Lab Week 03



**Session: 2022 – 2026**

**Submitted by:**

Abdur Rehman Kazim 2022-CS-115

**Supervised by:**

Sir Samyan Qayuum

Department of Computer Science

**University of Engineering and Technology**

**Lahore, Pakistan**

Contents

[**Searching Algorithms using Best First Search Framework:** 3](#_Toc177914272)

[**Breadth First Search** 3](#_Toc177914273)

[**Uniform Cost Search** 3](#_Toc177914274)

[**Depth First Search** 3](#_Toc177914275)

[**Iterative Deepening Search** 4](#_Toc177914276)

[**Problems:** 5](#_Toc177914277)

[**Missionaries and Cannibals** 5](#_Toc177914278)

[**8-Puzzle** 7](#_Toc177914279)

[**Maze Solver** 9](#_Toc177914280)

[**Water Jug** 11](#_Toc177914281)

[**Robot Navigator** 12](#_Toc177914282)

**Searching Algorithms using Best First Search Framework:**

problem = TransportationProblem(N=10)

**Breadth First Search**

Code:

bfs\_search = BestFirstSearch(problem, lambda state, cost: 0)

print(bfs\_search.search())

Output:

([1, 2, 4, 5, 10], 6)

**Uniform Cost Search**

Code:

ucs\_search = BestFirstSearch(problem, lambda state, cost: cost)

print(ucs\_search.search())

Output:

([1, 2, 4, 5, 10], 6)

**Depth First Search**

Code:

dfs\_search = BestFirstSearch(problem, lambda state, cost: -cost)

print(dfs\_search.search())

Output:

([1, 2, 4, 8, 9, 10], 7) Here the output shows dfs was not optimal.

**Iterative Deepening Search**

Class:

class IterativeDeepeningSearch:

    def \_\_init\_\_(self, problem):

        self.problem = problem

    def search(self):

        depthLimit = 0

        while True:

            result = self.depthLimitedSearch(depthLimit)

            if result is not None:

                return result

            depthLimit += 1

    def depthLimitedSearch(self, depthLimit):

        def ids\_priority(state, depth):

            return -depth if depth <= depthLimit else float('inf')

        search = BestFirstSearch(self.problem, ids\_priority)

        return search.search()

ids\_search = IterativeDeepeningSearch(problem)

print(ids\_search.search())

Output:

([1, 2, 4, 5, 10], 6)

**Problems:**

## **Missionaries and Cannibals**

Class:

class MissionariesAndCannibals:

    def \_\_init\_\_(self):

        self.graph = Graph()

        self.\_build\_graph()

    def \_build\_graph(self):

        visited = set()

        frontier = [self.start\_state()]

        actions = [[1, 0], [0, 1], [1, 1], [2, 0], [0, 2]]

        while frontier:

            state = frontier.pop()

            if state in visited:

                continue

            visited.add(state)

            self.graph.add\_node(state=state)

            m\_left, c\_left, boat = state

            for m, c in actions:

                if boat:    # boat at the right bank of the river

                    newState = (m\_left + m, c\_left + c, 0)

                else:       # at the left bank

                    newState = (m\_left - m, c\_left - c, 1)

                if self.validState(newState):

                    self.graph.add\_edge(state, f"{m}M and {c}C", 1, newState)

                    if newState not in visited:

                        frontier.append(newState)

    def start\_state(self):

        return (3, 3, 0)

    def is\_end(self, state):

        return state == (0, 0, 1)

    def validState(self, state):

        m\_left, c\_left, boat = state

        m\_right, c\_right = 3 - m\_left, 3 - c\_left

        return (0 <= m\_left <= 3) and (0 <= c\_left <= 3) and (0 <= m\_right <= 3) and (0 <= c\_right <= 3) and (m\_left == 0 or m\_left >= c\_left) and (m\_right == 0 or m\_right >= c\_right)

    def successors(self, state):

        node = self.graph.get\_node(state=state)

        return [(edge.action, edge.next\_node.state, edge.cost) for edge in node.edges]

problem = MissionariesAndCannibals()

BFS:

bfs\_search = BestFirstSearch(problem, lambda state, cost: 0)

print(bfs\_search.search())

Output:

([(3, 3, 0), (2, 2, 1), (3, 2, 0), (3, 0, 1), (3, 1, 0), (1, 1, 1), (2, 2, 0), (0, 2, 1), (0, 3, 0), (0, 1, 1), (0, 2, 0), (0, 0, 1)], 11)

Explanation:

State contains the number of missionaries on the left bank of the river followed by number of cannibals on the left bank and finally the position of the boat. 1 indicates right bank and 0 indicates left bank.

The output shows us a way the problem can be solved with each step indicating the missionaries and cannibals count on the left bank and boat position. The problem could be solved in minimum 11 steps.

DFS:

dfs\_search = BestFirstSearch(problem, lambda state, cost: -cost)

print(dfs\_search.search())

Output:

([(3, 3, 0), (2, 2, 1), (3, 2, 0), (3, 0, 1), (3, 1, 0), (1, 1, 1), (2, 2, 0), (0, 2, 1), (0, 3, 0), (0, 1, 1), (0, 2, 0), (0, 0, 1)], 11)

UCS:

ucs\_search = BestFirstSearch(problem, lambda state, cost: cost)

print(ucs\_search.search())

Output:

([(3, 3, 0), (2, 2, 1), (3, 2, 0), (3, 0, 1), (3, 1, 0), (1, 1, 1), (2, 2, 0), (0, 2, 1), (0, 3, 0), (0, 1, 1), (0, 2, 0), (0, 0, 1)], 11)

IDS:

ids\_search = IterativeDeepeningSearch(problem)

print(ids\_search.search())

Output:

([(3, 3, 0), (2, 2, 1), (3, 2, 0), (3, 0, 1), (3, 1, 0), (1, 1, 1), (2, 2, 0), (0, 2, 1), (0, 3, 0), (0, 1, 1), (0, 2, 0), (0, 0, 1)], 11)

## **8-Puzzle**

Class:

class Puzzle:

    def \_\_init\_\_(self, initial\_state):

        self.initial\_state = initial\_state

        self.graph = Graph()

        self.visited = set()

    def zero(self, state):

        for i, row in enumerate(state):

            for j, value in enumerate(row):

                if value == 0:

                    return i, j

    def start\_state(self):

        return self.initial\_state

    def is\_end(self, state):

        return state == ((1, 2, 3), (4, 5, 6), (7, 8, 0))

    def successors(self, state):

        # node = self.graph.get\_node(state)

        # return [(edge.action, edge.next\_node.state, edge.cost) for edge in node.edges]

        directions = [[1, 0, "down"], [0, 1, "right"], [-1, 0, "up"], [0, -1, "left"]]

        r, c = self.zero(state)

        successors = []

        for dr, dc, action in directions:

            new\_r, new\_c = r + dr, c + dc

            if 0 <= new\_r < 3 and 0 <= new\_c < 3:

                new\_state = [list(row) for row in state]

                new\_state[r][c], new\_state[new\_r][new\_c] = new\_state[new\_r][new\_c], new\_state[r][c]

                new\_state = tuple(tuple(row) for row in new\_state)

                if new\_state not in self.visited:

                    self.visited.add(new\_state)

                    successors.append((action, new\_state, 1))

        return successors

problem = Puzzle(((1, 4, 3), (5, 8, 6), (7, 2, 0)))

BFS:

bfs\_search = BestFirstSearch(problem, lambda state, cost: 0)

print(bfs\_search.search())

Output: None

Not working. Mainly due to such a large state space

DFS:

dfs\_search = BestFirstSearch(problem, lambda state, cost: -cost)

print(dfs\_search.search())

Output: None

Not working. Mainly due to such a large state space

UCS:

ucs\_search = BestFirstSearch(problem, lambda state, cost: cost)

print(ucs\_search.search())

Output: None

Not working. Mainly due to such a large state space

IDS:

ids\_search = IterativeDeepeningSearch(problem)

print(ids\_search.search())

Output: None

Not working. Mainly due to such a large state space

## **Maze Solver**

Class:

class MazeSolver(GridWorldProblem):

    def \_\_init\_\_(self, grid, start\_state, end\_state):

        super().\_\_init\_\_(grid)

        self.start = start\_state

        self.end = end\_state

    def start\_state(self):

        return self.start

    def is\_end(self, state):

        return state == self.end

grid = [[1, 0, 1, 1, 1],

        [1, 0, 1, 0, 1],

        [1, 1, 1, 0, 0],

        [0, 0, 1, 1, 1]]

maze = MazeSolver(grid, (0, 0), (3, 3))

BFS:

bfs\_search = BestFirstSearch(maze, lambda state, cost: 0)

print(bfs\_search.search())

Output: ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3)], 6)

DFS:

dfs\_search = BestFirstSearch(maze, lambda state, cost: -cost)

print(dfs\_search.search())

Output: ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3)], 6)

UCS:

ucs\_search = BestFirstSearch(maze, lambda state, cost: cost)

print(ucs\_search.search())

Output: ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3)], 6)

IDS:

ids\_search = IterativeDeepeningSearch(maze)

print(ids\_search.search())

Output: ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3)], 6)

## **Water Jug**

Class:

class WaterJug:

    def \_\_init\_\_(self):

        self.jug1 = 4

        self.jug2 = 3

        self.goal = 2

    def start\_state(self):

        return (0, 0)

    def is\_end(self, state):

        return state[0] == self.goal or state[1] == self.goal

    def successors(self, state):

        jug1, jug2 = state

        successors = [['fill jug1', (self.jug1, jug2), 1],

                      ['fill jug2', (jug1, self.jug2), 1],

                      [ 'pour jug1 to jug2', (jug1 - min(jug1, self.jug2 - jug2), jug2 + min(jug1, self.jug2 - jug2)), 1],

                      [ 'pour jug2 to jug1', (jug1 + min(jug2, self.jug1 - jug1), jug2 - min(jug2, self.jug1 - jug1)), 1],

                      [ "empty jug1", (0, jug2), 1],

                      [ 'empty jug2', (jug1, 0), 1]]

        return successors

BFS:

bfs\_search = BestFirstSearch(problem, lambda state, cost: 0)

print(bfs\_search.search())

Output: ([(0, 0), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3)], 6)

DFS:

dfs\_search = BestFirstSearch(problem, lambda state, cost: -cost)

print(dfs\_search.search())

Output: ([(0, 0), (0, 3), (3, 0), (3, 3), (4, 2)], 4)

UCS:

ucs\_search = BestFirstSearch(maze, lambda state, cost: cost)

print(ucs\_search.search())

Output: ([(0, 0), (0, 3), (3, 0), (3, 3), (4, 2)], 4)

IDS:

ids\_search = IterativeDeepeningSearch(maze)

print(ids\_search.search())

Output: ([(0, 0), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3)], 6)

Explanation:

In this case it can be seen that uniform cost search and depth first search are giving us the optimal solution. It is mainly because in uniform cost search we are prioritizing our heap of the cumulative cost of the solution and hence can get minimum cost.

## **Robot Navigator**

Class:

class RobotNavigator(MazeSolver):

    def \_\_init\_\_(self, grid, start\_state, end\_state):

        super().\_\_init\_\_(grid, start\_state, end\_state)

grid = [[1, 1, 1, 0, 1],

        [1, 0, 1, 1, 1],

        [1, 0, 0, 1, 0],

        [0, 1, 1, 1, 1]]

robot = RobotNavigator(grid, (0, 0), (1, 3))

BFS:

bfs\_search = BestFirstSearch(robot, lambda state, cost: 0)

print(bfs\_search.search())

Output: ([(0, 0), (0, 1), (0, 2), (1, 2), (1, 3)], 4)

DFS:

dfs\_search = BestFirstSearch(robot, lambda state, cost: -cost)

print(dfs\_search.search())

Output: ([(0, 0), (0, 1), (0, 2), (1, 2), (1, 3)], 4)

UCS:

ucs\_search = BestFirstSearch(robot, lambda state, cost: cost)

print(ucs\_search.search())

Output: ([(0, 0), (0, 1), (0, 2), (1, 2), (1, 3)], 4)

IDS:

ids\_search = IterativeDeepeningSearch(robot)

print(ids\_search.search())

Output: ([(0, 0), (0, 1), (0, 2), (1, 2), (1, 3)], 4)

Explanation:

Since robot navigator uses a grid with obstacles, we can solve this problem with the same architecture used in the maze problem. There we represented closed paths with 0s and here the obstacles.