COIS 4550H

Assignment 1

WInter 2025

Bachelor Of Computer Science, Trent University

COIS-4550H: Artificial Intelligence

Professor: Sheikh Ahmed Munieb

10th February, 2025

Dikshith Reddy Macherla - Student Id: 0789055

Roman Bamrah - Student Id: 0705310

<u>Assignment 1</u>

Question 1

1. **Depth First Search**

PseudoCode:

```
DFS(graph, start, goal):
    Create an empty stack (LIFO) and push the start node onto it
    Create an empty set to track visited nodes

    while stack is not empty:
        current_node = stack.pop()  # Remove the last added node

        if current_node is the goal:
            return "Goal Found"

        if current_node is not in visited:
            Add current_node to visited

            for each neighbor in graph[current_node] in reverse
order:
                if neighbor is not visited:
                    push neighbor onto the stack

    return "Goal Not Found"
```

Applying DFS Step by Step

Given Start Node: S

Goal Nodes: G1, G2, G3

a. **Goal State = G1**

Let's use a stack (LIFO structure) for DFS.

Step-by-Step Traversal

```
Start at S
Stack: [S]
Visited: {}
Expand S: Push [D, B, A] (in reverse order of adjacency)

Visit A (pop from stack)
Stack: [D, B]
Visited: {S, A}
Expand A: Push [G2, B] (in reverse order)

Visit B (pop from stack)
Stack: [D, G2]
Visited: {S, A, B}
Expand B: Push [C]

Visit C (pop from stack)
```

Stack: [D, G2]
Visited: {S, A, B, C}
Expand C: Push [S, G3, F]

Visit F (pop from stack)
Stack: [D, G2, S, G3]
Visited: {S, A, B, C, F}
Expand F: Push [G1]

Visit G1 (pop from stack)
Stack: [D, G2, S, G3]
Visited: {S, A, B, C, F, G1}
Goal G1 Found!

DFS Path to G1:

S → A → B → C → F → G1

b. Goal State = G2

Step-by-Step Traversal

Start at S
Stack: [S]
Visited: {}
Expand S: Push [D, B, A] (in reverse order)

Visit A (pop from stack)
Stack: [D, B]
Visited: {S, A}
Expand A: Push [G2, B] (in reverse order)

Visit G2 (pop from stack)
Stack: [D, B]
Visited: {S, A, G2}
Goal G2 Found!

DFS Path to G2:

S → A → G2

c. Goal State = G3

Step-by-Step Traversal

Start at S
Stack: [S]
Visited: {}
Expand S: Push [D, B, A] (in reverse order)

Visit A (pop from stack)
Stack: [D, B]
Visited: {S, A}
Expand A: Push [G2, B]

Visit B (pop from stack)
Stack: [D, G2]

```
Visited: {S, A, B}
Expand B: Push [C]

Visit C (pop from stack)
Stack: [D, G2]
Visited: {S, A, B, C}
Expand C: Push [S, G3, F]

Visit G3 (pop from stack)
Stack: [D, G2, S, F]
Visited: {S, A, B, C, G3}
Goal G3 Found!
```

DFS Path to G3:

S → A → B → C → G3

2. **Breadth First Search**

PseudoCode:

```
BFS(graph, start, goal):
    Create an empty queue (FIFO) and enqueue the start node
    Create an empty set to track visited nodes

    while queue is not empty:
        current_node = queue.dequeue()  # Remove the front node

        if current_node is the goal:
            return "Goal Found"

        if current_node is not in visited:
            Add current_node to visited

            for each neighbor in graph[current_node]:
                if neighbor is not visited:
                    enqueue neighbor
    return "Goal Not Found"
```

Applying BFS Step by Step

Given Start Node: S

Goal Nodes: G1, G2, G3

Let's use a queue (FIFO structure) for BFS.

Step-by-Step Traversal

```
Start at S
Queue: [S]
Visited: {}
Expand S: Enqueue [A, B, D]

Visit A (dequeue from queue)
Queue: [B, D]
Visited: {S, A}
```

```
Expand A: Enqueue [B, G2]

Visit B (dequeue from queue)
Queue: [D, B, G2]
Visited: {S, A, B}
Expand B: Enqueue [C]

Visit D (dequeue from queue)
Queue: [B, G2, C]
Visited: {S, A, B, D}
Expand D: Enqueue [C, E]

Visit G2 (dequeue from queue)
Queue: [C, E]
Visited: {S, A, B, D, G2}
```
Goal G2 Found

```
Visit C (dequeue from queue)
Queue: [E]
Visited: {S, A, B, D, G2, C}
Expand C: Enqueue [F, G3]

Visit E (dequeue from queue)
Queue: [F, G3]
Visited: {S, A, B, D, G2, C, E}
Expand E: Enqueue [G1]

Visit F (dequeue from queue)
Queue: [G3, G1]
Visited: {S, A, B, D, G2, C, E, F}
Expand F: Enqueue [G1] (already in queue)

Visit G3 (dequeue from queue)
Queue: [G1]
Visited: {S, A, B, D, G2, C, E, F, G3}
```
Goal G3 Found

```
Visit G1 (dequeue from queue)
Queue: []
Visited: {S, A, B, D, G2, C, E, F, G3, G1}
```
Goal G1 Found

BFS Paths to Goal States:

Goal G2 path: S → A → G2

Goal G3 path: S → B → C → G3

Goal G1 path: S → D → E → G1

3. **Uniform Cost Search**

PseudoCode:

```
import heapq

def UCS(graph, start, goal):
```

```
    priority_queue = []  # Min-heap for priority queue
    heapq.heappush(priority_queue, (0, start, []))  # (cost, node,
path)

    visited = set()

    while priority_queue:
        cost, current_node, path = heapq.heappop(priority_queue)  #
Get node with least cost

        if current_node in visited:
            continue

        path = path + [current_node]  # Add to path
        visited.add(current_node)

        if current_node == goal:
            return (path, cost)  # Return final path and cost

        for neighbor, weight in graph[current_node]:
            if neighbor not in visited:
                heapq.heappush(priority_queue, (cost + weight,
neighbor, path))

    return None  # No path found
```

Applying UCS Step by Step

Given Start Node: S

Goal Nodes: G1, G2, G3

a. **Goal State = G1**

Step-by-Step Traversal

```
Start at S
Priority Queue: [(0, S, [])]
Visited: {}

Expand S (0 cost)

Visit neighbors: A(6), B(11), D(6)
Priority Queue: [(6, A), (11, B), (6, D)]
Expand A (6 cost)

Visit neighbors: B(6+5=11), G2(6+10=16)
Priority Queue: [(6, D), (11, B), (11, B), (16, G2)]
Expand D (6 cost)

Visit neighbors: C(6+2=8), E(6+2=8), S(6+3=9)
Priority Queue: [(8, C), (8, E), (9, S), (11, B), (11, B),
(16, G2)]
Expand C (8 cost)

Visit neighbors: F(8+9=17), G3(8+6=14), S(8+9=17)
```

Priority Queue: [(8, E), (9, S), (11, B), (11, B), (14, G3), (16, G2), (17, F), (17, S)]
Expand E (8 cost)

Visit neighbors: G1(8+9=17)
Priority Queue: [(9, S), (11, B), (11, B), (14, G3), (16, G2), (17, F), (17, S), (17, G1)]
Expand G1 (17 cost)
Goal G1 Found!

UCS Path to G1:

S → D → E → G1

Total Cost = 17

b. Goal State = G2

Step-by-Step Traversal

Start at S
Priority Queue: [(0, S, [])]
Visited: {}

Expand S (0 cost)

Visit neighbors: A(6), B(11), D(6)
Priority Queue: [(6, A), (11, B), (6, D)]
Expand A (6 cost)

Visit neighbors: B(6+5=11), G2(6+10=16)
Priority Queue: [(6, D), (11, B), (11, B), (16, G2)]
Expand D (6 cost)

Visit neighbors: C(6+2=8), E(6+2=8), S(6+3=9)
Priority Queue: [(8, C), (8, E), (9, S), (11, B), (11, B), (16, G2)]
Expand G2 (16 cost)

Goal G2 Found!

UCS Path to G2:

S → A → G2

Total Cost = 16

c. Goal State = G3

Step-by-Step Traversal

Start at S
Priority Queue: [(0, S, [])]
Visited: {}

Expand S (0 cost)

```
Visit neighbors: A(6), B(11), D(6)
Priority Queue: [(6, A), (11, B), (6, D)]
Expand A (6 cost)

Visit neighbors: B(6+5=11), G2(6+10=16)
Priority Queue: [(6, D), (11, B), (11, B), (16, G2)]
Expand D (6 cost)

Visit neighbors: C(6+2=8), E(6+2=8), S(6+3=9)
Priority Queue: [(8, C), (8, E), (9, S), (11, B), (11, B),
(16, G2)]
Expand C (8 cost)

Visit neighbors: F(8+9=17), G3(8+6=14), S(8+9=17)
Priority Queue: [(8, E), (9, S), (11, B), (11, B), (14,
G3), (16, G2), (17, F), (17, S)]
Expand G3 (14 cost)

Goal G3 Found!
```

UCS Path to G3:

S → D → C → G3

Total Cost = 14

Question 2:

1. **Greedy Best First Search**

PseudoCode:

```
function GREEDY_BEST_FIRST_SEARCH(graph, start, goal, h):
    OPEN = a priority queue, ordered by h(n) in ascending order
    CLOSED = an empty set
    PARENT = an empty dictionary  # for path reconstruction

    OPEN.insert(start)
    PARENT[start] = null

    while OPEN is not empty:
        current = OPEN.pop()  # node with smallest h-value

        if current == goal:
            return ReconstructPath(PARENT, current)

        CLOSED.add(current)

        # Expand neighbors
        for (neighbor, cost) in graph[current]:
            if neighbor not in OPEN and neighbor not in CLOSED:
                PARENT[neighbor] = current
                OPEN.insert(neighbor)

    return "No path found"
```

```
function ReconstructPath(PARENT, node):
    path = []
    while node != null:
        path.prepend(node)
        node = PARENT[node]
    return path
```

Assumptions for Heuristic h(n)

Greedy Best-First Search uses a heuristic function h(n) that estimates the distance to the nearest goal. Let's assume the following heuristic values:

```
'S9': 9,
'A7': 7,
'B3': 3,
'D2': 2,
'C4': 4,
'E5': 5,
'F7': 7,
'G1': 0,
'G2': 0,
'G3': 0
```

Step-by-Step Execution of GBFS

Start at S9

Priority Queue: [(9, S9)]

Visited: {}

a. **Goal State = G1**

```
Expand S9 (h=9)

Visit neighbors: A7(7), B3(3), D2(2)
Priority Queue: [(2, D2), (3, B3), (7, A7)]
Expand D2 (h=2)

Visit neighbors: C4(4), E5(5), S9(9)
Priority Queue: [(3, B3), (4, C4), (5, E5), (7, A7)]
Expand B3 (h=3)

Visit neighbors: C4(4)
Priority Queue: [(4, C4), (4, C4), (5, E5), (7, A7)]
Expand C4 (h=4)

Visit neighbors: F7(7), G3(0), S9(9)
Priority Queue: [(0, G3), (5, E5), (7, A7), (7, F7)]
Expand E5 (h=5)

Visit neighbors: G1(0)
Priority Queue: [(0, G1), (0, G3), (7, A7), (7, F7)]
Expand G1 (h=0)
```

Goal G1 Found

Path to G1:
S9 → D2 → E5 → G1

b. **Goal State = G2**

Expand S9 (h=9)

Visit neighbors: A7(7), B3(3), D2(2)
Priority Queue: [(2, D2), (3, B3), (7, A7)]
Expand D2 (h=2)

Visit neighbors: C4(4), E5(5), S9(9)
Priority Queue: [(3, B3), (4, C4), (5, E5), (7, A7)]
Expand B3 (h=3)

Visit neighbors: C4(4)
Priority Queue: [(4, C4), (4, C4), (5, E5), (7, A7)]
Expand C4 (h=4)

Visit neighbors: F7(7), G3(0), S9(9)
Priority Queue: [(0, G3), (5, E5), (7, A7), (7, F7)]
Expand A7 (h=7)

Visit neighbors: B3(3), G2(0)
Priority Queue: [(0, G2), (0, G3), (5, E5), (7, F7)]
Expand G2 (h=0)

Goal G2 Found

Path to G2:
S9 → A7 → G2

c. **Goal State = G3**

Expand S9 (h=9)

Visit neighbors: A7(7), B3(3), D2(2)
Priority Queue: [(2, D2), (3, B3), (7, A7)]
Expand D2 (h=2)

Visit neighbors: C4(4), E5(5), S9(9)
Priority Queue: [(3, B3), (4, C4), (5, E5), (7, A7)]
Expand B3 (h=3)

Visit neighbors: C4(4)
Priority Queue: [(4, C4), (4, C4), (5, E5), (7, A7)]
Expand C4 (h=4)

Visit neighbors: F7(7), G3(0), S9(9)
Priority Queue: [(0, G3), (5, E5), (7, A7), (7, F7)]
Expand G3 (h=0)

Goal G3 Found

Path to G3:
S9 → B3 → C4 → G3

2. **A* Search**

PseudoCode:

```
A_STAR_SEARCH(graph, startNodes, h):

    # Priority queue (min-heap) ordered by f(n)
    frontier = PRIORITY_QUEUE(order_by="lowest f-value")

    # Keep track of visited expansions in a closed set (optional in
some implementations)
    closed_set = SET()

    # Best-known distance from start
    g = dictionary()
    # For reconstructing path
    parent = dictionary()

    # Initialize
    for s in startNodes:
        g[s] = 0
        f_s = g[s] + h(s)
        frontier.push(s, priority=f_s)

    while not frontier.empty():
        current = frontier.pop()  # node with smallest f

        if GoalTest(current):
            return ReconstructPath(current, parent), g[current]

        # Mark current as explored
        closed_set.add(current)

        # For each neighbor of current:
        for (neighbor, edgeCost) in graph[current]:
            tentative_g = g[current] + edgeCost
            if neighbor in closed_set:
                # If we have seen it in closed_set,
                # only proceed if we found a strictly better g:
                if tentative_g >= g.get(neighbor, infinity):
                    continue
                # Otherwise remove neighbor from closed so we can
revisit
                closed_set.remove(neighbor)

            if tentative_g < g.get(neighbor, infinity):
                # Found a new best path to neighbor
                parent[neighbor] = current
                g[neighbor] = tentative_g
                f_neighbor = tentative_g + h(neighbor)
                frontier.push_or_update(neighbor,
priority=f_neighbor)
```

```
    return None, infinity  # If we exhaust the frontier without
reaching a goal

def ReconstructPath(node, parent):
    path = []
    while node in parent:
        path.insert(0, node)
        node = parent[node]
    path.insert(0, node)  # Insert start
    return path
```

Assumptions for Heuristic h(n)

For A*, Let's use the same heuristic function h(n) as in Greedy

Best-First Search, which estimates the cost from a node to the goal:

```
'S9': 9,
'A7': 7,
'B3': 3,
'D2': 2,
'C4': 4,
'E5': 5,
'F7': 7,
'G1': 0,
'G2': 0,
'G3': 0
```

Step-by-Step Execution of A* Search

Start at S9

Priority Queue: [(0+9, 0, S9)]

Visited: {}

a. **Goal State = G1**

Expand S9 (g=0, h=9, f=9)

Visit neighbors: A7(6+7=13), B3(11+3=14), D2(6+2=8)
Priority Queue: [(8, 6, D2), (13, 6, A7), (14, 11, B3)]
Expand D2 (g=6, h=2, f=8)

Visit neighbors: C4(6+2=8+4=12), E5(6+2=8+5=13), S9(6+3=9+9=18)
Priority Queue: [(12, 8, C4), (13, 6, A7), (13, 8, E5), (14, 11, B3)]
Expand C4 (g=8, h=4, f=12)

Visit neighbors: F7(8+9=17+7=24), G3(8+6=14+0=14), S9(8+9=17+9=26)
Priority Queue: [(13, 6, A7), (13, 8, E5), (14, 11, B3), (14, 14, G3), (24, 17, F7)]
Expand E5 (g=8, h=5, f=13)

Visit neighbors: G1(8+9=17+0=17)

Priority Queue: [(13, 6, A7), (14, 11, B3), (14, 14, G3), (17, 17, G1), (24, 17, F7)]
Expand G1 (g=17, h=0, f=17)

Goal G1 Found

Optimal Path to G1:

S9 → D2 → E5 → G1

Total Cost = 17

b. **Goal State = G2**

Expand S9

Priority Queue: [(8, 6, D2), (13, 6, A7), (14, 11, B3)]
Expand A7

Visit neighbors: B3(6+5=11+3=14), G2(6+10=16+0=16)
Priority Queue: [(8, 6, D2), (14, 11, B3), (16, 16, G2)]
Expand G2

Goal G2 Found

Optimal Path to G2:

S9 → A7 → G2

Total Cost = 16

c. **Goal State = G3**

Expand S9

Priority Queue: [(8, 6, D2), (13, 6, A7), (14, 11, B3)]
Expand D2

Priority Queue: [(12, 8, C4), (13, 6, A7), (14, 11, B3)]
Expand C4

Visit neighbors: G3(8+6=14+0=14)
Priority Queue: [(13, 6, A7), (14, 11, B3), (14, 14, G3)]
Expand G3

Goal G3 Found

Optimal Path to G3:

S9 → D2 → C4 → G3

Total Cost = 14

3. **Beam Search**

PseudoCode:

```
function BEAM_SEARCH(graph, start, goals, h, w):
    # frontier is the set of nodes at the current 'layer'
    frontier = [start]
    PARENT = {}                # for path reconstruction
    PARENT[start] = None

    while frontier is not empty:
        # Check if any node in the frontier is a goal
        for node in frontier:
            if node in goals:
                return ReconstructPath(PARENT, node)

        # Generate all successors of the current frontier
        successors = []
        for node in frontier:
            for (nbr, cost) in graph[node]:
                # If not yet discovered
                if nbr not in PARENT:
                    PARENT[nbr] = node
                    successors.append(nbr)

        # If no successors, search ends (failure)
        if not successors:
            return "No path found"

        # Sort successors by heuristic (lowest is best)
        successors.sort(key=lambda x: h[x])

        # Keep only top-w nodes
        frontier = successors[:w]

    return "No path found"


function ReconstructPath(PARENT, goalNode):
    path = []
    node = goalNode
    while node is not None:
        path.insert(0, node)   # prepend
        node = PARENT[node]
    return path
```

Assumptions for Heuristic h(n)

Beam Search selects nodes based on their heuristic h(n). Let's assume the following heuristic values:

```
'S9': 9,
'A7': 7,
'B3': 3,
'D2': 2,
'C4': 4,
'E5': 5,
'F7': 7,
'G1': 0,
'G2': 0,
```

'G3': 0

Step-by-Step Execution of Beam Search

Given Start Node: S9

Goal Nodes: G1, G2, G3

Choosing Beam Width k=2 (only the top 2 nodes at each step)

a. **Goal State = G1**

Expand S9 (h=9)

Candidates: A7(7), B3(3), D2(2)
Select Top-2: D2(2), B3(3)
Expand D2 (h=2) and B3 (h=3)

Candidates from D2: C4(4), E5(5)
Candidates from B3: C4(4)
Select Top-2: C4(4), E5(5)
Expand C4 (h=4) and E5 (h=5)

Candidates from C4: F7(7), G3(0)
Candidates from E5: G1(0)
Select Top-2: G1(0), G3(0)
Expand G1 (h=0)

Goal G1 Found

Path to G1:

S9 → D2 → E5 → G1

b. **Goal State = G2**

Expand S9 (h=9)

Candidates: A7(7), B3(3), D2(2)
Select Top-2: D2(2), B3(3)
Expand D2 (h=2) and B3 (h=3)

Candidates from D2: C4(4), E5(5)
Candidates from B3: C4(4)
Select Top-2: C4(4), E5(5)
Expand A7 (h=7)

Candidates: B3(3), G2(0)
Select Top-2: G2(0), B3(3)
Expand G2 (h=0)

Goal G2 Found

Path to G2:

S9 → A7 → G2

c. **Goal State = G3**

Expand S9 (h=9)

Candidates: A7(7), B3(3), D2(2)
Select Top-2: D2(2), B3(3)
Expand D2 (h=2) and B3 (h=3)

Candidates from D2: C4(4), E5(5)
Candidates from B3: C4(4)
Select Top-2: C4(4), E5(5)
Expand C4 (h=4) and E5 (h=5)

Candidates from C4: F7(7), G3(0)
Candidates from E5: G1(0)
Select Top-2: G1(0), G3(0)
Expand G3 (h=0)

Goal G3 Found

Path to G3:

S9 → D2 → C4 → G3

Question 3:

When I'm developing an application for a device with limited memory (like a smartphone) and need to ensure I find the true shortest path, I would choose **A* search**. Let me explain why, and then walk through the space and time complexities.

My Rationale

1. **Optimality:**

A* ensures that I get the shortest (optimal) path if my heuristic is admissible, meaning it never overestimates the actual cost to reach the goal.

2. **Heuristic Guidance:**

Because A* uses a heuristic to guide which paths to explore first, it tends to expand far fewer nodes compared to algorithms that do not use a heuristic (like Breadth First Search or Uniform Cost Search). This targeted approach can be a big help in keeping memory usage in check on limited devices.

3. **Memory Constraints:**

While A* can still have high worst-case memory usage (since it needs to store explored and frontier nodes), a good heuristic significantly reduces the number of nodes that end up in memory at once. In many practical scenarios, it is much more memory-friendly than a blind or purely cost-based search.

Why Not the Other Algorithms?

a. **Greedy Best-First Search:**

It only uses the heuristic and ignores the cost so far. It can quickly chase down what looks like a promising path but may lead me away from the truly shortest path.

b. **Beam Search:**

It limits the number of paths explored, so it saves memory, but that same cutoff can mean missing the best path (no optimality guarantee).

c. **Depth-First Search (DFS):**

DFS only keeps a path to a leaf, so it uses minimal memory, but it doesn't guarantee the shortest path unless I exhaustively search—which can be very large in time.

d. **Breadth-First Search (BFS):**

If edges are all the same cost, BFS can find the shortest path, but it stores many nodes in memory at once (entire layers at a time).

e. **Uniform Cost Search (UCS):**

It guarantees the shortest path for varying costs but can still expand a very large number of nodes because it doesn't leverage a heuristic—it purely operates on the lowest current path cost.

Time and Space Complexities of A*

a. Time Complexity (Worst Case): $O(b^d)$, where b is the branching factor and d is the depth of the optimal solution. In practice, a good heuristic usually makes it much more efficient than BFS or UCS.

b. Space Complexity (Worst Case): $O(b^d)$ as well, mainly because I have to keep track of all expanded nodes (closed set) and the frontier (open set). With a strong heuristic, the number of nodes stored is often drastically smaller in real-world scenarios.

Conclusion

In my experience, A* is the most balanced choice for a memory-constrained device that still needs the true shortest path. Its heuristic guidance reduces unnecessary exploration, and when paired with a well-designed heuristic, it typically keeps both time and memory requirements manageable.

Question 4:

The core idea is that BFS can be viewed as UCS when all edges have the same (uniform) cost. Uniform-Cost Search (UCS) expands nodes in order of increasing path cost. When each edge has cost = 1 (or any fixed constant c), the "path cost" from the start state is simply the number of edges in that path (i.e., the depth). Thus, expanding nodes in order of increasing path cost = expanding nodes level by level, which is precisely BFS.

Below is a more step-by-step outline of the argument:

Breadth-First Search (BFS)

- Traverses a graph or tree level by level.
- When searching for a goal node, BFS explores all nodes at depth d before exploring any nodes at depth d+1.

Uniform-Cost Search (UCS)

- A generalization of Dijkstra's shortest path algorithm for exploring a state space.

- Maintains a priority queue (often called the frontier or open list) keyed by the current path cost from the start node.
- Always expands the node that has the lowest cumulative path cost so far.

Equal Edge Costs Imply Equivalent Expansions

When all edges in the graph have the same cost (assume each edge cost = 1 for simplicity):

1. The cost of reaching a node from the start is equal to the number of edges on the path from the start to that node.
2. UCS will store nodes in a priority queue sorted by this cost (cumulative number of edges).
    - The node with the smallest cumulative cost is expanded first.
3. However, sorting by the cumulative number of edges is exactly the same as expanding nodes in increasing order of their depth in the search tree (or levels in the graph).

Therefore, UCS in this scenario will expand:

- All nodes reachable with 1 edge (depth 1),
- Then all nodes reachable with 2 edges (depth 2),
- Then all nodes are reachable with 3 edges (depth 3), etc.

This is precisely what BFS does: exploring nodes level by level (depth by depth).

Key Observations

1. If edges do not all have equal cost, BFS and UCS diverge. BFS still explores by shallowest depth first (ignoring cost differences), whereas UCS always picks the node with the lowest total cost (not necessarily shallowest depth).

2. For BFS, we do not typically maintain a cost-to-reach in the frontier (we just maintain the queue by order of discovery). In UCS, we maintain a priority queue keyed by the path cost. But if all edges have cost = 1, the two approaches maintain the frontier in the same order.

Hence, BFS is simply UCS in the special case where all step costs are identical.