CBS 3004
ARTIFICIAL INTELLIGENCE
LAB ASSESMENT 1

ANISH DESAI
23BBS0119

1. Breadth First Search (BFS)

Algorithm

1. Create an empty list visited to record the traversal order.
2. Create a FIFO queue and place the start node in it. (queue = [start])
3. While the queue is **not** empty:
   i. Dequeue the front node and call it node.
   ii. Append node to visited.
   iii. Look up node's children with tree.get(node, []).
   iv. Enqueue **each** child at the end of the queue.
4. When the queue becomes empty, **return** visited.

Performance Measures (COTS)

Completeness: BFS is complete; it will always find a solution if one exists because it explores all nodes level by level.

Optimality: BFS is optimal if all edges have the same cost, as it finds the shortest path in terms of the number of edges.

Time Complexity: O(V+E)O(V + E)O(V+E), where VVV is the number of vertices (nodes) and EEE is the number of edges.In the worst case, BFS visits all nodes and edges once.

Space Complexity: O(V)O(V)O(V) due to storage of the visited list, queue, and tree structure.

Python Code

```
from collections import deque

def bfs_tree(tree, start):
    visited = []
    queue = deque([start])

    while queue:
        node = queue.popleft()
        visited.append(node)
        queue.extend(tree.get(node, []))

    return visited

tree = {}
nodes = int(input("Enter number of nodes: "))

for _ in range(nodes):
    node = input("Enter node: ")
    children = input(f"Enter children of {node} (comma-separated, leave
empty if none): ").replace(" ", "")
    if children:
        tree[node] = children.split(",")
    else:
        tree[node] = []

start_node = input("Enter starting node: ")
result = bfs_tree(tree, start_node)
print("BFS Traversal:", " → ".join(result))
```

Output

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish Desai/AppData/Lo
Enter number of nodes: 6
Enter node: A
Enter children of A (comma-separated, leave empty if none): B,C
Enter node: B
Enter children of B (comma-separated, leave empty if none): D
Enter node: C
Enter children of C (comma-separated, leave empty if none): E
Enter node: D
Enter children of D (comma-separated, leave empty if none): F
Enter node: E
Enter children of E (comma-separated, leave empty if none): F
Enter node: F
Enter children of F (comma-separated, leave empty if none):
Enter starting node: A
BFS Traversal: A → B → C → D → E → F → F
PS C:\Users\Anish Desai\Desktop\AI>
```

Observation

The BFS in your code explores nodes level-by-level, visiting all nodes at depth d before moving to depth d+1. This ensures that the first time a node is reached, it is via the shortest path in terms of the number of edges. However, for very wide trees, it can use a large amount of memory to store the frontier.

2. Depth First Search (DFS)

Algorithm

1. First, create a **queue** (a deque in the code) and add the designated start node to it. You also create an empty list called visited to store the final traversal order.
2. As long as the **queue is not empty**, continue to perform the next steps.
3. Remove the node from the **front** of the queue. Let's call this the current_node. Add this current_node to the end of your visited list.
4. Find all the children of the current_node and add them all to the **back** of the queue. This ensures that you visit all nodes at the current level before moving on to the next one.
5. Once the queue is empty, it means every reachable node has been visited. The algorithm concludes by returning the visited list.

Performance Measures (COTS)

Completeness: DFS is not complete in infinite depth spaces or when cycles exist without visited-checking. In finite graphs with visited-checking, it is complete.

• Optimality: DFS is not optimal because it may find a longer path to the goal before finding the shortest one.

• Time Complexity: O(V+E)O(V + E)O(V+E), where VVV = vertices, EEE = edges.

• Space Complexity: O(V)O(V)O(V) in recursive implementation due to the recursion stack and visited list.

Python Code

```
def dfs_tree(tree, start, visited=None):

    if visited is None:
        visited = []

    visited.append(start)

    for child in tree.get(start, []):
        if child not in visited:
            dfs_tree(tree, child, visited)

    return visited


tree = {}
nodes = int(input("Enter number of nodes: "))

for _ in range(nodes):
```

```
    node = input("Enter node: ")
    children = input(f"Enter children of {node} (comma-separated, leave
empty if none): ").replace(" ", "")
    if children:
        tree[node] = children.split(",")
    else:
        tree[node] = []

start_node = input("Enter starting node: ")
result = dfs_tree(tree, start_node)

print("DFS Traversal:", " → ".join(result))
```

Output

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish Desai/AppData/
Enter number of nodes: 6
Enter node: A
Enter children of A (comma-separated, leave empty if none): B,C
Enter node: B
Enter children of B (comma-separated, leave empty if none): D
Enter node: C
Enter children of C (comma-separated, leave empty if none): E
Enter node: D
Enter children of D (comma-separated, leave empty if none): F
Enter node: E
Enter children of E (comma-separated, leave empty if none): F
Enter node: F
Enter children of F (comma-separated, leave empty if none):
Enter starting node: A
BFS Traversal: A → B → C → D → E → F → F
PS C:\Users\Anish Desai\Desktop\AI>
```

Observation

DFS explores as far as possible along a branch before backtracking. This results in deep exploration first, which may find solutions quickly in some cases but is not guaranteed to find the shortest path. It uses less memory than BFS for wide trees but can get stuck in deep or infinite paths without visited-checking.

3. Uniform Cost Search (UCS)

Algorithm

1. First, create a **priority queue** to store paths to explore. Each item in the queue will be a tuple: (total_cost, current_node, path_so_far). The queue is automatically sorted by total_cost. Add the starting point to the priority queue with a cost of **0**, like this: (0, start_node, [start_node]). You also need a visited **set** to keep track of nodes that have already been fully explored.
2. As long as the **priority queue is not empty**, continue the loop. Inside the loop, pop the item with the **smallest cost**. This is always the most promising path to extend.
3. Check if the node from the path you just popped is the **goal**. If it is, you've successfully found the cheapest path. Return the path and its total cost. The algorithm can stop because it's guaranteed that any other path still in the queue will have a higher or equal cost.

4. If the current node is not the goal and has not already been visited, add it to the visited set. Then, for each of its neighbors that has not been visited, calculate the **new total cost** to reach that neighbor (i.e., current_path_cost + weight_of_edge). Push a new item for this neighbor onto the priority queue: (new_total_cost, neighbor_node, current_path + [neighbor_node]).
5. If the search loop finishes (meaning the priority queue becomes empty) and the goal was never reached, it signifies that there is **no path** from the start to the goal node. In this case, the function returns None.

Performance Measures (COTS)

• Completeness: UCS is complete, as it will always find a path if one exists (costs must be non-negative).

• Optimality: UCS is optimal, because it always expands the least-cost node first.

• Time Complexity: O(Elog⁡V)O(E \log V)O(ElogV) due to priority queue operations.

• Space Complexity: O(V)O(V)O(V) for storing visited nodes and queue data.

Python Code

```
import heapq

def ucs(graph, start, goal):
    if start == goal:
        return [start], 0

    queue = []
    heapq.heappush(queue, (0, start, [start]))
    visited = set()

    while queue:
        cost, node, path = heapq.heappop(queue)

        if node == goal:
            return path, cost

        if node not in visited:
            visited.add(node)
            for neighbor, weight in graph.get(node, []):
                if neighbor not in visited:
                    heapq.heappush(queue, (cost + weight, neighbor, path +
[neighbor]))

    return None, None


graph = {}
```

```
nodes = int(input("Enter number of nodes: "))

for _ in range(nodes):
    node = input("Enter node: ")
    edges = input(f"Enter edges from {node} in format neighbor:weight
(comma-separated, e.g., B:2,C:3): ").replace(" ", "")
    neighbors = []
    if edges:
        for edge in edges.split(","):
            n, w = edge.split(":")
            neighbors.append((n, int(w)))
    graph[node] = neighbors

start_node = input("Enter starting node: ")
goal_node = input("Enter goal node: ")

path, total_cost = ucs(graph, start_node, goal_node)

if path:
    print("Path found:", " → ".join(path))
    print("Total cost:", total_cost)
else:
    print("No path found.")
```

Output

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish Desai/AppData/Local/Microsoft/
Enter number of nodes: 6
Enter node: A
Enter edges from A in format neighbor:weight (comma-separated, e.g., B:2,C:3): B:1,C
Enter node: B
Enter edges from B in format neighbor:weight (comma-separated, e.g., B:2,C:3): D:2
Enter node: C
Enter edges from C in format neighbor:weight (comma-separated, e.g., B:2,C:3): E:3
Enter node: D
Enter edges from D in format neighbor:weight (comma-separated, e.g., B:2,C:3): F:2
Enter node: E
Enter edges from E in format neighbor:weight (comma-separated, e.g., B:2,C:3): F:1
Enter node: F
Enter edges from F in format neighbor:weight (comma-separated, e.g., B:2,C:3):
Enter starting node: A
Enter goal node: F
Path found: A → B → D → F
Total cost: 5
PS C:\Users\Anish Desai\Desktop\AI>
```

Observation

UCS guarantees the least cost path to the goal. Performs better than BFS in weighted graphs because it considers actual path cost. Can be slower if many paths with similar costs exist, as it explores multiple options before reaching the goal.

4. Depth Limited Search (DLS)

Algorithm

1. If visited is not provided, set visited ← [].
2. Append the current node: visited.append(current).
3. If current == target, **return** visited (goal reached).

4. If depth <= 0, **return** None (limit reached, stop exploring this path).
5. For each neighbor in graph.get(current, []):
   5.1 If neighbor is **not** in visited:
   a) Recursively call DLS on the neighbor with reduced depth:
   path ← DLS(graph, neighbor, target, depth - 1, visited.copy())
   b) If path is not None, **return** path (solution found downstream).
6. After exploring all neighbors without success, **return** None .

Performance Measures (COTS)

• Completeness: Not complete if the goal lies beyond the depth limit.

• Optimality: Not optimal, as it does not guarantee the shortest or least-cost path.

• Time Complexity: O(bl)O(b^l)O(bl) where bbb is branching factor and lll is depth limit.

• Space Complexity: O(l)O(l)O(l) due to recursion stack (linear in depth limit).

Python Code

```
def dls(graph, current, target, depth, visited=None):
    if visited is None:
        visited = []
    visited.append(current)
    if current == target:
        return visited
    if depth <= 0:
        return None
    for neighbor in graph.get(current, []):
        if neighbor not in visited:
            path = dls(graph, neighbor, target, depth - 1, visited.copy())
            if path:
                return path
    return None


graph = {}
n = int(input("Enter number of nodes: "))

for _ in range(n):
    node = input("Enter node: ")
    children = input(f"Enter children of {node} (comma-separated, leave
empty if none): ").replace(" ", "")
    if children:
        graph[node] = children.split(",")
    else:
```

```
        graph[node] = []

start = input("Enter starting node: ")
goal = input("Enter goal node: ")
limit = int(input("Enter depth limit: "))

result = dls(graph, start, goal, limit)

if result:
    print("Path found:", " → ".join(result))
else:
    print("No path found within depth limit")
```

Output

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish Desai/AppData/Loc
Enter number of nodes: 6
Enter node: A
Enter children of A (comma-separated, leave empty if none): B,C
Enter node: B
Enter children of B (comma-separated, leave empty if none): D
Enter node: C
Enter children of C (comma-separated, leave empty if none): E
Enter node: D
Enter children of D (comma-separated, leave empty if none): F
Enter node: E
Enter children of E (comma-separated, leave empty if none): F
Enter node: F
Enter children of F (comma-separated, leave empty if none):
Enter starting node: A
Enter goal node: F
Enter depth limit: 3
Path found: A → B → D → F
PS C:\Users\Anish Desai\Desktop\AI>
```

Observation

Works well when maximum depth of solution is known in advance. Avoids infinite loops in infinite search spaces. Risk of missing solution if depth limit is set too low.

5. Iterative Deepening Search (IDS)

Algorithm

1. If current_node == goal, return [current_node].
2. If depth == 0, return None (limit reached).
3. Mark current_node as visited on this path (e.g., append to visited).
4. For each child in tree.get(current_node, []):
   4.1 If child is not in visited:
      a) Call path = DLS(tree, child, goal, depth - 1, visited.copy()).
      b) If path is not None, return [current_node] + path.
5. After all children explored without success, return None.

Performance Measures (COTS)

• Completeness: Yes (if maximum depth limit ≥ depth of goal).

• Optimality: Yes (if path cost = depth).

• Time Complexity: O(bd)O(b^d)O(bd) where bbb is branching factor, ddd is depth of goal.

• Space Complexity: O(d)O(d)O(d) due to recursion stack.

Python Code

```
def dls(graph, current, target, depth, visited):
    if current == target:
        return [current]
    if depth <= 0:
        return None
    visited.append(current)
    for neighbor in graph.get(current, []):
        if neighbor not in visited:
            path = dls(graph, neighbor, target, depth - 1, visited.copy())
            if path:
                return [current] + path
    return None

def ids(graph, start, target, max_depth):
    for depth in range(max_depth + 1):
```

```
        path = dls(graph, start, target, depth, [])
        if path:
            return path
    return None


graph = {}
n = int(input("Enter number of nodes: "))

for _ in range(n):
    node = input("Enter node: ")
    children = input(f"Enter children of {node} (comma-separated, leave
empty if none): ").replace(" ", "")
    if children:
        graph[node] = children.split(",")
    else:
        graph[node] = []

start_node = input("Enter starting node: ")
goal_node = input("Enter goal node: ")
limit = int(input("Enter max depth limit: "))

result = ids(graph, start_node, goal_node, limit)

if result:
    print("Path found:", " → ".join(result))
else:
    print("No path found within depth limit")
```

Output

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish Desai/AppDat
Enter number of nodes: 6
Enter node: A
Enter children of A (comma-separated, leave empty if none): B,C
Enter node: B
Enter children of B (comma-separated, leave empty if none): D
Enter node: C
Enter children of C (comma-separated, leave empty if none): E
Enter node: D
Enter children of D (comma-separated, leave empty if none): F
Enter node: E
Enter children of E (comma-separated, leave empty if none): F
Enter node: F
Enter children of F (comma-separated, leave empty if none):
Enter starting node: A
Enter goal node: F
Enter max depth limit: 3
Path found: A → B → D → F
PS C:\Users\Anish Desai\Desktop\AI>
```

Observation

Combines benefits of BFS (completeness, optimality) and DFS (low memory use). Re-explores nodes multiple times, so time cost is slightly higher than BFS.

6. Bi-directional Search (BDS)

Algorithm

1. First, two separate searches are initialized. One starts at the **source** node and the other at the **goal** node. Each search maintains its own **queue** for nodes to visit and its

own **visited set** to keep track of the nodes it has already explored.

2. The algorithm then enters a loop that continues as long as both the forward search queue (from the source) and the backward search queue (from the goal) are not empty. The two searches effectively take turns expanding.
3. The search starting from the **source** takes one step. It dequeues a node, finds all its neighbors, and for each neighbor it hasn't seen before, it adds that neighbor to its own queue and its own visited set.
4. After processing each new neighbor, the forward search performs a critical check: it looks to see if this new neighbor already exists in the **visited set** of the backward search (the one coming from the goal). If it does, the two search frontiers have met. The search is successful, and this meeting node is returned.
5. If the forward search did not find a meeting point, the search starting from the **goal** takes its turn. It performs the exact same process in reverse: it dequeues a node, explores its neighbors, and adds them to its own queue and visited set.
6. Similarly, after visiting a new neighbor, the backward search checks if this node has already been visited by the forward search. If it has, a **meeting point** has been

found, and the search concludes successfully by returning this node.

7. If the main loop finishes because one of the queues has become empty, it means one of the search frontiers has been fully explored without ever intersecting the other. This signifies that there is **no path** between the source and the goal.

Performance Measures (COTS)

• Completeness: Yes (if the graph is finite and both sides are explored).

• Optimality: Yes (if each step has equal cost).

• Time Complexity: O(bd/2)O(b^{d/2})O(bd/2) — much faster than BFS when bbb is large.

• Space Complexity: O(bd/2)O(b^{d/2})O(bd/2) due to storage of two frontiers.

Python Code

```
from collections import deque

def bfs_step(graph, queue, visited, other_side_visited):
    if queue:
        current = queue.popleft()
        for neighbor in graph.get(current, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                if neighbor in other_side_visited:
                    return neighbor
    return None


def bidirectional_search(graph, source, target):
    if source == target:
        return [source]

    visited_from_start = set([source])
    visited_from_goal = set([target])
    queue_start = deque([source])
    queue_goal = deque([target])

    while queue_start and queue_goal:
        meet_point = bfs_step(graph, queue_start, visited_from_start,
visited_from_goal)
        if meet_point:
            return visited_from_start, visited_from_goal, meet_point

        meet_point = bfs_step(graph, queue_goal, visited_from_goal,
visited_from_start)
        if meet_point:
            return visited_from_start, visited_from_goal, meet_point
```

```
    return None, None, None


graph = {}
n = int(input("Enter number of nodes: "))

for _ in range(n):
    node = input("Enter node: ")
    neighbors = input(f"Enter neighbors of {node} (comma-separated):
").replace(" ", "").split(",")
    graph[node] = neighbors

start_node = input("Enter starting node: ")
goal_node = input("Enter goal node: ")

visited_start, visited_goal, meeting_point =
bidirectional_search(graph, start_node, goal_node)

if meeting_point:
    print(f"Search met at node: {meeting_point}")
    print("Start side visited:", " → ".join(visited_start))
    print("Goal side visited:", " → ".join(visited_goal))
else:
    print("No path found.")
```

Output

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish Desai/AppData/Local/Microso
Enter number of nodes: 6
Enter node: A
Enter neighbors of A (comma-separated): B:1,C:4
Enter node: B
Enter neighbors of B (comma-separated): D:2
Enter node: C
Enter neighbors of C (comma-separated): E:3
Enter node: D
Enter neighbors of D (comma-separated): F:2
Enter node: E
Enter neighbors of E (comma-separated): F:1
Enter node: F
Enter neighbors of F (comma-separated):
Enter starting node: A
Enter goal node: F
No path found.
PS C:\Users\Anish Desai\Desktop\AI>
```

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish Desai/App
AI/code1.py"
Enter number of nodes: 6
Enter node: A
Enter neighbors of A (comma-separated): B,C
Enter node: B
Enter neighbors of B (comma-separated): A,D
Enter node: C
Enter neighbors of C (comma-separated): A,E
Enter node: D
Enter neighbors of D (comma-separated): B,F
Enter node: E
Enter neighbors of E (comma-separated): C,F
Enter node: F
Enter neighbors of F (comma-separated): D,E
Enter starting node: A
Enter goal node: F
Search met at node: D
Start side visited: A → B → D → C
Goal side visited: F → E → D
PS C:\Users\Anish Desai\Desktop\AI>
```

Observation

Expands nodes from both ends, meeting in the middle. Significantly reduces time when the search space is large. Needs additional memory to store visited nodes from both directions.

7. Greedy Best First Search (GBFS)

Algorithm

1. Initialize an **open list** (priority queue) and insert (h[start], start).
2. Initialize an empty set visited to record explored nodes.
3. Initialize a dictionary parent with parent[start] = None.
4. While the open list is not empty:
   4.1 Remove from the open list the node with the smallest heuristic value; call it current.
   4.2 If current == goal:
      a) Reconstruct the path by following parent links from goal back to start.
      b) Reverse this path and return it.
   4.3 Add current to visited.
   4.4 For each neighbor of current in graph[current]:
      a) If neighbor is not in visited:
         — Set parent[neighbor] = current.
         — Add (h[neighbor], neighbor) to the open list.
5. If the open list becomes empty without finding the goal, return None.

Performance Measures (COTS)

• Completeness: No (can get stuck in loops if not tracking visited nodes).

• Optimality: No (chooses node with lowest h(n)h(n)h(n), not considering path cost).

• Time Complexity: O(bm)O(b^m)O(bm) worst case.

• Space Complexity: O(bm)O(b^m)O(bm) — stores nodes in priority queue.

Python Code

```
import heapq

def greedy_best_first_search(graph, start, target, h):
    open_nodes = []
    heapq.heappush(open_nodes, (h[start], start))
    visited = set()
    parent = {start: None}
```

```
    while open_nodes:
        _, current = heapq.heappop(open_nodes)

        if current == target:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]

        visited.add(current)

        for neighbor in graph.get(current, []):
            if neighbor not in visited:
                parent[neighbor] = current
                heapq.heappush(open_nodes, (h[neighbor], neighbor))

    return None


graph = {}
n = int(input("Enter number of nodes: "))

for _ in range(n):
    node = input("Enter node: ")
    neighbors = input(f"Enter neighbors of {node} (comma-separated):
").replace(" ", "").split(",")
    graph[node] = neighbors

heuristic = {}
print("Enter heuristic values for each node:")
for node in graph:
    heuristic[node] = int(input(f"h({node}): "))

start_node = input("Enter starting node: ")
```

```
goal_node = input("Enter goal node: ")

path = greedy_best_first_search(graph, start_node, goal_node,
heuristic)

if path:
    print("Path found:", " → ".join(path))
else:
    print("No path found.")
```

Output

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish
Enter number of nodes: 6
Enter node: A
Enter neighbors of A (comma-separated): B,C
Enter node: B
Enter neighbors of B (comma-separated): D
Enter node: C
Enter neighbors of C (comma-separated): E
Enter node: D
Enter neighbors of D (comma-separated): F
Enter node: E
Enter neighbors of E (comma-separated): F
Enter node: F
Enter neighbors of F (comma-separated):
Enter heuristic values for each node:
h(A): 7
h(B): 6
h(C): 2
h(D): 5
h(E): 3
h(F): 0
Enter starting node: A
Enter goal node: F
Path found: A → C → E → F
PS C:\Users\Anish Desai\Desktop\AI>
```

Observation

Always chooses the node with the lowest heuristic estimate to the goal. Fast for some cases but can be misled by poor heuristics. Not guaranteed to find the shortest path.

8. A* Search

Algorithm

1. Initialize an empty priority queue open and push (f=start_f, g=0, start) where start_f = h[start].
2. Initialize closed = set() (explored nodes).
3. Initialize parent = {start: None} and g_cost = {start: 0}.
4. While open is not empty:
   4.1 Pop the entry with the smallest f from open; let it be (f, g, current).
   4.2 If current == goal:
      a) Reconstruct the path by following parent from goal back to start and return the reversed path.
   4.3 Add current to closed.
   4.4 For each (neighbor, step_cost) in graph.get(current, []):
      a) tentative_g = g_cost[current] + step_cost.
      b) If neighbor in closed and tentative_g >= g_cost.get(neighbor, +∞), skip this neighbor.
      c) If tentative_g < g_cost.get(neighbor, +∞) (better path found):
         i. Set parent[neighbor] = current.
         ii. Set g_cost[neighbor] = tentative_g.
         iii. Compute f_cost = tentative_g + h[neighbor].
         iv. Push (f_cost, tentative_g, neighbor) into open.

5. If the loop ends without finding goal, return None.

Performance Measures (COTS)

• Completeness: Yes (if h(n)h(n)h(n) is admissible and graph is finite).

• Optimality: Yes (if h(n)h(n)h(n) is admissible and consistent).

• Time Complexity: O(bd)O(b^d)O(bd) worst case.

• Space Complexity: O(bd)O(b^d)O(bd) — stores all nodes in memory

Python Code

```
import heapq

def a_star_search(graph, source, target, h):
    open_list = []
    heapq.heappush(open_list, (h[source], 0, source))
    visited = set()
```

```
    parent = {source: None}
    g_cost = {source: 0}

    while open_list:
        f, g, current = heapq.heappop(open_list)

        if current == target:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]

        visited.add(current)

        for neighbor, cost in graph.get(current, []):
            temp_g = g_cost[current] + cost
            if neighbor not in visited or temp_g < g_cost.get(neighbor,
float('inf')):
                parent[neighbor] = current
                g_cost[neighbor] = temp_g
                f_cost = temp_g + h[neighbor]
                heapq.heappush(open_list, (f_cost, temp_g, neighbor))

    return None


graph = {}
n = int(input("Enter number of nodes: "))

for _ in range(n):
    node = input("Enter node: ")
    edges = input(f"Enter neighbors of {node} with costs (format: B:1,C:3):
").replace(" ", "").split(",")
    graph[node] = [(e.split(":")[0], int(e.split(":")[1])) for e in edges if e]
```

```
heuristic = {}
print("Enter heuristic values for each node:")
for node in graph:
    heuristic[node] = int(input(f"h({node}): "))

start_node = input("Enter starting node: ")
goal_node = input("Enter goal node: ")

path = a_star_search(graph, start_node, goal_node, heuristic)

if path:
    print("Path found:", " → ".join(path))
else:
    print("No path found.")
```

Output

```
PS C:\Users\Anish Desai\Desktop\AI> & "C:/Users/Anish Desai/AppData/Loca
AI/code1.py"
Enter number of nodes: 6
Enter node: A
Enter neighbors of A with costs (format: B:1,C:3): B:1,C:4
Enter node: B
Enter neighbors of B with costs (format: B:1,C:3): C:2,D:5
Enter node: C
Enter neighbors of C with costs (format: B:1,C:3): D:1,E:3
Enter node: D
Enter neighbors of D with costs (format: B:1,C:3): F:2
Enter node: E
Enter neighbors of E with costs (format: B:1,C:3): F:1
Enter node: F
Enter neighbors of F with costs (format: B:1,C:3):
Enter heuristic values for each node:
h(A): 7
h(B): 6
h(C): 4
h(D): 2
h(E): 3
h(F): 0
Enter starting node: A
Enter goal node: F
Path found: A → B → C → D → F
PS C:\Users\Anish Desai\Desktop\AI>
```

Observation

Combines uniform cost search and greedy best-first search by using $f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)$. Guarantees optimal path when heuristic is admissible. More efficient than Uniform Cost Search if the heuristic is good.