best local option every time.

We use the Manhattan distance as the estimation to guide the greedy best-first search.

- Sometimes, it works well and find the best solution (lucky).
- However, the result might be not optimal, and we have no chance to fix it.




```
graph = {
    (0, 0): [(0, 1, 7)],
    (0, 1): [(0, 2, 6)],
    (0, 2): [(1, 2, 5)],
    (1, 2): [(2, 2, 5)],
    (2, 2): [(2, 1, 5), (2, 3,
    (2, 1): [(2, 0, 6)],
    (2, 3): [(2, 4, 2)],
    (2, 0): [(3, 0, 5)],
    (2, 4): [(3, 4, 1), (1, 4,
    (3, 0): [(4, 0, 4)],
    (1, 4): [(0, 4, 2)],
    (3, 4): [(4, 4, 0)],
    (4, 0): [(4, 1, 3)],
    (0, 4): [],
    (4, 4): [],
    (4, 1): [(4, 2, 2)],
    (4, 2): [],
}
```

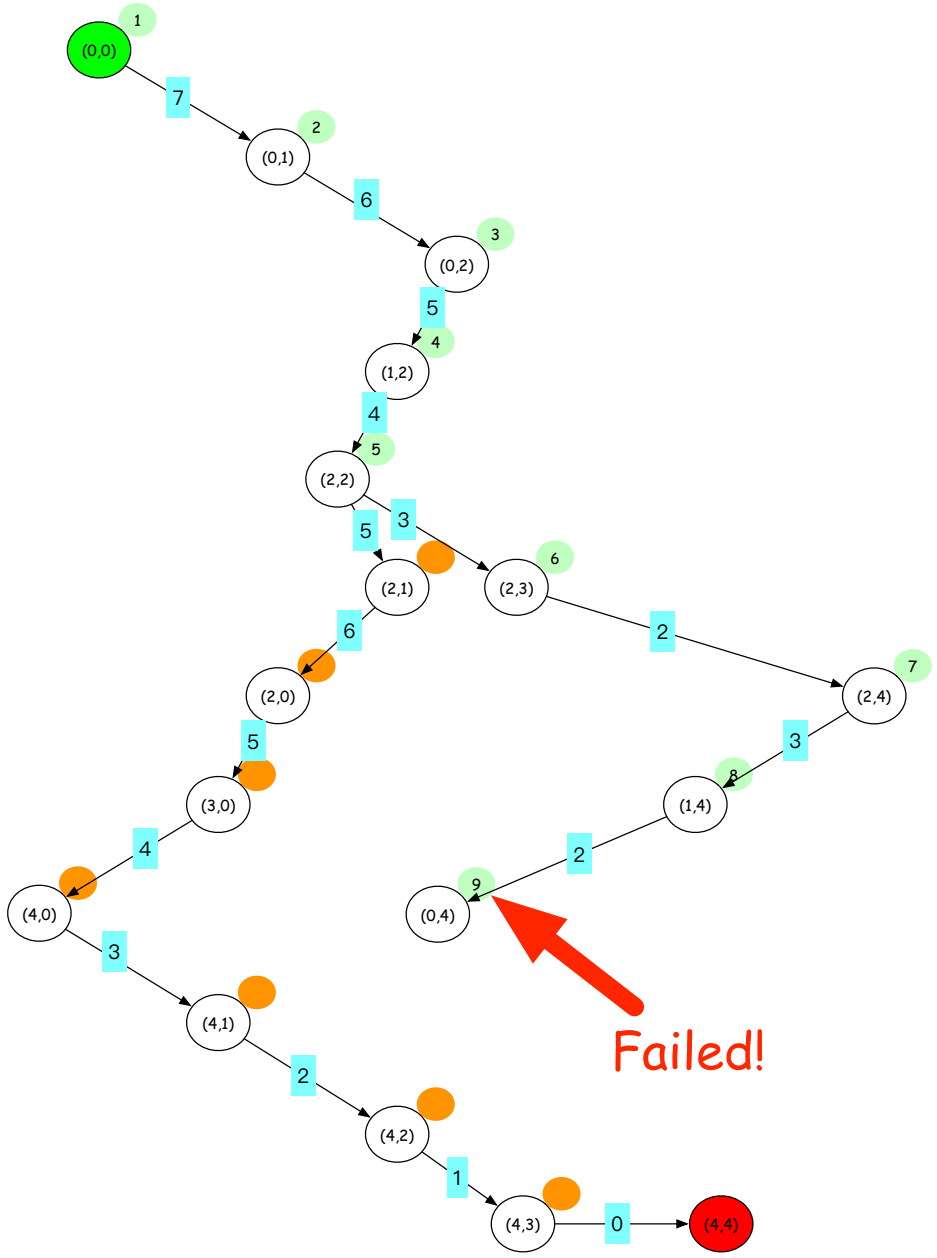
Example 3: Shortest Pathfinding

Greedy Best-First Search:

- In given weighted graph, just pick the best local option every time.

- However, the result might be not optimal, and we have no chance to fix it. Consider the example below.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0



Example 3: Shortest Pathfinding

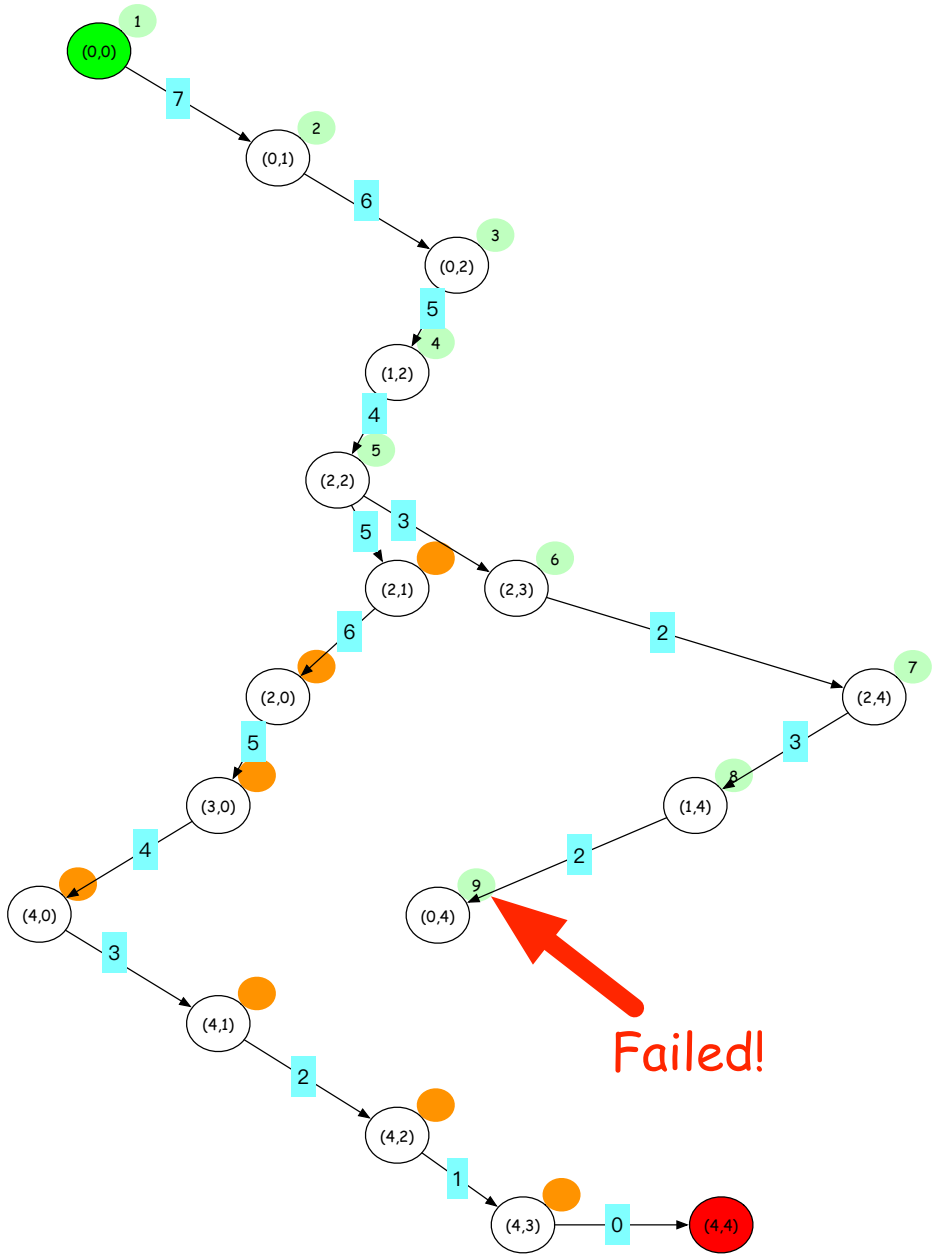
Greedy Best-First Search:

- In given weighted graph, just pick the best local option every time.

- However, the result might be not optimal, and we have no chance to fix it. Consider the example below.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0

Why?



Example 3: Shortest Pathfinding

Greedy Best-First Search:

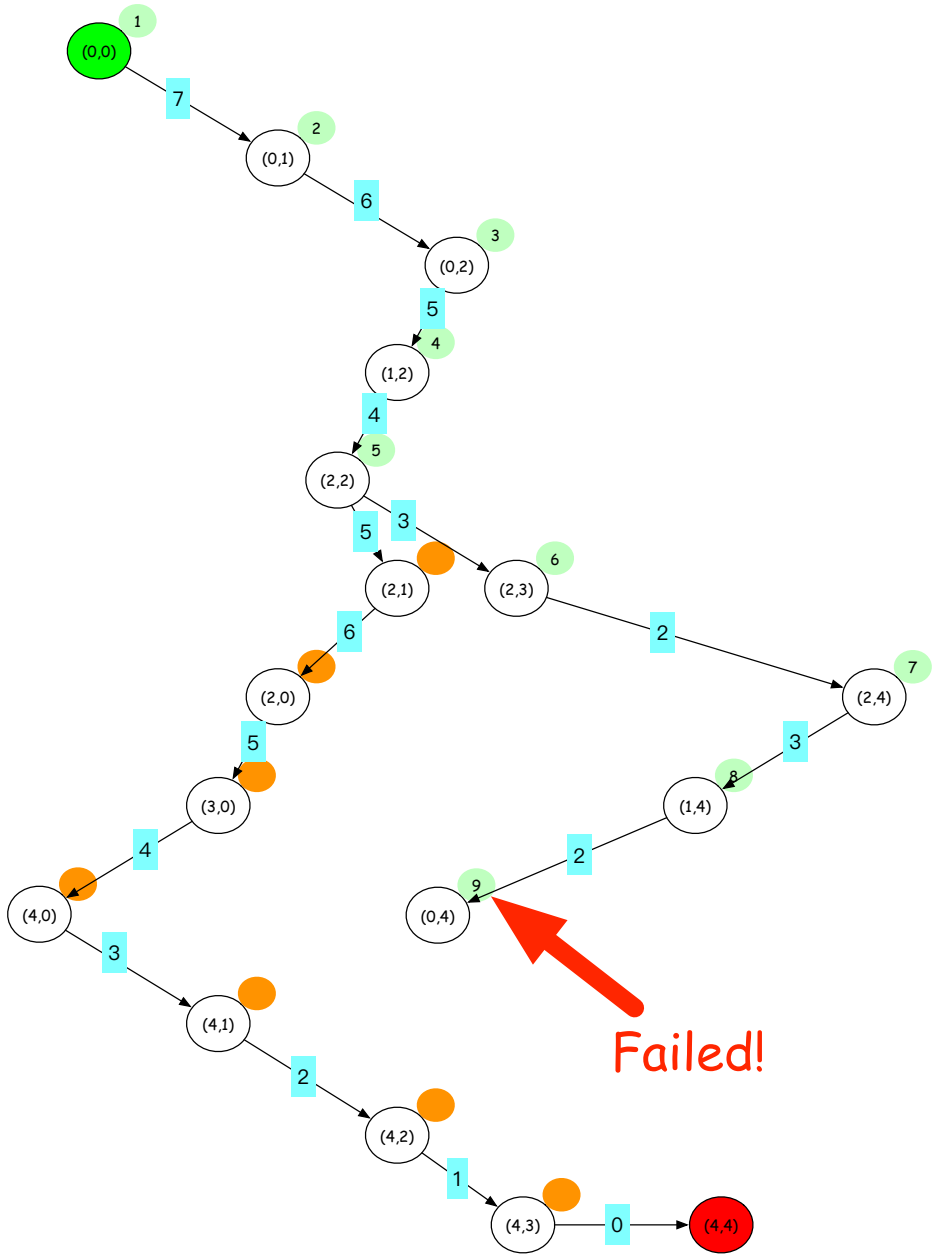
- In given weighted graph, just pick the best local option every time.

- However, the result might be not optimal, and we have no chance to fix it. Consider the example below.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0

Why?

The core idea of greedy best-first search is to always pursue the path with the smallest estimated future cost. However, the future is inherently uncertain, as the estimation of future costs may be inaccurate. In such cases, greedy best-first search not only risks failing to find the optimal solution but may even fail entirely.



Example 3: Shortest Pathfinding

Greedy Best-First Search:

- In given weighted graph, just pick the best local option every time.

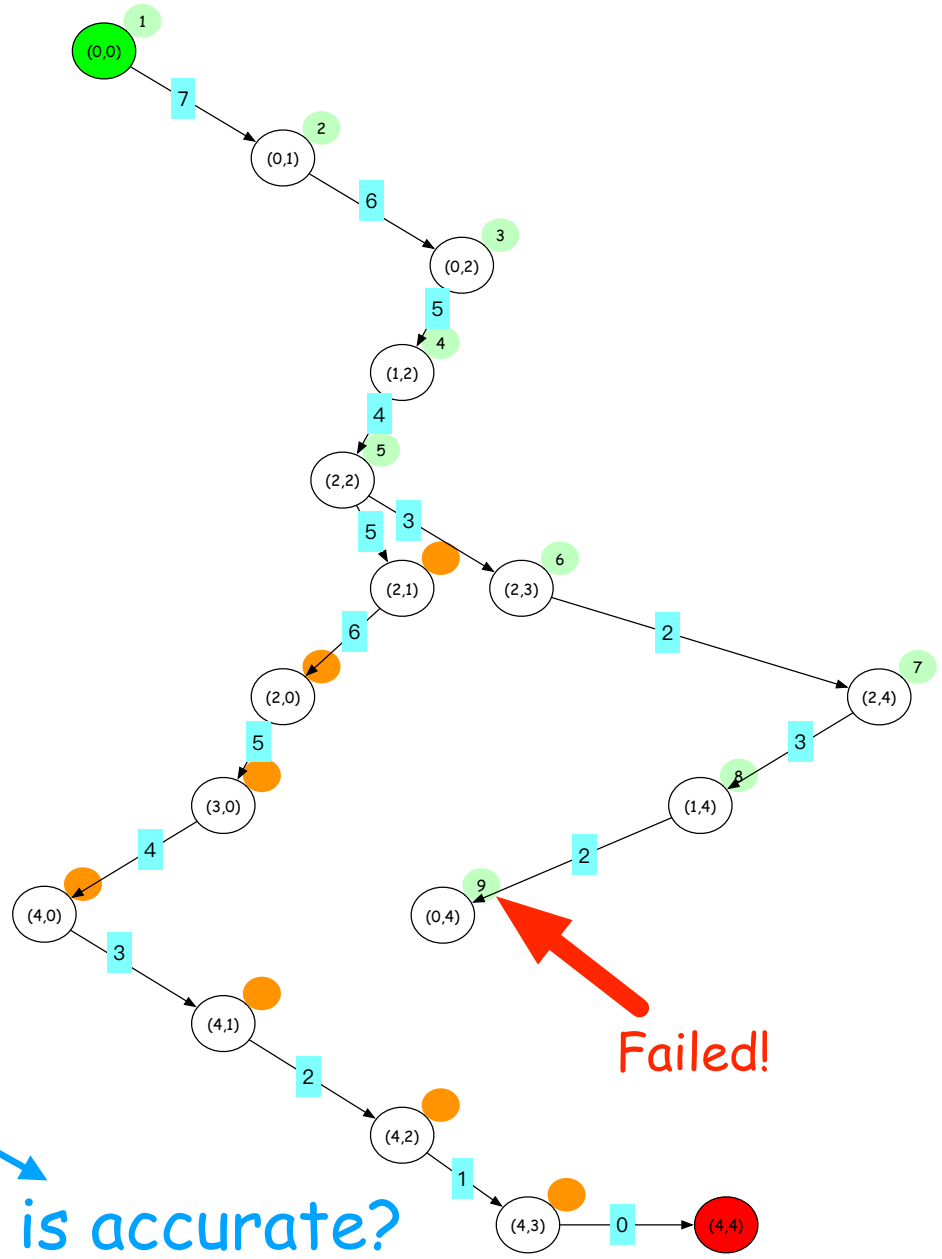
- However, the result might be not optimal, and we have no chance to fix it. Consider the example below.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0

Why?

The core idea of greedy best-first search is to always pursue the path with the smallest estimated future cost. However, the future is inherently uncertain, as **the estimation of future costs may be inaccurate**. In such cases, greedy best-first search not only risks failing to find the optimal solution but may even fail entirely.

What is accurate?

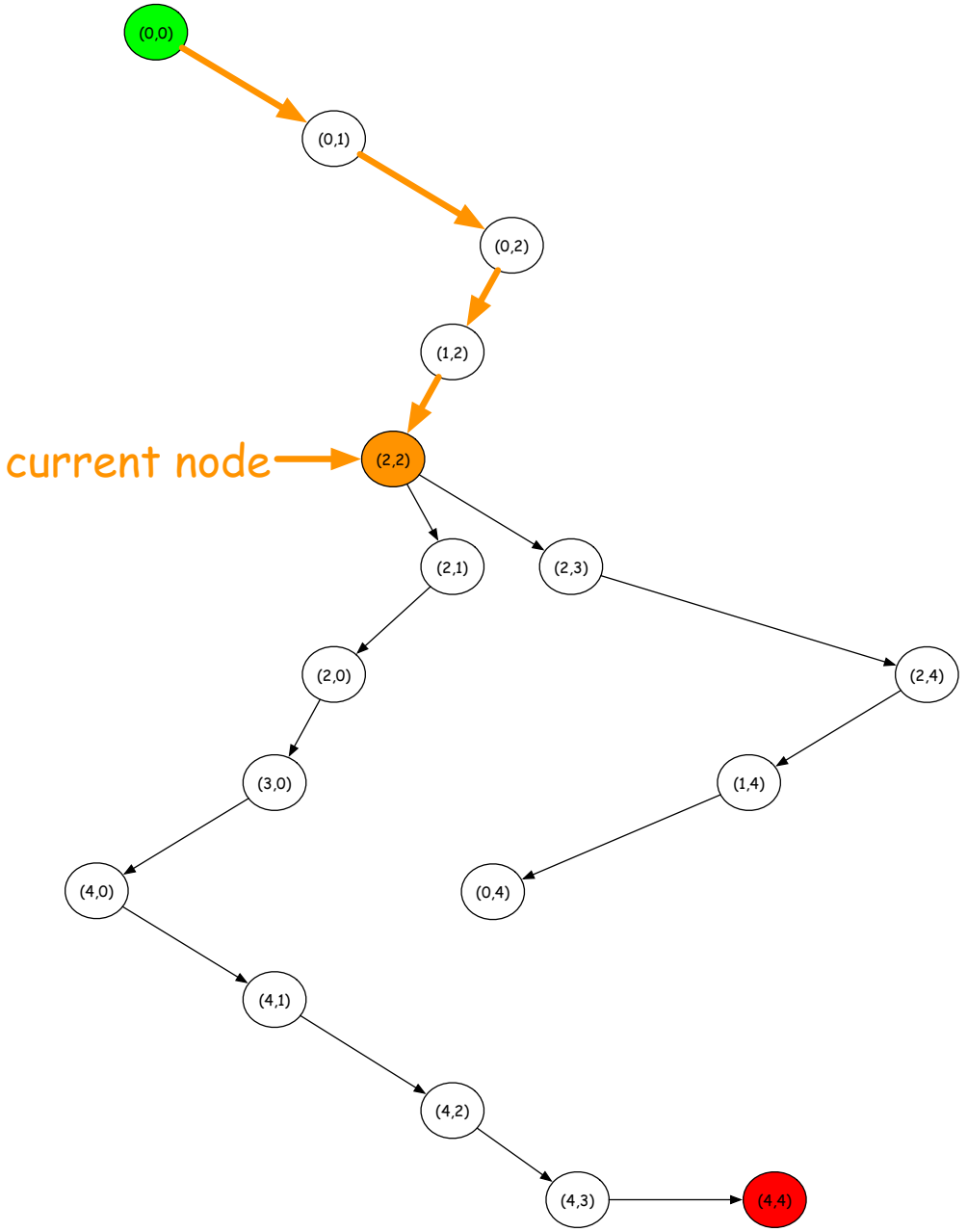


Example 3: Shortest Pathfinding

Dijkstra's Algorithm:

- The most accurate information we can obtain is the recorded events that have already occurred, specifically the cost incurred up to the current node.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0

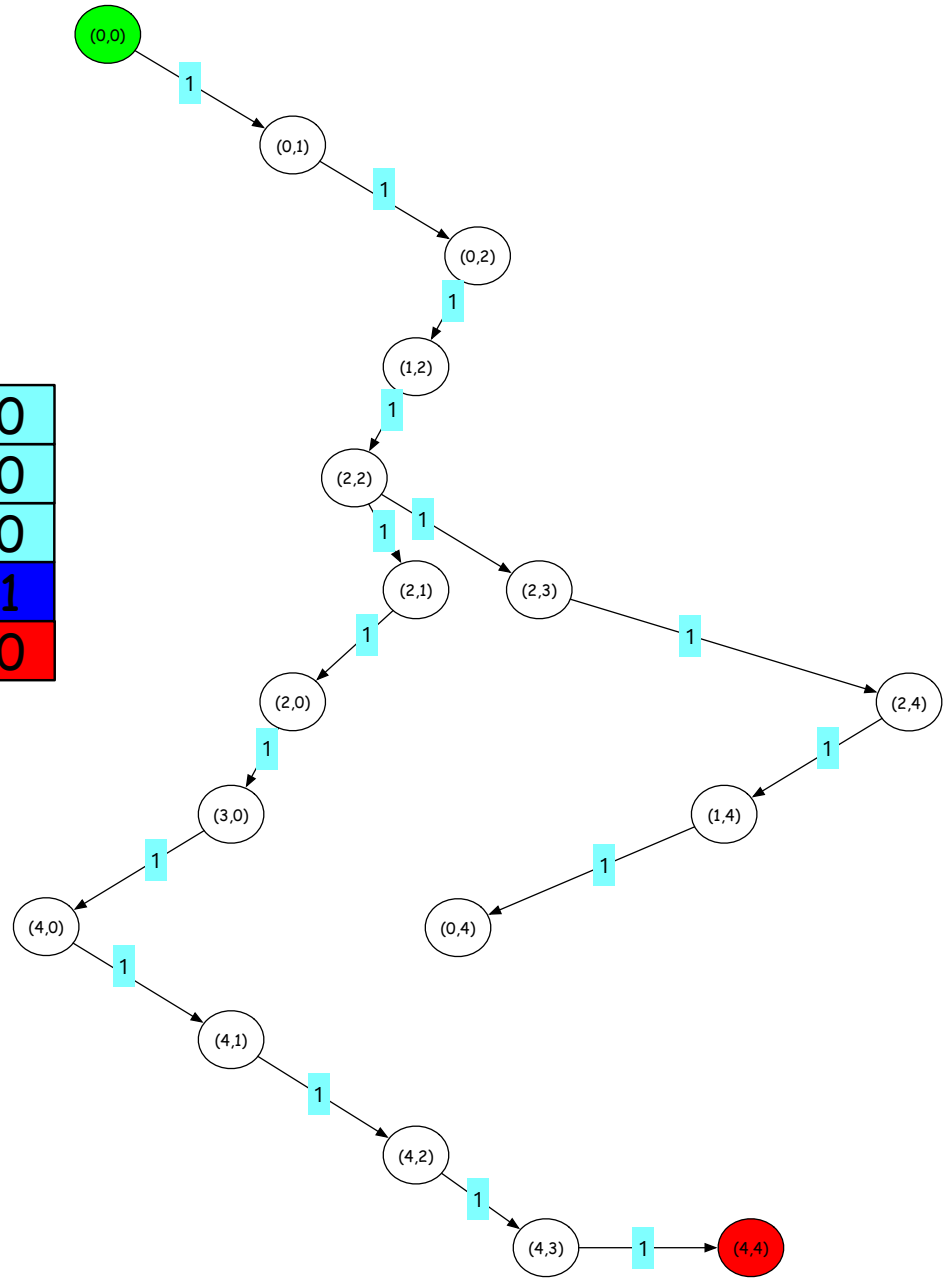


Example 3: Shortest Pathfinding

Dijkstra's Algorithm:

- The most accurate information we can obtain is the recorded events that have already occurred, specifically the cost incurred up to the current node.
- Weighted graph.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0



Example 3: Shortest Pathfinding

Dijkstra's Algorithm:

- The most accurate information we can obtain is the recorded events that have already occurred, specifically the cost incurred up to the current node.
- Weighted graph.
- The algorithm uses a **priority queue** (min-heap) to always expand the node with the smallest current distance.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0

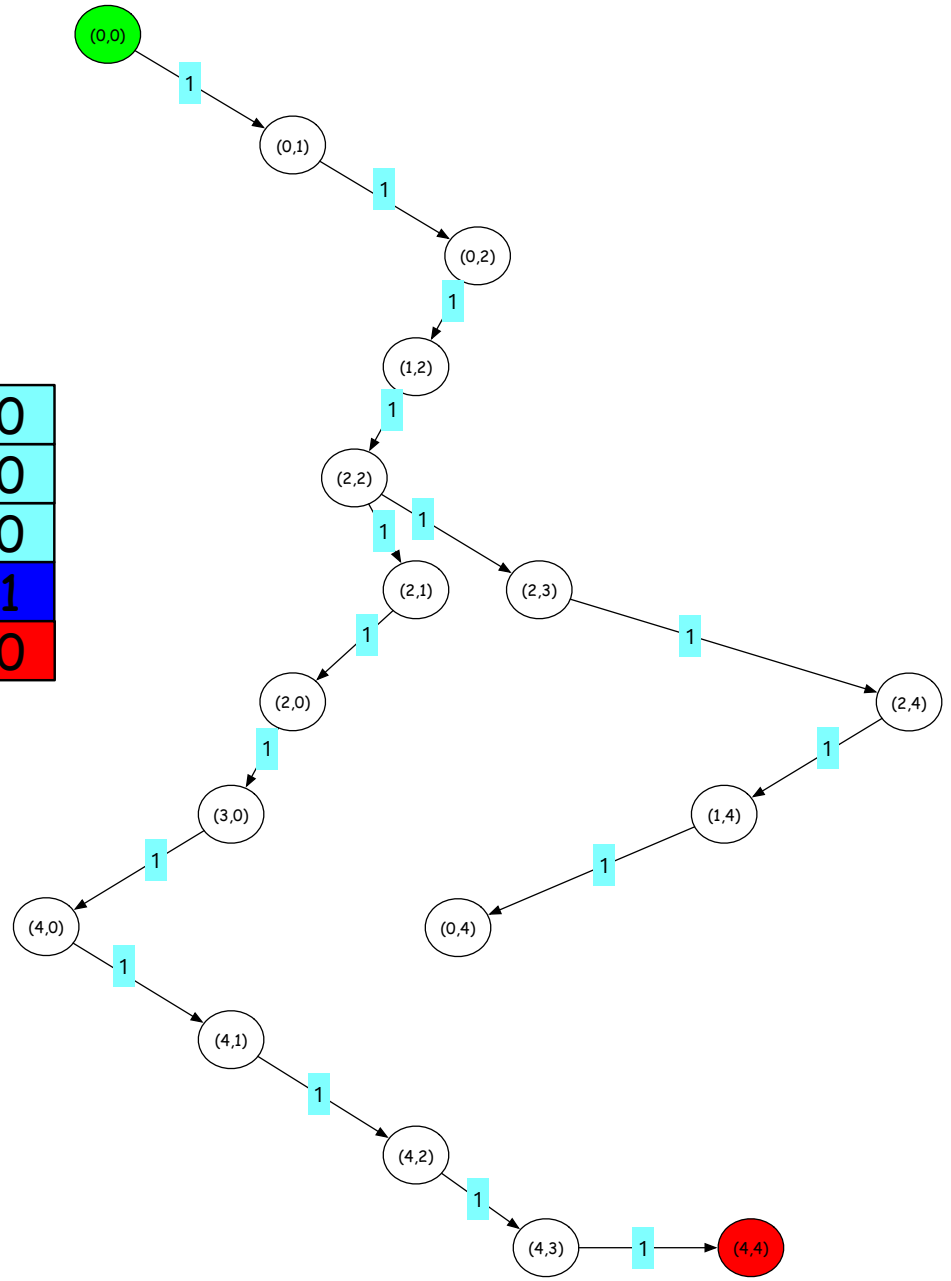
When the current node is (2, 2)

(2, 1, 5)	(2, 3, 5)			

Assume we pop (2, 1)

(2, 3, 5)	(2, 0, 6)			

We will pop out (2, 3) then
(It's like an insurance for exit wrong paths)

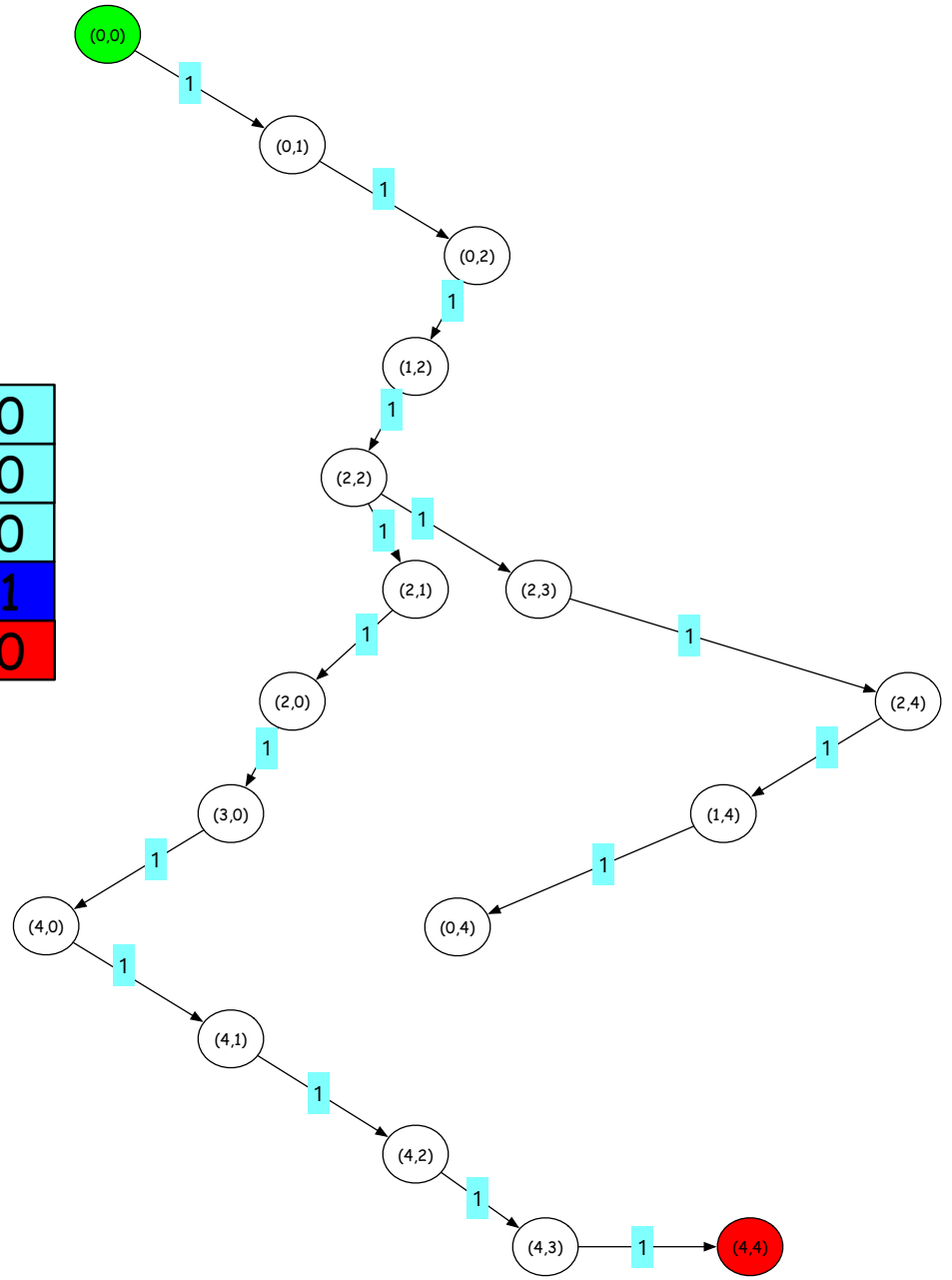


Example 3: Shortest Pathfinding

Dijkstra's Algorithm:

- The most accurate information we can obtain is the recorded events that have already occurred, specifically the cost incurred up to the current node.
- Weighted graph.
- The algorithm uses a **priority queue** (min-heap) to always expand the node with the smallest current distance.
- The process ends when the target node is reached or all reachable nodes are processed.
  - if found, the best (**Optimality**).
  - if reached all, no result.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0

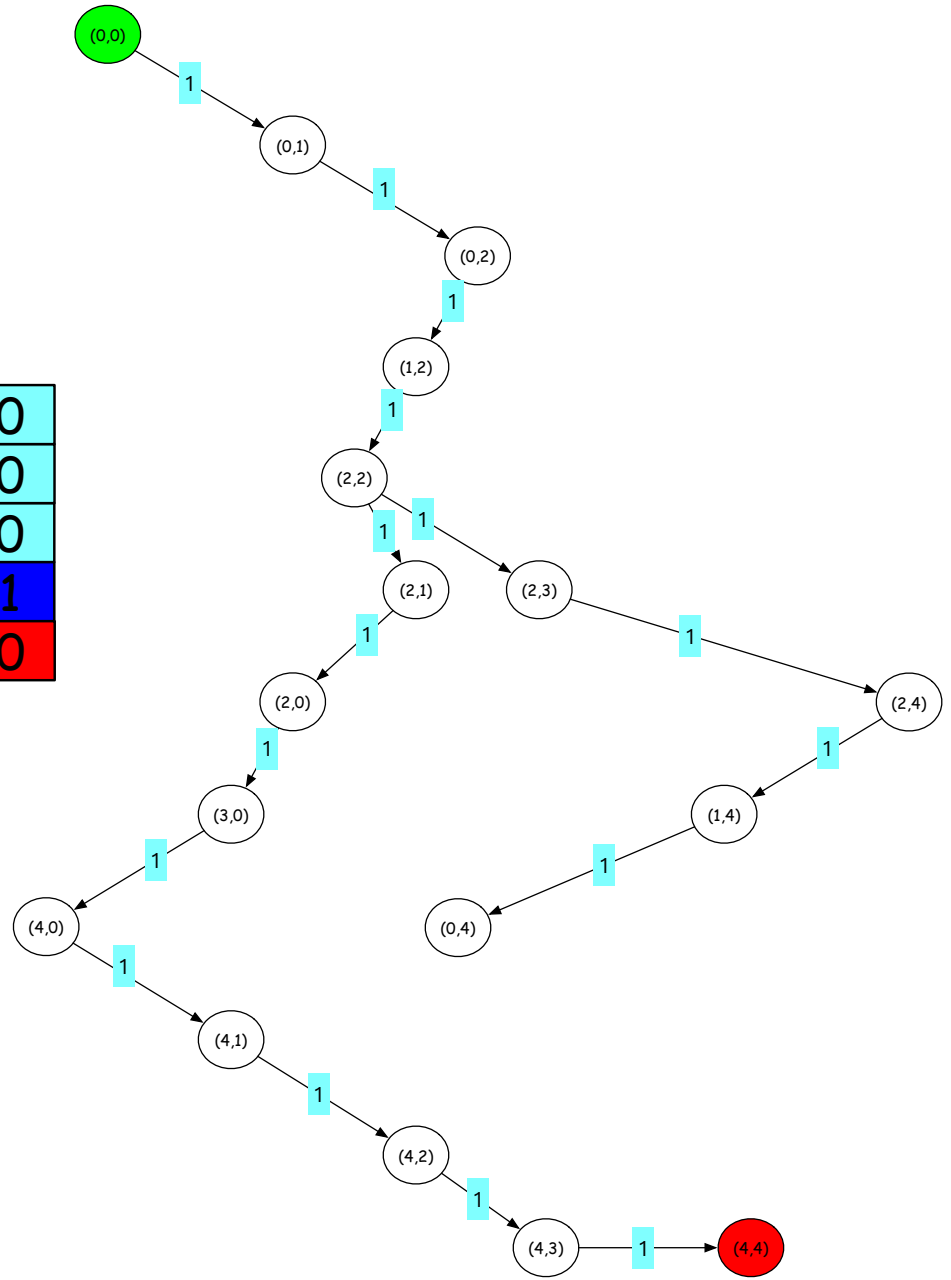


Example 3: Shortest Pathfinding

Dijkstra's Algorithm:

- The most accurate information we can obtain is the recorded events that have already occurred, specifically the cost incurred up to the current node.
- Weighted graph.
- The algorithm uses a **priority queue** (min-heap) to always expand the node with the smallest current distance.
- The process ends when the target node is reached or all reachable nodes are processed.
  - if found, the best (**Optimality**).
  - if reached all, no result.

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0



Past vs. Future

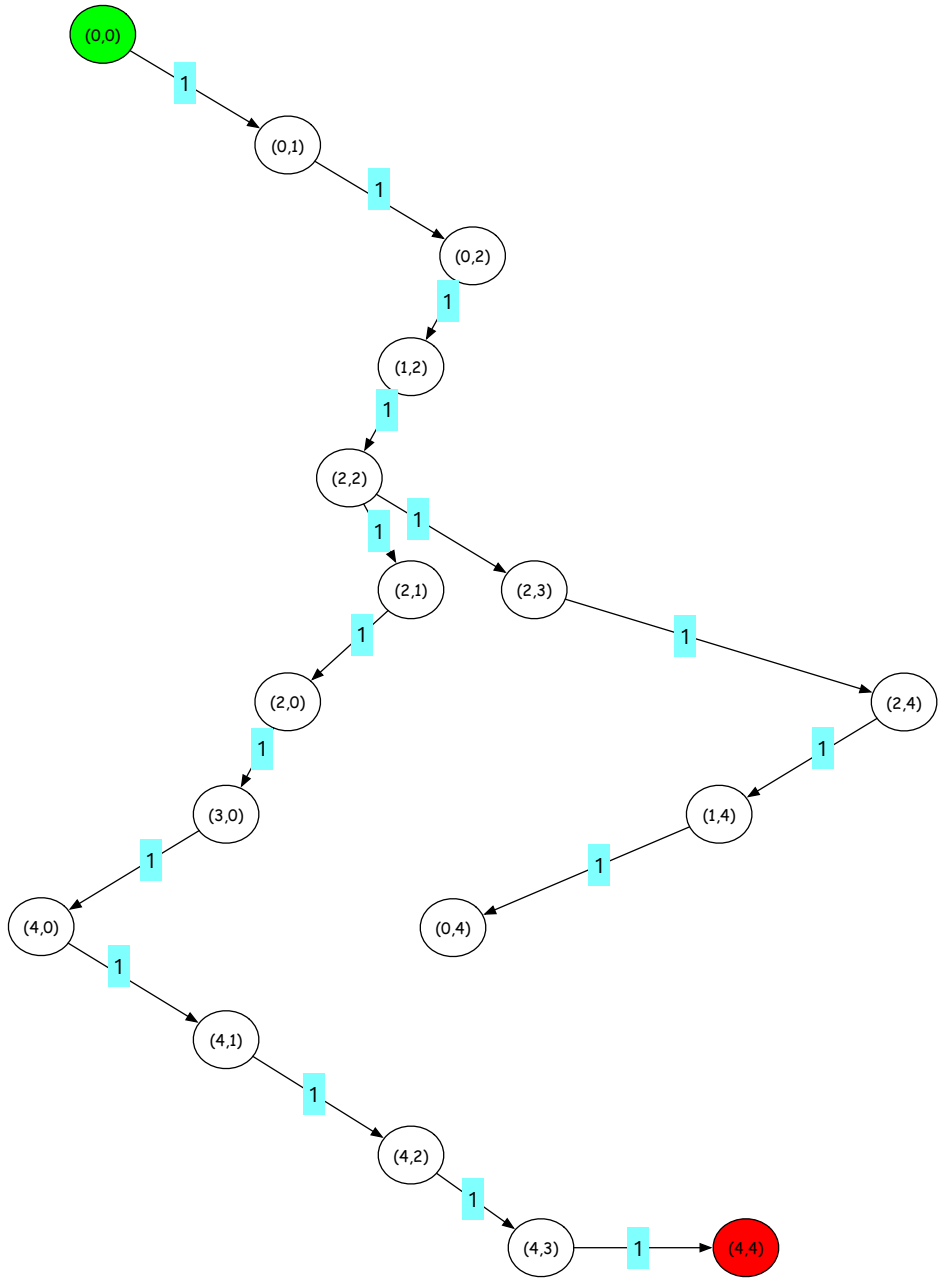
$g(n)$ vs. $h(n)$

Example 3: Shortest Pathfinding

Dijkstra's Algorithm:

- The most accurate information we can obtain is the recorded events that have already occurred, specifically the cost incurred up to the current node.

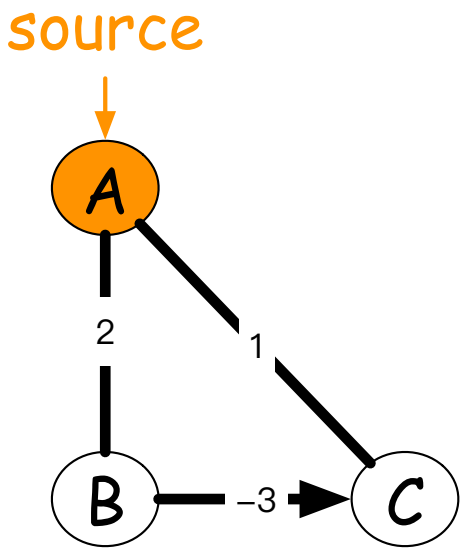
```
Func Dijkstra(graph, source):
    Output: Shortest distances (dist) from source to all
nodes and predecessors (prev)

    for each node v in graph:
        dist[v] = ∞
    dist[source] = 0

    Create a min-priority queue Q
    Add source into the Q

    While Q is not empty:
        dist_so_far, u = Q.pop()
        for each neighbor v of u:
            if dist[u] + weight(u, v) < dist[v]:
                dist[v] = dist[u] + weight(u, v)
                push v into the Q

    return dist and prev
```



Example 3: Shortest Pathfinding

Dijkstra's Algorithm:

- The most accurate information we can obtain is the recorded events that have already occurred, specifically the cost incurred up to the current node.
- (Non-negative) Weighted graph.
  - Greedy Assumption:
    - Dijkstra's algorithm works by always expanding the node with the smallest current distance.
    - Once a node's shortest distance is finalized, the algorithm assumes that no shorter path to this node will ever be found.
    - This assumption is true only if all edge weights are non-negative.



Initiaization

Dis(A)=0, Dis(B)=inf, Dis(C)=inf

Choose A:

Dis(A)=0, Dis(B)=2, Dis(C)=1

Choose C: (do nothing)

Dis(A)=0, Dis(B)=2, Dis(C)=1

Choose B:

2 + -3 = -1 < Dis(C)

Example 3: Shortest Pathfinding

Dijkstra's Algorithm:

- The most accurate information we can obtain is the recorded events that have already occurred, specifically the cost incurred up to the current node.
- Time complexity: $O((|V| + |E|)\log|V|)$
  - When Using a Binary Min-Heap
    - Extracting the minimum node takes $O(\log|V|)$
      - At most V nodes in the heap, $O(|V|\log|V|)$
    - Updating nodes takes $O(\log|V|)$
      - At most relexing $|E|$ edges, $O(|E|\log|V|)$

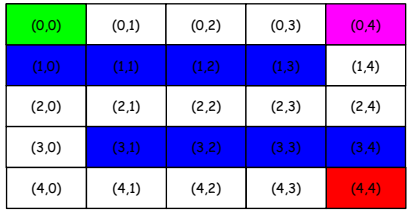
Relaxation is the process of checking whether a shorter path to a node can be found by passing through another node, and if so, updating the shortest known distance.

```
Func Dijkstra(graph, source):

    for each node v in graph:
        dist[v] = ∞
    dist[source] = 0

    Create a min-priority queue Q
    Add source into the Q

    While Q is not empty:
        dist_so_far, u = Q.pop()
        for each neighbor v of u:
            if dist[u] + weight(u, v) < dist[v]:
                dist[v] = dist[u] + weight(u, v)
                push v into the Q

    return dist and prev
```

Example 3: Shortest Pathfinding

A* search: Heuristic = PAST & FUTURE

- For the current node n, e.g., $n = (0,4)$



  - cost function $g(n)$
    - The cost from the start node to the current node n. It's how far (how much cost) you've already traveled.
    - $g((0,4)) = 4$
  - heuristic function $h(n)$
    - It estimates the cost from the current node n to the goal node.
    - It's an educated guess of how much farther it is to the goal.
    - $h((0,4)) = 4$
  - The total cost function $f(n)$
    - the sum of $g(n)$ and $h(n)$, i.e., $f(n) = g(n) + h(n)$
    - $f((0,4)) = g((0,4)) + h((0,4)) = 4 + 4 = 8$

A* can balance between paths that are already known (using $g(n)$)
and paths that are likely to be shorter (using $h(n)$).

Example 3: Shortest Pathfinding

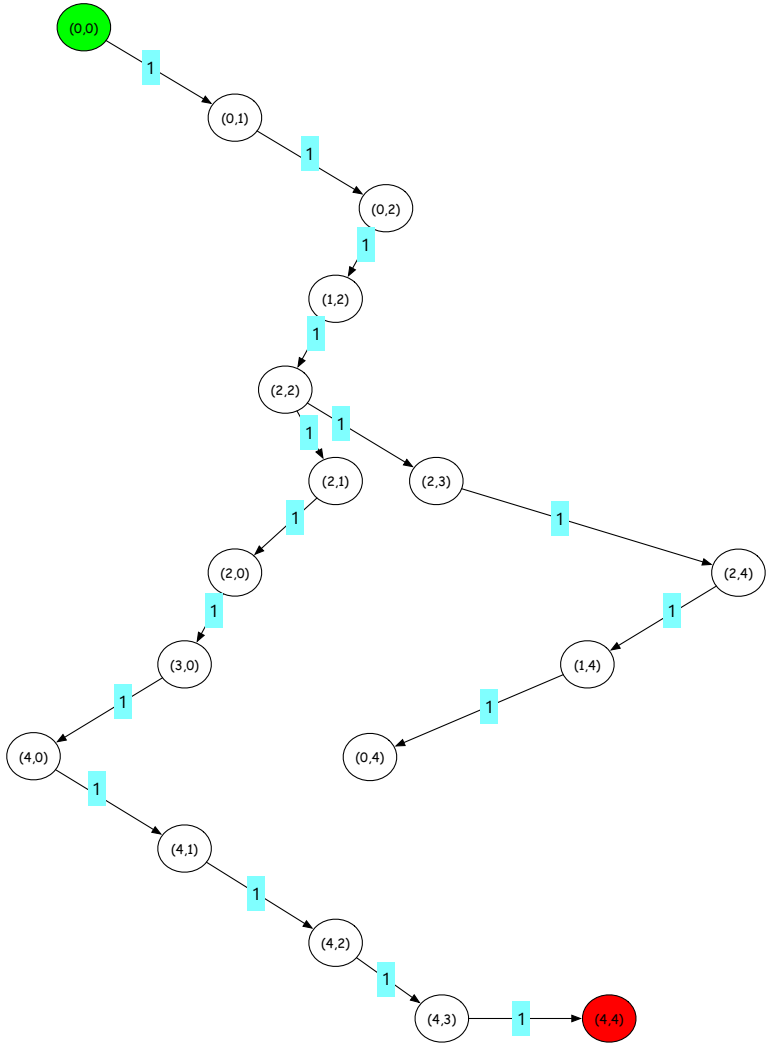
A* search: Heuristic = PAST & FUTURE

- How A* Works:
  1. **Start at the initial node**
     1. e.g., $n = (0,0)$
     2. The initial cost $g((0,0)) = 0$, as we haven't moved yet.
     3. $f((0,0)) = g((0,0)) + h((0,0)) = h((0,0))$
  2. **Explore neighboring nodes**
     1. For each neighbor, e.g., $n = (0,1)$ and $n = (1,0)$
        1. $g(n)$: The cost to reach the current node from the initial node.
        2. $h(n)$: An estimate of how far this neighbor is from the goal node.
  3. **Update the total cost**
     1. For each neighboring node, compute $f(n)$.
     2. put $f(n)$ into a **priority queue** (a list of nodes to explore, ordered by their $f(n)$ values).
  4. **Choose the node with the smallest** $f(n)$ from the queue and repeat the process.
  5. **Stop when you reach the goal node**.
     1. No need to traverse all nodes. The first time A* reaches the goal, it will have found the shortest path because of how $f(n)$ is calculated.

Example 3: Shortest Pathfinding

A* search:

Heuristic = PAST & FUTURE

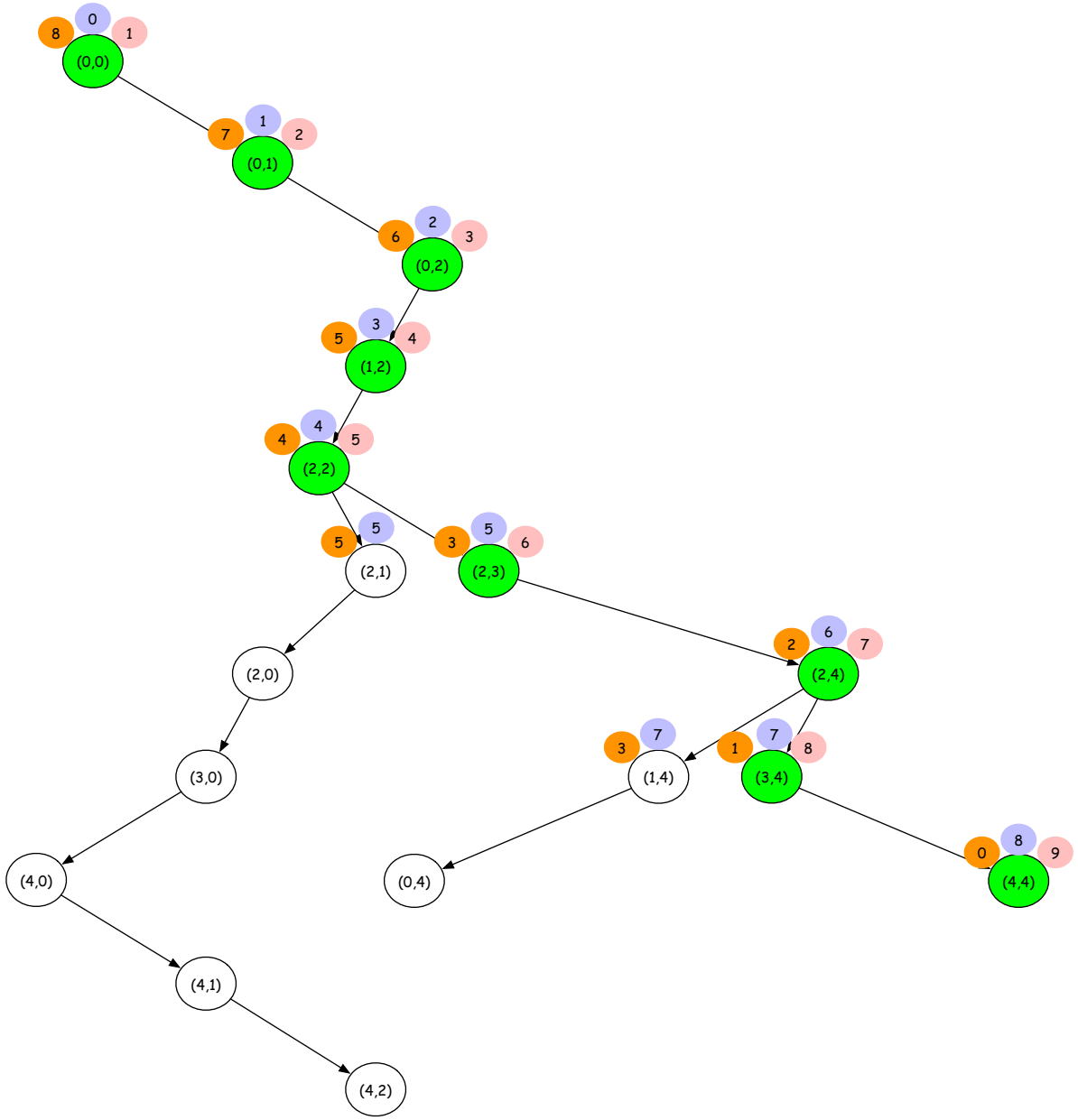



```
A_Star_Search(Grid, Start, Goal):
    Initialize an empty priority queue Q
    Initialize a VISITED to keep track of visited nodes
    Initialize a dictionary "g_score" where g_score[node] = ∞ for all nodes, except
g_score[Start] = 0
    Initialize a dictionary "f_score" where f_score[node] = ∞ for all nodes, except
f_score[Start] = h(Start, Goal)
    Initialize a came_from to track the path

    Add Start to Q with priority f_score[Start]

    while Q is not empty:
        current = Q.pop()  // Get the node in Q with the lowest f_score

        if current == Goal:
            return // Found the shortest path

        Mark current as visited

        for each neighbor of current in the 4 possible directions:
            if neighbor is not walkable or neighbor is in "visited":
                continue

            tentative_g_score = g_score[current] + 1  // Each move costs 1

            if tentative_g_score < g_score[neighbor]:
                Edit came_from // This path to neighbor is better, so record it
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + h(neighbor, Goal)  // f(n)
= g(n) + h(n)

                if neighbor is not in Q:
                    Add neighbor to Q with priority f_score[neighbor]

    return -1  // No path found
```

Example 3: Shortest Pathfinding

A* search: Heuristic = PAST & FUTURE

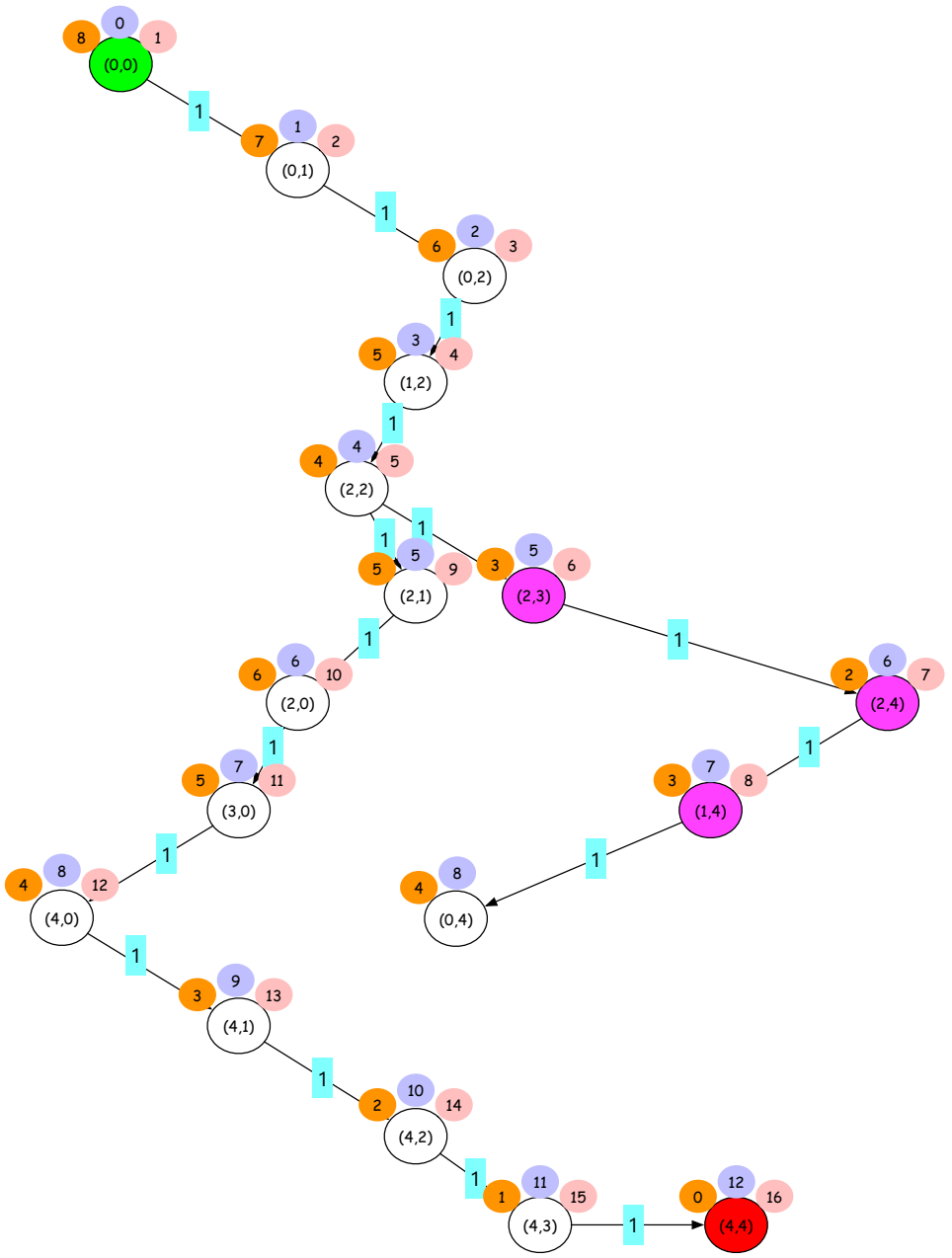
0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	1	0



Example 3: Shortest Pathfinding

A* search: Heuristic = PAST & FUTURE

0	0	0	1	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	1
0	0	0	0	0



Example 3: Shortest Pathfinding

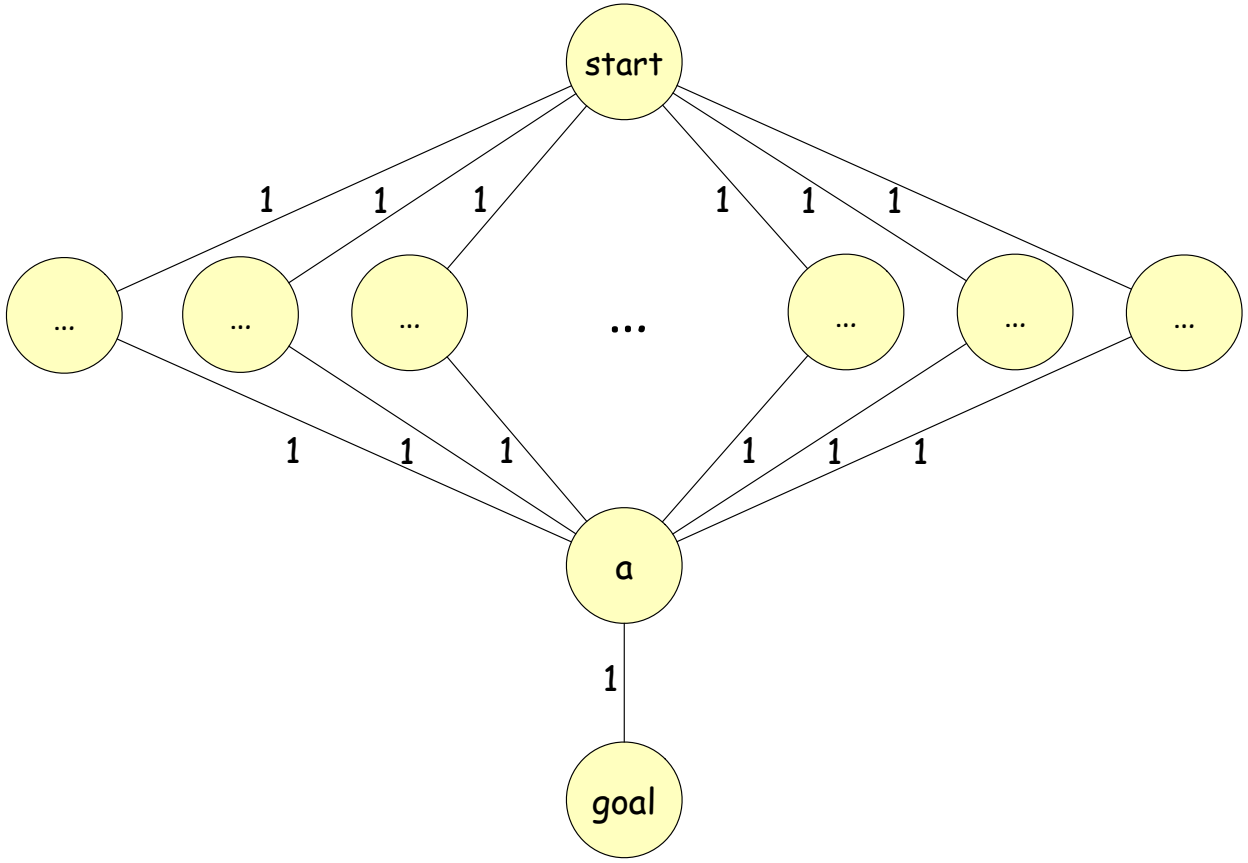
A* search: Heuristic = PAST & FUTURE

- A* search
  - Pros
    - Efficient Search: A* focuses on the most promising nodes first by leveraging the actual cost (g(n)) and heuristic estimate (h(n)); No need to traverse all nodes.
    - Guaranteed Optimality: A* guarantees the Shortest Path as long as the heuristic is admissible and consistent.
    - Customizable with Heuristics
      - Manhattan distance
      - Euclidean distance
      - Chebyshev distance
      - etc

- A* search
  - Cons
    - Memory-Intensive
      - Stores ALL explored nodes in memory. This can become a significant problem in large search spaces as memory usage grows rapidly.
    - Depends Heavily on the Heuristic
      - A poorly chosen or weak heuristic can cause A* to behave inefficiently, exploring many unnecessary nodes.
      - If the heuristic is NOT admissible or inconsistent, A* may not find the optimal solution.

Example 3: Shortest Pathfinding

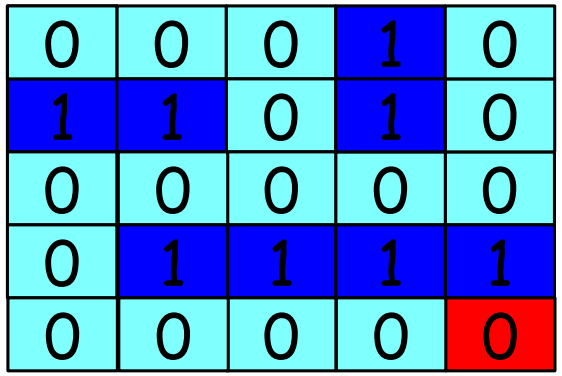
A* search: Heuristic = PAST & FUTURE



- A* search
  - Cons
    - Memory-Intensive
      - Stores ALL explored nodes in memory. This can become a significant problem in large search spaces as memory usage grows rapidly.
    - Depends Heavily on the Heuristic
      - A poorly chosen or weak heuristic can cause A* to behave inefficiently, exploring many unnecessary nodes.
      - If the heuristic is NOT admissible or inconsistent, A* may not find the optimal solution.

Summary

- Uninformed Search
  - Search without guidance, i.e., blind search
  - Breadth-first Search (BFS)
  - Depth-first Search (DFS)

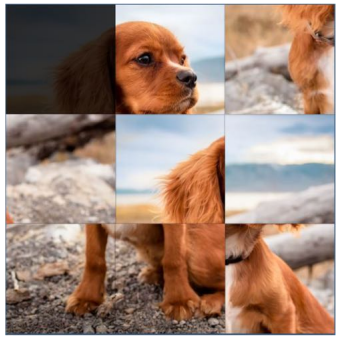


- Informed Search
  - Search with guidance, i.e., heuristic search
    - common heuristics
      - Manhattan Distance
      - Chebyshev Distance
      - Eunlidean Distance
  - Greedy Best-first Search (leverage the estimation regarding future)
  - Dijkstra's Algorithm (leverage the collected info regarding past)
  - A* search (leverage both past and future)

- Define
  - node
  - edge
- Construct plain graph/tree
- Construct weighted graph/tree

Homework: Eight Puzzle Game

The 8-puzzle is a sliding puzzle that consists of a 3×3 grid containing 8 numbered tiles (1 to 8) and one empty space.



The goal is to rearrange the tiles from an arbitrary initial configuration into a specified goal configuration by sliding the tiles into the empty space.

1	2	3
4	6	8
7		5

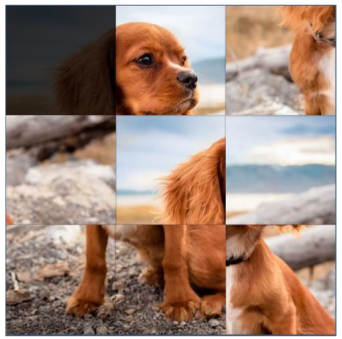
initial state

A series of transformations

1	2	3
4	5	6
7	8	

goal state

Homework: Eight Puzzle Game



1. Can we transfer this problm into a general graph search problem? (node & edge)
    1. If it's okay, provide the graph; if not, provide the analysis to show why.
2. Slove this problem using dynamic construction of the search space.
    1. Implement BFS
    2. Implement DFS
    3. Compare the performance and provide your understanding.
3. Analyze how to build a weighted tree for this problem?
    1. Implement greedy best-first search, report your result.
    2. Implement dijkstra's algorithm, report your result.
    3. Implement A* search, report your result.