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 :

```
from abc import ABC, abstractmethod
class SearchProblem:
    def start_node(self):
        pass
    def is_goal(self):
        pass
    def neighbors(self):
        pass
```

```
    def heuristic(self):
        return 0

class Edge:
    def __init__(self, start, end, cost = 1):
        self.start = start
        self.end = end
        self.cost = cost

    def __repr__(self):
        return f"{self.start} --> {self.end}"

class SearchProblemFromExplicitGraph(SearchProblem):
    def __init__(self, nodes, edges, start, goals = set(), hmap = {}):
        self.nodes = nodes
        self.edges = edges
        self.neighs = {k:[x for x in edges if x.start==k] for k in nodes}
        self.goals = goals
        self.start = start
        self.hmap = hmap

    def start_node(self):
        return self.start

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

    def neighbors(self, node):
        return self.neighs[node]

class Path:
    def __init__(self, node, parent_path=None, edge=None):
        self.node = node
        self.parent_path = parent_path
        self.edge = edge

    def nodes(self):
        if self.parent_path:
            return self.parent_path.nodes() +  (self.node,)
        return (self.node,)

    def __repr__(self):
        if self.parent_path:
            return f"{self.parent_path} --> {self.node}"
        return f"{self.node}"

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

    def search(self):
        while self.frontier:
            print(f"Frontier: {[p for p in self.frontier]}")
```

```
            path = self.frontier.pop(0)
            if self.problem.is_goal(path.node):
                print(f"\nSolution: {path}\n")
            else:
                print(f"Expanding: {path}")
            for edge in self.problem.neighbors(path.node):
                if edge.end not in path.nodes():
                    self.frontier.insert(0, Path(edge.end, path, edge))

        print(f"No more solutions")

nodes_graph1 = {'A', 'B', 'C', 'D', 'G'}
edges_graph1 = [
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_graph1 = 'A'
goals_graph1 = {'G'}
problem_graph1 = SearchProblemFromExplicitGraph(nodes_graph1, edges_graph1,
start_graph1, goals_graph1)
searcher1 = Searcher(problem_graph1)
searcher1.search()
```

Testing :

```
Frontier: [A]
Expanding: A
Frontier: [A --> C, A --> B]
Expanding: A --> C
Frontier: [A --> C --> D, A --> C --> B, A --> B]
Expanding: A --> C --> D
Frontier: [A --> C --> D --> G, A --> C --> B, A --> B]

Solution: A --> C --> D --> G

Frontier: [A --> C --> B, A --> B]
Expanding: A --> C --> B
Frontier: [A --> C --> B --> G, A --> C --> B --> D, A --> B]

Solution: A --> C --> B --> G

Frontier: [A --> C --> B --> D, A --> B]
Expanding: A --> C --> B --> D
Frontier: [A --> C --> B --> D --> G, A --> B]

Solution: A --> C --> B --> D --> G
```

```
Frontier: [A --> B]
Expanding: A --> B
Frontier: [A --> B --> G, A --> B --> D]

Solution: A --> B --> G

Frontier: [A --> B --> D]
Expanding: A --> B --> D
Frontier: [A --> B --> D --> G]

Solution: A --> B --> D --> G

No more solutions
```

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 :

```
import heapq

class SearchProblem:
    def start_node(self):
        pass
    def is_goal(self, node):
        pass
    def neighbors(self, node):
        pass
    def heuristic(self, node):
        return 0

class Edge:
    def __init__(self, start, end, cost = 1):
        self.start = start
        self.end = end
        self.cost = cost

    def __repr__(self):
        return f"{self.start} --> {self.end}"

class SearchProblemFromExplicitGraph(SearchProblem):
    def __init__(self, nodes, edges, start, goals = set(), hmap = {}):
        self.nodes = nodes
        self.edges = edges
        self.neighs = {k:[x for x in edges if x.start==k] for k in nodes}
        self.goals = goals
        self.start = start
        self.hmap = hmap

    def start_node(self):
        return self.start

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

    def neighbors(self, node):
        return self.neighs[node]

class Path:
    def __init__(self, node, parent_path = None, edge = None):
        self.node = node
        self.parent_path = parent_path
        self.edge = edge
```

```
    def __repr__(self):
        if self.parent_path:
            return f"{self.parent_path} --> {self.node}"
        return f"{self.node}"

    def distance(self):
        if self.parent_path:
            return self.edge.cost + self.parent_path.distance()
        return 0

class Searcher:
    def __init__(self, problem):
        self.problem = problem
        self.insert_index = 0
        self.frontier = [(0, self.insert_index, Path(problem.start_node()))]
        self.explored = set()

    def search(self):
        while self.frontier:
            path = heapq.heappop(self.frontier)[2]
            if self.problem.is_goal(path.node):
                return path
            if path.node not in self.explored:
                self.explored.add(path.node)
                for edge in self.problem.neighbors(path.node):
                    new_path = Path(edge.end, path, edge)
                    new_dist = new_path.distance()
                    self.insert_index += 1
                    heapq.heappush(self.frontier, (new_dist, self.insert_index,
new_path))
        return None

problem = SearchProblemFromExplicitGraph(
    nodes = {'A','B','C','D','G'},
    edges = [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'}
)

uniformCostSearcher = Searcher(problem)
solution = uniformCostSearcher.search()
print("Path to Goal Found:")
print(solution)
```

Testing :

```
Path to Goal Found:
A --> C --> B --> D --> G
```

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 :

1. A*

```
import heapq

class Node:
    def __init__(self, state, parent=None):
        self.state = state.copy()
        self.parent = parent

    def compare(self, node):
        for i in range(3):
            for j in range(3):
                if node.state[i][j] != self.state[i][j]:
                    return False
        return True

    def find_blank(self):
        for i in range(3):
            for j in range(3):
                if self.state[i][j] is None:
```

```
                    return i, j

    def manhattan_heuristic(self):
        heuristic = 0
        for row in range(3):
            for col in range(3):
                num = self.state[row][col]
                if num is None:
                    continue
                exp_row = (num - 1) // 3
                exp_col = (num - 1) % 3

                heuristic += abs(exp_row - row) + abs(exp_col - col)
        return heuristic

    def moves(self):
        if self.parent is None:
            return 0
        return self.parent.moves() + 1

    def aStarHeuristic(self):
        return self.moves() + self.manhattan_heuristic()

    def toString(self):
        string = ""
        for i in range(3):
            for j in range(3):
                if self.state[i][j] is None:
                    string += '  '
                else:
                    string += str(self.state[i][j]) + ' '
            string += '\n'
        return string

    def __repr__(self):
        return self.toString()

    def get_path(self):
        string = ""
        if self.parent:
            string += self.parent.get_path() + '\n'
        string += self.toString()
        return string

class Solution:
    def __init__(self, state):
        node = Node(state)
        self.index = 0
        self.frontier = [(node.aStarHeuristic(), self.index, node)]
        self.explored = []
        self.best = None

    def create_node(self, parent, direction, i, j):
        new_state = [parent.state[0].copy(), parent.state[1].copy(),
```

```
parent.state[2].copy()]
        if direction == 'UP':
            new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], None
        elif direction == 'DOWN':
            new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], None
        elif direction == 'RIGHT':
            new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], None
        else:
            new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], None
        return Node(new_state, parent)

    def solve(self):
        while self.frontier:
            # Pop the frontier
            node = heapq.heappop(self.frontier)[2]

            # Check if the heuristic of this node is any better than the best
solution so far
            if(self.best is not None and node.moves() >= self.best.moves()):
                continue

            # Check if this node is already present in explored
            if any(explored_node.compare(node) for explored_node in
self.explored):
                continue

            # Check if it is a goal node
            if node.manhattan_heuristic() == 0:
                if(self.best is None or (self.best.moves() > node.moves())):
                    self.best = node
                continue

            # Otherwise, set as explored
            self.explored.append(node)

            # Add its children into the frontier
            i, j = node.find_blank()

            if i > 0:
                up_node = self.create_node(node, 'UP', i, j)
                self.index += 1
                heapq.heappush(self.frontier, (up_node.aStarHeuristic(),
self.index, up_node))
            if i < 2:
                down_node = self.create_node(node, 'DOWN', i, j)
                self.index += 1
                heapq.heappush(self.frontier, (down_node.aStarHeuristic(),
self.index, down_node))
            if j < 2:
                right_node = self.create_node(node, 'RIGHT', i, j)
                self.index += 1
                heapq.heappush(self.frontier, (right_node.aStarHeuristic(),
self.index, right_node))
            if j > 0:
```

```
                left_node = self.create_node(node, 'LEFT', i, j)
                self.index += 1
                heapq.heappush(self.frontier, (left_node.aStarHeuristic(),
self.index, left_node))

        return self.best


state = [[1, 2, 3], [None, 4, 6], [7, 5, 8]]
solution = Solution(state)

result = solution.solve()
print(result.get_path())
```

2. Greedy Best First Search

```
import heapq

class Node:
    def __init__(self, state, parent=None):
        self.state = state.copy()
        self.parent = parent

    def compare(self, node):
        for i in range(3):
            for j in range(3):
                if node.state[i][j] != self.state[i][j]:
                    return False
        return True

    def find_blank(self):
        for i in range(3):
            for j in range(3):
                if self.state[i][j] is None:
                    return i, j

    def manhattan_heuristic(self):
        heuristic = 0
        for row in range(3):
            for col in range(3):
                num = self.state[row][col]
                if num is None:
                    continue
                exp_row = (num - 1) // 3
                exp_col = (num - 1) % 3

                heuristic += abs(exp_row - row) + abs(exp_col - col)
        return heuristic

    def toString(self):
        string = ""
        for i in range(3):
```

```
            for j in range(3):
                if self.state[i][j] is None:
                    string += '  '
                else:
                    string += str(self.state[i][j]) + ' '
            string += '\n'
        return string

    def __repr__(self):
        return self.toString()

    def get_path(self):
        string = ""
        if self.parent:
            string += self.parent.get_path() + '\n'
        string += self.toString()
        return string

class Solution:
    def __init__(self, state):
        node = Node(state)
        self.index = 0
        self.frontier = [(node.manhattan_heuristic(), self.index, node)]
        self.explored = []

    def create_node(self, parent, direction, i, j):
        new_state = [parent.state[0].copy(), parent.state[1].copy(),
parent.state[2].copy()]
        if direction == 'UP':
            new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j],
new_state[i][j]
        elif direction == 'DOWN':
            new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j],
new_state[i][j]
        elif direction == 'RIGHT':
            new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1],
new_state[i][j]
        else:
            new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1],
new_state[i][j]
        return Node(new_state, parent)

    def solve(self):
        while self.frontier:
            # Pop the frontier
            node = heapq.heappop(self.frontier)[2]

            # Check if this node is already present in explored
            if any(explored_node.compare(node) for explored_node in
self.explored):
                continue

            # Check if it is a goal node
            if node.manhattan_heuristic() == 0:
```

```
                return node

            # Otherwise, set as explored
            self.explored.append(node)

            # Add its children into the frontier
            i, j = node.find_blank()

            if i > 0:
                up_node = self.create_node(node, 'UP', i, j)
                self.index += 1
                heapq.heappush(self.frontier, (up_node.manhattan_heuristic(),
self.index, up_node))
            if i < 2:
                down_node = self.create_node(node, 'DOWN', i, j)
                self.index += 1
                heapq.heappush(self.frontier, (down_node.manhattan_heuristic(),
self.index, down_node))
            if j < 2:
                right_node = self.create_node(node, 'RIGHT', i, j)
                self.index += 1
                heapq.heappush(self.frontier, (right_node.manhattan_heuristic(),
self.index, right_node))
            if j > 0:
                left_node = self.create_node(node, 'LEFT', i, j)
                self.index += 1
                heapq.heappush(self.frontier, (left_node.manhattan_heuristic(),
self.index, left_node))

        return None


state = [[1, 2, 3], [None, 4, 6], [7, 5, 8]]
solution = Solution(state)

result = solution.solve()
if result:
    print(result.get_path())
else:
    print("No solution found")
```

Testing :

1. A*

```
1 2 3
  4 6
7 5 8

1 2 3
4   6
7 5 8
```

```
1 2 3
4 5 6
7   8

1 2 3
4 5 6
7 8
```

2. Greedy Best First Search

```
1 2 3
  4 6
7 5 8

1 2 3
4   6
7 5 8

1 2 3
4 5 6
7   8

1 2 3
4 5 6
7 8
```

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 :

```
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent

    def __repr__(self):
        return str(self.state)

    def get_path(self):
        current = self
        path = []
        while current:
            path.append(current.state)
            current = current.parent
        path.reverse()
        return path

    def next_states(self):
        x, y, z = self.state
        next_states = []

        # Transfer from x to y and z
        if x > 0:
            if y < 5:  # Transfer x to y
                transfer = min(x, 5 - y)
                next_states.append((x - transfer, y + transfer, z))
            if z < 3:  # Transfer x to z
                transfer = min(x, 3 - z)
```

```
                next_states.append((x - transfer, y, z + transfer))

        # Transfer from y to x and z
        if y > 0:
            if x < 8:  # Transfer y to x
                transfer = min(y, 8 - x)
                next_states.append((x + transfer, y - transfer, z))
            if z < 3:  # Transfer y to z
                transfer = min(y, 3 - z)
                next_states.append((x, y - transfer, z + transfer))

        # Transfer from z to x and y
        if z > 0:
            if x < 8:  # Transfer z to x
                transfer = min(z, 8 - x)
                next_states.append((x + transfer, y, z - transfer))
            if y < 5:  # Transfer z to y
                transfer = min(z, 5 - y)
                next_states.append((x, y + transfer, z - transfer))

        return next_states

class Solution:
    def __init__(self):
        self.explored = set()
        self.frontier = []

    def search(self, initial_state):
        start_node = Node(initial_state)
        self.frontier.append(start_node)

        while self.frontier:
            current_node = self.frontier.pop(0)
            state = current_node.state

            if state[0] == 4 or state[1] == 4 or state[2] == 4:
                return current_node.get_path()

            if state not in self.explored:
                self.explored.add(state)
                for next_state in current_node.next_states():
                    if next_state not in self.explored:
                        next_node = Node(next_state, current_node)
                        self.frontier.append(next_node)

        return None  # No solution found

initial_state = (8, 0, 0)
solution = Solution()
path = solution.search(initial_state)
print(*path, sep="\n")
```

Testing :

```
(8, 0, 0)
(3, 5, 0)
(3, 2, 3)
(6, 2, 0)
(6, 0, 2)
(1, 5, 2)
(1, 4, 3)
```

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 :

1. Local Search

```
import random

def init(n):
    l = list(range(n))
    random.shuffle(l)
    return tuple(l)

def neighbors(state, n):
    lst = []
    for i in range(n):
        for j in range(i + 1, n):
            new = state[:i] + (state[j],) + state[i + 1:j] + (state[i],) + state[j
+ 1:]
            lst.append(new)
    return lst

def evaluate(state, n):
    c = 0
    for i in range(n):
        for j in range(i + 1, n):
            if abs(i - j) == abs(state[i] - state[j]):
                c += 1
    return c

def print_board(state):
    n = len(state)
    board = [['.' for _ in range(n)] for _ in range(n)]
    for row, col in enumerate(state):
        board[row][col] = 'Q'
    for row in board:
        print(" ".join(row))
    print("\n")

def local_search(n):
```

```
    cur_state = init(n)
    cur_val = evaluate(cur_state, n)
    print("Initial board:")
    print_board(cur_state)
    while cur_val > 0:
        xx = neighbors(cur_state, n)
        for i in xx:
            x = evaluate(i, n)
            if x < cur_val:
                cur_val = x
                cur_state = i
                print(f"Current state with evaluation {cur_val}:")
                print_board(cur_state)
                break
        else:
            print("\nRandom Restart!\n")
            return local_search(n)
            break
    print("Solution found:")
    print_board(cur_state)
    return cur_state

n = int(input("No. of rows = "))
local_search(n)
```

2. Stochastic Search

```
import random

def no_attacking_pairs(board):
    """ Count the number of pairs of queens that are attacking each other. """
    n = len(board)
    count = 0
    for i in range(n):
        for j in range(i + 1, n):
            if (board[i] == board[j] or
                abs(board[i] - board[j]) == abs(i - j)):
                count += 1
    return count

def possible_successors(conf):
    n = len(conf)
    state_value = {}

    for i in range(n):
        for j in range(n):
            if j != conf[i]:
                x = conf[:i] + [j] + conf[i + 1:]
                ap = no_attacking_pairs(x)
                state_value[ap] = x

    min_conflicts = min(state_value.keys())
```

```
    return state_value[min_conflicts], min_conflicts

def print_board(board):
    """ Display the board with queens as 'Q' and empty spaces as '.' """
    n = len(board)
    for row in range(n):
        line = ""
        for col in range(n):
            if board[row] == col:
                line += "Q "
            else:
                line += ". "
        print(line)
    print("\n")

def random_restart(n):
    global iteration
    iteration += 1
    print(f"\nRandom Restart #{iteration}")
    l = [random.randint(0, n - 1) for _ in range(n)]
    print_board(l)
    return l

def eight_queens(initial):
    conflicts = no_attacking_pairs(initial)
    print("Initial configuration:")
    print_board(initial)

    while conflicts > 0:
        new, new_conflicts = possible_successors(initial)
        if new_conflicts < conflicts:
            conflicts = new_conflicts
            initial = new
            print("New configuration with fewer conflicts:")
            print_board(initial)
        else:
            initial = random_restart(len(initial))
            conflicts = no_attacking_pairs(initial)

    print("Solution found:")
    print_board(initial)
    return initial

iteration = 0
n = int(input('No. of rows = '))
board = random_restart(n)

solution = eight_queens(board)
print("Number of random restarts =", iteration)
print("Final configuration of the board =")
print_board(solution)
```

Testing :

1. Local Search

```
No. of rows = 4
Initial board:
. . . Q
Q . . .
. Q . .
. . Q .


Current state with evaluation 1:
Q . . .
. . . Q
. Q . .
. . Q .


Current state with evaluation 0:
. Q . .
. . . Q
Q . . .
. . Q .


Solution found:
. Q . .
. . . Q
Q . . .
. . Q .
```

2. Stochastic Search

```
No. of rows = 4

Random Restart #1
. . Q .
. . . Q
. . . Q
. Q . .


Initial configuration:
. . Q .
. . . Q
. . . Q
. Q . .


New configuration with fewer conflicts:
. . Q .
```

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


Solution found:
. . Q .
Q . . .
. . . Q
. Q . .


Number of random restarts = 1
Final configuration of the board =
. . 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 :

```
import random
import itertools
```

```
class Variable(object):
    def __init__(self, name, domain, position=None):
        self.name = name
        self.domain = domain
        self.position = position if position else (random.random(),
random.random())
        self.size = len(domain)

    def __str__(self):
        return self.name

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

    def __eq__(self, other):
        if isinstance(other, Variable):
            return self.name == other.name
        return False

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



class Constraint(object):
    def __init__(self, scope, condition, string=None, position=None):
        self.scope = scope
        self.condition = condition
        if string is None:
            self.string = f"{self.condition.__name__}({self.scope})"
        else:
            self.string = string
        self.position = position

    def __repr__(self):
        return self.string

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

    def holds(self, assignment):
        return self.condition(*tuple(assignment[v] for v in self.scope))

class CSP(object):
    def __init__(self, title, variables, constraints):
        self.title = title
        self.variables = variables
        self.constraints = constraints
        self.var_to_const = {var:set() for var in self.variables}

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

```
    def __str__(self):
        return str(self.title)

    def __repr__(self):
        return f"CSP({self.title}, {self.variables}, {([str(c) for c in
self.constraints])}"

    def consistent(self, assignment):
        return all(con.holds(assignment)
                   for con in self.constraints
                   if con.can_evaluate(assignment))


def n_queens_csp(n):
    variables = [Variable(f"Q{i}", list(range(n))) for i in range(n)]

    constraints = []

    for (var1, var2) in itertools.combinations(variables, 2):
        constraints.append(Constraint([var1, var2], lambda x, y: x != y, f"{var1}
!= {var2}"))

    for (var1, var2) in itertools.combinations(variables, 2):
        row1 = int(var1.name[1:])
        row2 = int(var2.name[1:])
        constraints.append(Constraint([var1, var2], lambda x, y, row_diff=abs(row1
- row2): abs(x - y) != row_diff,
                                      f"abs({var1} - {var2}) != {abs(row1 -
row2)}"))

    csp = CSP(f"{n}-Queens Problem", variables, constraints)

    solution = backtrack({}, csp)

    return solution


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

    unassigned = [x for x in csp.variables if x not in assignment]
    var = unassigned[0]

    for value in var.domain:
        new_assignment = assignment.copy()
        new_assignment[var] = value

        if csp.consistent(new_assignment):
            result = backtrack(new_assignment, csp)
            if result is not None:
                return result

    return None
```

```
def print_n_queens_solution(n, solution):
    # Create an empty board
    board = [['.' for _ in range(n)] for _ in range(n)]

    # Place queens on the board
    for var, value in solution.items():
        row = int(var.name[1])
        col = value
        board[row][col] = 'Q'

    # Print the board
    for row in reversed(board):
        print(' '.join(row))
    print()


solution = n_queens_csp(8)
print_n_queens_solution(8, solution)
```

Testing :

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

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. With 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 :

1. Without Alpha-Beta pruning

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.left = None
        self.right = None

def minimax(node, is_maximizing_player, path):
    if node.left is None and node.right is None:
        return node.value

    if is_maximizing_player:
        left = minimax(node.left, False, path)
        right = minimax(node.right, False, path)

        if left > right:
            path.append("left")
            return left
        else:
            path.append("right")
            return right
    else:
        left = minimax(node.left, True, path)
        right = minimax(node.right, True, path)

        if left < right:
            path.append("left")
            return left
        else:
            path.append("right")
            return right

def print_path(path):
    print(" -> ".join(path[::-1]))

root = Node()
```

```
root.left = Node()
root.right = Node()

root.left.left = Node(3)
root.left.right = Node(2)
root.right.left = Node(5)
root.right.right = Node(7)

path = []

root_value = minimax(root, True, path)
print_path(path)
print(root_value)
```

2. With Alpha-Beta pruning

```
class Node:
    def __init__(self, name, isMax, children=None, value=None):
        self.name = name
        self.isMax = isMax
        self.children = children if children is not None else []
        self.value = value

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

    def __repr__(self):
        return f"{self.name} ({'Max' if self.isMax else 'Min'})"

def minimax_alpha_beta(node, depth, alpha=float('-inf'), beta=float('inf')):
    if node.value is not None:  # Leaf node
        return node.value, [node.name]

    if node.isMax:
        best_score = float('-inf')
        best_path = []

        for child in node.children:
            score, path = minimax_alpha_beta(child, depth + 1, alpha, beta)
            if score > best_score:
                best_score = score
                best_path = [node.name] + path
            alpha = max(alpha, best_score)
            if beta <= alpha:  # Beta cut-off
                break
        return best_score, best_path

    else:
        best_score = float('inf')
        best_path = []

        for child in node.children:
```

```
            score, path = minimax_alpha_beta(child, depth + 1, alpha, beta)
            if score < best_score:
                best_score = score
                best_path = [node.name] + path
            beta = min(beta, best_score)
            if beta <= alpha:  # Alpha cut-off
                break
        return best_score, best_path


# Sample game tree for testing
# Create nodes (leaf nodes with values, intermediate nodes without values)
leaf1 = Node(name="Leaf1", isMax=None, value=3)
leaf2 = Node(name="Leaf2", isMax=None, value=5)
leaf3 = Node(name="Leaf3", isMax=None, value=2)
leaf4 = Node(name="Leaf4", isMax=None, value=9)
leaf5 = Node(name="Leaf5", isMax=None, value=0)
leaf6 = Node(name="Leaf6", isMax=None, value=7)
leaf7 = Node(name="Leaf7", isMax=None, value=4)
leaf8 = Node(name="Leaf8", isMax=None, value=1)

# Intermediate nodes
nodeA = Node(name="A", isMax=True, children=[leaf1, leaf2])
nodeB = Node(name="B", isMax=True, children=[leaf3, leaf4])
nodeC = Node(name="C", isMax=False, children=[leaf5, leaf6])
nodeD = Node(name="D", isMax=False, children=[leaf7, leaf8])

# Root node
root = Node(name="Root", isMax=True, children=[nodeA, nodeB, nodeC, nodeD])

# Run minimax with alpha-beta pruning
score_ab, path_ab = minimax_alpha_beta(root, 0)
print("Score:", score_ab)
print("Path:", path_ab)
```

Testing :

1. Without Alpha-Beta pruning

```
right -> left -> right
5
```

2. With Alpha-Beta pruning

```
Score: 9
Path: ['Root', 'B', 'Leaf4']
```

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

3. Implement a bottom-up proof procedure for definite clauses in PL to compute the fixed point consequence set of a knowledge base.

4. 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 :

1. Top-Down Approach

```
class Clause(object):
    def __init__(self, head, body=[]):
        self.head = head
        self.body = body

class Askable(object):
    def __init__(self, atom):
        self.atom = atom

class KB:
    def __init__(self, statements=[]):
        self.statements = statements
        self.clauses = []
        self.askables = []
```

```
        self.atom_to_clauses = {}

        for statement in statements:
            if isinstance(statement, Clause):
                self.add_clause(statement)
            elif isinstance(statement, Askable):
                self.askables.append(statement.atom)

    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)

    def clauses_for_atom(self, a):
        return self.atom_to_clauses.get(a, [])

# Example of creating a knowledge base
triv_KB = KB([
    Clause('i_am', ['i_think']),
    Clause('i_think'),
    Clause('i_smell', ['i_exist'])
])

# Bottom-up proof procedure for definite clauses in PL
def bottom_up_prove(kb):
    known_atoms = set()
    new_atoms = True

    while new_atoms:
        new_atoms = False
        for clause in kb.clauses:
            if clause.head not in known_atoms and all(body_atom in known_atoms for
body_atom in clause.body):
                known_atoms.add(clause.head)
                new_atoms = True

    return known_atoms

# Top-down proof procedure for definite clauses in PL
def top_down_prove(kb, goal):
    def prove_recursive(subgoal):
        if subgoal in known:
            return True
        for clause in kb.clauses_for_atom(subgoal):
            if all(prove_recursive(atom) for atom in clause.body):
                known.add(subgoal)
                return True
        return False

    known = set()
    return all(prove_recursive(g) for g in goal)

# Example usage
```

```
goal = ['i_am']
result = top_down_prove(triv_KB, goal)
print(f"Can the goal {goal} be proved? {result}")
result = bottom_up_prove(triv_KB, goal)
print(f"Can the goal {goal} be proved? {result}")
```

Testing :

1. Top-Down Approach

```
Can the goal ['i_am'] be proved? True
```

2. Bottom-Up Approach

```
Can the goal ['i_am'] be proved? True
```

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 :

```
# Function to calculate and print a sample Joint Probability Distribution
def get_input_and_print_probability():
    print("Enter the states for the following variables (leave blank for
unknown):")
    e_state = input("Exam Level (e) [difficult/easy]: ").strip().lower() or None
    i_state = input("IQ (i) [high/low]: ").strip().lower() or None
    m_state = input("Marks (m) [high/low]: ").strip().lower() or None
    a_state = input("Admission (a) [yes/no]: ").strip().lower() or None
    s_state = input("Aptitude Score (s) [good/poor]: ").strip().lower() or None

    valid_states_e = list(P_e.keys())
    valid_states_i = list(P_i.keys())
    valid_states_m = ['high', 'low']
    valid_states_a = ['yes', 'no']
    valid_states_s = ['good', 'poor']

    states_to_check = {
        'e': valid_states_e if e_state is None else [e_state],
        'i': valid_states_i if i_state is None else [i_state],
        'm': valid_states_m if m_state is None else [m_state],
        'a': valid_states_a if a_state is None else [a_state],
        's': valid_states_s if s_state is None else [s_state],
    }

    total_jpd = 0
    print("\nCalculating JPD for the following combinations:")
    for e in states_to_check['e']:
        for i in states_to_check['i']:
            for m in states_to_check['m']:
                for a in states_to_check['a']:
                    for s in states_to_check['s']:
                        jpd = calculate_jpd(e, i, m, a, s)
                        total_jpd += jpd
                        print(f"P(e={e}, i={i}, m={m}, a={a}, s={s}) = {jpd:.4f}")
    print(f"\nTotal Joint Probability for the given states = {total_jpd:.4f}")

# Print all CPDs in table format
print_cpd_exam_level()
print_cpd_iq()
```

```
print_cpd_marks()
print_cpd_admission()
print_cpd_aptitude_score()

# Print Joint Probability Distribution Formula and Sample Calculation
print_jpd_table()
print_jpd_formula()
# Call the function to get input and print the probability
get_input_and_print_probability()
```

Testing :

```
CPD for Exam Level (e):
+----------------+------+
| e              | P(e) |
+----------------+------+
| difficult      | 0.4  |
| easy           | 0.6  |
+----------------+------+

CPD for IQ (i):
+----------------+------+
| i              | P(i) |
+----------------+------+
| high           | 0.7  |
| low            | 0.3  |
+----------------+------+

CPD for Marks (m):
+----------------+----------------+----------------+----------------+
| e              | i              | P(m=high)      | P(m=low)       |
+----------------+----------------+----------------+----------------+
| difficult      | high           | 0.9            | 0.1            |
| difficult      | low            | 0.4            | 0.6            |
| easy           | high           | 0.6            | 0.4            |
| easy           | low            | 0.1            | 0.9            |
+----------------+----------------+----------------+----------------+

CPD for Admission (a):
+----------------+----------------+----------------+
| m              | P(a=yes)       | P(a=no)        |
+----------------+----------------+----------------+
| high           | 0.8            | 0.2            |
| low            | 0.5            | 0.5            |
+----------------+----------------+----------------+

CPD for Aptitude Score (s):
+----------------+----------------+----------------+
| i              | P(s=good)      | P(s=poor)      |
+----------------+----------------+----------------+
| high           | 0.6            | 0.4            |
| low            | 0.3            | 0.7            |
```

```
+----------------+----------------+----------------+

Joint Probability Distribution Table:
+----------------+----------------+----------------+----------------+-------------
---+-----------------+
| e              | i              | m              | a              | s
| P(e, i, m, a, s)|
+----------------+----------------+----------------+----------------+-------------
---+-----------------+
| difficult      | high           | high           | yes            | good
| 0.1210          |
| difficult      | high           | high           | yes            | poor
| 0.0806          |
| difficult      | high           | high           | no             | good
| 0.0302          |
| difficult      | high           | high           | no             | poor
| 0.0202          |
| difficult      | high           | low            | yes            | good
| 0.0084          |
| difficult      | high           | low            | yes            | poor
| 0.0056          |
| difficult      | high           | low            | no             | good
| 0.0084          |
| difficult      | high           | low            | no             | poor
| 0.0056          |
| difficult      | low            | high           | yes            | good
| 0.0115          |
| difficult      | low            | high           | yes            | poor
| 0.0269          |
| difficult      | low            | high           | no             | good
| 0.0029          |
| difficult      | low            | high           | no             | poor
| 0.0067          |
| difficult      | low            | low            | yes            | good
| 0.0108          |
| difficult      | low            | low            | yes            | poor
| 0.0252          |
| difficult      | low            | low            | no             | good
| 0.0108          |
| difficult      | low            | low            | no             | poor
| 0.0252          |
| easy           | high           | high           | yes            | good
| 0.1210          |
| easy           | high           | high           | yes            | poor
| 0.0806          |
| easy           | high           | high           | no             | good
| 0.0302          |
| easy           | high           | high           | no             | poor
| 0.0202          |
| easy           | high           | low            | yes            | good
| 0.0504          |
| easy           | high           | low            | yes            | poor
| 0.0336          |
| easy           | high           | low            | no             | good
```

```
| 0.0504          |
| easy            | high           | low            | no             | poor
| 0.0336          |
| easy            | low            | high           | yes            | good
| 0.0043          |
| easy            | low            | high           | yes            | poor
| 0.0101          |
| easy            | low            | high           | no             | good
| 0.0011          |
| easy            | low            | high           | no             | poor
| 0.0025          |
| easy            | low            | low            | yes            | good
| 0.0243          |
| easy            | low            | low            | yes            | poor
| 0.0567          |
| easy            | low            | low            | no             | good
| 0.0243          |
| easy            | low            | low            | no             | poor
| 0.0567          |
+----------------+----------------+----------------+----------------+-------------
---+----------------+
Joint Probability Distribution Formula:
P(e, i, m, a, s) = P(e) * P(i) * P(m | e, i) * P(a | m) * P(s | i)

Where:
 P(e): Probability of Exam Level
 P(i): Probability of IQ
 P(m | e, i): Probability of Marks given Exam Level and IQ
 P(a | m): Probability of Admission given Marks
 P(s | i): Probability of Aptitude Score given IQ

Enter the states for the following variables (leave blank for unknown):
Exam Level (e) [difficult/easy]:
IQ (i) [high/low]:
Marks (m) [high/low]: low
Admission (a) [yes/no]:
Aptitude Score (s) [good/poor]: good

Calculating JPD for the following combinations:
P(e=difficult, i=high, m=low, a=yes, s=good) = 0.0084
P(e=difficult, i=high, m=low, a=no, s=good) = 0.0084
P(e=difficult, i=low, m=low, a=yes, s=good) = 0.0108
P(e=difficult, i=low, m=low, a=no, s=good) = 0.0108
P(e=easy, i=high, m=low, a=yes, s=good) = 0.0504
P(e=easy, i=high, m=low, a=no, s=good) = 0.0504
P(e=easy, i=low, m=low, a=yes, s=good) = 0.0243
P(e=easy, i=low, m=low, a=no, s=good) = 0.0243

Total Joint Probability for the given states = 0.1878
```

Mini Project: Dots and Boxes

Date : 12/11/2024

PROBLEM STATEMENT:

The goal of this project is to develop an artificial intelligence (AI) system for "Dots and Boxes," a two-player strategy game. This project focuses on implementing AI techniques to create an intelligent opponent for human players and to enable automated gameplay between AI agents.

Game Description:

Dots and Boxes is played on a grid of dots where players take turns drawing horizontal or vertical lines between adjacent dots. When a player completes a square (box) by drawing the fourth side, they claim it and earn an extra turn. The objective is to claim the most boxes by the end of the game.

Project Components:

Core Game Logic:

Develop the foundational rules and structure of Dots and Boxes, including the game board setup, move validation, and win conditions. Ensure that players alternate turns, with each player earning an additional turn upon completing a box.

AI Integration:

Implement an AI opponent using Alpha-Beta Pruning to optimize decision-making by minimizing possible losses and maximizing gains. Use an enhanced heuristic evaluation function to analyze board states and choose optimal moves, with strategic considerations for different phases of the game.

User Interface:

Design a user-friendly graphical interface with Raylib, allowing human players to play against the AI. Automated Testing for AI Evaluation: Develop scripts for AI-vs-AI matches to assess the performance of different strategies and evaluate effectiveness in various game scenarios.

Project Overview:

This project creates an AI opponent for "Dots and Boxes," a two-player game where players take turns drawing lines on a grid to complete squares. The goal is to claim more squares than the opponent by carefully choosing moves that secure boxes while limiting easy opportunities for the other player. The AI uses smart decision-making methods, such as Alpha-Beta Pruning and Heuristic Evaluation strategies, to play effectively against a human. The board is displayed with red lines for the human player and blue for the AI. Players can choose between playing against the AI or watching two AI opponents play each other.

This project demonstrates how AI can be applied to make strategic moves in turn-based games, providing insights into building challenging, competitive game opponents.

Game Setup:

Board Configuration:

The game is played on a square grid of dots, with board sizes typically set to 3x3, 4x4, or larger, depending on the desired level of complexity. The board is represented as a 2D grid where each line (between dots) is either unmarked, red (human), or blue (AI), indicating which player has claimed that line.

Player Roles:

1. There are two players, designated as the Human and the AI.
2. The Human player, using red lines, always starts the game.
3. Players take turns drawing horizontal or vertical lines between adjacent dots.
4. If a player completes a square (box) by drawing the fourth side, they claim that box and earn an extra turn.
5. Players can either be human-controlled or AI-controlled.
6. The setup allows for Human vs. AI flexible testing and interaction.

AI Configuration:

The AI uses Alpha-Beta Pruning combined with heuristic evaluation to optimize decision-making. This algorithms enable the AI to analyze the board, choose advantageous moves, and play strategically with efficient computation.

KEY FEATURES

Board Representation:

The game board is modeled as a 2D array of dots and lines (horizontal and vertical), where each cell represents a square that can be claimed by either player. Each line segment (horizontal or vertical) is tracked to determine whether it has been filled, allowing quick checking of completed squares.

Player Moves:

The human player (red) and AI (Blue) take turns connecting dots to form lines. When a player completes the fourth line of a square, they claim the box and are rewarded with an extra turn. The AI uses Alpha-Beta Pruning combined with a heuristic evaluation function to analyze the board and decide on optimal moves.

AI Algorithms:

Alpha-Beta Pruning:

This algorithm efficiently searches through possible moves by eliminating less promising branches, allowing the AI to evaluate more options within a limited depth.

Heuristic Evaluation:

The heuristic function evaluates the board by scoring the number of claimed boxes, potential future moves, and control over available lines. This scoring helps the AI prioritize moves that maximize its advantage.

Game Progress and Termination:

The game ends when all boxes on the board are claimed. The player with the most boxes is declared the winner. A graphical notification shows the winner when the game concludes in Human vs. AI mode.

ALGORITHM:

Algorithm Overview

The AI makes decisions by evaluating each possible move using Alpha-Beta Pruning combined with a Heuristic Evaluation Function. The pruning reduces the number of moves the AI evaluates by cutting off branches that won't improve the outcome, while the heuristic function scores the board states based on desirable configurations, such as control of boxes.

Algorithm Details

1. **Alpha-Beta Pruning with Minimax:** Objective: Find the best possible move for the AI while assuming the player also plays optimally. Alpha-Beta Pruning is applied to eliminate moves that will not influence the outcome, making the algorithm faster.
2. **Heuristic Evaluation Function:** Since exploring all moves is infeasible, a heuristic function (findScores) estimates board desirability based on AI score, player score, and control score.

   Score Components:
   AI Score: Increases when the AI claims a box.
   Player Score: Increases when the player claims a box.
   Control Score: Counts the number of unfilled boxes, as potential control over boxes is beneficial for future moves.

Algorithm Steps

```
Define Constants:
    Set MAX_DEPTH to limit the depth of recursive search for the Alpha-Beta
Pruning. This prevents excessive recursion and keeps decision-making responsive.

Move Evaluation with immediate_move:
    For each move, create copies of the current board (h_bars, v_bars, boxes) to
simulate the move without affecting the actual game state.<br>
    Apply the move and update the board with updateBoxes, marking completed boxes
for the current player.<br>
    Recur to the next depth level by calling immediate_move with the opposite
player.

Alpha-Beta Pruning Logic:
    Track the best scores for both the AI (maximizing player) and the opponent
(minimizing player) using alpha and beta.<br>
    For the maximizing player (AI):
        Update best_score if the move improves the score.
        Update alpha with the new best score.
        Prune the branch if alpha >= beta.
    For the minimizing player (player):
        Update best_score if the move reduces the score.
        Update beta with the new best score.
```

```
        Prune the branch if beta <= alpha.

Generate Available Moves (get_available_moves):
    Collect all unfilled horizontal and vertical bars as potential moves.
    Use these moves to simulate different board states in immediate_move.

Heuristic Evaluation Function (findScores):
    Calculate scores for AI and player by counting boxes they control.
    Include a control score that encourages the AI to aim for potential future
moves by favoring unfilled boxes.

Main Decision Function (think_immediate):
    Create copies of the original board to preserve the game state.
    Run immediate_move to get the best move for the AI by evaluating each possible
move and selecting the one with the highest score.
```

Pseudocode Summary

```
function think_immediate(boxes_orig, h_bars_orig, v_bars_orig, all_tex):
    boxes, h_bars, v_bars = copy(board state)
    best_move, _ = immediate_move(0, boxes, h_bars, v_bars, all_tex)
    return best_move
function immediate_move(player_turn, boxes, h_bars, v_bars, all_tex, depth, alpha,
beta):
    if depth == MAX_DEPTH or near endgame:
        return None, findScores(boxes, all_tex)

    best_move, best_score = initialize based on player type
    available_moves = get_available_moves(h_bars, v_bars, all_tex)
    for move in available_moves:
        simulate move on copied board state
        updateBoxes(player_turn, new_boxes, new_h_bars, new_v_bars, all_tex)
        _, score = immediate_move(opposite player, new_boxes, new_h_bars,
new_v_bars, all_tex, depth + 1, alpha, beta)

        if maximizing player:
            if score > best_score: update best_score, best_move
            alpha = max(alpha, best_score)
            if beta <= alpha: break
        else:
            if score < best_score: update best_score, best_move
            beta = min(beta, best_score)
            if beta <= alpha: break

    return best_move, best_score

function findScores(boxes, all_tex):
    ai_score, player_score, control_score = calculate from board state
    return ai_score - player_score + 0.5 * control_score
```

```
function get_available_moves(h_bars, v_bars, all_tex):
    return list of unfilled bars (moves)
```

Key Advantages of This Approach

Efficient Decision-Making: By integrating Alpha-Beta Pruning into the decision-making process, the AI is able to avoid evaluating every possible move within the game tree. Instead, it eliminates branches that will not influence the final decision. This selective search reduces computational demands, allowing the AI to make decisions faster and freeing up resources to focus on high-impact moves. This efficiency is particularly important as the game progresses, when the number of potential moves may increase exponentially, making a full search impractical without pruning.

Strategic Move Selection: The AI's heuristic evaluation function enables it to make more strategic decisions by weighing not only immediate score impacts but also potential control of the board. For example, the AI considers factors such as the number of tiles it has placed and the positioning of those tiles—favoring central control and clusters of tiles that can limit the opponent's options. By assessing each move in light of these strategic factors, the AI is guided toward choices that have the potential to yield greater long-term advantages, making it more likely to seize control of the game and maintain it.

Adaptability: The algorithm's structure allows the AI to adapt dynamically to various game states and opponent strategies. As each move is evaluated based on the current board setup and the opponent's recent actions, the AI is capable of responding in real time, adjusting its strategy to exploit open areas, reinforce its own advantages, and counter the opponent's moves. This adaptability ensures that the AI remains competitive and presents a challenging opponent, as it does not rely solely on predefined moves but rather recalibrates its approach continuously throughout the game. By combining Alpha-Beta Pruning with a nuanced heuristic function that considers both immediate and long-term gains, the AI is equipped to make calculated, strategic decisions that enhance both its efficiency and effectiveness. This approach not only improves game performance but also creates a challenging and engaging experience for the player.

MODULES AND FUNCTIONS:

pyray and raylib Modules

These modules are used for game development, handling graphics, input, and audio. These are the main modules referenced in the code.

```
init_window(screen_width, screen_height, title)
toggle_fullscreen()
init_audio_device()
load_sound(file_name)
load_texture(file_name)
get_mouse_position()
is_mouse_button_down(button)
is_mouse_button_released(button)
clear_background(color)
draw_text(text, x, y, font_size, color)
draw_texture_rec(texture, source_rec, position, color)
begin_drawing()
```

```
end_drawing()
set_target_fps(fps)
window_should_close()
check_collision_point_rec(point, rec)
play_sound(sound)
unload_texture(texture)
unload_sound(sound)
close_audio_device()
close_window()
get_key_pressed()
measure_text(text, font_size)
draw_rectangle_rec(rectangle, color)
draw_rectangle_lines(x, y, width, height, color)
```

PlayerDeets.py module

This module handles input for the player's username and moves to the game board setup screen. It has `playerDeets.py` functions that can be used to perform the tasks.

BoardDeets.py module

The function `boardDeets()` is called when the player enters their name and presses the button. It's responsible for setting up the game board.

Game.py module

This module is responsible for managing the game logic, handling player and AI turns, detecting valid moves, filling completed boxes, and updating the game state. It alternates turns between the human player (Red) and the AI (Blue), checks for collisions when placing lines, and ends the game when all boxes are filled, displaying the final score. The functions provided by this module are `checkCollision()`, `fillBox()`, `game()`.

AI.py module

This module implements a Minimax algorithm with Alpha-Beta Pruning to enable AI decision-making in the Dots and Boxes game. It evaluates the game state by checking available moves, simulating potential moves, and selecting the best move for the AI based on a scoring function. The AI considers up to a maximum depth of 3 for decision-making, balancing between maximizing its score and minimizing the player's score. It provides `think()` function for finding the next move by the computer to play against the human player.

EndGame.py module

The module helps to display the game over the screen at the end of a Dots and Boxes game. The screen shows the results (who won or if it's a tie), the score, and two buttons—Play Again and Quit. The player can choose to restart the game or quit the application. It handles button interactions with mouse hover, press, and release states. It returns `endGame()` function that helps display the results of the game.

SOURCE CODE:

1. Main.py

```
from pyray import *
from raylib import *
from playerDeets import *

def main():
    screen_width = 1920
    screen_height = 1080

    init_window(screen_width, screen_height, "dOts and BOxes")
    toggle_fullscreen()
    init_audio_device()

    fx_button = load_sound("assets/buttonfx.wav")
    button = load_texture("assets/start-button-open.png")

    frame_height = button.height / 3
    source_rec = [0, frame_height, button.width, frame_height]

    btn_bounds = [screen_width / 2 - button.width / 2,
                  screen_height / 2 - button.height / 3 / 2,
                  button.width,
                  frame_height]

    btn_state = 0       # Button state: 0-NORMAL, 1-MOUSE_HOVER, 2-PRESSED
    btn_action = False  # Button action should be activated

    set_target_fps(120)

    while not window_should_close():
        mouse_point = get_mouse_position()
        btn_action = False

        draw_text("WelcOme tO dOts and bOxes :D", screen_width//2-150,
screen_height//2-100, 20, GRAY)

        if check_collision_point_rec(mouse_point, btn_bounds):
            if is_mouse_button_down(MOUSE_BUTTON_LEFT):
                btn_state = 2
            else:
                btn_state = 1

            if is_mouse_button_released(MOUSE_BUTTON_LEFT):
                btn_action = True
        else:
            btn_state = 0

        offset_x = -1 if btn_state != 0 else 0
        offset_y = -1 if btn_state != 0 else 0

        if btn_action:
            play_sound(fx_button)
            clear_background(RAYWHITE)
            playerDeets()
```

```
            break

        begin_drawing()

        clear_background(RAYWHITE)
        draw_texture_rec(button, source_rec, [btn_bounds[0] + offset_x,
btn_bounds[1] + offset_y], WHITE)
        end_drawing()

    unload_texture(button)
    unload_sound(fx_button)

    close_audio_device()
    close_window()

if __name__ == "__main__":
    main()
```

2. PlayerDeets.py

```
from pyray import *
from raylib import *
from boardDeets import *
from game import *

def playerDeets():
    screen_width = 1920
    screen_height = 1080
    set_target_fps(60)

    input_box = Rectangle(screen_width // 2 - 200, screen_height // 2 - 100, 400,
50)
    player_name = ""
    active = False
    color_inactive = LIGHTGRAY
    color_active = DARKGRAY
    color = color_inactive

    # Load textures
    fx_button = load_sound("assets/buttonfx.wav")
    button_disabled = load_texture("assets/enter-button-disabled.png")
    button_open = load_texture("assets/enter-button-open.png")
    button = button_disabled
    btn_change = True

    frame_height = button.height / 3
    source_rec = [0, frame_height, button.width, frame_height]

    btn_bounds = [screen_width // 2 - button.width // 2 + 10,
                  screen_height // 2 - 50,
                  button.width,
```

```
                frame_height]

    click_text_x, click_text_y = screen_width // 2 - 115, screen_height // 2 - 150
    enter_text_x, enter_text_y = screen_width // 2 - 115, screen_height // 2 - 130

    btn_state = 0       # Button state: 0-NORMAL, 1-MOUSE_HOVER, 2-PRESSED
    btn_action = False  # Button action should be activated

    while not window_should_close():
        button = button_disabled
        if check_collision_point_rec(get_mouse_position(), input_box):
            if is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
                active = not active
            color = color_active if active else color_inactive
        else:
            if is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
                active = not active
                color = color_inactive

        if active:
            key = get_key_pressed()
            if key == KEY_BACKSPACE:
                player_name = player_name[:-1]
            elif 32 <= key <= 126:
                player_name += chr(key)

        if player_name:
            button = button_open

        mouse_point = get_mouse_position()
        btn_action = False
        if check_collision_point_rec(mouse_point, btn_bounds):
            if is_mouse_button_down(0):
                btn_state = 2
            else:
                btn_state = 1

            if is_mouse_button_released(0):
                btn_action = True
        else:
            btn_state = 0

        if btn_action and player_name:
            clear_background(RAYWHITE)
            play_sound(fx_button)
            boardDeets(player_name)
            break

        offset_x = -1 if btn_state != 0 else 0
        offset_y = -1 if btn_state != 0 else 0

        begin_drawing()
        clear_background(RAYWHITE)
```

```
        draw_text("Click on the input box!", click_text_x, click_text_y, 20, GRAY)
        draw_text("Enter your username:", enter_text_x, enter_text_y, 20, GRAY)
        draw_rectangle_rec(input_box, color)
        draw_rectangle_lines(int(input_box.x), int(input_box.y),
int(input_box.width), int(input_box.height), DARKGRAY)
        draw_text(player_name, int(input_box.x) + 5, int(input_box.y) + 8, 40,
GRAY)

        draw_texture_rec(button, source_rec, [btn_bounds[0] + offset_x,
btn_bounds[1] + offset_y], WHITE)

        if active:
            if (get_time() * 2) % 2 < 1:
                draw_text("_", int(input_box.x) + 8 + measure_text(player_name,
40), int(input_box.y) + 12, 40, GRAY)

        end_drawing()

    unload_texture(button_disabled)
    unload_texture(button_open)
    close_window()
```

3. BoardDeets.py

```
from pyray import *
from raylib import *
from game import *

def boardDeets(player_name):
    rows = 6
    cols = 6

    screen_width = 1920
    screen_height = 1080
    set_target_fps(60)

    input_box_rows = Rectangle(screen_width // 2 - 65, screen_height // 2 - 100,
50, 50)
    input_box_cols = Rectangle(screen_width // 2 + 30, screen_height // 2 - 100,
50, 50)

    rows_text = ""
    cols_text = ""
    active_rows = False
    active_cols = False
    color_inactive = LIGHTGRAY
    color_active = DARKGRAY
    color_rows = color_inactive
    color_cols = color_inactive

    fx_button = load_sound("assets/buttonfx.wav")
```

```
    button_disabled = load_texture("assets/enter-button-disabled.png")
    button_open = load_texture("assets/enter-button-open.png")
    button = button_disabled
    btn_change = True

    frame_height = button.height / 3
    source_rec = [0, frame_height, button.width, frame_height]

    btn_bounds = [screen_width // 2 - button.width // 2 + 10,
                  screen_height // 2 - 50,
                  button.width,
                  frame_height]

    btn_state = 0       # Button state: 0-NORMAL, 1-MOUSE_HOVER, 2-PRESSED
    btn_action = False  # Button action should be activated

    while not window_should_close():
        button = button_disabled
        if is_mouse_button_pressed(0):
            if check_collision_point_rec(get_mouse_position(), input_box_rows):
                active_rows = True
                active_cols = False
                color_rows = color_active if active_rows else color_inactive
                color_cols = color_active if active_cols else color_inactive
            elif check_collision_point_rec(get_mouse_position(), input_box_cols):
                active_cols = True
                active_rows = False
                color_rows = color_active if active_rows else color_inactive
                color_cols = color_active if active_cols else color_inactive
            else:
                active_rows = False
                active_cols = False
                color_rows = color_inactive
                color_cols = color_inactive

        if active_rows:
            key = get_key_pressed()
            if key == KEY_BACKSPACE:
                rows_text = rows_text[:-1]
            elif 32 <= key <= 126:
                rows_text += chr(key)

        if active_cols:
            key = get_key_pressed()
            if key == KEY_BACKSPACE:
                cols_text = cols_text[:-1]
            elif 32 <= key <= 126:
                cols_text += chr(key)

        if rows_text and cols_text and rows_text.isnumeric() and
cols_text.isnumeric():
            button = button_open

        mouse_point = get_mouse_position()
```

```
        btn_action = False
        if check_collision_point_rec(mouse_point, btn_bounds):
            if is_mouse_button_down(0):
                btn_state = 2
            else:
                btn_state = 1

            if is_mouse_button_released(0):
                btn_action = True
        else:
            btn_state = 0

        if btn_action:
            try:
                rows = int(rows_text)
                cols = int(cols_text)
                if 1 <= rows < 10 and 1 <= cols < 10:
                    clear_background(RAYWHITE)
                    play_sound(fx_button)
                    game(player_name, rows, cols)
                    break
            except ValueError:
                print("Enter valid number")

        offset_x = -1 if btn_state != 0 else 0
        offset_y = -1 if btn_state != 0 else 0

        begin_drawing()
        clear_background(RAYWHITE)

        draw_text("Enter number of rows (1-9) x columns (1-9):", screen_width//2-
200, screen_height // 2 - 130, 20, GRAY)
        draw_rectangle_rec(input_box_rows, color_rows)
        draw_rectangle_lines(int(input_box_rows.x), int(input_box_rows.y),
int(input_box_rows.width), int(input_box_rows.height), DARKGRAY)
        draw_text(rows_text, int(input_box_rows.x) + 5, int(input_box_rows.y) + 8,
40, GRAY)

        draw_text("x", screen_width//2-5, screen_height//2-100, 50, GRAY)
        draw_rectangle_rec(input_box_cols, color_cols)
        draw_rectangle_lines(int(input_box_cols.x), int(input_box_cols.y),
int(input_box_cols.width), int(input_box_cols.height), DARKGRAY)
        draw_text(cols_text, int(input_box_cols.x) + 5, int(input_box_cols.y) + 8,
40, GRAY)

        draw_texture_rec(button, source_rec, [btn_bounds[0] + offset_x,
btn_bounds[1] + offset_y], WHITE)

        if active_rows and (get_time() * 2) % 2 < 1:
            draw_text("_", int(input_box_rows.x) + 8 + measure_text(rows_text,
40), int(input_box_rows.y) + 12, 40, GRAY)
        elif active_cols and (get_time() * 2) % 2 < 1:
            draw_text("_", int(input_box_cols.x) + 8 + measure_text(cols_text,
40), int(input_box_cols.y) + 12, 40, GRAY)
```

```
        end_drawing()

    unload_texture(button_disabled)
    unload_texture(button_open)
    close_window()
```

4. Game.py

```
import time
from pyray import *
from raylib import *
from ai import think
from endGame import endGame

def checkCollision(mouse_point, h_bar_pos, v_bar_pos, row_n, col_n):
    for x in range(col_n):
        for y in range(row_n + 1):
            bar_rect = h_bar_pos[x][y]
            if check_collision_point_rec(mouse_point, bar_rect):
                return [0, y, x]

    for x in range(col_n + 1):
        for y in range(row_n):
            bar_rect = v_bar_pos[x][y]
            if check_collision_point_rec(mouse_point, bar_rect):
                return [1, y, x]
    return None

def fillBox(boxes, h_bars, v_bars, row_n, col_n, gray_h_tex, gray_v_tex,
red_win_tex, blue_win_tex, player_turn):
    ok = 0
    for x in range(col_n):
        for y in range(row_n):
            if(boxes[y][x] == None):
                if(h_bars[y][x] != gray_h_tex and h_bars[y+1][x] != gray_h_tex and
v_bars[y][x] != gray_v_tex and v_bars[y][x+1] != gray_v_tex):
                    ok = 1
                    if(player_turn):
                        boxes[y][x] = red_win_tex
                    else:
                        boxes[y][x] = blue_win_tex
    if ok:
        return player_turn
    return not player_turn

def game(player_name, row_n, col_n):
    dot_tex = load_texture("assets/dot.png")
    red_h_tex = load_texture("assets/red-bar-horizontal.png")
    red_v_tex = load_texture("assets/red-bar-vertical.png")
```

```
    blue_h_tex = load_texture("assets/blue-bar-horizontal.png")
    blue_v_tex = load_texture("assets/blue-bar-vertical.png")
    gray_h_tex = load_texture("assets/gray-bar-horizontal.png")
    gray_v_tex = load_texture("assets/gray-bar-vertical.png")

    red_win_tex = load_texture("assets/red-win.png")
    blue_win_tex = load_texture("assets/blue-win.png")

    all_tex = [dot_tex, red_h_tex, red_v_tex, blue_h_tex, blue_v_tex, gray_h_tex,
gray_v_tex, red_win_tex, blue_win_tex]

    screen_width = 1920
    screen_height = 1080

    h_bars = [[gray_h_tex for y in range(col_n)] for x in range(row_n+1)]
    v_bars = [[gray_v_tex for y in range(col_n+1)] for x in range(row_n)]

    start_dot_x = (screen_width//2) - ((col_n//2) * 100) - dot_tex.width//2
    start_dot_y = (screen_height//2) - ((row_n//2) * 100) - dot_tex.height//2

    h_bar_pos = [[Rectangle(start_dot_x + dot_tex.width//2 + x * 100 - 10,
start_dot_y + dot_tex.height//2 + y * 100 - 10 ,95, 30,BLACK) for y in
range(row_n+1)] for x in range(col_n)]
    v_bar_pos = [[Rectangle(start_dot_x + dot_tex.width//2 + x * 100 - 10,
start_dot_y + dot_tex.height//2 + y * 100 - 10, 30, 95,BLACK) for y in
range(row_n)] for x in range(col_n+1)]

    boxes = [[None for y in range(col_n)] for x in range(row_n)]

    red_points = 0
    blue_points = 0
    player_turn = True

    while not window_should_close():

        filled_boxes = 0
        for row in boxes:
            for box in row:
                if(box != None):
                    filled_boxes+=1

        if(filled_boxes == row_n * col_n):
            time.sleep(1.5)
            clear_background(WHITE)
            endGame(player_name, red_points, blue_points)
            break

        player_turn = fillBox(boxes, h_bars, v_bars, row_n, col_n, gray_h_tex,
gray_v_tex, red_win_tex, blue_win_tex, not player_turn)

        mouse_point = get_mouse_position()
        result = None
        if player_turn and is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
            result = checkCollision(mouse_point, h_bar_pos, v_bar_pos, row_n,
```

```
col_n)

        if not player_turn:
            time.sleep(1)
            result = think(boxes, h_bars, v_bars, all_tex)

        if result is not None:
            if result[0] == 0:
                if h_bars[result[1]][result[2]] == gray_h_tex:
                    h_bars[result[1]][result[2]] = red_h_tex if player_turn else
blue_h_tex
                    player_turn = not player_turn
            else:
                if v_bars[result[1]][result[2]] == gray_v_tex:
                    v_bars[result[1]][result[2]] = red_v_tex if player_turn else
blue_v_tex
                    player_turn = not player_turn

        red_points = sum(1 for row in boxes for box in row if box == red_win_tex)
        blue_points = sum(1 for row in boxes for box in row if box ==
blue_win_tex)

        begin_drawing()
        clear_background(RAYWHITE)

        for x in range(col_n+1):
            for y in range(row_n+1):
                pos_x = start_dot_x + x * 100
                pos_y = start_dot_y + y * 100
                draw_texture(dot_tex, pos_x, pos_y, WHITE)

        for x in range(col_n):
            for y in range(row_n+1):
                pos_x = start_dot_x + x * 100 + 50
                pos_y = start_dot_y + y * 100
                draw_texture(h_bars[y][x], pos_x, pos_y, WHITE)

        for x in range(col_n+1):
            for y in range(row_n):
                pos_x = start_dot_x + x * 100
                pos_y = start_dot_y + y * 100 + 50
                draw_texture(v_bars[y][x], pos_x, pos_y, WHITE)

        for x in range(col_n):
            for y in range(row_n):
                if(boxes[y][x] != None):
                    pos_x = start_dot_x + x * 100 + 50
                    pos_y = start_dot_y + y * 100 + 50
                    draw_texture(boxes[y][x], pos_x, pos_y, WHITE)

        if player_turn:
            draw_text("Red Turn", 20, 20, 30, RED)
            draw_text("Blue Turn", 20, 60, 30, GRAY)
        else:
```

```
            draw_text("Red Turn", 20, 20, 30, GRAY)
            draw_text("Blue Turn", 20, 60, 30, BLUE)

        draw_text(f"Red Points: {red_points}", 20, 100, 30, RED)
        draw_text(f"Blue Points: {blue_points}", 20, 140, 30, BLUE)

        end_drawing()
```

5. AI.py

```
import sys
import copy
import math
sys.setrecursionlimit(10**6)

MAX_DEPTH = 3

def updateBoxes(player_turn, boxes, h_bars, v_bars, all_tex):
    [dot_tex, red_h_tex, red_v_tex, blue_h_tex, blue_v_tex, gray_h_tex,
gray_v_tex, red_win_tex, blue_win_tex] = all_tex
    for i in range(len(boxes)):
        for j in range(len(boxes[0])):
            if (h_bars[i][j] != gray_h_tex and h_bars[i + 1][j] != gray_h_tex and
                v_bars[i][j] != gray_v_tex and v_bars[i][j + 1] != gray_v_tex):
                if boxes[i][j] is None:
                    boxes[i][j] = blue_win_tex if player_turn == 0 else
red_win_tex

def findScores(boxes, all_tex):
    [dot_tex, red_h_tex, red_v_tex, blue_h_tex, blue_v_tex, gray_h_tex,
gray_v_tex, red_win_tex, blue_win_tex] = all_tex
    ai_score, player_score = 0, 0
    for row in boxes:
        for ele in row:
            if ele == blue_win_tex:
                ai_score += 1
            elif ele == red_win_tex:
                player_score += 1
    control_score = sum(1 for row in boxes for box in row if box is None)
    return (ai_score - player_score) + 0.5 * control_score

def immediate_move(player_turn, boxes, h_bars, v_bars, all_tex, depth=0,
alpha=float('-inf'), beta=float('inf')):
    if depth == MAX_DEPTH:
        return None, findScores(boxes, all_tex)

    best_move = None
    best_score = float('-inf') if player_turn == 0 else float('inf')
    available_moves = get_available_moves(h_bars, v_bars, all_tex)

    for move in available_moves:
```

```
        new_h_bars = [row[:] for row in h_bars]
        new_v_bars = [row[:] for row in v_bars]
        new_boxes = [row[:] for row in boxes]

        if move[0] == 0:
            new_h_bars[move[1]][move[2]] = all_tex[3] if player_turn == 0 else
all_tex[1]
        else:
            new_v_bars[move[1]][move[2]] = all_tex[4] if player_turn == 0 else
all_tex[2]

        updateBoxes(player_turn, new_boxes, new_h_bars, new_v_bars, all_tex)

        _, score = immediate_move(1 - player_turn, new_boxes, new_h_bars,
new_v_bars, all_tex, depth + 1, alpha, beta)

        if player_turn == 0:
            if score > best_score:
                best_score = score
                best_move = move
            alpha = max(alpha, best_score)
        else:
            if score < best_score:
                best_score = score
                best_move = move
            beta = min(beta, best_score)

        if beta <= alpha:
            break

    return best_move, best_score

def get_available_moves(h_bars, v_bars, all_tex):
    available_moves = []
    for i, row in enumerate(h_bars):
        for j, ele in enumerate(row):
            if ele == all_tex[5]:
                available_moves.append([0, i, j])
    for i, row in enumerate(v_bars):
        for j, ele in enumerate(row):
            if ele == all_tex[6]:
                available_moves.append([1, i, j])
    return available_moves

def think(boxes_orig, h_bars_orig, v_bars_orig, all_tex):
    boxes = [[x for x in row] for row in boxes_orig]
    h_bars = [[x for x in row] for row in h_bars_orig]
    v_bars = [[x for x in row] for row in v_bars_orig]

    best_move, _ = immediate_move(0, boxes, h_bars, v_bars, all_tex)
    return best_move
```

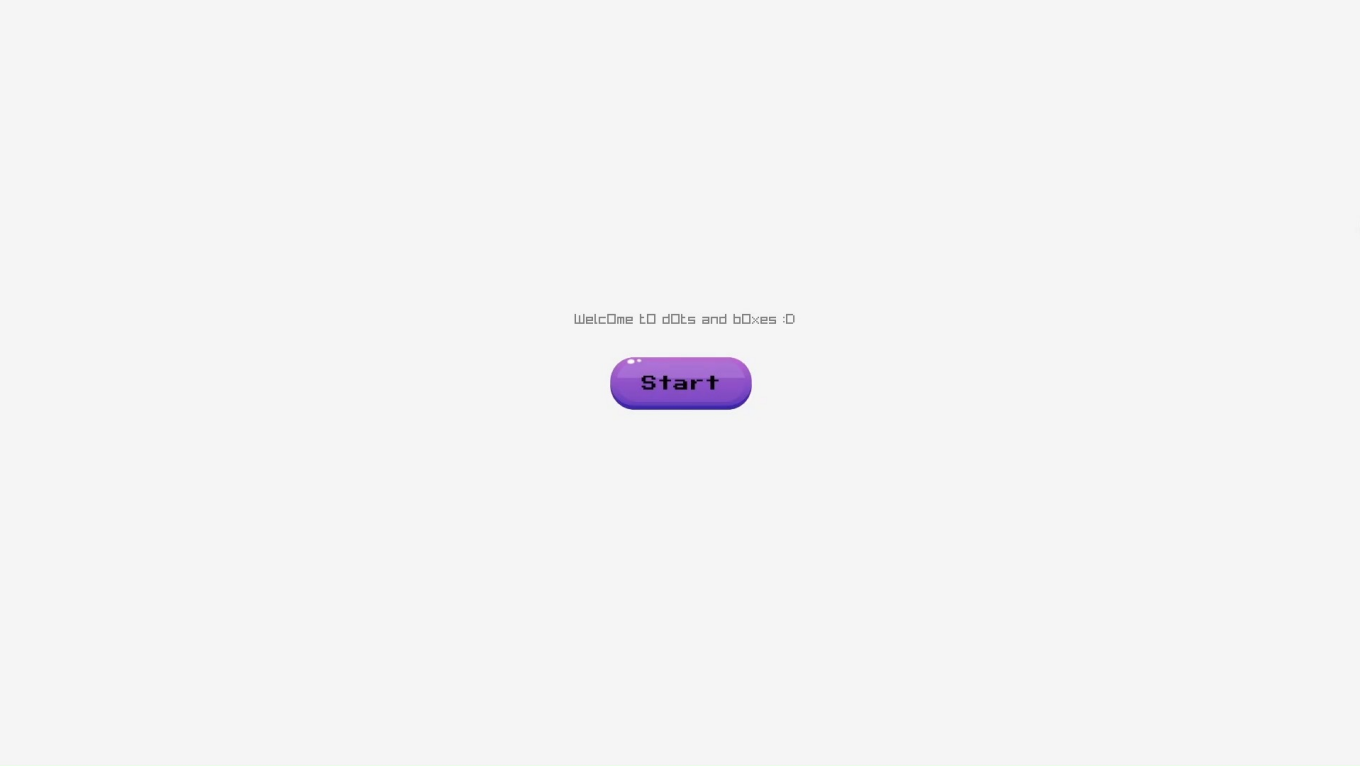
6. EndGame.py

```
import time
from pyray import *
from raylib import *

def endGame(player_name, red_points, blue_points):
    from boardDeets import boardDeets
    screen_width = 1920
    screen_height = 1080

    fx_button = load_sound("assets/buttonfx.wav")
    play_again_button = load_texture("assets/play-again-button.png")
    quit_button = load_texture("assets/quit-button.png")

    button_width = 230
    button_height = 80

    play_again_source_rec = [play_again_button.width/2 - button_width/2,
play_again_button.height/2-button_height/2, button_width, button_height]
    quit_source_rec = [quit_button.width/2- button_width/2, quit_button.height/2-
button_height/2, button_width, button_height]

    play_again_btn_bounds = [screen_width // 2 - button_width // 2,
                             screen_height * 2 / 3 - button_height / 2,
                             button_width, button_height]

    quit_btn_bounds = [screen_width // 2 - button_width // 2,
                       screen_height * 2 / 3 - button_height / 2 + 130,
                       button_width, button_height]

    play_again_btn_state = 0
    quit_btn_state = 0

    set_target_fps(60)

    while not window_should_close():
        mouse_point = get_mouse_position()

        result_text = ""
        if red_points > blue_points:
            result_text = f"{player_name} Wins!"
        elif red_points < blue_points:
            result_text = "AI Wins!"
        else:
            result_text = "It's a Tie!"

        begin_drawing()
        clear_background(RAYWHITE)
        draw_text(f"Game Over", screen_width // 2 - 100, screen_height // 2 - 150,
40, GRAY)
        draw_text(result_text, screen_width // 2 - 100, screen_height // 2 - 100,
```

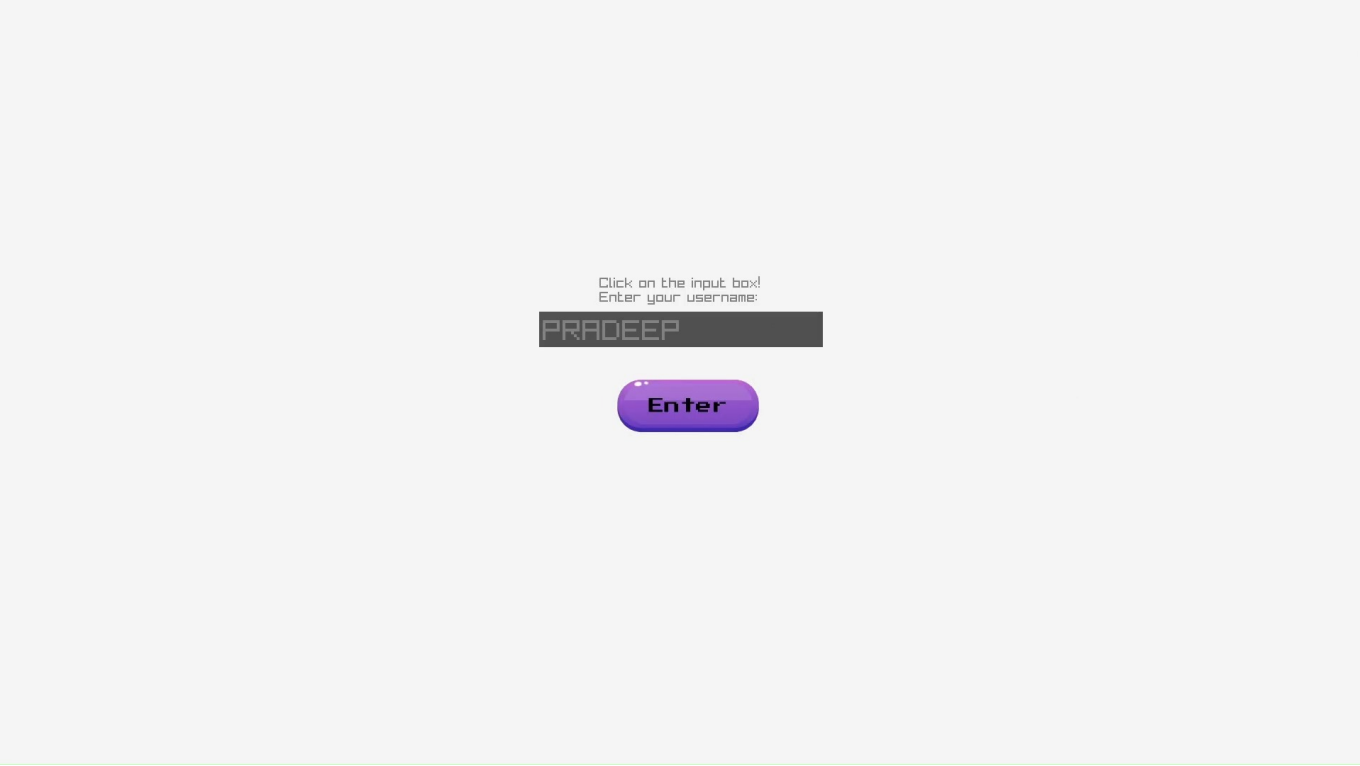
```
30, DARKGRAY)
        draw_text(f"Red Points: {red_points}", screen_width // 2 - 100,
screen_height // 2 - 30, 20, RED)
        draw_text(f"Blue Points: {blue_points}", screen_width // 2 - 100,
screen_height // 2, 20, BLUE)

        if check_collision_point_rec(mouse_point, play_again_btn_bounds):
            if is_mouse_button_down(MOUSE_BUTTON_LEFT):
                play_again_btn_state = 2
            else:
                play_again_btn_state = 1
            if is_mouse_button_released(MOUSE_BUTTON_LEFT):
                clear_background(WHITE)
                boardDeets(player_name)
                break
        else:
            play_again_btn_state = 0

        if check_collision_point_rec(mouse_point, quit_btn_bounds):
            if is_mouse_button_down(MOUSE_BUTTON_LEFT):
                quit_btn_state = 2
            else:
                quit_btn_state = 1
            if is_mouse_button_released(MOUSE_BUTTON_LEFT):
                play_sound(fx_button)
                close_window()
                break
        else:
            quit_btn_state = 0

        offset_x_play_again = -1 if play_again_btn_state != 0 else 0
        offset_y_play_again = -1 if play_again_btn_state != 0 else 0
        draw_texture_rec(play_again_button, play_again_source_rec,
                         [play_again_btn_bounds[0] + offset_x_play_again,
                          play_again_btn_bounds[1] + offset_y_play_again], WHITE)

        offset_x_quit = -1 if quit_btn_state != 0 else 0
        offset_y_quit = -1 if quit_btn_state != 0 else 0
        draw_texture_rec(quit_button, quit_source_rec,
                         [quit_btn_bounds[0] + offset_x_quit,
                          quit_btn_bounds[1] + offset_y_quit], WHITE)

        end_drawing()

    unload_texture(play_again_button)
    unload_texture(quit_button)
    close_window()
```

OUTPUT SCREENSHOTS

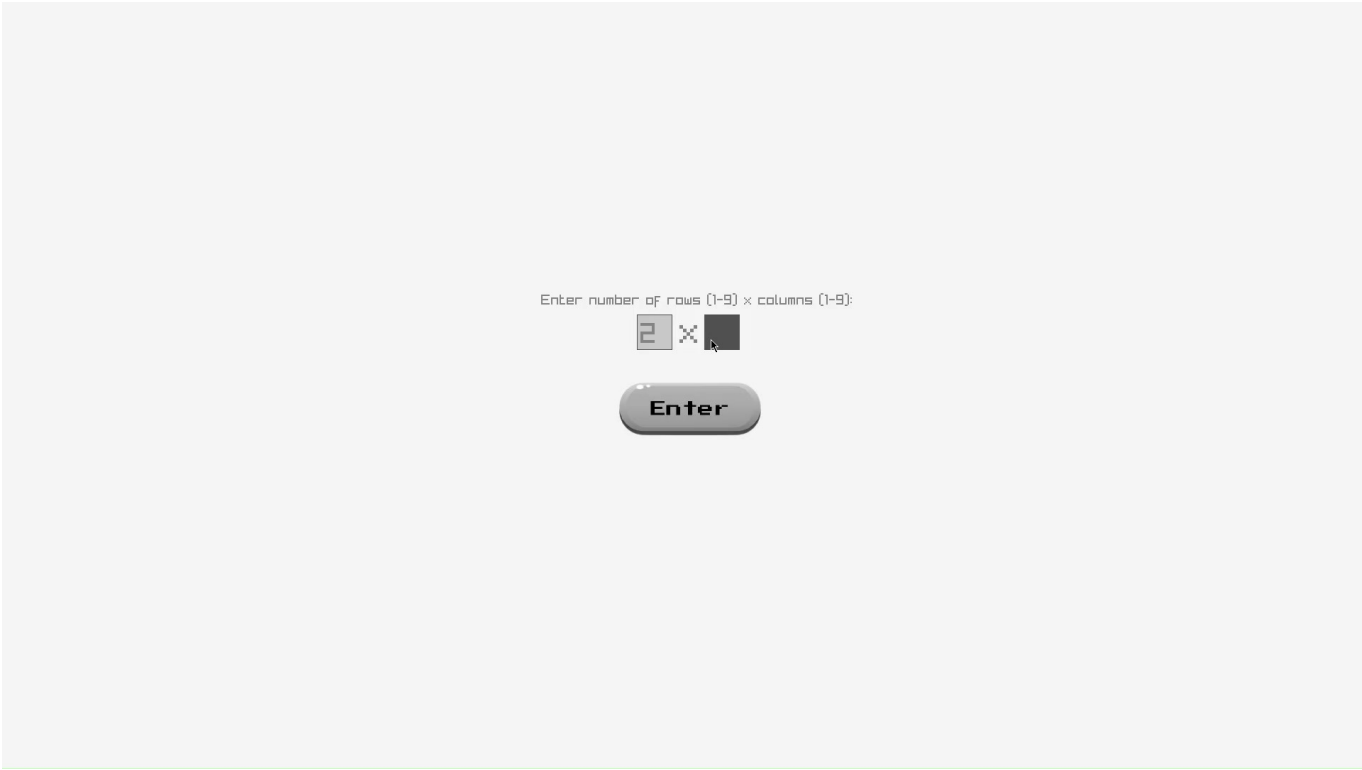
Title Page View



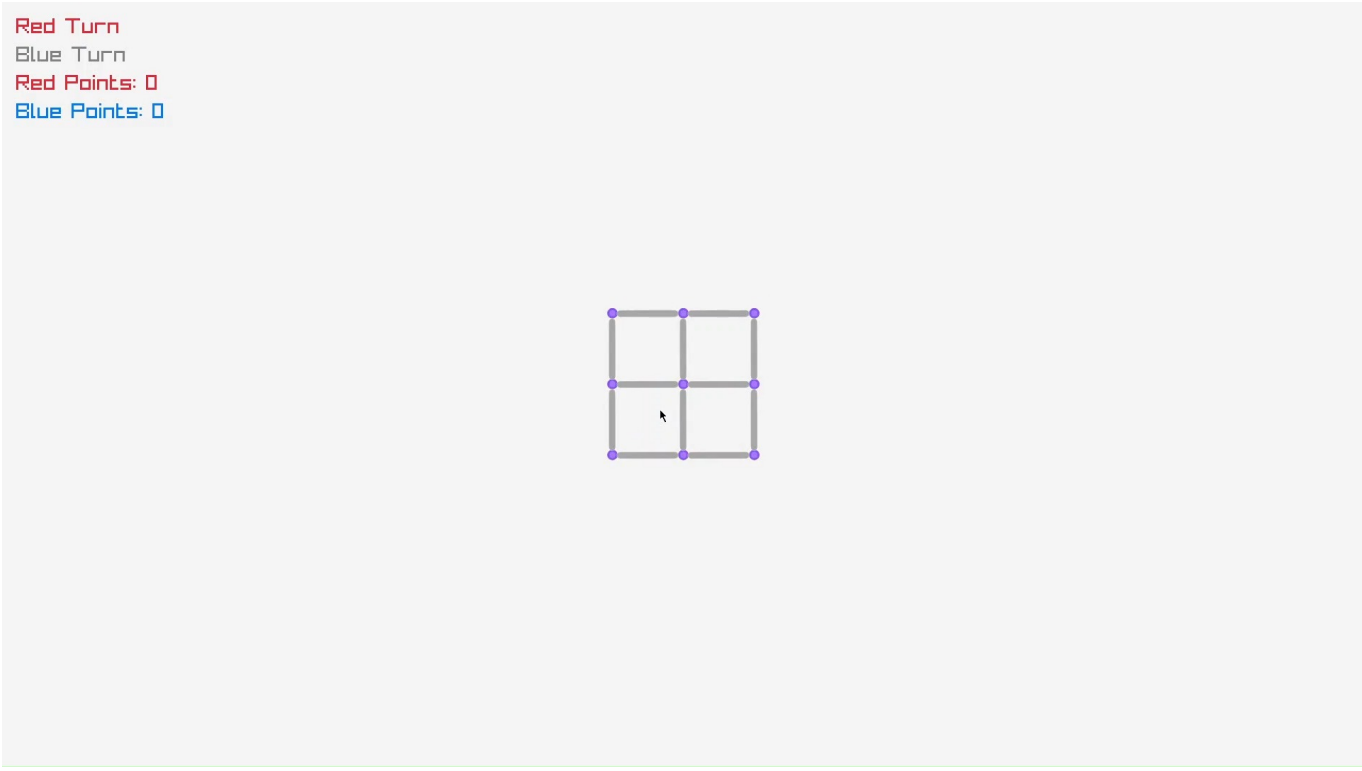
Name Page View



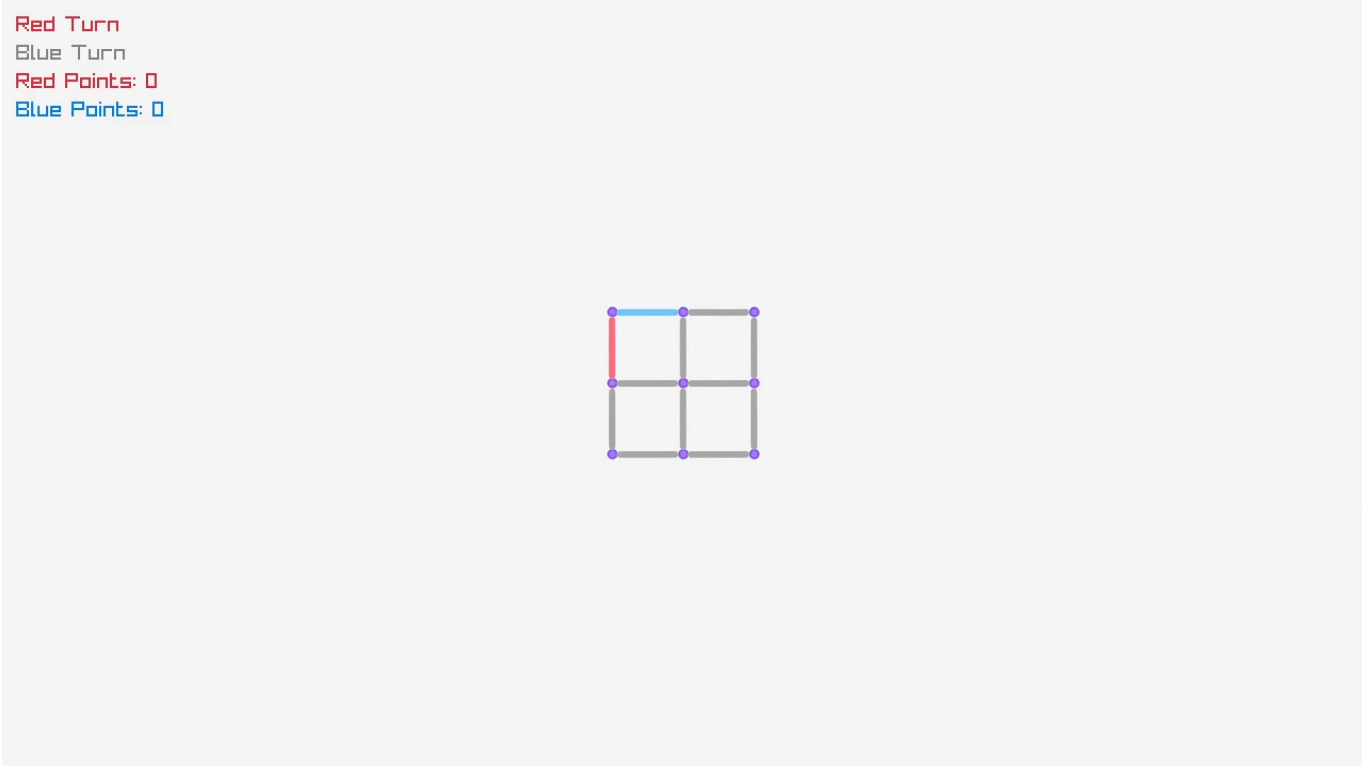
Board Dimension Page View



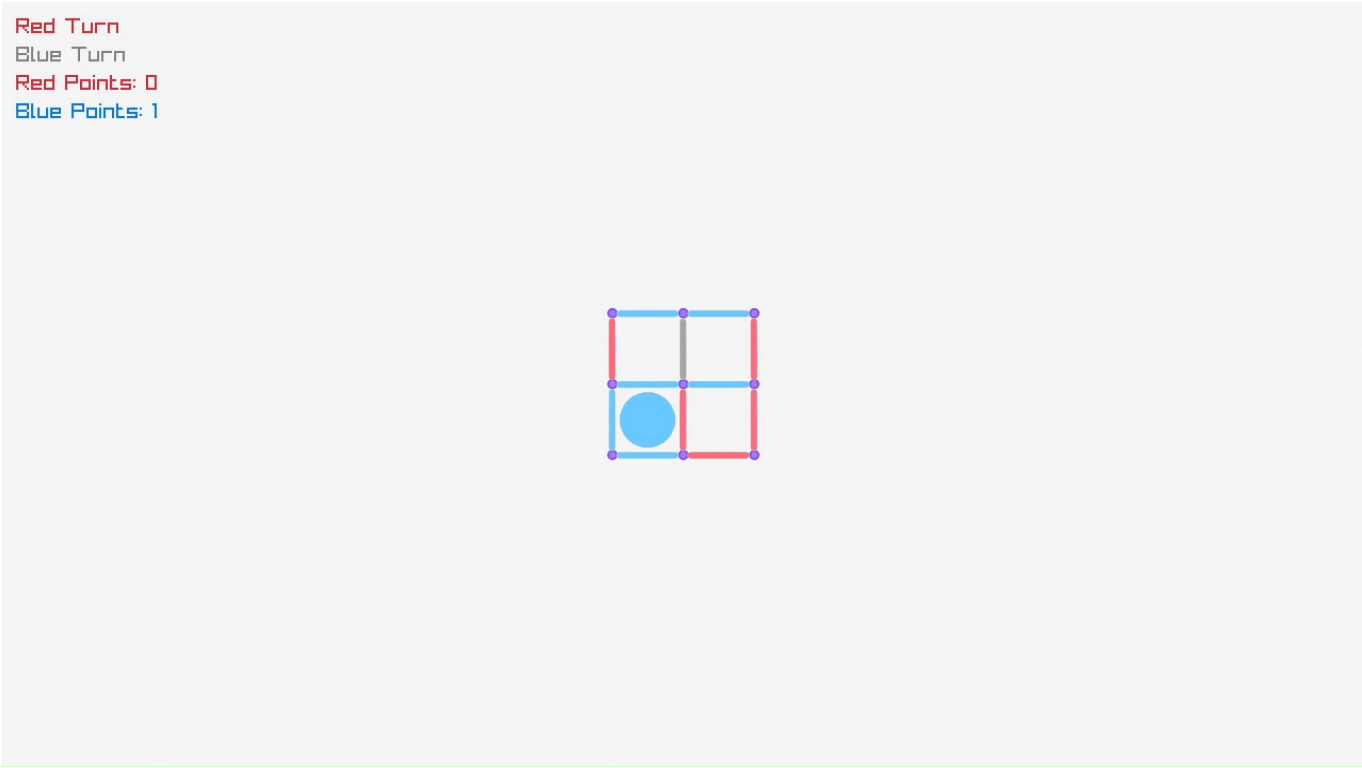
Inital Board Setup



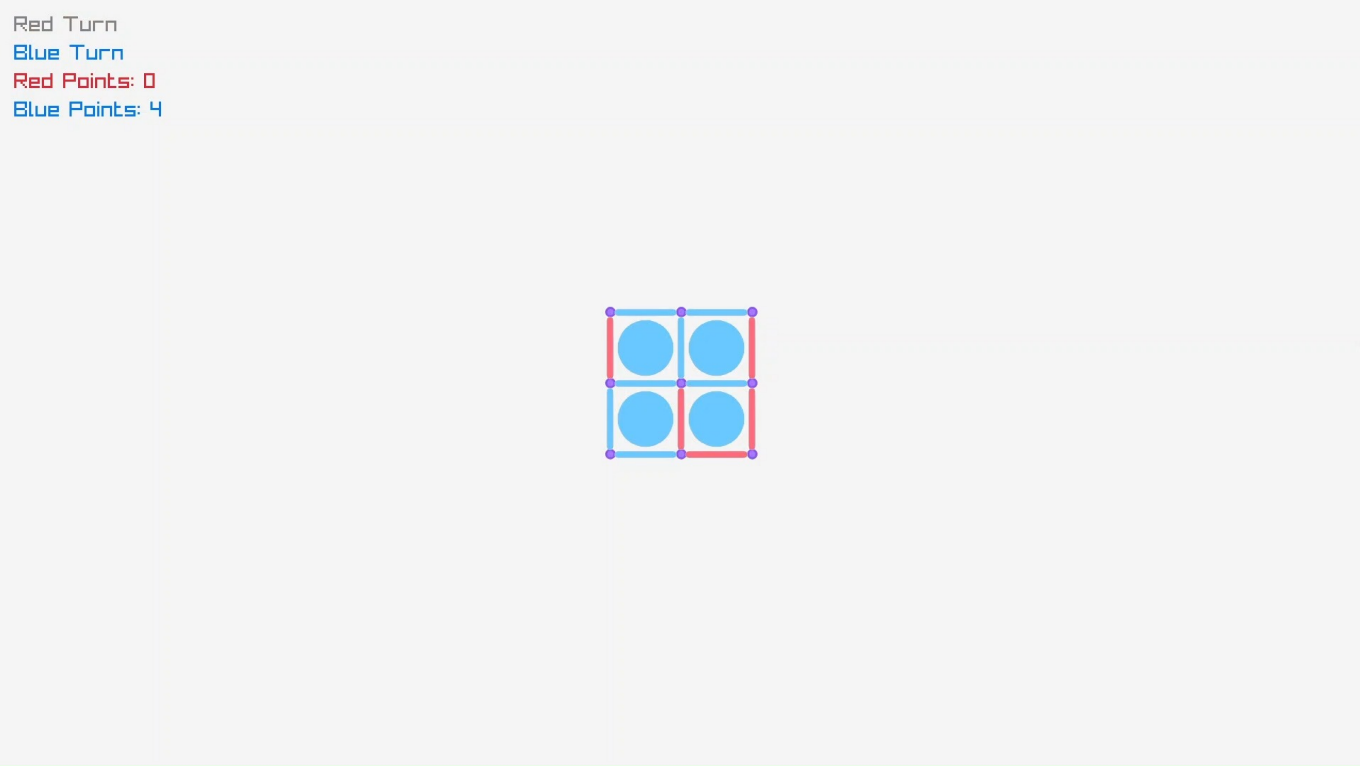
First Move



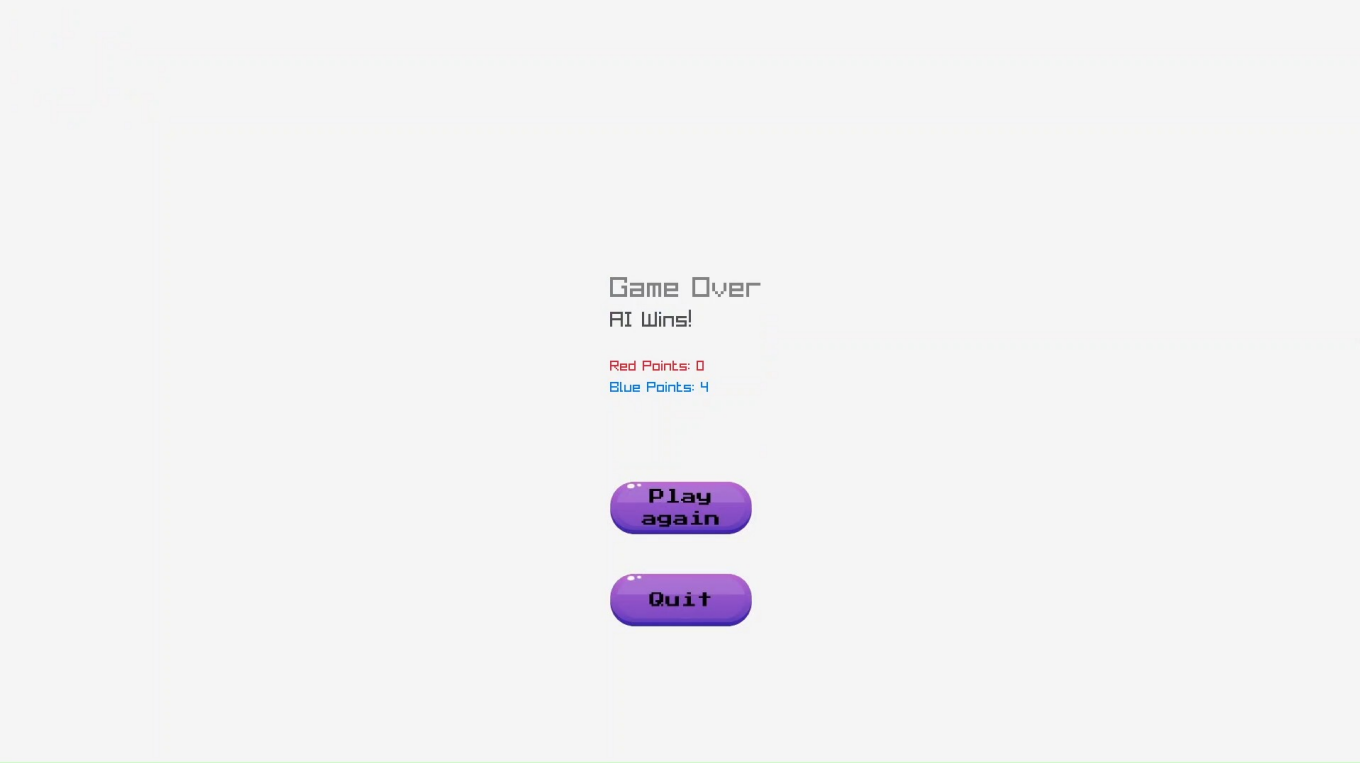
AI Scores First Point



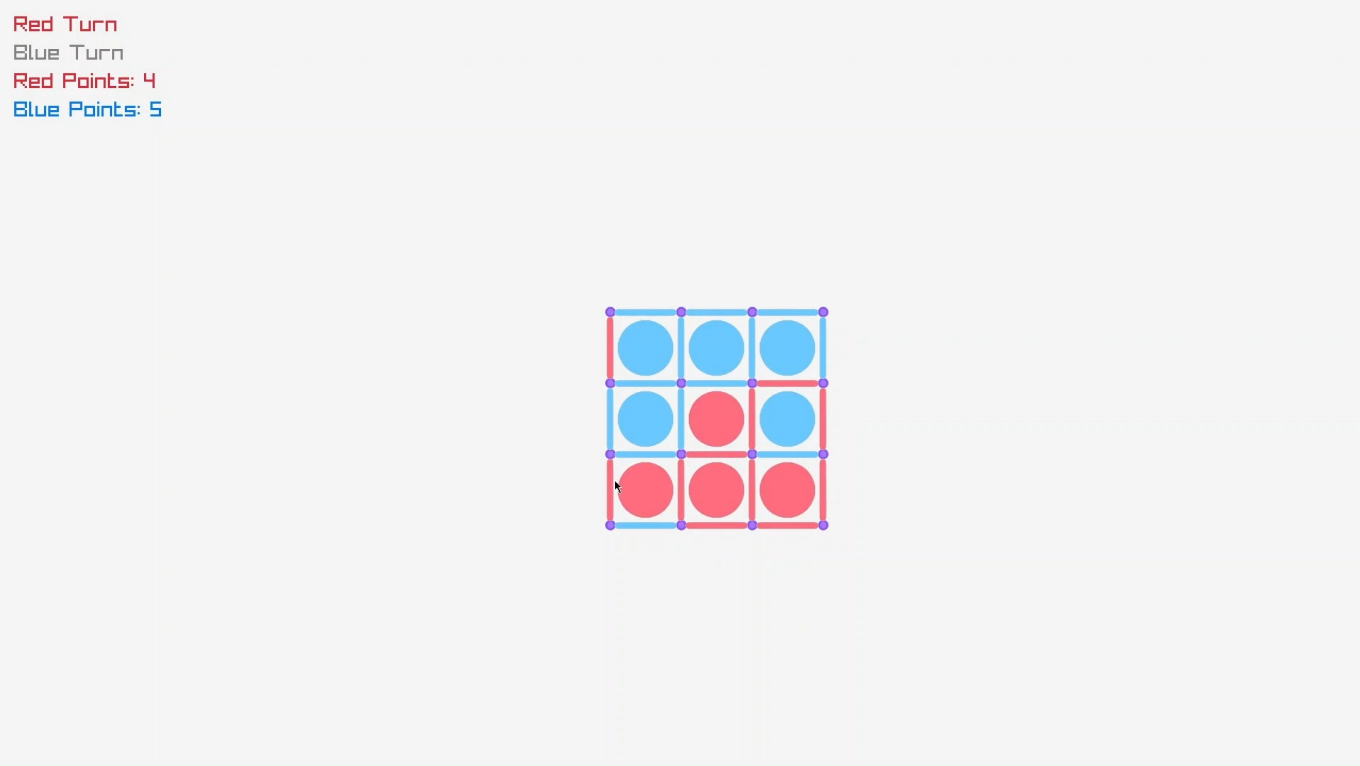
Players Out Of Moves



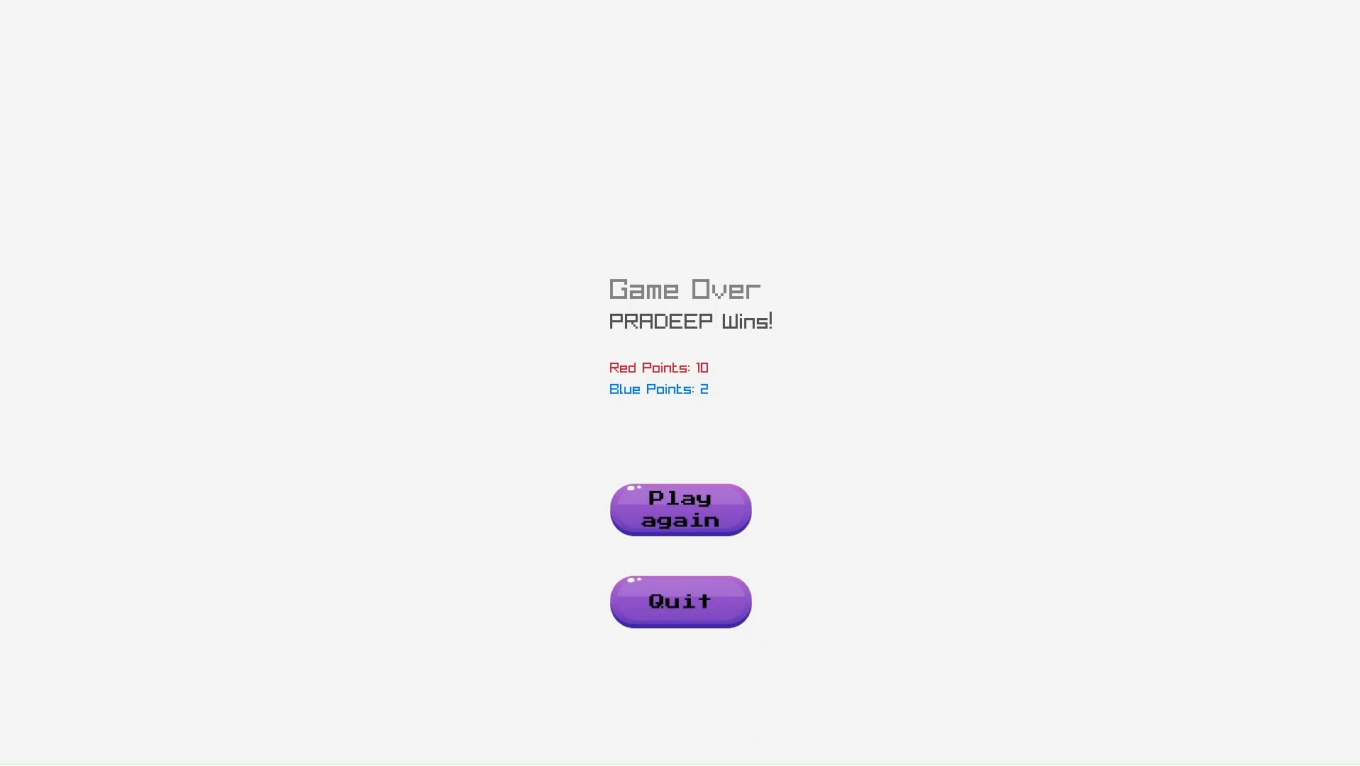
Result Page View shows that AI has won the game



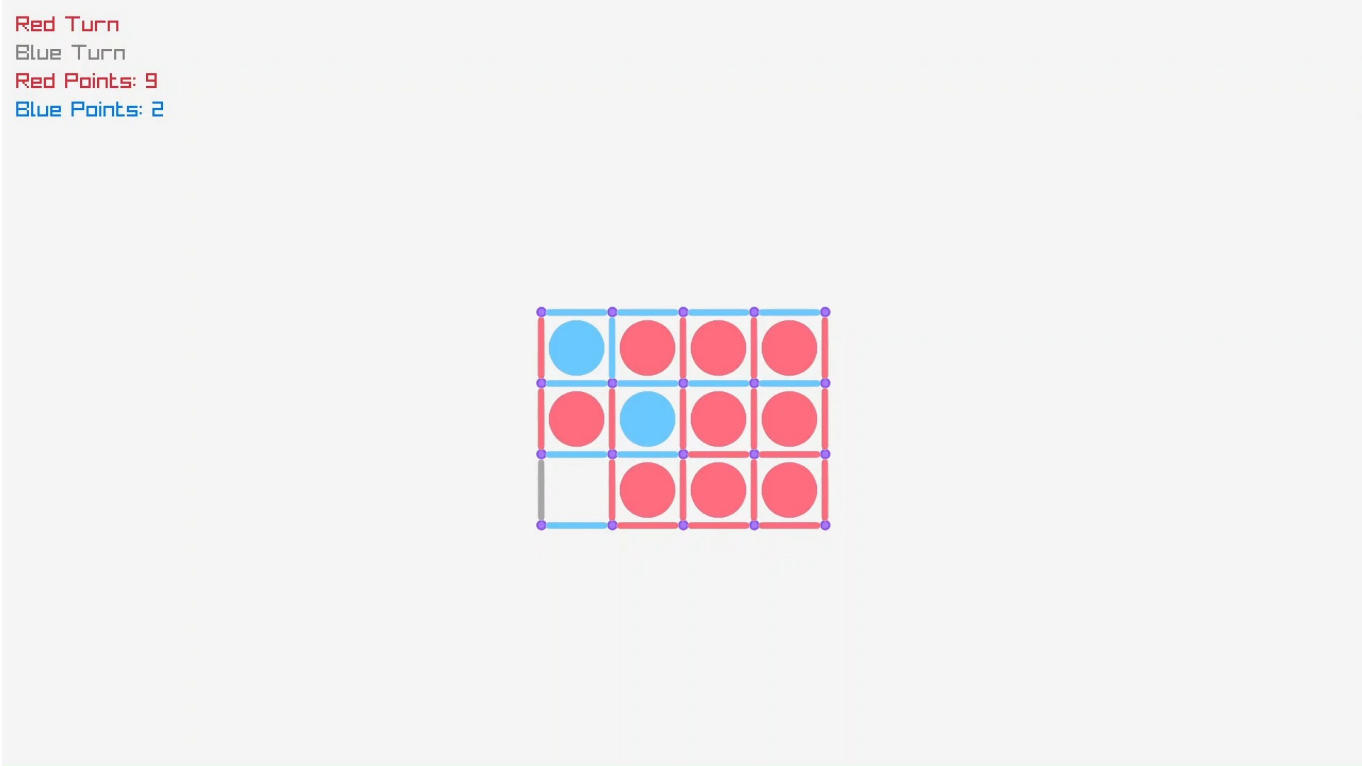
Board View of another game with AI



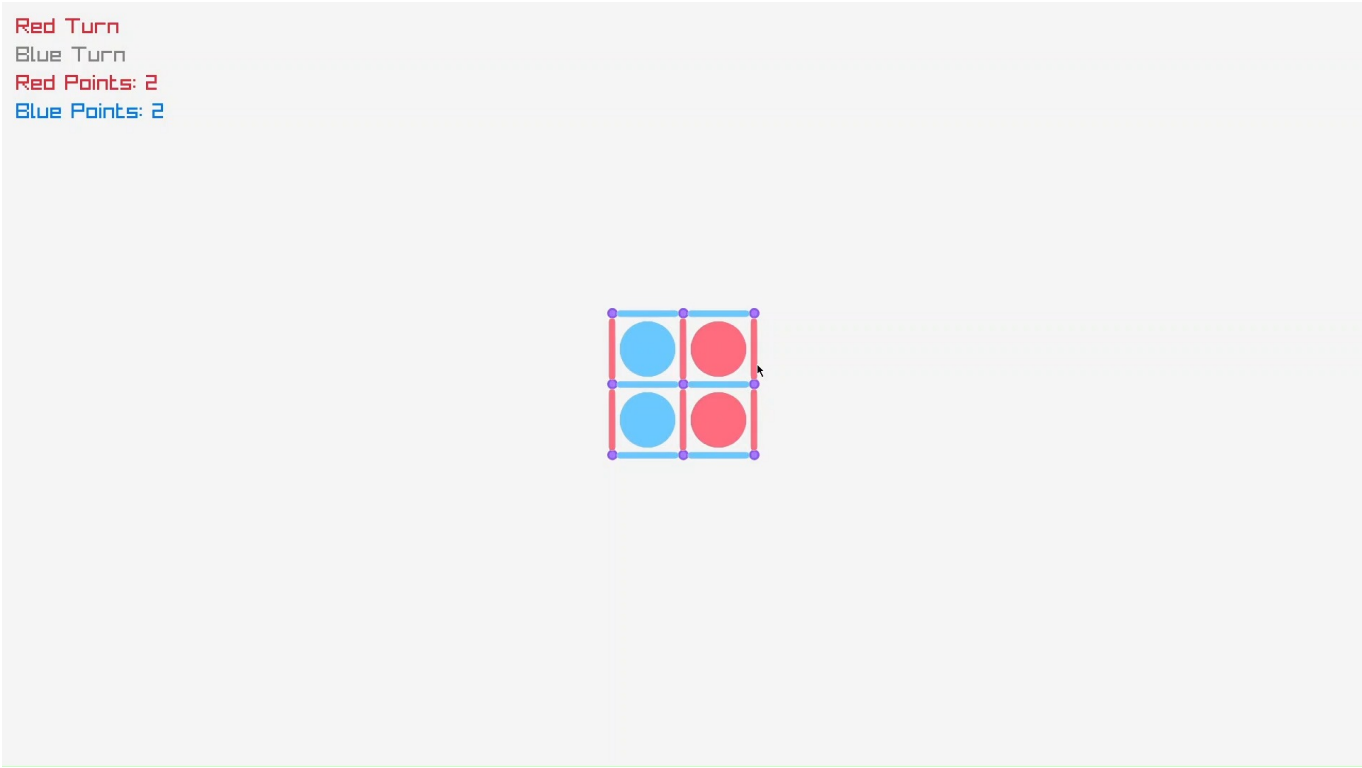
Result Page View of another game with AI



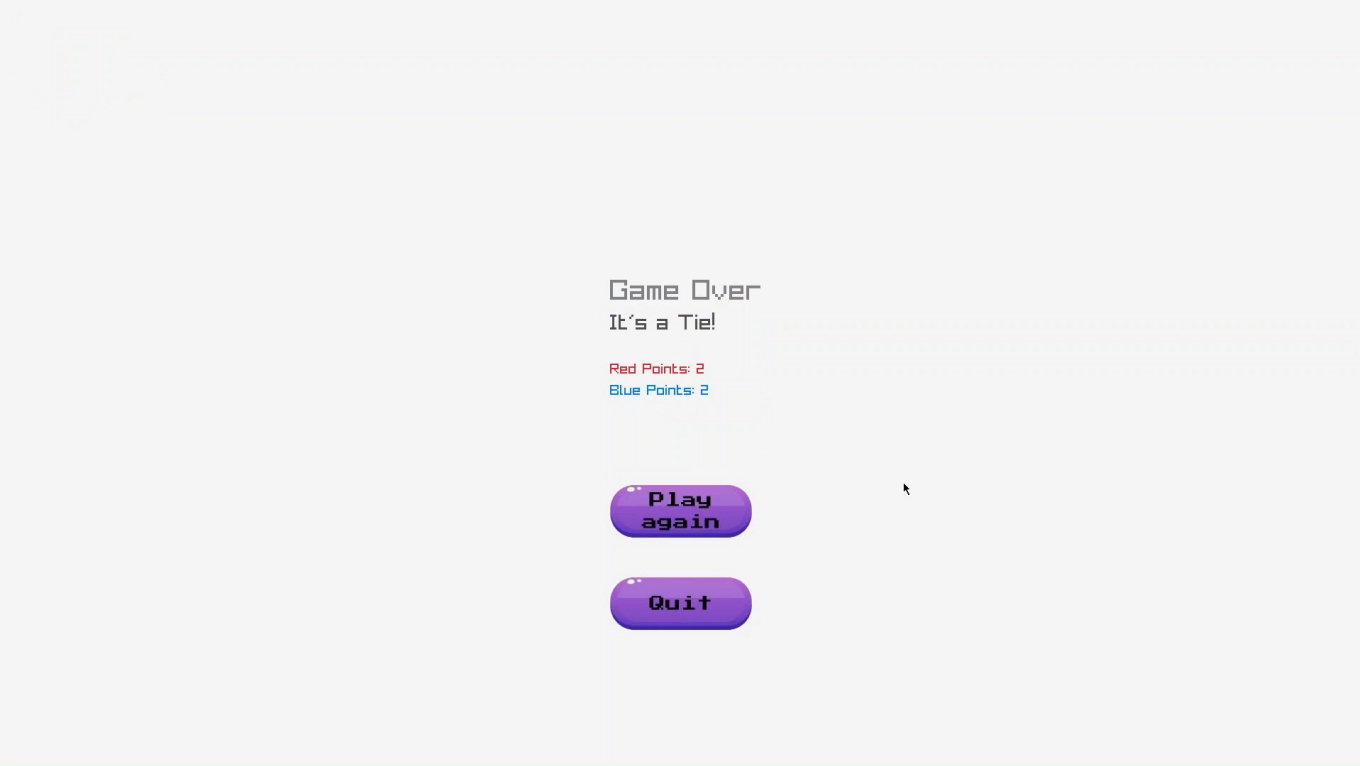
Board View with player leading the game



Result Page View shows that the player and AI has made equal number of points



Result Page View shows that game is tied



CONCLUSION:

The Dots and Boxes AI project demonstrates the effectiveness of using AI to create a challenging and engaging game opponent through a Minimax search algorithm with Alpha-Beta Pruning. By employing this strategy, the AI is able to make strategic decisions that simulate a competitive human player, evaluating potential moves in real-time. The Alpha-Beta Pruning algorithm allows the AI to efficiently explore the game tree, pruning branches that won't affect the outcome, ensuring that the AI can choose optimal moves without the need to examine every possible option.

In addition to the search algorithm, the AI uses a custom-designed heuristic evaluation function that scores the current game state based on factors such as box ownership and control of the board. This evaluation guides the AI to make smart, goal-oriented moves, balancing the pursuit of immediate wins with long-term control of the game.

The project was implemented using Python, with game mechanics and visual elements facilitated by the Raylib library, which made it easy to create a visually engaging and interactive game interface. The AI's decision-making process was refined through extensive testing, including manual evaluation and experimenting with different strategies like the basic Minimax algorithm and the more advanced Minimax with Alpha-Beta Pruning. The final implementation focuses on improving the AI's responsiveness to varying game scenarios, ensuring that it provides a challenging opponent in every game session.

This project highlights the power of AI in board games, delivering a fun and engaging player experience that remains computationally efficient. The flexible structure of the code allows for future expansion, with potential to integrate additional strategies, more complex game variations, or even machine learning-based approaches in the future. The Dots and Boxes AI successfully creates an intelligent opponent while serving as a solid foundation for exploring advanced AI techniques in game development.