CS571 -ARTIFICIALINTELLIGENCELAB

ASSIGNMENT-3:UCS IDS

Intaj Choudhury – 2211MC09
Ankit Anand – 2311MC04
Khushbu Bharti – 2311MC21

Question
The task is to check if we can reach from any random start grid to the mentioned target grid by moving the Blank space ('B'). In one step, the Blank space can move either top or down or left or right. It can′t move to the already occupied square.

1. Write a program to find the goal state from the given starting state using Uniform Cost Search (UCS) and Iterative Deepening Search (IDS)

Solution

1. **Algorithm:**

Uniform Cost Search -

1. Initialize an empty priority queue (usually implemented as a min-heap) called open_list.
2. Create a set called closed_set to keep track of visited nodes.
3. Insert the start node into the open_list with a cost of 0.
4. While the open_list is not empty: a. Remove the node with the lowest cost from the open_list. This is the current node to be expanded. b. If the current node is the goal node, the algorithm terminates, and the path is found. c. Otherwise, add the current node to the closed_set to mark it as visited. d. For each neighbour of the current node that has not been visited: i. If the neighbour is not in the open_list, add it with the cost equal to the cumulative cost to reach the current node plus the cost of the edge to the neighbour. ii. If the neighbour is already in the open_list with a higher cost, update its cost in the open_list. e. Repeat steps 4a through 4d.

The order followed in bfs, dfs, ucs and ids is :
Right, Left, Up, Down.

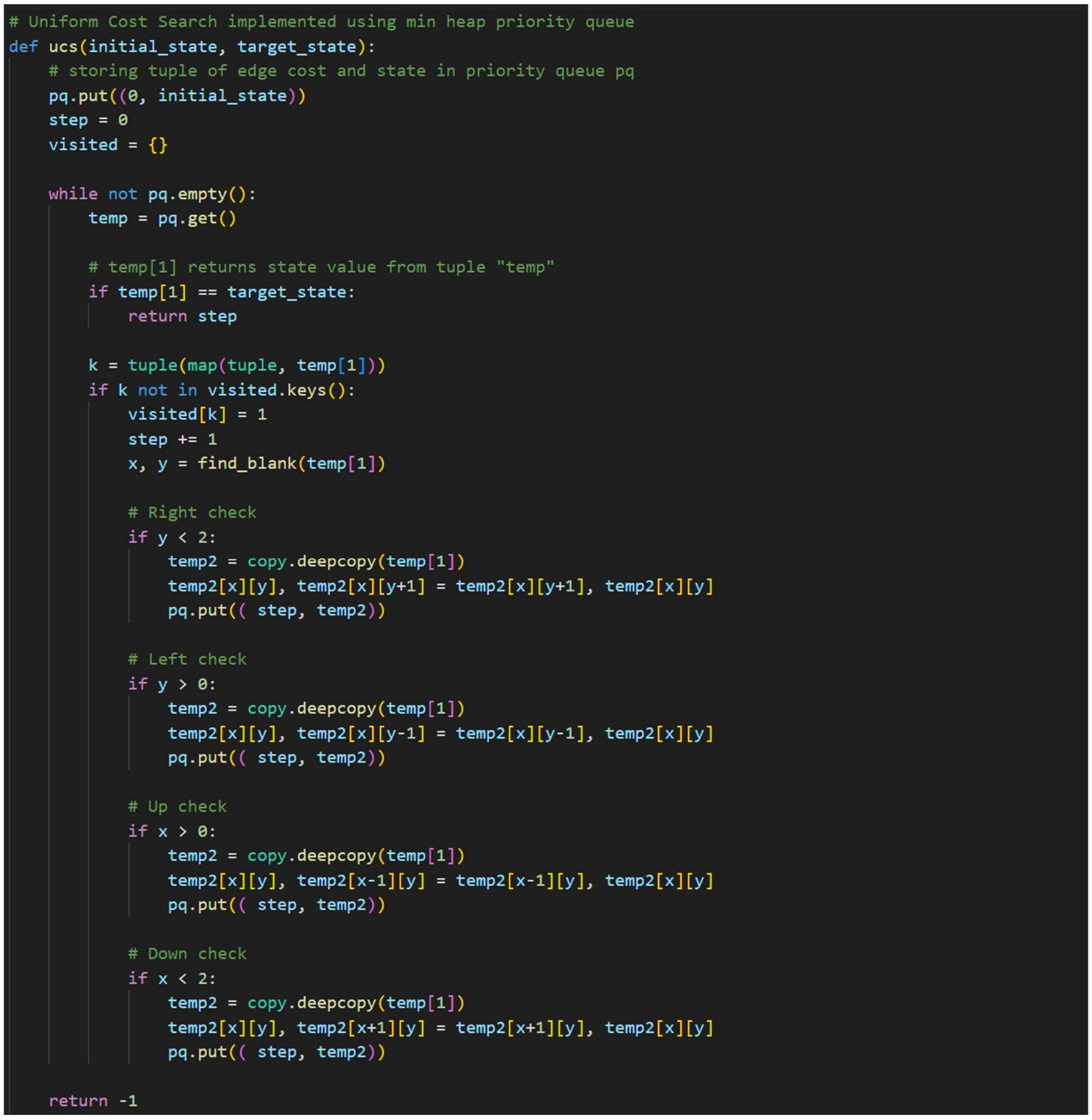
Iterative Deepening Depth-First Search (IDDFS) -

1. Set the initial depth limit to 0.
2. Enter a loop that continues until a solution is found or all depths are exhausted: a. Initialize an empty dictionary called "visited" to keep track of visited nodes. b. Call the Depth-Limited DFS function with the root node, goal node, current depth limit, and the "visited" dictionary. c. If the Depth-Limited DFS function returns a non-null result (a path), exit the loop. d. Increment the depth limit by 1.
3. Depth-Limited DFS function: a. Accept the current node, goal node, current depth limit, and the "visited" dictionary as parameters. b. If the depth limit is less than 0, return null (backtrack due to depth limit reached). c. If the current node is the goal node, return a path containing only the goal node. d. If the current node is already visited, return null (backtrack to avoid cycles). e. Mark the current node as visited in the "visited" dictionary. f. Loop through each neighbor of the current node: i. Recursively call the Depth-Limited DFS function with the neighbor, goal node, depth limit - 1, and the "visited" dictionary. ii. If a non-null path is returned from the recursive call: - Append the current node to the beginning of the path. - Return the path. g. If no path to the goal is found from the current node, return null.
4. In the main program: a. Obtain the start node and the goal node. b. Call the IDDFS function with the start node and goal node. c. If a non-null path is returned from the IDDFS function: i. Process and use the path for the solution. d. If the IDDFS function returns null for all depth limits: i. Conclude that the goal is not reachable from the start node.

2. Assume the cost of the edge between any two nodes at a given level are identical and equal to 1 (eg., the cost of edge between node at level n and node at level n+1 is 1 but the cost of edge between node at level n and n+2 is 2 as the node at level n+2 can be reached by traversing nodes at n+ level).

Solution

Uniform Cost Search Algorithm is being implemented using the given edge cost assumption



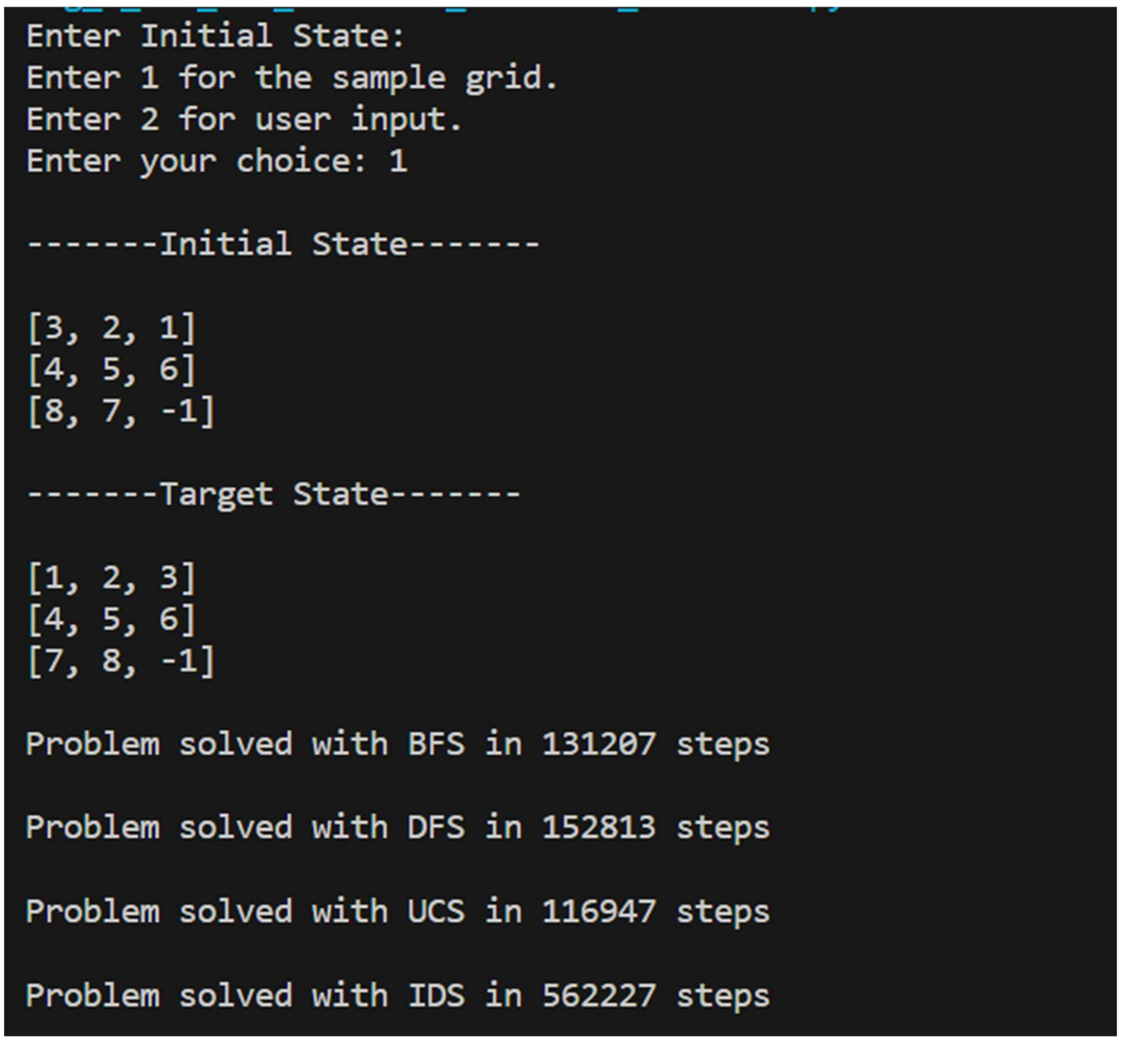
```
# Uniform Cost Search implemented using min heap priority queue
def ucs(initial_state, target_state):
    # storing tuple of edge cost and state in priority queue pq
    pq.put((0, initial_state))
    step = 0
    visited = {}

    while not pq.empty():
        temp = pq.get()

        # temp[1] returns state value from tuple "temp"
        if temp[1] == target_state:
            return step

        k = tuple(map(tuple, temp[1]))
        if k not in visited.keys():
            visited[k] = 1
            step += 1
            x, y = find_blank(temp[1])

            # Right check
            if y < 2:
                temp2 = copy.deepcopy(temp[1])
                temp2[x][y], temp2[x][y+1] = temp2[x][y+1], temp2[x][y]
                pq.put(( step, temp2))

            # Left check
            if y > 0:
                temp2 = copy.deepcopy(temp[1])
                temp2[x][y], temp2[x][y-1] = temp2[x][y-1], temp2[x][y]
                pq.put(( step, temp2))

            # Up check
            if x > 0:
                temp2 = copy.deepcopy(temp[1])
                temp2[x][y], temp2[x-1][y] = temp2[x-1][y], temp2[x][y]
                pq.put(( step, temp2))

            # Down check
            if x < 2:
                temp2 = copy.deepcopy(temp[1])
                temp2[x][y], temp2[x+1][y] = temp2[x+1][y], temp2[x][y]
                pq.put(( step, temp2))

    return -1
```

Target :
[1, 2, 3]
[4, 5, 6]
[7, 8, -1]

Sample Input :
[3, 2, 1],
[4, 5, 6],
[8, 7, -1]

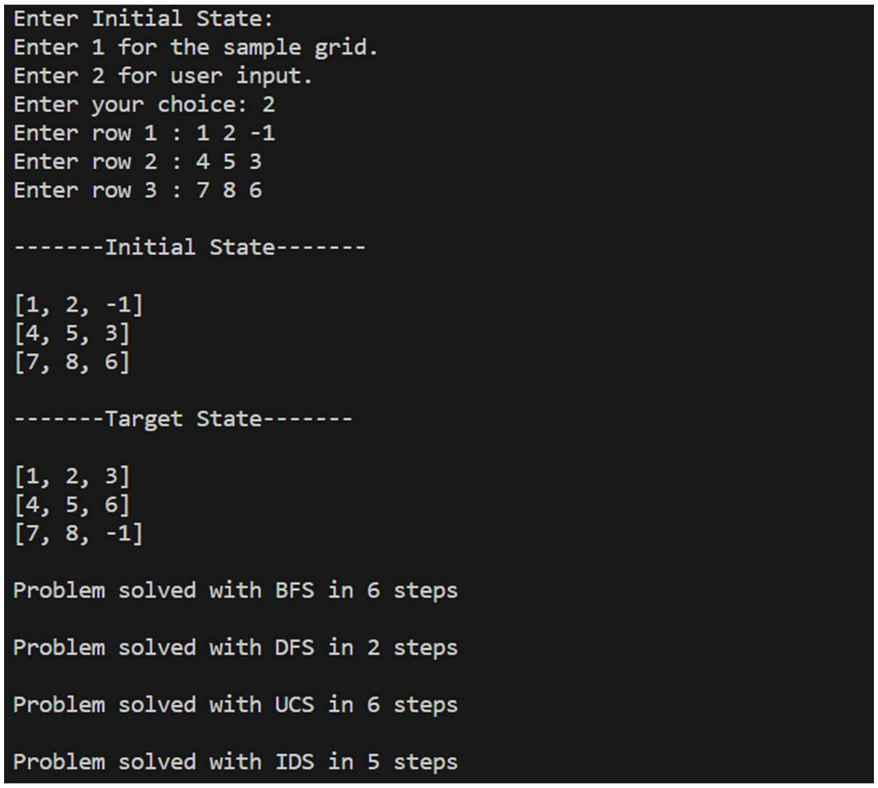
3. Compare the UCS and IDS with BFS and DFS (from Assignment-1). Report the no. of steps each algorithm took to reach the goal node and which algorithm is optimal.

Case1: Sample grid is taken as the initial state



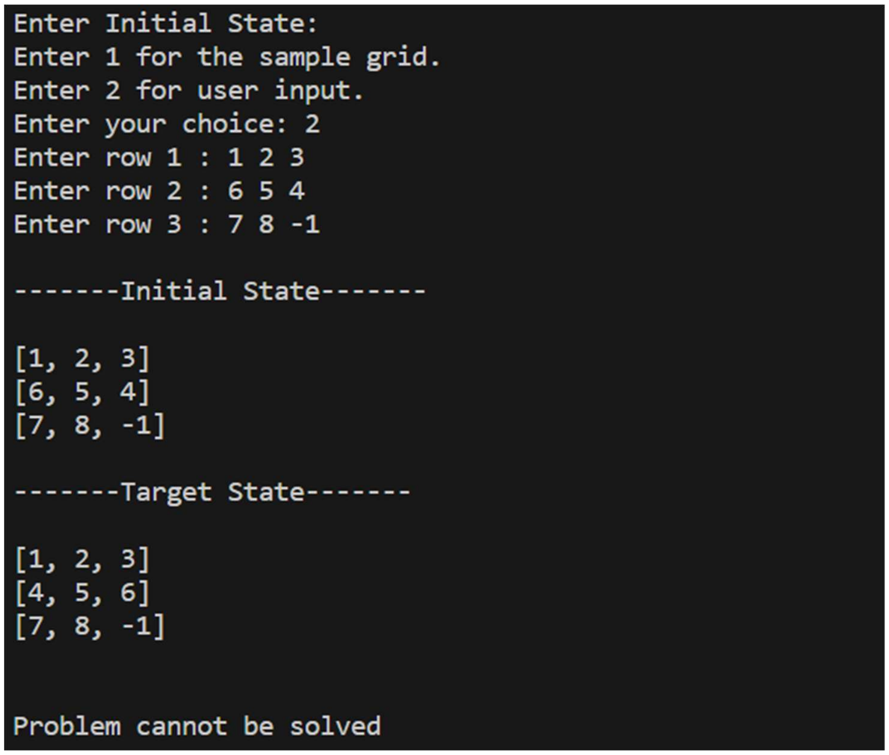
```
Enter Initial State:
Enter 1 for the sample grid.
Enter 2 for user input.
Enter your choice: 1

-------Initial State-------

[3, 2, 1]
[4, 5, 6]
[8, 7, -1]

-------Target State-------

[1, 2, 3]
[4, 5, 6]
[7, 8, -1]

Problem solved with BFS in 131207 steps

Problem solved with DFS in 152813 steps

Problem solved with UCS in 116947 steps

Problem solved with IDS in 562227 steps
```

Case2: Random input grid



```
Enter Initial State:
Enter 1 for the sample grid.
Enter 2 for user input.
Enter your choice: 2
Enter row 1 : 1 2 -1
Enter row 2 : 4 5 3
Enter row 3 : 7 8 6

-------Initial State-------

[1, 2, -1]
[4, 5, 3]
[7, 8, 6]

-------Target State-------

[1, 2, 3]
[4, 5, 6]
[7, 8, -1]

Problem solved with BFS in 6 steps

Problem solved with DFS in 2 steps

Problem solved with UCS in 6 steps

Problem solved with IDS in 5 steps
```

Case3: When puzzle is not solvable.



```
Enter Initial State:
Enter 1 for the sample grid.
Enter 2 for user input.
Enter your choice: 2
Enter row 1 : 1 2 3
Enter row 2 : 6 5 4
Enter row 3 : 7 8 -1

-------Initial State-------

[1, 2, 3]
[6, 5, 4]
[7, 8, -1]

-------Target State-------

[1, 2, 3]
[4, 5, 6]
[7, 8, -1]


Problem cannot be solved
```

In summary, Uniform Cost Search is better than Breadth-First Search when:

1. **Edge Costs Vary:** If the graph has varying edge costs and you want to find the path with the lowest accumulated cost, UCS is the preferred choice.
2. **Weighted Graphs:** When working with weighted graphs (positive edge weights), UCS is more appropriate since it takes edge costs into account.

Breadth-First Search, on the other hand, is generally preferred when:

1. **Unweighted Graphs:** For unweighted graphs or graphs where all edge weights are equal, BFS is a simpler and more efficient choice.
2. **Shortest Path by Number of Edges:** If you are interested in finding the shortest path in terms of the number of edges, BFS guarantees finding the shortest such path.

Uniform Cost Search is generally better than Depth-First Search when:

1. **Optimal Path Finding:** If you need to find the optimal path (i.e., the path with the lowest cumulative cost) in terms of edge weights, UCS is preferable. UCS explores paths with lower costs first, ensuring that the first solution found is the one with the minimum cost.
2. **Non-Uniform Edge Costs:** When the graph has varying edge costs and you need to find the path with the least total cost, UCS is more appropriate. It takes into account the weights of edges, ensuring that the lowest-cost paths are explored first.
3. **Completeness and Optimality:** UCS is both complete (will find a solution if one exists) and optimal (will find the least-cost solution) as long as edge costs are non-negative. DFS, on the other hand, might not be optimal or complete, especially if the search space is large or infinite.

Depth-First Search might be preferred in specific situations such as:

1. **Memory Efficiency:** DFS uses less memory compared to UCS since it only needs to store the path to the current node, while UCS needs to store the entire path to every node along with their associated costs.
2. **Exploring Deeply:** If the search space is vast and the solution is likely to be deep within the tree or graph, DFS might reach the solution faster since it explores deeper before backtracking.
3. **Non-Optimal Solutions:** If you're not interested in finding the optimal solution in terms of cost, and just need any valid solution quickly, DFS might be more suitable.

Iterative Deepening Depth-First Search is better than Breadth-First Search when:

1. **Memory Efficiency:** IDDFS is memory efficient compared to BFS. It only keeps track of the current path and doesn't require storing the entire level of nodes in memory, making it more suitable for large search spaces.
2. **Branching Factor:** If the branching factor (the number of child nodes per node) is high, IDDFS can be more efficient in terms of memory usage compared to BFS. BFS needs to store all nodes at the current level in memory, which can quickly become a problem with a high branching factor.
3. **Complete and Optimal:** Similar to BFS, IDDFS is complete (will find a solution if one exists) and can find an optimal solution as long as edge costs are uniform and non-negative.
4. **Depth-Limited Exploration:** IDDFS explores nodes in a depth-limited manner, incrementing the depth limit with each iteration. This can be useful when you have some intuition about the approximate depth of the solution but don't want to commit to exploring the entire breadth of the search space like BFS does.

Breadth-First Search might still be preferable in situations where:

1. **Shortest Path by Number of Edges:** If you need to find the shortest path in terms of the number of edges, BFS guarantees finding the shortest such path. IDDFS doesn't guarantee finding the shortest path unless you implement additional checks.
2. **Optimal Solution in Terms of Edge Weights:** If you need the optimal solution considering varying edge costs, Uniform Cost Search or A* might be more appropriate than either BFS or IDDFS.
3. **Small Branching Factor:** If the branching factor is small and memory constraints are not a concern, BFS can be more efficient in terms of search time since it doesn't require repeated exploration of the same nodes at different depths.

IDDFS might be a better choice than DFS in the following situations:

1. **Limited Memory:** IDDFS uses only a small amount of memory, comparable to traditional DFS. This makes it suitable for cases where memory is limited or you're dealing with large search spaces.
2. **Unknown Depth Limit:** If you don't have prior knowledge about the depth of the solution or the optimal depth for searching, IDDFS can be a more adaptive choice. It allows you to perform depth-limited search iteratively, gradually increasing the depth limit until a solution is found.
3. **Deep Solutions:** IDDFS can be more efficient when searching for solutions that are deep in the search tree. Traditional DFS might get stuck exploring a deep branch of the tree before finding a solution, while IDDFS explores deeper levels in each iteration.
4. **Completeness and Optimality:** IDDFS is complete (will find a solution if one exists) and can find an optimal solution if edge costs are uniform and non-negative, similar to traditional DFS. This is in contrast to BFS, which might require a lot of memory and time for deeper search spaces.

However, it's important to note that IDDFS also inherits some of the disadvantages of traditional DFS, such as not being optimal or complete if the graph has cycles or negative edge weights.