Lab Week 3

Session: 2022 – 2026

Submitted by:

Muhammad Saad Akmal 2022-CS-148

Submitted To:

Sir Samyan

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Problem 1

```
class Problem1:
    def __init__(self, N , goal):
        self.N = N
        self.goal = goal

    def start_state(self):
        return self.N

    def is_goal(self, state):
        return state == self.goal

    def cost(slef,s,a):
        return 1


    def actions(self , state):
        actions = []

        if state[2] == 0:

            if state[0]-1 >= 0:

                actions.append('Move right with one canabel')

            if state[1]-1 >= 0:
                actions.append('Move right with one missionary')

            if state[1]-1 >= 0 and  state[0]-1 >= 0:
                actions.append('Move right with both')

        if state[2] == 1:
            actions.append('Move left')

        return actions


    def transition(self, state , action):

        if action == "Move right with one canabel":
            return (state[0]-1 , state[1] ,1)

        elif action == "Move right with one missionary":
            return (state[0] , state[1]-1 ,1)

        elif action == "Move right with both":
            return (state[0]-1 , state[1]-1 ,1)

        elif action == "Move left":
            return (state[0] , state[1],0)
```

Output:

BFS result is ([(3, 3, 0), (2, 2, 1), (2, 2, 0), (1, 1, 1), (1, 1, 0), (0, 0, 1)], 5)

DFS result is ([(3, 3, 0), (2, 2, 1), (2, 2, 0), (1, 1, 1), (1, 1, 0), (0, 0, 1)], 5)

UCS result is ([(3, 3, 0), (2, 2, 1), (2, 2, 0), (1, 1, 1), (1, 1, 0), (0, 0, 1)], 5)

IDS result is ([(3, 3, 0), (2, 2, 1), (2, 2, 0), (1, 1, 1), (1, 1, 0), (0, 0, 1)], 5)

Problem 2:

```
class GridWorldProblem:
    def __init__(self, grid , goal):
        self.grid = grid
        self.graph = Graph()
        self.goal = goal
        self._build_graph()

    def _build_graph(self):
        rows = len(self.grid)
        cols = len(self.grid[0])
        for row in range(rows):
            for col in range(cols):
                state = (row, col)

                if state[1] + 1 < state[1]:
                    self.graph.add_edge(state, 'Right', 1, state[1] + 1)

                if state[1] - 1 < state[1]:
                    self.graph.add_edge(state, 'Left', 1, state[1] + 1)

                if state[0] + 1 < state[0]:
                    self.graph.add_edge(state, 'Down', 1, state[0] + 1)

                if state[0] + 1 <state[0]:
                    self.graph.add_edge(state, 'Up', 1, state[0] + 1)

    def start_state(self):
        return self.grid

    def is_goal(self, state):
        return state == self.goal

    def cost(self,state,action):
        return 1

    def actions(self,state):

        actions = []
        row_idx , col_idx = self.find_space(state)

        if col_idx + 1 < 3:

            actions.append('Right')

        if col_idx - 1 >= 0:
            actions.append('Left')

        if row_idx + 1 < 3:
            actions.append('Down')

        if row_idx - 1 >= 0:
            actions.append('Up')

        return actions
```

```python
def transition(self, state , action):

    row_idx , col_idx = self.find_space(state)
    if action == 'Right':
        new_state = self.swap(state , (row_idx , col_idx) , (row_idx , col_idx+1))
        return new_state

    if action == 'Left':
        new_state = self.swap(state , (row_idx , col_idx) , (row_idx , col_idx-1))
        return new_state

    if action == 'Down':

        new_state = self.swap(state , (row_idx , col_idx) , (row_idx + 1 , col_idx))
        return new_state

    if action == 'Up':

        new_state = self.swap(state , (row_idx , col_idx) , (row_idx - 1 , col_idx))
        return new_state
```

```python
def swap(self, grid, pos1, pos2):

    temp_list = [list(row) for row in grid]

    row1, col1 = pos1
    row2, col2 = pos2

    temp_list[row1][col1], temp_list[row2][col2] = temp_list[row2][col2], temp_list[row1][col1]

    return tuple(tuple(row) for row in temp_list)
```

```python
def find_space(self , grid):
    for row_idx, row in enumerate(grid):
        if 0 in row:
            col_idx = row.index(0)
            return (row_idx , col_idx)
```

Output:

Initial state: ((1, 2, 3), (4, 0, 6), (7, 5, 8))

Final State:((1, 2, 3), (4, 5, 6), (7, 8, 0))

BFS result is ([((1, 2, 3), (4, 0, 6), (7, 5, 8)), ((1, 2, 3), (4, 5, 6), (7, 0, 8)), ((1, 2, 3), (4, 5, 6), (7, 8, 0))], 2)

DFS result too large, Cost:320

UCS result is ([((1, 2, 3), (4, 0, 6), (7, 5, 8)), ((1, 2, 3), (4, 5, 6), (7, 0, 8)), ((1, 2, 3), (4, 5, 6), (7, 8, 0))], 2)

IDS result is ([((1, 2, 3), (4, 0, 6), (7, 5, 8)), ((1, 2, 3), (4, 5, 6), (7, 0, 8)), ((1, 2, 3), (4, 5, 6), (7, 8, 0))], 2)

Problem 3:

```
class GridWorldProblem:
    def __init__(self, grid , start , goal):
        self.grid = grid
        self.start = start
        self.goal = goal
        self.graph = Graph()
        self._build_graph()

    def _build_graph(self):
        rows = len(self.grid)
        cols = len(self.grid[0])
        for row in range(rows):
            for col in range(cols):
                state = (row, col)

                if state[1] + 1 < state[1]:
                    self.graph.add_edge(state, 'Right', 1, state[1] + 1)

                if state[1] - 1 < state[1]:
                    self.graph.add_edge(state, 'Left', 1, state[1] + 1)

                if state[0] + 1 < state[0]:
                    self.graph.add_edge(state, 'Down', 1, state[0] + 1)

                if state[0] + 1 <state[0]:
                    self.graph.add_edge(state, 'Up', 1, state[0] + 1)

    def start_state(self):
        return self.start

    def is_goal(self, state):
        return state == self.goal

    def cost(slef,state,action):
        return 1


    def actions(self,state):
        row , col = state
        actions = []
        rows = len(self.grid)
        cols = len(self.grid[0])

        if col+1 < cols and self.grid[row][col+1] != 1:
            actions.append('Right')
        if col-1 >= 0 and self.grid[row][col-1] != 1:
            actions.append('Left')
        if row+1 < rows and self.grid[row+1][col] != 1:
            actions.append('Down')
        if row-1 >= 0 and self.grid[row-1][col] != 1:
            actions.append('Up')

        return actions


    def transition(self, state , action):
        row , col = state

        if action == 'Right':
            return (row,col+1)
        if action == 'Left':
            return (row,col-1)
        if action == 'Down':
            return (row+1,col)
        if action == 'Up':
            return (row-1,col)
```

Output:

BFS result is ([(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3), (3, 4), (3, 5), (4, 5), (5, 5)], 10)

DFS result is ([(0, 0), (0, 1), (1, 1), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (5, 2), (5, 3), (5, 4), (5, 5)], 12)

UCS result is ([(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3), (3, 4), (3, 5), (4, 5), (5, 5)], 10)

IDS result is ([(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3), (3, 4), (3, 5), (4, 5), (5, 5)], 10)

Problem 4:

```
class WaterJugProblem:
    def start_state(self):
        return (0,0)

    def is_goal(self, state):
        return state[0] == 2 or state[1] == 2

    def cost (self,state,action):
        return 1

    def actions(self,state):

        actions = []

        if state[0] != 4:

            actions.append('Fill 4 gallon')

        if state[1] != 3:

            actions.append('Fill 3 gallon')

        if state[0] <= state[1]:
            actions.append('Pour from 3 to 4 gallon')

        if state[1] <= state[0]:
            actions.append('Pour from 4 to 3 gallon')

        if state[0] != 0:

            actions.append('Empty 4 gallon')

        if state[1] != 0:
            actions.append('Empty 3 gallon')

        return actions
```

```
def transition(self, state , action):

    if action == 'Fill 4 gallon':
        new_State = (4,state[1])
        return new_State

    if action == 'Fill 3 gallon':
        new_State = (3,state[1])
        return new_State


    if action == 'Pour from 3 to 4 gallon':
        j1 = state[0]
        j2 = state[1]
        while j2 != 0 and j1 != 4:
            j1 = j1+1
            j2 = j2-1

        new_State = (j1,j2)

        return new_State

    if action == 'Pour from 4 to 3 gallon':
        j1 = state[0]
        j2 = state[1]
        while j1!=0 and j2 != 3:
            j1 = j1-1
            j2 = j2+1

        new_State = (j1,j2)
        return new_State

    if action == 'Empty 4 gallon':
        j1=0
        j2 = state[1]
        return (j1,j2)

    if action == 'Empty 3 gallon':
        j2=0
        j1 = state[0]
        return (j1,j2)
```

Output:

BFS result is ([(0, 0), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3)], 6)

DFS result is ([(0, 0), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3)], 6)

UCS result is ([(0, 0), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3)], 6)

IDS result is ([(0, 0), (3, 0), (0, 3), (4, 3), (4, 0), (1, 3), (1, 0), (0, 1), (3, 1), (4, 1), (2, 3)], 9)

Problem 5:

```
class RobotNavigation:
    def __init__(self, grid , start , goal):
        self.grid = grid
        self.start = start
        self.goal = goal
        self.graph = Graph()
        self._build_graph()

    def _build_graph(self):
        rows = len(self.grid)
        cols = len(self.grid[0])
        for row in range(rows):
            for col in range(cols):
                state = (row, col)

                if state[1] + 1 < state[1]:
                    self.graph.add_edge(state, 'Right', 1, state[1] + 1)

                if state[1] - 1 < state[1]:
                    self.graph.add_edge(state, 'Left', 1, state[1] + 1)

                if state[0] + 1 < state[0]:
                    self.graph.add_edge(state, 'Down', 1, state[0] + 1)

                if state[0] + 1 <state[0]:
                    self.graph.add_edge(state, 'Up', 1, state[0] + 1)

    def start_state(self):
        return self.start

    def is_goal(self, state):
        return state == self.goal

    def cost(slef,state,action):
        return 1
```

```
    def actions(self,state):
        row , col = state
        actions = []

        rows = len(self.grid)
        cols = len(self.grid[0])


        if col+1 < cols and self.grid[row][col+1] != 1:
            actions.append('Right')
        if col-1 >= 0 and self.grid[row][col-1] != 1:
            actions.append('Left')
        if row+1 < rows and self.grid[row+1][col] != 1:
            actions.append('Down')
        if row-1 >= 0 and self.grid[row-1][col] != 1:
            actions.append('Up')

        return actions


    def transition(self, state , action):
        row , col = state

        if action == 'Right':
            return (row,col+1)
        if action == 'Left':
            return (row,col-1)
        if action == 'Down':
            return (row+1,col)
        if action == 'Up':
            return (row-1,col)
```

Output:

BFS result is ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5)], 10)

DFS result is ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5)], 10)

UCS result is ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5)], 10)

IDS result is ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5)], 10)