UCS2504 - Foundations of Artificial Intelligence

Assignment 1

Date : 01/08/2024

Problem Description :

1 Representing Search Problems :

A search problem consists of

- a start node
- a neighbors function that, given a node, returns an enumeration of the edges from the node
- a specification of a goal in terms of a Boolean function that takes a node and returns true if the node is a goal
- a (optional) heuristic function that, given a node, returns a non-negative real number. The heuristic function defaults to zero.

As far as the searcher is concerned a node can be anything. In the simple examples, the node is a string. Define an abstract class Search problem with methods start node(), is goal(), neighbors() and heuristic().

The neighbors is a list of edges. A (directed) edge consists of two nodes, a from node and a to node. The edge is the pair (from node,to node), but can also contain a non-negative cost (which defaults to 1) and can be labeled with an action. Implement a class Edge. Define a suitable repr () method to print the edge.

2 Explicit Representation of Search Graph :

The first representation of a search problem is from an explicit graph (as opposed to one that is generated as needed). An explicit graph consists of

- a set of nodes
- a list of edges
- a start node
- a set of goal nodes
- (optionally) a dictionary that maps a node to a heuristic value for that node

To define a search problem, we need to define the start node, the goal predicate, the neighbors function and the heuristic function. Define a concrete class Search problem from explicit graph(Search problem).

Give a title string also to the search problem. Define a suitable repr () method to print the graph.

3 Paths :

A searcher will return a path from the start node to a goal node. Represent the path in terms of a recursive data structure that can share subparts. A path is either:

- a node (representing a path of length 0) or
- an initial path and an edge, where the from node of the edge is the node at the end of initial.

Implement a class Path(). Define a suitable repr () method to print the path.

4 Example Search Problems :

Using Search problem from explicit graph, represent the following graphs.

For example, the first graph can be created with the code from searchProblem import Edge, Search_problem_from_explicit_graph, Search_problem problem1 = Search_problem_from_explicit_graph('Problem 1', {'A','B','C','D','G'}, [Edge('A','B',3), Edge('A','C',1), Edge('B','D',1), Edge('B','G',3), Edge('C','B',1), Edge('C','D',3), Edge('D','G',1)], start = 'A', goals = {'G'})

5 Searcher :

A Searcher for a problem is given can be asked repeatedly for the next path. To solve a problem, you can construct a Searcher object for the problem and then repeatedly ask for the next path using search. If there are no more paths, None is returned. Implement Searcher class with DFS (Depth-First Search).

To use depth-first search to find multiple paths for problem1, copy and paste the following into Python's read-evaluate-print loop; keep finding next solutions until there are no more:

Depth-first search for problem1; do the following: searcher1 = Searcher(searchExample.problem1) searcher1.search() # find first solution searcher1.search() # find next solution (repeat until no solutions)

Algorithm:

```
Input: problem
Output: solution, or failure

frontier ← [initial state of problem]
explored = {}
while frontier is not empty do
  node ← remove a node from frontier
  if node is a goal state then
    return solution
  end
  add node to explored
  add the successor nodes to frontier only if not in frontier or explored
end
return failure
```

Code :

```
# 1) Representing Search Problems

from abc import ABC, abstractmethod

class Search_problem(ABC):
    @abstractmethod
    def start_node(self):
        pass
```

```
    @abstractmethod
    def is_goal(self, node):
        pass

    @abstractmethod
    def neighbors(self, node):
        pass

    def heuristic(self, n):
        return 0

class Edge:
    def __init__(self, from_node, to_node, cost=1, action=None):
        self.from_node = from_node
        self.to_node = to_node
        self.action = action
        self.cost = cost
        assert cost >= 0, (f"Cost cannot be negative: {self}, cost={cost}")

    def __repr__(self):
        return f"Edge({self.from_node} -> {self.to_node}, cost={self.cost},
action={self.action})"

# 2)  Explicit Representation of Search Graph

class Search_problem_from_explicit_graph(Search_problem):
    def __init__(self, title, nodes, edges, start, goals, heuristic=None):
        self.title = title
        self.nodes = nodes
        self.edges = edges
        self.start = start
        self.goals = goals
        self.heuristic_dict = heuristic if heuristic else {}

    def start_node(self):
        return self.start

    def is_goal(self, node):
        return node in self.goals

    def neighbors(self, node):
        return [edge for edge in self.edges if edge.from_node == node]

    def heuristic(self, node):
        return self.heuristic_dict.get(node, 0)

    def __repr__(self):
        return f"SearchProblemFromExplicitGraph('{self.title}', {self.nodes},
{self.edges}, start='{self.start}', goals={self.goals}, heuristic=
{self.heuristic_dict})"


# 3) Paths
```

```
class Path:
    def __init__(self, node, edge=None, previous_path=None):
        self.node = node
        self.edge = edge
        self.previous_path = previous_path
        self.total_cost = edge.cost + previous_path.total_cost if edge and
previous_path else 0

    def __repr__(self):
        path_str = self.node
        if self.previous_path:
            path_str = f"{self.previous_path} -> {self.node}"
        return f"{path_str} (Total cost: {self.total_cost})"

# 4) Example Search Problems

problem1 = Search_problem_from_explicit_graph('Problem 1',
    {'A', 'B', 'C', 'D', 'G'},
    [Edge('A', 'B', 3), Edge('A', 'C', 1), Edge('B', 'D', 1), Edge('B', 'G', 3),
     Edge('C', 'B', 1), Edge('C', 'D', 3), Edge('D', 'G', 1)],
    start='A',
    goals={'G','D'}
)

problem2 = Search_problem_from_explicit_graph('Problem 2',
    {'A','B','C','D','E','G','H','J'},
    [Edge('A','B',1), Edge('B','C',3), Edge('B','D',1), Edge('D','E',3),
    Edge('D','G',1), Edge('A','H',3), Edge('H','J',1)],
    start = 'A',
    goals = {'J'}
)

problem3 = Search_problem_from_explicit_graph('Problem 3',
    {'A', 'B', 'C', 'D', 'E', 'G', 'H', 'J'},
    [Edge('A', 'B', 2), Edge('A', 'C', 3), Edge('A', 'D', 4), Edge('B', 'E', 2),
     Edge('B', 'F', 3), Edge('C', 'J', 7), Edge('D', 'H', 4), Edge('F', 'D', 2),
     Edge('H', 'G', 3), Edge('J', 'G', 4)],
    start='A',
    goals={'G'}
)

# 5) Searcher

class Searcher:
    def __init__(self, problem):
        self.problem = problem
        self.frontier = [Path(problem.start_node())]

    def search(self):
        while self.frontier:
            path = self.frontier.pop()
            node = path.node
```

```
            if self.problem.is_goal(node):
                return path

            for edge in self.problem.neighbors(node):
                new_path = Path(edge.to_node, edge, path)
                self.frontier.append(new_path)

        return None

# Depth-first search for problem1
print("Paths for problem1 using DFS")
searcher1 = Searcher(problem1)
result = searcher1.search()
while result is not None:
    print(result)
    result = searcher1.search()

# Depth-first search for problem2
print("\nPaths for problem2 using DFS")
searcher2 = Searcher(problem2)
result = searcher2.search()
while result is not None:
    print(result)
    result = searcher2.search()

# Depth-first search for problem3
print("\nPaths for problem3 using DFS")
searcher3 = Searcher(problem3)
result = searcher3.search()
while result is not None:
    print(result)
    result = searcher3.search()
```

Testing :

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/SearchProblems - Assignment -
1.py"
Paths for problem1 using DFS
A (Total cost: 0) -> C (Total cost: 1) -> D (Total cost: 4)
A (Total cost: 0) -> C (Total cost: 1) -> B (Total cost: 2) -> G (Total cost: 5)
A (Total cost: 0) -> C (Total cost: 1) -> B (Total cost: 2) -> D (Total cost: 3)
A (Total cost: 0) -> B (Total cost: 3) -> G (Total cost: 6)
A (Total cost: 0) -> B (Total cost: 3) -> D (Total cost: 4)

Paths for problem2 using DFS
A (Total cost: 0) -> H (Total cost: 3) -> J (Total cost: 4)

Paths for problem3 using DFS
```

```
A (Total cost: 0) -> D (Total cost: 4) -> H (Total cost: 8) -> G (Total cost: 11)
A (Total cost: 0) -> C (Total cost: 3) -> J (Total cost: 10) -> G (Total cost: 14)
A (Total cost: 0) -> B (Total cost: 2) -> F (Total cost: 5) -> D (Total cost: 7) -
> H (Total cost: 11) -> G (Total cost: 14)
```

Assignment 2

Date : 08/08/2024

Problem Description :

1 Representing Search Problems :

A search problem consists of

- a start node
- a neighbors function that, given a node, returns an enumeration of the edges from the node
- a specification of a goal in terms of a Boolean function that takes a node and returns true if the node is a goal
- a (optional) heuristic function that, given a node, returns a non-negative real number. The heuristic function defaults to zero.

As far as the searcher is concerned a node can be anything. In the simple examples, the node is a string. Define an abstract class Search problem with methods start node(), is goal(), neighbors() and heuristic().

The neighbors is a list of edges. A (directed) edge consists of two nodes, a from node and a to node. The edge is the pair (from node,to node), but can also contain a non-negative cost (which defaults to 1) and can be labeled with an action. Implement a class Edge. Define a suitable repr () method to print the edge.

2 Explicit Representation of Search Graph :

The first representation of a search problem is from an explicit graph (as opposed to one that is generated as needed). An explicit graph consists of

- a set of nodes
- a list of edges
- a start node
- a set of goal nodes
- (optionally) a dictionary that maps a node to a heuristic value for that node

To define a search problem, we need to define the start node, the goal predicate, the neighbors function and the heuristic function. Define a concrete class Search problem from explicit graph(Search problem).

Give a title string also to the search problem. Define a suitable repr () method to print the graph. 3 Paths :

A searcher will return a path from the start node to a goal node. Represent the path in terms of a recursive data structure that can share subparts. A path is either:

- a node (representing a path of length 0) or
- an initial path and an edge, where the from node of the edge is the node at the end of initial.

Implement a class Path(). Define a suitable repr () method to print the path.

4 Example Search Problems :

Using Search problem from explicit graph, represent the following graphs.

For example, the first graph can be created with the code from searchProblem import Edge, Search_problem_from_explicit_graph, Search_problem problem1 = Search_problem_from_explicit_graph('Problem 1', {'A','B','C','D','G'}, [Edge('A','B',3), Edge('A','C',1), Edge('B','D',1), Edge('B','G',3), Edge('C','B',1), Edge('C','D',3), Edge('D','G',1)], start = 'A', goals = {'G'})

5 Frontier as a Priority Queue In many of the search algorithms, such as Uniform Cost Search, A* and other best-first searchers, the frontier is implemented as a priority queue. Use Python's built-in priority queue implementations heapq (read the Python documentation, https://docs.python. org/3/library/heapq.html). Implement FrontierPQ. A frontier is a list of triples. The first element of each triple is the value to be minimized. The second element is a unique index which specifies the order that the elements were added to the queue, and the third element is the path that is on the queue. The use of the unique index ensures that the priority queue implementation does not compare paths; whether one path is less than another is not defined. It also lets us control what sort of search (e.g., depth-first or breadth-first) occurs when the value to be minimized does not give a unique next path. Use a variable frontier index to maintain the total number of elements of the frontier that have been created.

6 Searcher A Searcher for a problem can be asked repeatedly for the next path. To solve a problem, you can construct a Searcher object for the problem and then repeatedly ask for the next path using search. If there are no more paths, None is returned. Implement Searcher class using using the FrontierPQ class.

Algorithm:

```
Input: problem
Output: solution, or failure

frontier ← Priority Queue
Add starting node to frontier
explored ← Set
while frontier is not empty do
  path ← remove the frontier node with shortest distance
  v ← path.node
  if v is a goal node then return solution
  if v is not in explored
    for each successor w of v do
        new_path ← path + v
        new_cost ← path.cost + heuristic(u)
        Add new_path to Frontier
return failure
```

Code :

```
# 1) Representing search problems

from abc import ABC, abstractmethod
import heapq

class Search_problem(ABC):
    @abstractmethod
```

```
    def start_node(self):
        pass

    @abstractmethod
    def is_goal(self, node):
        pass

    @abstractmethod
    def neighbors(self, node):
        pass

    def heuristic(self, n):
        return 0

class Edge:
    def __init__(self, from_node, to_node, cost=1, action=None):
        self.from_node = from_node
        self.to_node = to_node
        self.action = action
        self.cost = cost
        assert cost >= 0, (f"Cost cannot be negative: {self}, cost={cost}")

    def __repr__(self):
        return f"Edge({self.from_node} -> {self.to_node}, cost={self.cost},
action={self.action})"

# 2) Explicit Representation of Search Graph

class Search_problem_from_explicit_graph(Search_problem):
    def __init__(self, title, nodes, edges, start, goals, heuristic=None):
        self.title = title
        self.nodes = nodes
        self.edges = edges
        self.start = start
        self.goals = goals
        self.heuristic_dict = heuristic if heuristic else {}

    def start_node(self):
        return self.start

    def is_goal(self, node):
        return node in self.goals

    def neighbors(self, node):
        return [edge for edge in self.edges if edge.from_node == node]

    def heuristic(self, node):
        return self.heuristic_dict.get(node, 0)

    def __repr__(self):
        return f"SearchProblemFromExplicitGraph('{self.title}', {self.nodes},
{self.edges}, start='{self.start}', goals={self.goals}, heuristic=
{self.heuristic_dict})"
```

```
# 3) Paths

class Path:
    def __init__(self, node, edge=None, previous_path=None):
        self.node = node
        self.edge = edge
        self.previous_path = previous_path
        self.total_cost = edge.cost + previous_path.total_cost if edge and
previous_path else 0

    def __repr__(self):
        if self.previous_path is None:
            return f"{self.node}"
        else:
            return f"{self.previous_path} -> {self.node}"

    def get_full_path(self):
        if self.previous_path is None:
            return [self.node]
        else:
            return self.previous_path.get_full_path() + [self.node]

    def get_total_cost(self):
        return self.total_cost

# 4) Example Search Problems

problem1 = Search_problem_from_explicit_graph('Problem 1',
    {'A', 'B', 'C', 'D', 'G'},
    [Edge('A', 'B', 3), Edge('A', 'C', 1), Edge('B', 'D', 1), Edge('B', 'G', 3),
     Edge('C', 'B', 1), Edge('C', 'D', 3), Edge('D', 'G', 1)],
    start='A',
    goals={'D'}
)

problem2 = Search_problem_from_explicit_graph('Problem 2',
    {'A','B','C','D','E','G','H','J'},
    [Edge('A','B',1), Edge('B','C',3), Edge('B','D',1), Edge('D','E',3),
    Edge('D','G',1), Edge('A','H',3), Edge('H','J',1)],
    start = 'A',
    goals = {'H'}
)

problem3 = Search_problem_from_explicit_graph('Problem 3',
    {'A', 'B', 'C', 'D', 'E', 'G', 'H', 'J'},
    [Edge('A', 'B', 2), Edge('A', 'C', 3), Edge('A', 'D', 4), Edge('B', 'E', 2),
     Edge('B', 'F', 3), Edge('C', 'J', 7), Edge('D', 'H', 4), Edge('F', 'D', 2),
     Edge('H', 'G', 3), Edge('J', 'G', 4)],
    start='A',
    goals={'J'}
)

# 5) Frontier as a Priority Queue
```

```
class FrontierPQ:
    def __init__(self):
        self.elements = []
        self.counter = 0

    def add(self, path):
        heapq.heappush(self.elements, (path.total_cost, self.counter, path))
        self.counter += 1

    def pop(self):
        return heapq.heappop(self.elements)[2]

    def is_empty(self):
        return len(self.elements) == 0

# 6) Searcher

class Searcher:
    def __init__(self, problem):
        self.problem = problem
        self.frontier = FrontierPQ()
        self.frontier.add(Path(problem.start_node()))
        self.explored = set()

    def search(self):
        while not self.frontier.is_empty():
            path = self.frontier.pop()
            node = path.node

            if self.problem.is_goal(node):
                return path

            if node not in self.explored:
                self.explored.add(node)
                for edge in self.problem.neighbors(node):
                    new_path = Path(edge.to_node, edge, path)
                    self.frontier.add(new_path)

        return None

#Dijkstra search for problem1
print("Paths for problem 1 using Dijkstra's algorithm")
searcher1 = Searcher(problem1)
result = searcher1.search()
if result:
    print("Path:", result.get_full_path())
    print("Total Cost:", result.get_total_cost())

# Dijkstra search for problem2
print("\nPaths for problem 2 using Dijkstra's algorithm")
searcher2 = Searcher(problem2)
result = searcher2.search()
if result:
    print("Path:", result.get_full_path())
```

```
    print("Total Cost:", result.get_total_cost())

# Dijkstra search for problem3
print("\nPaths for problem 3 using Dijkstra's algorithm")
searcher3 = Searcher(problem3)
result = searcher3.search()
if result:
    print("Path:", result.get_full_path())
    print("Total Cost:", result.get_total_cost())
```

Testing :

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/SearchProblemDijkstra -
Assignment - 2.py"
Paths for problem 1 using Dijkstra's algorithm
Path: ['A', 'C', 'B', 'D']
Total Cost: 3

Paths for problem 2 using Dijkstra's algorithm
Path: ['A', 'H']
Total Cost: 3

Paths for problem 3 using Dijkstra's algorithm
Path: ['A', 'C', 'J']
Total Cost: 10
```

Assignment 3

Date : 12/08/2024

Problem Description 1:

In a 3×3 board, 8 of the squares are filled with integers 1 to 9, and one square is left empty. One move is sliding into the empty square the integer in any one of its adjacent squares. The start state is given on the left side of the figure, and the goal state given on the right side. Find a sequence of moves to go from the start state to the goal state.

1. Formulate the problem as a state space search problem.
2. Find a suitable representation for the states and the nodes.
3. Solve the problem using any of the uninformed search strategies.
4. We can use Manhattan distance as a heuristic h(n). The cheapest cost from the current node to the goal node, can be estimated as how many moves will be required to transform the current node into the goal node. This is related to the distance each tile must travel to arrive at its destination, hence we sum the Manhattan distance of each square from its home position.
5. An alternative heuristic should consider the number of tiles that are “out-of-sequence”. An out of sequence score can be computed as follows:

- a tile in the center counts 1,
- a tile not in the center counts 0 if it is followed by its proper successor as defined by the goal arrangement,
- otherwise, a tile counts 2.

6. Use anyone of the two heuristics, and implement Greedy Best-First Search.
7. Use anyone of the two heuristics, and implement A* Search

Algorithm:

1. A*

```
Input: problem
Output: solution, or failure

frontier ← Priority Queue
add starting node to frontier with priority = heuristic(start) + 0

while frontier is not empty do
    path ← remove node from frontier with lowest priority
    node ← path.node
    add node to explored set

    for each neighbor of node do
        if neighbor not in explored set then
            new_path ← Path(neighbor, path, edge)
            if neighbor is a goal node then
                return new_path as solution
```

```
            frontier.add((heuristic(neighbor) + g(new_path), new_path))

return failure
```

2. Greedy Best First Search

```
Input: problem
Output: solution, or failure

frontier ← Priority Queue
add starting node to frontier with priority = heuristic(start)

while frontier is not empty do
    path ← remove node from frontier with lowest priority
    node ← path.node
    add node to explored set

    for each neighbor of node do
        if neighbor not in explored set then
            new_path ← Path(neighbor, path, edge)
            if neighbor is a goal node then
                return new_path as solution

            frontier.add((heuristic(neighbor), new_path))

return failure
```

*Code for A and Greedy Best First Search:**

```
import heapq
from collections import deque

#This problem is formulated as a state space search problem
class PuzzleState:
    #Representation for the states and nodes
    def __init__(self, board, parent=None, action=None, cost=0):
        # Parameters:
        # board: The current configuration of the puzzle.
        # parent: The parent state from which this state was generated.
        # action: The move that led to this state.
        # cost: The cost to reach this state from the start state.

        self.board = board
        self.parent = parent
        self.action = action
        self.cost = cost

    def __lt__(self, other):
        return self.cost < other.cost
```

```
    def is_goal(self, goal):
        return self.board == goal

    def get_blank_position(self):
        return self.board.index(0)

    def generate_children(self):
        # Generate all possible child states from the current state by sliding a
tile into the blank space.
        # Returns: A list of child PuzzleState objects.

        children = []
        blank_pos = self.get_blank_position()
        moves = [(1, 0, "Down"), (-1, 0, "Up"), (0, 1, "Right"), (0, -1, "Left")]
        x, y = divmod(blank_pos, 3)

        for dx, dy, action in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_blank_pos = nx * 3 + ny
                new_board = self.board[:]
                new_board[blank_pos], new_board[new_blank_pos] =
new_board[new_blank_pos], new_board[blank_pos]
                children.append(PuzzleState(new_board, self, action))

        return children

    def print_solution(self):
        # Recursively print the sequence of moves from the initial state to the
goal state.
        if self.parent:
            self.parent.print_solution()
        self.print_board()

    def print_board(self):
        # Print the current board configuration in a 3x3 grid format.
        for i in range(3):
            print(self.board[i*3:(i+1)*3])
        print("\n")

    def get_move_count(self):
        # Count the number of moves from the initial state to the current state.
        if self.parent is None:
            return 0
        return 1 + self.parent.get_move_count()

# Breadth-First Search (BFS) - Uninformed Search Strategy
def bfs(start, goal):
    frontier = deque([start])
    explored = set()

    while frontier:
        current = frontier.popleft()
```

```
        if current.is_goal(goal):
            current.print_solution()
            move_count = current.get_move_count()
            print(f"Number of moves: {move_count}")
            return move_count

        explored.add(tuple(current.board))

        for child in current.generate_children():
            if tuple(child.board) not in explored:
                frontier.append(child)

# Manhattan Distance as a heuristic
def manhattan_distance(state, goal):
    distance = 0
    for i in range(1, 9):
        current_pos = state.board.index(i)
        goal_pos = goal.index(i)
        current_x, current_y = divmod(current_pos, 3)
        goal_x, goal_y = divmod(goal_pos, 3)
        distance += abs(current_x - goal_x) + abs(current_y - goal_y)
    return distance

# Out-of-sequence as a heuristic
def out_of_sequence(state, goal):
    score = 0
    for i in range(8):
        if i == 4:
            if state.board[i] != goal[i]:
                score += 1
        elif state.board[i] != 0 and state.board[i] != goal[i]:
            score += 2
    return score

# Greedy Best-First Search
def greedy_best_first_search(start, goal, heuristic):
    frontier = []
    heapq.heappush(frontier, (heuristic(start, goal), start))
    explored = set()

    while frontier:
        _, current = heapq.heappop(frontier)

        if current.is_goal(goal):
            current.print_solution()
            move_count = current.get_move_count()
            print(f"Number of moves: {move_count}")
            return move_count

        explored.add(tuple(current.board))

        for child in current.generate_children():
            if tuple(child.board) not in explored:
```

```
                heapq.heappush(frontier, (heuristic(child, goal), child))

# A* Search
def a_star_search(start, goal, heuristic):
    frontier = []
    heapq.heappush(frontier, (heuristic(start, goal), start))
    explored = set()

    while frontier:
        _, current = heapq.heappop(frontier)

        if current.is_goal(goal):
            current.print_solution()
            move_count = current.get_move_count()
            print(f"Number of moves: {move_count}")
            return move_count

        explored.add(tuple(current.board))

        for child in current.generate_children():
            new_cost = current.cost + 1
            if tuple(child.board) not in explored or new_cost < child.cost:
                child.cost = new_cost
                priority = new_cost + heuristic(child, goal)
                heapq.heappush(frontier, (priority, child))

# Puzzle State Example
start_state = PuzzleState([1, 2, 3, 4, 0, 5, 6, 7, 8])
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

# Store the number of moves for each search method and heuristic
results = {}

# Breadth-First Search (BFS)
print("Breadth-First Search:")
results['BFS'] = bfs(start_state, goal_state)

# Greedy Best-First Search with Manhattan Distance
print("\nGreedy Best-First Search with Manhattan Distance:")
results['Greedy Best-First (Manhattan Distance)'] =
greedy_best_first_search(start_state, goal_state, manhattan_distance)

# A* Search with Manhattan Distance
print("\nA* Search with Manhattan Distance:")
results['A* (Manhattan Distance)'] = a_star_search(start_state, goal_state,
manhattan_distance)

# Greedy Best-First Search with Out-of-sequence Score
print("\nGreedy Best-First Search with Out-of-sequence Score:")
results['Greedy Best-First (Out-of-sequence Score)'] =
greedy_best_first_search(start_state, goal_state, out_of_sequence)

# A* Search with Out-of-sequence Score
print("\nA* Search with Out-of-sequence Score:")
```

```
results['A* (Out-of-sequence Score)'] = a_star_search(start_state, goal_state,
out_of_sequence)

# Summary of the number of moves for each strategy
print("\nSummary of Number of Moves:")
for method, moves in results.items():
    print(f"{method}: {moves} moves")
```

Testing :

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/8PuzzleProblem_Algorithm -
Assignment - 3.py"
Breadth-First Search:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]


[1, 2, 3]
[4, 5, 0]
[6, 7, 8]


[1, 2, 3]
[4, 5, 8]
[6, 7, 0]


[1, 2, 3]
[4, 5, 8]
[6, 0, 7]


[1, 2, 3]
[4, 5, 8]
[0, 6, 7]


[1, 2, 3]
[0, 5, 8]
[4, 6, 7]


[1, 2, 3]
[5, 0, 8]
[4, 6, 7]
```

```
[1, 2, 3]
[5, 6, 8]
[4, 0, 7]


[1, 2, 3]
[5, 6, 8]
[4, 7, 0]


[1, 2, 3]
[5, 6, 0]
[4, 7, 8]


[1, 2, 3]
[5, 0, 6]
[4, 7, 8]


[1, 2, 3]
[0, 5, 6]
[4, 7, 8]


[1, 2, 3]
[4, 5, 6]
[0, 7, 8]


[1, 2, 3]
[4, 5, 6]
[7, 0, 8]


[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


Number of moves: 14

Greedy Best-First Search with Manhattan Distance:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]


[1, 2, 3]
[0, 4, 5]
[6, 7, 8]


[1, 2, 3]
```

```
[6, 4, 5]
[0, 7, 8]


[1, 2, 3]
[6, 4, 5]
[7, 0, 8]


[1, 2, 3]
[6, 0, 5]
[7, 4, 8]


[1, 2, 3]
[0, 6, 5]
[7, 4, 8]


[1, 2, 3]
[7, 6, 5]
[0, 4, 8]


[1, 2, 3]
[7, 6, 5]
[4, 0, 8]


[1, 2, 3]
[7, 0, 5]
[4, 6, 8]


[1, 2, 3]
[7, 5, 0]
[4, 6, 8]


[1, 2, 3]
[7, 5, 8]
[4, 6, 0]


[1, 2, 3]
[7, 5, 8]
[4, 0, 6]


[1, 2, 3]
[7, 5, 8]
[0, 4, 6]
```

```
[1, 2, 3]
[0, 5, 8]
[7, 4, 6]


[1, 2, 3]
[5, 0, 8]
[7, 4, 6]


[1, 2, 3]
[5, 8, 0]
[7, 4, 6]


[1, 2, 3]
[5, 8, 6]
[7, 4, 0]


[1, 2, 3]
[5, 8, 6]
[7, 0, 4]


[1, 2, 3]
[5, 0, 6]
[7, 8, 4]


[1, 2, 3]
[0, 5, 6]
[7, 8, 4]


[1, 2, 3]
[7, 5, 6]
[0, 8, 4]


[1, 2, 3]
[7, 5, 6]
[8, 0, 4]


[1, 2, 3]
[7, 5, 6]
[8, 4, 0]


[1, 2, 3]
[7, 5, 0]
[8, 4, 6]
```

```
[1, 2, 3]
[7, 0, 5]
[8, 4, 6]


[1, 2, 3]
[7, 4, 5]
[8, 0, 6]


[1, 2, 3]
[7, 4, 5]
[0, 8, 6]


[1, 2, 3]
[0, 4, 5]
[7, 8, 6]


[1, 2, 3]
[4, 0, 5]
[7, 8, 6]


[1, 2, 3]
[4, 5, 0]
[7, 8, 6]


[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


Number of moves: 30

A* Search with Manhattan Distance:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]


[1, 2, 3]
[4, 5, 0]
[6, 7, 8]


[1, 2, 3]
[4, 5, 8]
[6, 7, 0]
```

```
[1, 2, 3]
[4, 5, 8]
[6, 0, 7]


[1, 2, 3]
[4, 5, 8]
[0, 6, 7]


[1, 2, 3]
[0, 5, 8]
[4, 6, 7]


[1, 2, 3]
[5, 0, 8]
[4, 6, 7]


[1, 2, 3]
[5, 6, 8]
[4, 0, 7]


[1, 2, 3]
[5, 6, 8]
[4, 7, 0]


[1, 2, 3]
[5, 6, 0]
[4, 7, 8]


[1, 2, 3]
[5, 0, 6]
[4, 7, 8]


[1, 2, 3]
[0, 5, 6]
[4, 7, 8]


[1, 2, 3]
[4, 5, 6]
[0, 7, 8]


[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
```

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


Number of moves: 14

Greedy Best-First Search with Out-of-sequence Score:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]


[1, 2, 3]
[0, 4, 5]
[6, 7, 8]


[1, 2, 3]
[6, 4, 5]
[0, 7, 8]


[1, 2, 3]
[6, 4, 5]
[7, 0, 8]


[1, 2, 3]
[6, 4, 5]
[7, 8, 0]


[1, 2, 3]
[6, 4, 0]
[7, 8, 5]


[1, 2, 3]
[6, 0, 4]
[7, 8, 5]


[1, 2, 3]
[6, 8, 4]
[7, 0, 5]


[1, 2, 3]
[6, 8, 4]
[7, 5, 0]
```

```
[1, 2, 3]
[6, 8, 0]
[7, 5, 4]


[1, 2, 3]
[6, 0, 8]
[7, 5, 4]


[1, 2, 3]
[6, 5, 8]
[7, 0, 4]


[1, 2, 3]
[6, 5, 8]
[7, 4, 0]


[1, 2, 3]
[6, 5, 0]
[7, 4, 8]


[1, 2, 3]
[6, 0, 5]
[7, 4, 8]


[1, 2, 3]
[0, 6, 5]
[7, 4, 8]


[1, 2, 3]
[7, 6, 5]
[0, 4, 8]


[1, 2, 3]
[7, 6, 5]
[4, 0, 8]


[1, 2, 3]
[7, 6, 5]
[4, 8, 0]


[1, 2, 3]
[7, 6, 0]
[4, 8, 5]
```

```
[1, 2, 3]
[7, 0, 6]
[4, 8, 5]


[1, 2, 3]
[0, 7, 6]
[4, 8, 5]


[1, 2, 3]
[4, 7, 6]
[0, 8, 5]


[1, 2, 3]
[4, 7, 6]
[8, 0, 5]


[1, 2, 3]
[4, 7, 6]
[8, 5, 0]


[1, 2, 3]
[4, 7, 0]
[8, 5, 6]


[1, 2, 3]
[4, 0, 7]
[8, 5, 6]


[1, 2, 3]
[4, 5, 7]
[8, 0, 6]


[1, 2, 3]
[4, 5, 7]
[0, 8, 6]


[1, 2, 3]
[0, 5, 7]
[4, 8, 6]


[1, 2, 3]
[5, 0, 7]
[4, 8, 6]
```

```
[1, 2, 3]
[5, 7, 0]
[4, 8, 6]


[1, 2, 3]
[5, 7, 6]
[4, 8, 0]


[1, 2, 3]
[5, 7, 6]
[4, 0, 8]


[1, 2, 3]
[5, 7, 6]
[0, 4, 8]


[1, 2, 3]
[0, 7, 6]
[5, 4, 8]


[1, 2, 3]
[7, 0, 6]
[5, 4, 8]


[1, 2, 3]
[7, 4, 6]
[5, 0, 8]


[1, 2, 3]
[7, 4, 6]
[0, 5, 8]


[1, 2, 3]
[0, 4, 6]
[7, 5, 8]


[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]


Number of moves: 42

A* Search with Out-of-sequence Score:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]


[1, 2, 3]
[4, 5, 0]
[6, 7, 8]


[1, 2, 3]
[4, 5, 8]
[6, 7, 0]


[1, 2, 3]
[4, 5, 8]
[6, 0, 7]


[1, 2, 3]
[4, 5, 8]
[0, 6, 7]


[1, 2, 3]
[0, 5, 8]
[4, 6, 7]


[1, 2, 3]
[5, 0, 8]
[4, 6, 7]


[1, 2, 3]
[5, 6, 8]
[4, 0, 7]


[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
```

```
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]


[1, 2, 3]
[5, 0, 6]
[4, 7, 8]


[1, 2, 3]
[0, 5, 6]
[4, 7, 8]


[1, 2, 3]
[4, 5, 6]
[0, 7, 8]


[1, 2, 3]
[4, 5, 6]
[7, 0, 8]


[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


Number of moves: 14

Summary of Number of Moves:
BFS: 14 moves
Greedy Best-First (Manhattan Distance): 30 moves
A* (Manhattan Distance): 14 moves
Greedy Best-First (Out-of-sequence Score): 42 moves
A* (Out-of-sequence Score): 14 moves
```

Problem Description 2:

You are given an 8-litre jar full of water and two empty jars of 5- and 3-litre capacity. You have to get exactly 4 litres of water in one of the jars. You can completely empty a jar into another jar with space or completely fill up a jar from another jar.

1. Formulate the problem: Identify states, actions, initial state, goal state(s). Represent the state by a 3-tuple. For example, the intial state state is (8,0,0). (4,1,3) is a goal state (there may be other goal states also).
2. Use a suitable data structure to keep track of the parent of every state. Write a function to print the sequence of states and actions from the initial state to the goal state.
3. Write a function next states(s) that returns a list of successor states of a given state s.
4. Implement Breadth-First-Search algorithm to search the state space graph for a goal state that produces the required sequence of pourings. Use a Queue as frontier that stores the discovered states yet be explored. Use a dictionary for explored that is used to store the explored states.
5. Modify your program to trace the contents of the Queue in your algorithm. How many states are explored by your algorithm?

Algorithm:

```
Input: problem
Output: solution, or failure
frontier ← Queue
add starting node to frontier
parent[start] ← None

while frontier is not empty do
    path ← remove node from frontier
    node ← path.node
    add node to explored set

    for each neighbor of node do
        if neighbor not in explored set then
            new_path ← Path(neighbor, path)
            if neighbor is a goal node then
                return new_path as solution

            frontier.append(new_path)

return failure
```

Code :

```
from collections import deque

# Define the state of the water jugs
class WaterJugState:
    def __init__(self, jugs, parent=None, action=None):
```

```
        self.jugs = jugs
        self.parent = parent
        self.action = action

    def is_goal(self, goal_state):
        return self.jugs == goal_state

    def __eq__(self, other):
        return self.jugs == other.jugs

    def __hash__(self):
        return hash(self.jugs)

# Generate all possible next states from the current state
def next_states(state):
    successors = []
    jug_sizes = (8, 5, 3)

    for i in range(3):
        for j in range(3):
            if i != j:
                new_jugs = list(state.jugs)
                transfer_amount = min(new_jugs[i], jug_sizes[j] - new_jugs[j])
                new_jugs[i] -= transfer_amount
                new_jugs[j] += transfer_amount

                action = f"Pour {transfer_amount}L from jug {i+1} to jug {j+1}"
                successors.append(WaterJugState(tuple(new_jugs), state, action))

    return successors

# Perform BFS to find a solution state that matches the goal state
def bfs(initial_state, goal_state):
    #Queue as a frontier
    frontier = deque([initial_state])
    explored = {}
    state_count = 0

    while frontier:
        current_state = frontier.popleft()
        state_count += 1
        if current_state.is_goal(goal_state):
            return current_state, state_count
        explored[current_state.jugs] = current_state

        for next_state in next_states(current_state):
            if next_state.jugs not in explored and all(next_state.jugs != s.jugs
for s in frontier):
                frontier.append(next_state)

    return None, state_count

# Print the sequence of states and actions from initial to goal state
def print_solution(state):
```

```
    if state.parent:
        print_solution(state.parent)
    # Print the current state and action leading to this state
    action_str = state.action if state.action else "None"
    print(f"State: {state.jugs}, Action: {action_str}")

# Get the target state from the user
goal_state = tuple(map(int, input("Enter the goal state as three integers (e.g., 4
1 3): ").split()))

# Initial state of the water jugs
initial_state = WaterJugState((8, 0, 0))

goal_state, explored_states = bfs(initial_state, goal_state)

if goal_state:
    print("Solution found:")
    print_solution(goal_state)
    print(f"Goal state reached: {goal_state.jugs}")
    print(f"Total states explored: {explored_states}")
else:
    print("No solution found")
```

Testing :

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/Decantation_Problem - Assignment
- 3.py"
Enter the goal state as three integers (e.g., 4 1 3): 4 1 3
Solution found:
State: (8, 0, 0), Action: None
State: (5, 0, 3), Action: Pour 3L from jug 1 to jug 3
State: (5, 3, 0), Action: Pour 3L from jug 3 to jug 2
State: (2, 3, 3), Action: Pour 3L from jug 1 to jug 3
State: (2, 5, 1), Action: Pour 2L from jug 3 to jug 2
State: (7, 0, 1), Action: Pour 5L from jug 2 to jug 1
State: (7, 1, 0), Action: Pour 1L from jug 3 to jug 2
State: (4, 1, 3), Action: Pour 3L from jug 1 to jug 3
Goal state reached: (4, 1, 3)
Total states explored: 16
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/Decantation_Problem - Assignment
- 3.py"
Enter the goal state as three integers (e.g., 4 1 3): 4 4 0
Solution found:
State: (8, 0, 0), Action: None
State: (3, 5, 0), Action: Pour 5L from jug 1 to jug 2
State: (3, 2, 3), Action: Pour 3L from jug 2 to jug 3
State: (6, 2, 0), Action: Pour 3L from jug 3 to jug 1
```

```
State: (6, 0, 2), Action: Pour 2L from jug 2 to jug 3
State: (1, 5, 2), Action: Pour 5L from jug 1 to jug 2
State: (1, 4, 3), Action: Pour 1L from jug 2 to jug 3
State: (4, 4, 0), Action: Pour 3L from jug 3 to jug 1
Goal state reached: (4, 4, 0)
Total states explored: 15
```

Assignment 4

Date : 29/08/2024

Problem Description :

Place 8 queens “safely” in a 8×8 chessboard – no queen is under attack from any other queen (in horizontal, vertical and diagonal directions). Formulate it as a constraint satisfaction problem.

- One queen is placed in each column.
- Variables are the rows in which queens are placed in the columns
- Assignment: 8 row indexes.
- Evaluation function: the number of attacking pairs in 8-queens Implement a local search algorithm to find one safe assignment.

Algorithm:

1. Local Search

```
Input: problem
Output: solution, or failure

current ← initial state of problem
while true do
    neighbors ← generate neighbors of current
    best_neighbor ← find the best state in neighbors

    if best_neighbor is better than current then
        current ← best_neighbor
    else
        return current as solution
```

2. Stochastic Search

```
Input: problem
Output: solution, or failure

current ← initial solution of problem
while stopping criteria not met do
    if current is a valid solution then
        return current as solution

    neighbor ← randomly select a neighbor of current
    neighbor_value ← evaluate(neighbor)

    if neighbor_value < evaluate(current) then
        current ← neighbor
    else
        if random() < acceptance_probability(current, neighbor_value) then
```

```
            current ← neighbor
return failure
```

Code :

```
from random import randint

# Set the board size (8x8)
N = 8

# Randomly configure the board with queens at random positions
def configureRandomly(board, state):
    for i in range(N):
        state[i] = randint(0, 100000) % N
        board[state[i]][i] = 1

def printBoard(board):
    for i in range(N):
        print(' '.join("Q" if board[i][j] == 1 else "." for j in range(N)))

# Print the state (positions of queens on the board)
def printState(state):
    print(*state)

# Compare two states to see if they are the same
def compareStates(state1, state2):
    for i in range(N):
        if state1[i] != state2[i]:
            return False
    return True

# Fill the board with a given value (used to reset the board)
def fill(board, value):
    for i in range(N):
        for j in range(N):
            board[i][j] = value

# Calculate the number of attacking queens on the board
def calculateObjective(board, state):
    attacking = 0
    for i in range(N):
        row = state[i]

        # Check left in the row
        col = i - 1
        while col >= 0 and board[row][col] != 1:
            col -= 1
        if col >= 0 and board[row][col] == 1:
            attacking += 1

        # Check right in the row
        col = i + 1
```

```
        while col < N and board[row][col] != 1:
            col += 1
        if col < N and board[row][col] == 1:
            attacking += 1

        # Check upper-left diagonal
        row = state[i] - 1
        col = i - 1
        while col >= 0 and row >= 0 and board[row][col] != 1:
            col -= 1
            row -= 1
        if col >= 0 and row >= 0 and board[row][col] == 1:
            attacking += 1

        # Check lower-right diagonal
        row = state[i] + 1
        col = i + 1
        while col < N and row < N and board[row][col] != 1:
            col += 1
            row += 1
        if col < N and row < N and board[row][col] == 1:
            attacking += 1

        # Check lower-left diagonal
        row = state[i] + 1
        col = i - 1
        while col >= 0 and row < N and board[row][col] != 1:
            col -= 1
            row += 1
        if col >= 0 and row < N and board[row][col] == 1:
            attacking += 1

        # Check upper-right diagonal
        row = state[i] - 1
        col = i + 1
        while col < N and row >= 0 and board[row][col] != 1:
            col += 1
            row -= 1
        if col < N and row >= 0 and board[row][col] == 1:
            attacking += 1

    return int(attacking / 2)

# Board Creation
def generateBoard(board, state):
    fill(board, 0)
    for i in range(N):
        board[state[i]][i] = 1

def copyState(state1, state2):
    for i in range(N):
        state1[i] = state2[i]

# Get the neighbor of the current state (modifying one queen's position)
```

```
def getNeighbour(board, state):
    opBoard = [[0 for _ in range(N)] for _ in range(N)]
    opState = [0 for _ in range(N)]

    copyState(opState, state)
    generateBoard(opBoard, opState)

    opObjective = calculateObjective(opBoard, opState)

    NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
    NeighbourState = [0 for _ in range(N)]
    copyState(NeighbourState, state)
    generateBoard(NeighbourBoard, NeighbourState)

    for i in range(N):
        for j in range(N):
            if j != state[i]:
                NeighbourState[i] = j
                NeighbourBoard[NeighbourState[i]][i] = 1
                NeighbourBoard[state[i]][i] = 0
                temp = calculateObjective(NeighbourBoard, NeighbourState)

                # If the new state is better or equal, update the state
                if temp <= opObjective:
                    opObjective = temp
                    copyState(opState, NeighbourState)
                    generateBoard(opBoard, opState)

                # Undo the move and restore the original state
                NeighbourBoard[NeighbourState[i]][i] = 0
                NeighbourState[i] = state[i]
                NeighbourBoard[state[i]][i] = 1

    # Update the current state to the best neighbor found
    copyState(state, opState)
    fill(board, 0)
    generateBoard(board, state)

# Hill climbing algorithm to solve the N-Queens problem
def hillClimbing(board, state):

    neighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
    neighbourState = [0 for _ in range(N)]

    copyState(neighbourState, state)
    generateBoard(neighbourBoard, neighbourState)

    while True:
        copyState(state, neighbourState)
        generateBoard(board, state)
        getNeighbour(neighbourBoard, neighbourState)

        # If the state does not change, print the solution
        if compareStates(state, neighbourState):
```

```
            printBoard(board)
            break

        # If the objective value remains the same, randomize a queen's position
        elif calculateObjective(board, state) ==
calculateObjective(neighbourBoard, neighbourState):
            neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N
            generateBoard(neighbourBoard, neighbourState)

# Function calls
state = [0] * N
board = [[0 for _ in range(N)] for _ in range(N)]
configureRandomly(board, state)

hillClimbing(board, state)
```

Testing :

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/8Queens_Problem - Assignment -
4.py"
. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .
```

Assignment 5

Date : 05/09/2024

Problem Description :

1. Class Variable Define a class Variable consisting of a name and a domain. The domain of a variable is a list or a tuple, as the ordering will matter in the representation of constraints. We would like to create a Variable object, for example, as X = Variable('X', {1,2,3})

2. Class Constraint Define a class Constraint consisting of

- A tuple (or list) of variables called the scope.
- A condition, a Boolean function that takes the same number of arguments as there are variables in the scope. The condition must have a name property that gives a printable name of the function; built-in functions and functions that are defined using def have such a property; for other functions you may need to define this property.
- An optional name We would like to create a Variable object, for example, as Constraint([X,Y],lt) where lt is a function that tests whether the first argument is less than the second one. Add the following methods to the class. def can_evaluate(self, assignment): """ assignment is a variable:value dictionary returns True if the constraint can be evaluated given assignment """ def holds(self,assignment): """returns the value of Constraint evaluated in assignment. precondition: all variables are assigned in assignment, ie self.can_evaluate(assignment) """

3. Class CSP A constraint satisfaction problem (CSP) requires:

- variables: a list or set of variables
- constraints: a set or list of constraints. Other properties are inferred from these:
- var to const is a mapping fromvariables to set of constraints, such that var to const[var] is the set of constraints with var in the scope. Add a method consistent(assignment) to class CSP that returns true if the assignment is consistent with each of the constraints in csp (i.e., all of the constraints that can be evaluated evaluate to true).

We may create a CSP problem, for example, as

X = Variable('X', {1,2,3}) Y = Variable('Y', {1,2,3}) Z = Variable('Z', {1,2,3}) csp0 = CSP("csp0", {X,Y,Z}, [Constraint([X,Y],lt), Constraint([Y,Z],lt)])

The CSP csp0 has variables X, Y and Z, each with domain {1, 2, 3}. The con straints are X < Y and Y < Z.

4. 8-Queens Place 8 queens "safely" in a 8×8 chessboard – no queen is under attack from any other queen (in horizontal, vertical and diagonal directions). Formulate it as a constraint satisfaction problem.

- One queen is placed in each column.
- Variables are the rows in which queens are placed in the columns
- Assignment: 8 row indexes. Represent it as a CSP.

5. Simple DFS Solver Solve CSP using depth-first search through the space of partial assignments. This takes in a CSP problem and an optional variable ordering (a list of the variables in the CSP). It returns a generator of the solutions.

Algorithm:

```
Input: assignment, CSP (Constraint Satisfaction Problem)
Output: solution, or failure

function backtrack(assignment, csp):
    if length of assignment equals number of csp variables then
        return assignment

    unassigned ← variables in csp not in assignment
    var ← first variable in unassigned

    for each value in var.domain do
        new_assignment ← copy of assignment
        new_assignment[var] ← value

        if csp is consistent with new_assignment then
            result ← backtrack(new_assignment, csp)
            if result is not None then
                return result

    return None
```

Code :

```
# 1) Class Variable
class Variable:
    def __init__(self, name, domain):
        self.name = name
        self.domain = domain

    def __repr__(self):
        return f"Variable({self.name}, {self.domain})"

# 2) Class Constraint
class Constraint:
    def __init__(self, scope, condition, name=None):
        self.scope = scope  # List of variables
        self.condition = condition  # Condition function
        self.name = name if name else condition.__name__  # Default name is the
function's name

    def can_evaluate(self, assignment):
        return all(var in assignment for var in self.scope)

    def holds(self, assignment):
        if not self.can_evaluate(assignment):
            return False
        values = [assignment[var] for var in self.scope]
        return self.condition(*values)
```

```
# 3) Class CSP
class CSP:
    def __init__(self, name, variables, constraints):
        self.name = name
        self.variables = variables
        self.constraints = constraints
        self.var_to_const = {var: set() for var in variables}

        for constraint in constraints:
            for var in constraint.scope:
                self.var_to_const[var].add(constraint)

    def consistent(self, assignment):
        # Returns True if the assignment is consistent with all constraints.
        for constraint in self.constraints:
            if constraint.can_evaluate(assignment) and not
constraint.holds(assignment):
                return False
        return True

# 4) 8 - Queens
# Not attacking condition
def not_attacking(row1, col1, row2, col2):
    return row1 != row2 and abs(row1 - row2) != abs(col1 - col2)

def different_rows(*args):
    return len(set(args)) == len(args)

# Defining the constraints
columns = range(8)
variables = [Variable(f'Q{col}', list(range(8))) for col in columns]

constraints = []
constraints.append(Constraint(variables, different_rows))

# Add constraints for diagonal attacks
for col1 in columns:
    for col2 in range(col1 + 1, 8):
        constraints.append(
            Constraint(
                [variables[col1], variables[col2]],
                lambda row1, row2, col1=col1, col2=col2: not_attacking(row1, col1,
row2, col2)
            )
        )

# Create CSP for 8-Queens
csp_8_queens = CSP("8-Queens", variables, constraints)

# 5) Simple DFS Solver
# DFS Solution
def dfs_solver(csp, variable_order=None):
    if variable_order is None:
```

```
        variable_order = list(csp.variables)

    def backtrack(assignment):
        if len(assignment) == len(csp.variables):
            yield assignment
            return

        var = next(v for v in variable_order if v not in assignment)
        for value in var.domain:
            local_assignment = assignment.copy()
            local_assignment[var] = value

            if csp.consistent(local_assignment):
                yield from backtrack(local_assignment)

    return backtrack({})

# Printing the solution
def print_8_queens_solution(solution):
    board = [['.'] * 8 for _ in range(8)]
    for var, row in solution.items():
        col = int(var.name[1])
        board[row][col] = 'Q'

    for row in board:
        print(" ".join(row))
    print("\n")

solutions = dfs_solver(csp_8_queens)

for idx, solution in enumerate(solutions, 1):
    word = str(input(("Enter Yes to see the next solution, else Enter No to exit
the program: ")))
    if word == "Yes":
        print(f"Solution {idx}:")
        print_8_queens_solution(solution)
    elif word == "No":
        print("Exiting")
        break
    else:
        print("Invalid Command")
```

Testing :

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/CSP_8Queens_Problem - Assignment
- 5.py"
Enter Yes to see the next solution, else Enter No to exit the program: Yes
Solution 1:
Q . . . . . .
```

```
. . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .


Enter Yes to see the next solution, else Enter No to exit the program: Yes
Solution 2:
Q . . . . . . .
. . . . . Q .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .


Enter Yes to see the next solution, else Enter No to exit the program: Yes
Solution 3:
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .


Enter Yes to see the next solution, else Enter No to exit the program: No
Exiting
PS C:\Users\Mugilkrishna D U>
```

Assignment 6

Date : 05/09/2024

Problem Description :

Consider two-player zero-sum games, where a player only wins when another player loses. This can be modeled with a single utility which one agent (the maximizing agent) is trying maximize and the other agent (the minimizing agent) is trying to minimize. Define a class Node to represent a node in a game tree.

class Node(Displayable): """A node in a search tree. It has a name a string isMax is True if it is a maximizing node, otherwise it is minimizing node children is the list of children value is what it evaluates to if it is a leaf. """ Create the game tree given below:

1. Implement minimax algorithm for a zero-sum two player game as a function minimax(node, depth). Let minimax(node, depth) return both the score and the path. Test it on the game tree you have created.
2. Modify the minimax function to include αβ-pruning.

Algorithm:

1. Without Alpha-beta pruning

```
function minimax(node):
    if node is a leaf then
        return evaluate(node), None

    if node is a maximizing node then
        max_score ← -∞
        max_path ← None
        for each child in node.children() do
            score, path ← minimax(child)
            if score > max_score then
                max_score ← score
                max_path ← (child.name, path)

        return max_score, max_path

    else  // node is a minimizing node
        min_score ← ∞
        min_path ← None
        for each child in node.children() do
            score, path ← minimax(child)
            if score < min_score then
                min_score ← score
                min_path ← (child.name, path)

        return min_score, min_path
```

2. Without Alpha-beta pruning

```
function minimax(node, alpha, beta):
    if node is a leaf then
        return evaluate(node), None

    if node is a maximizing node then
        max_path ← None
        for each child in node.children() do
            score, path ← minimax(child, alpha, beta)
            if score >= beta then
                return score, None
            if score > alpha then
                alpha ← score
                max_path ← (child.name, path)
        return alpha, max_path

    else  // node is a minimizing node
        min_path ← None
        for each child in node.children() do
            score, path ← minimax(child, alpha, beta)
            if score <= alpha then
                return score, None
            if score < beta then
                beta ← score
                min_path ← (child.name, path)
        return beta, min_path
```

Code :

```
# Class Node to initialize, add children and evaluate inequality
class Node:
    def __init__(self, name, isMax, value=None, range_value=None):
        self.name = name
        self.isMax = isMax
        self.value = value
        self.range_value = range_value  # Inequality attribute
        self.children = []  # Children Node List

    def add_child(self, child_node):
        self.children.append(child_node)

    def evaluate(self):
        if self.value is not None:
            return self.value
        elif self.range_value is not None:
            if self.range_value.startswith('<='):
                return float(self.range_value[2:])
            elif self.range_value.startswith('>='):
                return float(self.range_value[2:])
        return float('-inf')

# 1) Minimax algorithm for a zero-sum two player game as a function minimax(node,
```

```
depth).

def minimax(node, depth):
    if not node.children or depth == 0:
        return node.evaluate(), [node.name]

    if node.isMax:
        max_eval = float('-inf')
        best_path = []
        for child in node.children:
            eval, path = minimax(child, depth - 1)
            if eval > max_eval:
                max_eval = eval
                best_path = path
        return max_eval, [node.name] + best_path
    else:
        min_eval = float('inf')
        best_path = []
        for child in node.children:
            eval, path = minimax(child, depth - 1)
            if eval < min_eval:
                min_eval = eval
                best_path = path
        return min_eval, [node.name] + best_path

# 2) Minimax with alpha-beta pruning including inequality handling

def minimax_alpha_beta(node, depth, alpha=float('-inf'), beta=float('inf')):
    if not node.children or depth == 0:  # Leaf node or depth limit
        return node.evaluate(), [node.name]

    if node.isMax:
        max_eval = float('-inf')
        best_path = []
        for child in node.children:
            eval, path = minimax_alpha_beta(child, depth - 1, alpha, beta)
            if eval > max_eval:
                max_eval = eval
                best_path = path
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # β cutoff
        return max_eval, [node.name] + best_path
    else:
        min_eval = float('inf')
        best_path = []
        for child in node.children:
            eval, path = minimax_alpha_beta(child, depth - 1, alpha, beta)
            if eval < min_eval:
                min_eval = eval
                best_path = path
            beta = min(beta, eval)
            if beta <= alpha:
                break  # α cutoff
```

```
        return min_eval, [node.name] + best_path

# Game Tree simulation using MiniMax Algorithm and Alpha-Beta Pruning
# Creating the game tree

root = Node("A", isMax=True, value=5)
b = Node("B", isMax=False, value=5)
c = Node("C", isMax=False, value=2)
d = Node("D", isMax=True, value=5)
e = Node("E", isMax=True, value=6)
f = Node("F", isMax=True, value=2)
g = Node("G", isMax=True)
h = Node("H", isMax=False, value=3)
i = Node("I", isMax=False, value=5)
j = Node("J", isMax=False, value=6)
k = Node("K", isMax=False, value=9)
l = Node("L", isMax=False, value=1)
m = Node("M", isMax=False, value=2)
n = Node("N", isMax=False, value=0)
o = Node("O", isMax=False, value=-1)

# Constructing the tree
root.add_child(b)
root.add_child(c)
b.add_child(d)
b.add_child(e)
c.add_child(f)
c.add_child(g)
d.add_child(h)
d.add_child(i)
e.add_child(j)
e.add_child(k)
f.add_child(l)
f.add_child(m)
g.add_child(n)
g.add_child(o)

# Minimax Algorithm
score, path = minimax(root, depth=3)
print(f"Optimal Score: {score}")
print(f"Path: {' -> '.join(path)}")

# Alpha-Beta Pruning
score_ab, path_ab = minimax_alpha_beta(root, depth=3)
print(f"Optimal Score with αβ-Pruning: {score_ab}")
print(f"Path with αβ-Pruning: {' -> '.join(path_ab)}")
```

Testing :

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
```

```
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/Minimax_Algorithm - Assignment -
6.py"
Optimal Score: 5
Path: A -> B -> D -> I
Optimal Score with αβ-Pruning: 5
Path with αβ-Pruning: A -> B -> D -> I
```

Assignment 7

Date : 25/09/2024

Problem Description :

1 Knowledge Base

Define a class for Clause. A clause consists of a head (an atom) and a body. A body is represented as a list of atoms. Atoms are represented as strings. class Clause(object): """A definite clause""" def **init**(self,head,body=[]): """clause with atom head and lost of atoms body""" self.head=head self.body = body

Define a class Askable to represent atoms askable from the user. class Askable(object): """An askable atom""" def **init**(self,atom): """clause with atom head and lost of atoms body""" self.atom=atom

Define a class KB to represent a knowldege base. A knowledge base is a list of clauses and askables. In order to make top-down inference faster, create a dictionary that maps each atom into the set of clauses with that atom in the head.

class KB(Displayable): """A knowledge base consists of a set of clauses. This also creates a dictionary to give fast access to the clauses with an atom in head. 1 """ def **init**(self, statements=[]): self.statements = statements self.clauses = ... self.askables = ... self.atom_to_clauses = {} ... def add_clause(self, c): ... def clauses_for_atom(self,a): ...

With Clause and KB classes, we can define a trivial example KB as shown below: triv_KB = KB([Clause('i_am', ['i_think']), Clause('i_think'), Clause('i_smell', ['i_exist'])])

Represent the electrical domain of Example 5.8 of Poole and Macworth.

2 Proof Procedures

1. Implement a bottom-up proof procedure for definite clauses in PL to compute the fixed point consequence set of a knowledge base.
2. Implement a top-down proof procedure prove(kb, goal) for definite clauses in PL. It takes kb, a knowledge base KB and goal as inputs, where goal is a list of atoms. It returns True if kb ⊢ goal.

Algorithm:

1. Top-Down Approach

```
function prove(KB, ans_body, indent=""):
    print(indent + 'yes <- ' + join(ans_body with " & "))

    if ans_body is not empty then
        selected ← ans_body[0]

        if selected is an askable in KB then
            ask user if selected is true
            if user confirms selected is true then
                return prove(KB, ans_body[1:], indent + " ")
            else
```

```
                return False

        else
            for each clause in KB.clauses_for_atom(selected) do
                if prove(KB, clause.body + ans_body[1:], indent + " ") then
                    return True

            return False

    else
        return True
```

2. Bottom-Up Approach

```
function fixed_point(KB):
    fp ← ask_askables(KB)
    added ← True

    while added do
        added ← False  // Indicates if an atom was added this iteration

        for each clause in KB.clauses do
            if clause.head is not in fp and all elements of clause.body are in fp
then
                add clause.head to fp
                added ← True
                print(clause.head, "added to fixed point due to clause:", clause)

    return fp
```

Code :

```
# Class Clause represents a definite clause with a head and a body of atoms
class Clause(object):
    def __init__(self, head, body=[]):
        # Clause with atom head and list of atoms body
        self.head = head
        self.body = body

class Askable(object):
    # Represents atoms that can be queried or asked
    def __init__(self, atom):
        self.atom = atom

class KB:
    # A knowledge base consists of a set of clauses and askables
    def __init__(self, statements=[]):
        self.statements = statements
```

```
        self.clauses = []
        self.askables = []
        self.atom_to_clauses = {}
        self.askable_values = {}  # Stores responses to askable atoms
        for s in statements:
            if isinstance(s, Clause):
                self.add_clause(s)
            elif isinstance(s, Askable):
                self.askables.append(s.atom)

    # Adding new clauses to the knowledge base
    def add_clause(self, c):
        self.clauses.append(c)
        if c.head not in self.atom_to_clauses:
            self.atom_to_clauses[c.head] = []
        self.atom_to_clauses[c.head].append(c)

    # Returning the list of clauses where the atom a is the head
    def clauses_for_atom(self, a):
        return self.atom_to_clauses.get(a, [])

    # Asking the user about askable atoms
    def ask_user(self, atom):
        if atom not in self.askable_values:
            response = input(f"Is '{atom}' true? (yes/no): ").strip().lower()
            self.askable_values[atom] = (response == "yes")
        return self.askable_values[atom]

    # Bottom-Up Proof
    def bottom_up_proof(self):
        known_atoms = set()
        added = True
        while added:
            added = False
            for clause in self.clauses:
                if all(b in known_atoms for b in clause.body) and clause.head not
in known_atoms:
                    known_atoms.add(clause.head)
                    added = True
        return known_atoms

    # Top-Down Proof
    def prove(self, goal):
        def recursive_prove(goals, known_atoms):
            if not goals:
                return True
            current_goal = goals[0]
            if current_goal in known_atoms:
                return recursive_prove(goals[1:], known_atoms)
            if current_goal in self.askables:
                if self.ask_user(current_goal):
                    known_atoms.add(current_goal)
                    return recursive_prove(goals[1:], known_atoms)
                else:
```

```
                    return False
            for clause in self.clauses_for_atom(current_goal):
                if recursive_prove(clause.body + goals[1:], known_atoms):
                    known_atoms.add(current_goal)
                    return True
            return False

        return recursive_prove(goal, set())

# Expanded Knowledge Base with Askable atoms
expanded_KB = KB([
    Clause('a', ['b', 'c', 'g']),
    Clause('b', ['d', 'e']),
    Clause('b', ['g', 'e']),
    Clause('c', ['e']),
    Clause('d'),
    Clause('e'),
    Clause('f', ['a', 'g']),
    Clause('h', ['i']),
    Askable('g'),
    Askable('l')
])

# Bottom-up proof
consequence_set = expanded_KB.bottom_up_proof()
print("Bottom-up Proof Consequence Set:", consequence_set)

# Top-down proof for different goals
goal_1 = ['a']
goal_2 = ['b']
goal_3 = ['f']
goal_4 = ['g']
goal_5 = ['h']

is_provable_1 = expanded_KB.prove(goal_1)
is_provable_2 = expanded_KB.prove(goal_2)
is_provable_3 = expanded_KB.prove(goal_3)
is_provable_4 = expanded_KB.prove(goal_4)
is_provable_5 = expanded_KB.prove(goal_5)

print(f"Top-down Proof for goal {goal_1}: {is_provable_1}")
print(f"Top-down Proof for goal {goal_2}: {is_provable_2}")
print(f"Top-down Proof for goal {goal_3}: {is_provable_3}")
print(f"Top-down Proof for goal {goal_4}: {is_provable_4}")
print(f"Top-down Proof for goal {goal_5}: {is_provable_5}")
```

Testing :

1. Top-Down Approach

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/Proof_Procedures - Assignment -
7.py"
Bottom-up Proof Consequence Set: {'b', 'e', 'd', 'c'}
Is 'g' true? (yes/no): yes
Top-down Proof for goal ['a']: True
Top-down Proof for goal ['b']: True
Top-down Proof for goal ['f']: True
Top-down Proof for goal ['g']: True
Top-down Proof for goal ['h']: False
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/Proof_Procedures - Assignment -
7.py"
Bottom-up Proof Consequence Set: {'c', 'e', 'b', 'd'}
Is 'g' true? (yes/no): no
Top-down Proof for goal ['a']: False
Top-down Proof for goal ['b']: True
Top-down Proof for goal ['f']: False
Top-down Proof for goal ['g']: False
Top-down Proof for goal ['h']: False
PS C:\Users\Mugilkrishna D U>
```

Assignment 8

Date : 25/09/2024

Problem Description :

Inference using Bayesian Network (BN) – Joint Probability Distribution The given Bayesian Network has 5 variables with the dependency between the variables as shown below:

1. The marks (M) of a student depends on:

- Exam level (E): This is a discrete variable that can take two values, (difficult, easy) and
- IQ of the student (I): A discrete variable that can take two values (high, low)

2. The marks (M) will, in turn, predict whether he/she will get admitted (A) to a university.
3. The IQ (I) will also predict the aptitude score (S) of the student.

Write functions to

1. Construct the given DAG representation using appropriate libraries.
2. Read and print the Conditional Probability Table (CPT) for each variable.
3. Calculate the joint probability distribution of the BN using 5 variables. Observation: Write the formula for joint probability distribution and explain each parameter. Justify the answer with the advantage of BN.

Algorithm:

```
Input: Prior probabilities for hypotheses H
Output: Posterior Probability P(H|E)

function bayes_algorithm(P_H, P_E_given_H):
    P_E ← 0
    for each hypothesis H in P_H:
        P_E ← P_E + P(E | H) * P(H)  \

    for each hypothesis H in P_H:
        P_H_given_E[H] ← (P(E | H) * P(H)) / P_E

    return P_H_given_E
```

Code :

```
import networkx as nx
import matplotlib.pyplot as plt

# 1) Constructing the DAG
bn_graph = nx.DiGraph()

# Add nodes for each variable in the Bayesian Network
```

```
bn_graph.add_nodes_from(["IQ Level", "Exam Level", "Marks", "Aptitude Score",
"Admission"])

# Add directed edges based on the specified dependencies
edges = [
    ("Exam Level", "Marks"),
    ("Marks", "Admission"),
    ("IQ Level", "Marks"),
    ("IQ Level", "Aptitude Score")
]
bn_graph.add_edges_from(edges)

plt.figure(figsize=(10, 8))
pos = {
    "IQ Level": (0, 1),
    "Exam Level": (1, 1),
    "Marks": (0.5, 0),
    "Aptitude Score": (0, 0.5),
    "Admission": (0.5, -0.5)
}
nx.draw(
    bn_graph, pos, with_labels=True, node_size=8000, node_color="skyblue",
    font_size=11, font_weight="bold", arrows=True, edge_color="black",
linewidths=3
)

plt.title("Bayesian Network Structure", fontsize=18, fontweight='bold')
plt.grid(False)
plt.axis('off')
plt.show()

# 2) Conditional Probability Tables (CPT)
P_IQ = {0: 0.8, 1: 0.2}  # IQ Level
P_Exam = {0: 0.7, 1: 0.3}  # Exam Level

P_Marks_given_IQ_Exam = {
    (0, 0): {0: 0.6, 1: 0.4},
    (0, 1): {0: 0.1, 1: 0.9},
    (1, 0): {0: 0.5, 1: 0.5},
    (1, 1): {0: 0.8, 1: 0.2}
}

P_Aptitude_given_IQ = {
    0: {0: 0.75, 1: 0.25},
    1: {0: 0.4, 1: 0.6}
}

P_Admission_given_Marks = {
    0: {0: 0.6, 1: 0.4},
    1: {0: 0.9, 1: 0.1}
}

# Function to print CPT
def print_cpt(table, variable_name, given_vars=None):
```

```
    print(f"\nConditional Probability Table for {variable_name}")
    if given_vars:
        print(f"Given {given_vars}:")
    if isinstance(list(table.values())[0], dict):
        # Nested dictionary structure
        for condition, outcomes in table.items():
            cond_str = f"{condition}" if isinstance(condition, tuple) else f"
{given_vars}={condition}"
            for outcome, prob in outcomes.items():
                print(f"P({variable_name}={outcome} | {cond_str}) = {prob}")
    else:
        for outcome, prob in table.items():
            print(f"P({variable_name}={outcome}) = {prob}")

print("\nConditional Probability Tables (CPTs):")
print_cpt(P_IQ, "IQ Level")
print_cpt(P_Exam, "Exam Level")
print_cpt(P_Marks_given_IQ_Exam, "Marks", given_vars="IQ Level and Exam Level")
print_cpt(P_Aptitude_given_IQ, "Aptitude Score", given_vars="IQ Level")
print_cpt(P_Admission_given_Marks, "Admission", given_vars="Marks")

# 3) User Input for Joint Probability Distribution Calculation
print("\nPlease enter the values for the following variables:")
iq = int(input("IQ Level (0 for False, 1 for True): "))
exam = int(input("Exam Level (0 for False, 1 for True): "))
marks = int(input("Marks (0 for False, 1 for True): "))
aptitude = int(input("Aptitude Score (0 for False, 1 for True): "))
admission = int(input("Admission (0 for False, 1 for True): "))

# Function for Joint Probability Distribution
def joint_probability(iq, exam, marks, aptitude, admission):
    P_I = P_IQ[iq]
    P_E = P_Exam[exam]
    P_M_given_IE = P_Marks_given_IQ_Exam[(iq, exam)][marks]
    P_A_given_I = P_Aptitude_given_IQ[iq][aptitude]
    P_AD_given_M = P_Admission_given_Marks[marks][admission]

    # Joint probability is the product of all conditional probabilities
    return P_I * P_E * P_M_given_IE * P_A_given_I * P_AD_given_M

probability = joint_probability(iq, exam, marks, aptitude, admission)
print(f"\nJoint Probability P(IQ={iq}, Exam={exam}, Marks={marks}, Aptitude=
{aptitude}, Admission={admission}):", probability)
```

Testing :

```
PS C:\Users\Mugilkrishna D U> & "C:/Users/Mugilkrishna D
U/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Mugilkrishna D
U/Desktop/My Files/SSN/SEM5/FOUNDATIONS OF AI/LAB/BayesNet - Assignment - 8.py"

Conditional Probability Tables (CPTs):
```

```
Conditional Probability Table for IQ Level
P(IQ Level=0) = 0.8
P(IQ Level=1) = 0.2

Conditional Probability Table for Exam Level
P(Exam Level=0) = 0.7
P(Exam Level=1) = 0.3

Conditional Probability Table for Marks
Given IQ Level and Exam Level:
P(Marks=0 | (0, 0)) = 0.6
P(Marks=1 | (0, 0)) = 0.4
P(Marks=0 | (0, 1)) = 0.1
P(Marks=1 | (0, 1)) = 0.9
P(Marks=0 | (1, 0)) = 0.5
P(Marks=1 | (1, 0)) = 0.5
P(Marks=0 | (1, 1)) = 0.8
P(Marks=1 | (1, 1)) = 0.2

Conditional Probability Table for Aptitude Score
Given IQ Level:
P(Aptitude Score=0 | IQ Level=0) = 0.75
P(Aptitude Score=1 | IQ Level=0) = 0.25
P(Aptitude Score=0 | IQ Level=1) = 0.4
P(Aptitude Score=1 | IQ Level=1) = 0.6

Conditional Probability Table for Admission
Given Marks:
P(Admission=0 | Marks=0) = 0.6
P(Admission=1 | Marks=0) = 0.4
P(Admission=0 | Marks=1) = 0.9
P(Admission=1 | Marks=1) = 0.1

Please enter the values for the following variables:
IQ Level (0 for False, 1 for True): 1
Exam Level (0 for False, 1 for True): 0
Marks (0 for False, 1 for True): 0
Aptitude Score (0 for False, 1 for True): 1
Admission (0 for False, 1 for True): 0

Joint Probability P(IQ=1, Exam=0, Marks=0, Aptitude=1, Admission=0):
0.025199999999999997
PS C:\Users\Mugilkrishna D U>
![Bayes Network](bayes_img-2.png)
```

AI GAME REPORT

DEFEND THE NEXUS

Project Report Submitted By

Mugilkrishna D U 3122 22 5001 073

Nikilesh Jayaguptha 3122 22 5001 081

Department of Computer Science and Engineering

1. INTRODUCTION :

1.1 GAME OVERVIEW

Defend the Nexus is an engaging, strategy-based game where players must strategically position and upgrade towers to protect a vital energy source, known as the Nexus, from waves of diverse enemies. The players must prevent the enemies from reaching the Nexus by placing towers along the path. These towers automatically attach to the enemies that pass by. The game emphasizes resource management, requiring players to balance investments in towers and upgrades to survive increasingly difficult waves.

1.2 GAME FEATURES

Dynamic Pathfinding Using A star Algorithm:

The game employs the A* pathfinding algorithm, allowing enemies to find the optimal route from their spawn point to the Nexus. The algorithm calculates paths in real-time, making enemy movement more unpredictable and challenging for the player.

Strategic Tower Placement and Upgrades:

Players can strategically place up to five turrets on designated grid positions (represented by map data) to defend the Nexus. Each turret has customizable properties, such as attack_cooldown, attack_timer, hitpos, and damage, which define their firing rate, range, and impact. Turrets automatically target enemies within range, adding a layer of strategy and requiring players to consider placement and upgrade timing.

Resource Management and Progressively Difficult Waves:

The game introduces a balance between defense and resource allocation. Players must allocate resources wisely, choosing between adding new turrets and upgrading existing ones as waves become progressively more difficult. Enemy health is represented visually through color changes, from BLUE (full health) to BLACK (near defeat), adding feedback for players to adjust their strategies.

Real-Time Enemy Updates and Damage System:

The code includes dynamic, real-time updates for enemy positions, health, and progression along the path. Each enemy's health is modified upon receiving damage from turrets, and if it reaches zero, the enemy is removed from the list, while the turret logs a kill and accumulates damage statistics.

Detailed Turret and Enemy Performance Metrics:

Turrets track individual performance metrics, such as kills and damage dealt, throughout the game. These metrics are displayed at the end of each wave, giving players insights into the effectiveness of their turret placements and damage output. Total kills and damage output are calculated and printed for strategic assessment.

Visual Map Rendering with Customizable Colors:

The game's environment is created using a customizable 2D grid map, rendered with various colors for different map elements, such as GREEN for open paths, BROWN for path sections, RED for the Nexus points, and DARKGRAY for possible turret placements. This intuitive layout helps players plan their defense strategy at a glance.

Real-Time Mouse Interactions for Intuitive Gameplay:

Mouse interactions allow players to select turret positions in real-time. When the left mouse button is clicked, a turret is placed on the specified grid location if it meets game conditions. This simple, interactive control scheme enhances user engagement, enabling quick adjustments and fast-paced decision-making.

Adaptive Enemy Waves with Spawn Management:

The game features an adaptive spawn management system, where enemy waves spawn based on a timer and a maximum enemy limit. The variable spawn_interval controls the frequency of enemy spawns, ensuring a steady increase in difficulty.

1.3 SCOPE OF THE GAME

Game Strategy

Resource Management: Players must allocate resources efficiently, balancing between adding new turrets and upgrading existing ones.

Defensive Positioning: Strategically place turrets along the enemy path to maximize their impact and cover critical areas.

Adaptability: Players adjust strategies to counter different enemy types and escalating wave difficulties.

Player Engagement

Progressive Challenge: Enemy waves increase in complexity, keeping players engaged and continuously challenged.

Real-Time Feedback: Immediate feedback on turret performance and enemy health adds an interactive, dynamic experience.

Rewarding Performance Metrics: Players receive end-of-game stats on turret effectiveness, total damage, and kills, encouraging replay and improvement.

Educational Benefits

Strategic Thinking: Encourages planning and adaptability as players tackle various wave patterns and enemy behaviors.

Pathfinding and AI Awareness: Introduces players to A* pathfinding and basic AI mechanics used in enemy movement.

Math and Logic Skills: Players use logical reasoning and numerical thinking for resource allocation and performance tracking.

Game Customization and Extensions

Adjustable Difficulty: Options for increased enemy health, faster wave intervals, or resource scarcity provide varied challenge levels.

Expansion Potential: Additional turret types, enemy abilities, and customizable maps can extend gameplay and create new strategies.

AI Integration: With more advanced AI opponents, players could practice solo or experiment with different defensive tactics.

2. METHODOLOGY :

2.1 ALGORITHM SELECTION

In creating AI for our competitive game, selecting the appropriate algorithm is essential for achieving optimal AI performance and strategy. For this game, we implemented two distinct approaches to accommodate both User vs AI and AI vs AI gameplay modes. The AI for these modes is built upon advanced search algorithms that allow for efficient and strategic decision-making.

Normal Game Mode (User vs AI): A-star Algorithm In User vs AI mode, we used the A* (A-star) algorithm, a powerful search-based algorithm that is highly effective for decision-making in complex, path-dependent environments.

A star Algorithm: The A* algorithm combines elements of pathfinding with heuristic optimization, making it ideal for scenarios where the AI needs to evaluate the cost of reaching future game states. It works by balancing two factors:

g(n): The actual cost of reaching a specific game state from the initial position. h(n): A heuristic estimate of the cost to reach a favorable end state from the current position.

By combining these two factors, A* prioritizes moves that are likely to lead to a beneficial game outcome, searching paths that appear promising based on both immediate gains and future potential. This makes it a highly effective algorithm for User vs AI gameplay, where the AI's goal is to anticipate the user's moves and navigate the game space toward a winning strategy.

Simulated Mode (AI vs AI): Simulated Annealing For AI vs AI simulations, we use Simulated Annealing, an algorithm that introduces probabilistic variation to allow the AI to explore a broader range of moves and strategies.

Simulated Annealing: This algorithm allows the AI to make decisions with an element of randomness, balancing the pursuit of optimal moves with a tolerance for suboptimal moves. The degree of randomness is controlled by a temperature parameter, which decreases over time to reduce the likelihood of choosing lower-quality moves as the game progresses. This approach enables the AI to explore diverse strategies and adapt dynamically within the game space, creating a varied and realistic AI performance.

3. IMPLEMENTATION :

3.1 DESIGN DETAILS

In our game design, there are three core components that structure the gameplay and user experience across both User vs AI and AI vs AI modes:

1. Game Board Matrix

The primary layout for the game is based on a grid matrix that serves as the playing area. Each position in this grid represents a potential move space, allowing the user and AI to navigate and strategize based on available moves. The dimensions of the matrix may vary based on the game specifications but typically follow a standard square or rectangular layout to facilitate easy navigation and consistent visual symmetry.

2. Move Nodes

Each grid cell within the matrix contains a node that represents an individual move space. These nodes serve as the focal points for both user and AI moves, where each move corresponds to occupying a specific node within the matrix. The node's state is updated visually as either empty, occupied by the user, or occupied by the AI, allowing for clear and immediate feedback to both players.

3. Move Selector and Navigation Interface

A navigation bar or selection indicator is located at the top of the screen, allowing users to select columns (in User vs AI mode) or visualize AI moves (in AI vs AI mode). In User vs AI mode, users interact with this bar to select their move position. The bar dynamically highlights potential move spaces within a column, giving users a clear guide for where their move will be placed. For AI vs AI simulations, the navigation interface updates automatically to display AI move decisions, giving a real-time view of AI actions and strategy throughout the game.

3.3 IMPLEMENTATION OF THE ALGORITHMS

Simulated Annealing

Code

```
import random
import math

def simulated_annealing(board, temp, cooling_rate):
    current_state = board
    current_value = evaluate(current_state)

    while temp > 1:
        neighbor_state = generate_neighbor(current_state)
        neighbor_value = evaluate(neighbor_state)

        # Calculate the acceptance probability
        if neighbor_value > current_value:
            acceptance_probability = 1.0
        else:
            acceptance_probability = math.exp((neighbor_value - current_value) /
```

```
temp)

        # Decide whether to accept the neighbor state
        if acceptance_probability > random.uniform(0, 1):
            current_state = neighbor_state
            current_value = neighbor_value

        # Decrease temperature
        temp *= cooling_rate

    return current_state

def generate_neighbor(state):
    # Generate a neighboring state by randomly moving one piece
    neighbor = [row[:] for row in state]
    # Randomly select a position and modify it slightly
    # Example logic: swap positions, or add/subtract values in the state
    return neighbor

def evaluate(state):
    # Evaluation function for scoring the board state
    score = 0
    # Scoring logic based on board configuration
    return score
```

The Simulated Annealing algorithm, utilized in AI vs AI gameplay mode, starts with a current game state (the board) and iteratively attempts to find a better solution by exploring neighboring states. At each step, it evaluates the neighboring state and decides whether to accept it based on a calculated acceptance probability, which allows it to accept even non-optimal moves early in the process. Over time, as the temperature (temp) decreases, the likelihood of accepting worse states declines, focusing the search on more promising states. This approach helps approximate an optimal solution efficiently by gradually narrowing the search space.

A star Algorithm

Code

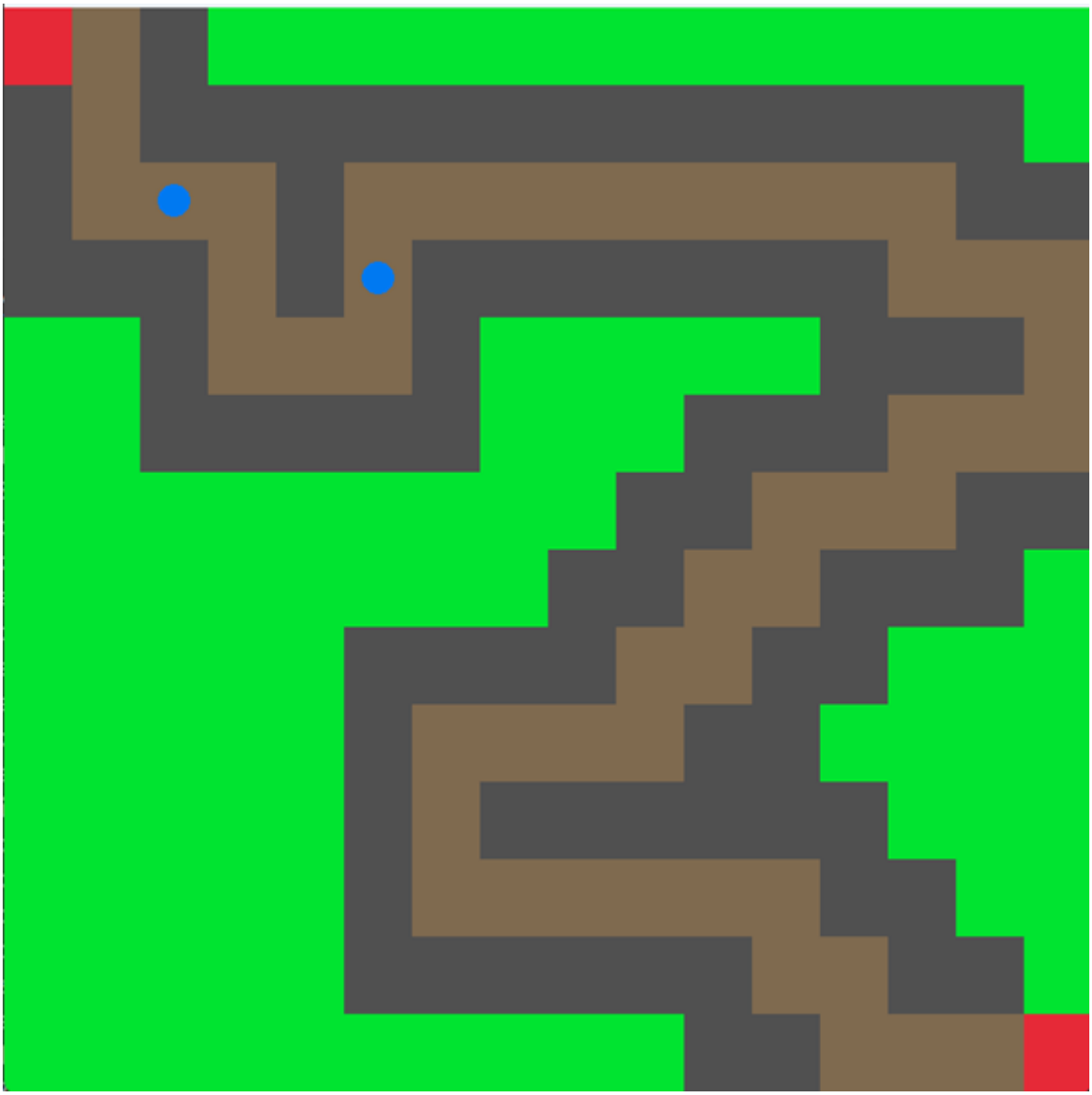
```
import heapq

class GameState:
    def __init__(self, board, g_cost, h_cost, parent=None):
        self.board = board
        self.g_cost = g_cost   # Cost from start to this node
        self.h_cost = h_cost   # Heuristic cost estimate to goal
        self.parent = parent
        self.f_cost = g_cost + h_cost

    def __lt__(self, other):
        return self.f_cost < other.f_cost

def a_star(board, goal_state):
```

```
    open_list = []
    closed_list = set()
    start_state = GameState(board, 0, heuristic(board, goal_state))

    heapq.heappush(open_list, start_state)

    while open_list:
        current_state = heapq.heappop(open_list)

        if is_goal_state(current_state.board, goal_state):
            return reconstruct_path(current_state)

        closed_list.add(str(current_state.board))

        for neighbor in generate_neighbors(current_state):
            if str(neighbor.board) in closed_list:
                continue

            neighbor.g_cost = current_state.g_cost + 1
            neighbor.h_cost = heuristic(neighbor.board, goal_state)
            neighbor.f_cost = neighbor.g_cost + neighbor.h_cost

            heapq.heappush(open_list, neighbor)

    return None

def heuristic(state, goal):
    # Heuristic function to estimate cost to goal
    h_value = 0
    # Calculation based on state
    return h_value

def generate_neighbors(state):
    # Generates neighboring states based on possible moves
    neighbors = []
    # Logic to produce neighboring game states
    return neighbors

def reconstruct_path(state):
    path = []
    while state:
        path.append(state.board)
        state = state.parent
    return path[::-1]
```

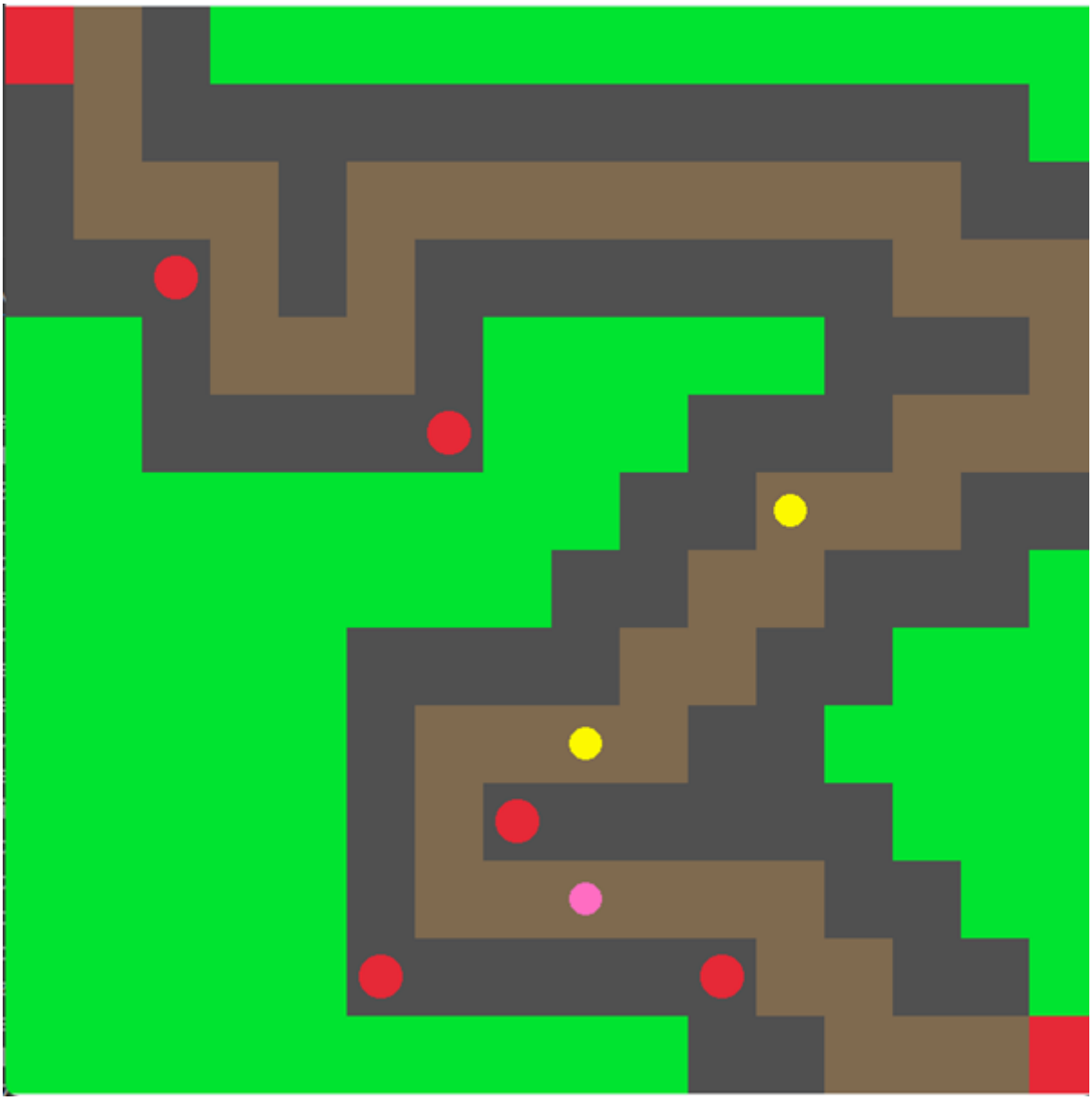
The A* algorithm, implemented for User vs AI gameplay mode, is a pathfinding algorithm that evaluates each possible game state to determine the most efficient move sequence. The algorithm calculates both the cumulative cost from the starting state (g_cost) and an estimated cost to the goal state (h_cost). These values are summed to provide an f_cost, prioritizing states with lower scores. Each potential move generates a new neighbor state, and the algorithm continues until it either reaches the goal state or exhausts options. This approach provides an optimal and efficient search path for the AI to follow in real-time gameplay.

4. GAME EXPERIENCE

Game Environment



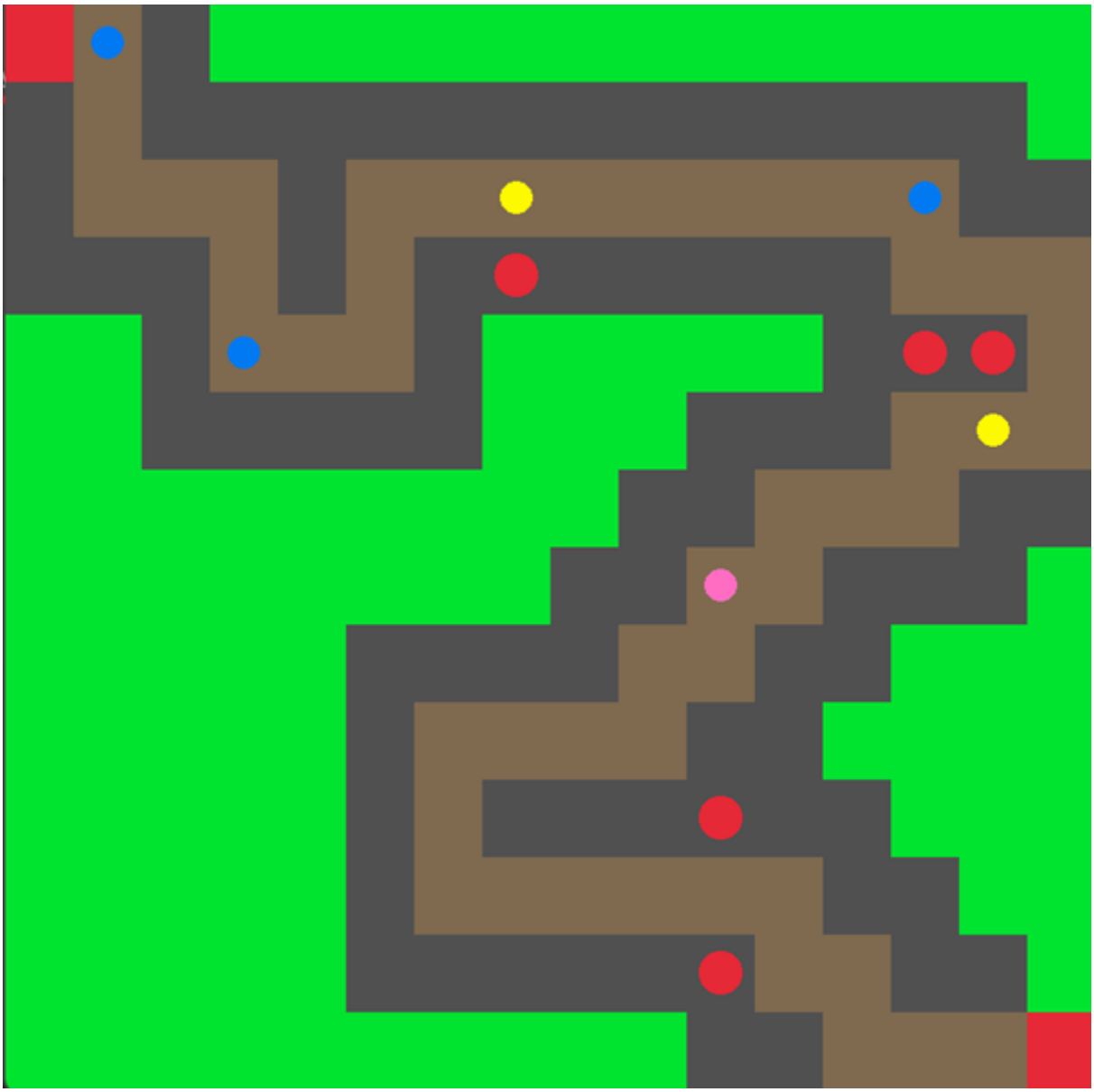
Using A star algorithm - USER VS AI



Terminal

```
[{'position': (3, 2), 'kills': 0, 'damage': 21}, {'position': (5, 6), 'kills': 0,
'damage': 15}, {'position': (12, 10), 'kills': 3, 'damage': 18}, {'position': (10,
7), 'kills': 0, 'damage': 21}, {'position': (12, 5), 'kills': 2, 'damage': 15},
90, 5]
```

Using Simulated Annealing



Terminal

```
[{'position': (1, 10), 'kills': 0, 'damage': 15}, {'position': (1, 0), 'kills': 0,
'damage': 21}, {'position': (12, 13), 'kills': 3, 'damage': 18}, {'position': (10,
10), 'kills': 1, 'damage': 21}, {'position': (9, 10), 'kills': 0, 'damage': 18},
93, 4]
```

5. IMPROVEMENTS

Performance: Enhancing the AI's simulated annealing parameters (cooling rate, initial temperature) and refining A* heuristics can yield more efficient and challenging gameplay. Higher processing capabilities could also support increased depth in evaluations for smarter moves.

AI Difficulty: AI difficulty could be raised by refining the evaluation functions in both algorithms. For simulated annealing, strategic neighbor generation would increase AI competitiveness, while improved heuristics in A* could make the AI more challenging for experienced players.

Adaptability: Implementing dynamic difficulty adjustments based on player performance would allow the AI to respond and scale in real time, making games more engaging for players of varying skill levels.

UI/UX: Improving the interface with clear indicators for current turns, remaining moves, and recent AI choices would enhance clarity. Additionally, highlighting key game events, like winning moves or critical placements, would make gameplay visually informative and immersive.

6. CONCLUSION

In conclusion, integration of A* for normal game mode with simulated annealing for AI vs AI provides a good balance between the nature of dynamics and stability of the game. A* ensures real-time optimal moves for user interaction, while simulated annealing introduces unpredictability along with strategic exploration for opponents in AI games. The play-through of these two algorithms will bring an engaging and adaptable experience to the game. There are opportunities to realize and enhance AI difficulty, performance, and responsiveness without upsetting the game balance. The more the UI elements and optimized algorithms, the more this project evolves into more intelligent strategic play.