Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Savut Dan Cosmin and Tocan Robert-Alexandru

Group:30432

Email:cosminsavut@gmail.com and roberttocan@yahoo.com

Teaching Assistant: Marius Adrian Stoica
stoica.ma.marius@student.utcluj.ro

Contents

Chapter 1

A1: Search

1.1 Introduction

In the realm of Artificial Intelligence (AI), search algorithms are essential for solving problems by exploring the problem space to find solutions. This chapter focuses on agents that solve problems by searching, comparing uninformed search strategies (such as depth-first search, breadth-first search, and uniform cost search) against informed search strategies (like the A* algorithm). The strengths of the A* algorithm are highlighted through various heuristics. These search strategies are illustrated through a game-based framework, specifically the Pacman agent. The Pacman game also provides a platform for practicing adversarial search.

1.2 Search Algorithms in AI

Search algorithms in AI are designed to help agents find the optimal solution by exploring different scenarios and alternatives. A search problem typically includes a search space, a start state, and a goal state. By simulating various scenarios, search algorithms enable AI agents to identify the optimal path or solution for a given task.

AI agents use logic to process the initial state and try to reach the goal state, making the AI systems heavily reliant on the efficiency of the search algorithms that ate used. These agents perform tasks without requiring software literacy from users, aiming to achieve goals by developing action plans. Completion of these plans involves evaluating all possible alternatives and choosing the best path.

1.3 Uninformed Search in AI

Uninformed search algorithms, also known as blind search algorithms, do not use additional information about the goal state other than what is provided in the problem definition. The strategies to reach the goal differ only by the order and length of actions. Uninformed search algorithms explore the search space systematically without considering the cost of reaching the goal.

1.3.1 Depth-First Search (DFS)

Depth-First Search (DFS) explores the deepest nodes in the search tree first using a stack to manage the frontier. It is a systematic search method, but it's still blind and that can be inefficient for large search spaces. For the implementation a stack that is part of the utils of the project as used.

1.3.2 Breadth-First Search (BFS)

Breadth-First Search (BFS) explores the shallowest nodes in the search tree first using a queue. It is complete and optimal for uniform cost problems but can be memory-intensive. For the BFS implementation we used a queue.

1.3.3 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) explores the node with the least total cost first using a priority queue based on the distance from the start node to the node to be explored. We used problem.getCostOfActions(pathToNode) since the cost of each action is one and it can be deduced from the path. It guarantees an optimal solution but can be slow for large search spaces.

1.4 Informed Search in AI

Informed search algorithms use additional information about the goal state to guide the search process more efficiently. This information is typically provided by a heuristic function that estimates how close a state is to the goal state. Informed search algorithms include A* search, Best-First search, and Greedy search.

1.4.1 A* Search

A* search combines the cost to reach a node and a heuristic to estimate the cost to the goal. It uses a priority queue to manage the frontier and is both complete and optimal if the heuristic is admissible and consistent. For the implementation, we used the priority queue defined in util and the nullHeuristic already defined.

1.5 Comparison of Search Strategies

Uninformed search strategies, such as DFS, BFS, and UCS, explore the search space systematically without additional information. They are simple to implement but can be inefficient for complex problems. In contrast, informed search strategies like A* use heuristics to guide the search process, making them more efficient and goal-directed.

1.6 Conclusion

Search algorithms are fundamental in AI for finding optimal solutions to problems. Uninformed search algorithms provide a basic framework for exploring the search space, while informed search algorithms enhance efficiency through heuristics. The Pacman game serves as an excellent platform for practicing and understanding these concepts, demonstrating the practical applications of search strategies in AI.

1.7 Eat all Food and Avoid Ghosts problem

For this problem a layout with food and at least one ghost will be used. The solution we found is not optimal and it is based on one that finds a reasonable path quickly (when there are no ghosts).

It is based on a greedy strategy that uses a modified BFS to find the closest dot. The agent then performs the first action in the path to this closest path if it is far enough from a ghost. If a ghost is too close to the square to which our agent would advance, it continues the modified BFS searching for another dot. If no dot is found, towards which the agent can advance, it will perform the first action available that keeps a safe distance from the ghosts.

The implementation differs from the BFS in the following ways:

- The algorithm has a different goal state. It calls another isGoalState function which checks if the state contains a dot.
- The algorithm is called after each action the agent performs.
- The algorithm returns only the first step of the path to the closest dot
- Extra checks at each found goal state, to check if the action gets the agent too close to a ghost
- Extra logic at the end of the algorithm, in case no available path was found towards any dot.

Bibliography

[1] https://www.javatpoint.com/search-algorithms-in-ai

[2] Berkeley Pacman Documentation https://inst.eecs.berkeley.edu/~cs188/fa24/projects/proj1/#q7-4-pts-eating-all-the-dots

[3] Themiscodes Github https://github.com/Themiscodes/Berkeley-Pacman-Projects/tree/main

[4] AI_Labs documentation

Appendix A

Your original code

Below are the implementations of various search algorithms used in the Pacman AI projects. Each algorithm solves a search problem using a different strategy.

A.1 Depth First Search (DFS)

DFS explores the deepest nodes in the search tree first. It uses a stack to manage the frontier.

```
def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
    "*** YOUR CODE HERE ***"
    stack = util.Stack()
    visited = []
    # we use a stack for LIFO strategy
    if problem.isGoalState(problem.getStartState()):
        return []

    stack.push((problem.getStartState(), []))
    # the initial state is pushed and popped from the stack
    while not stack.isEmpty():
        current_state, path = stack.pop()
        if problem.isGoalState(current_state):
            return path

        visited.append(current_state)
        successors = problem.getSuccessors(current_state)
        for succ in successors:
            # succ example ((5, 4), 'South', 1)
            if succ[0] not in visited:
                stack.push((succ[0], path + [succ[1]]))
                # path is updated with the direction
```

A.2 Breadth First Search (BFS)

BFS explores the shallowest nodes in the search tree first. It uses a queue to manage the frontier.

```
def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
```

```
"* YOUR CODE HERE *"
queue = util.Queue()
visited = []
#we are using a queue for FIFO strategy
if problem.isGoalState(problem.getStartState()):
    return []
queue.push((problem.getStartState(),[]))
while not queue.isEmpty():
    current_state, path = queue.pop()
    if problem.isGoalState(current_state):
        return path

    visited.append(current_state)
    successors = problem.getSuccessors(current_state)
    for succ in successors:
        if succ[0] not in visited:
            queue.push((succ[0], path + [succ[1]]))
            visited.append(succ[0])
            #the difference from DFS is that we mark all the
            successors as visited
return []
```

A.3 Uniform Cost Search (UCS)

UCS explores the node of least total cost first. It uses a priority queue to manage the frontier based on the cost.

```
def uniformCostSearch(problem: SearchProblem) -> List[Directions]:
    """Search the node of least total cost first using Uniform
    Cost Search (UCS)."""
    priority_queue = util.PriorityQueue()
    start_state = problem.getStartState()
    priority_queue.push((start_state, []), 0)
    visited = []

    while not priority_queue.isEmpty():
        current_state, path = priority_queue.pop()
        if problem.isGoalState(current_state):
            return path

        if current_state not in visited:
            visited.append(current_state)
            for succ in problem.getSuccessors(current_state):
                succ_state, succ_action, succ_cost = succ
                if succ not in visited:
                    priority_queue.push((succ_state, path + [succ_action]),
                    problem.getCostOfActions(path + [succ_action]))
                    # works because cost is one at each step

    return []
```

A.4 A* Search

A* search uses both the cost to reach the node and a heuristic to estimate the cost to the goal. It uses a priority queue to manage the frontier.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) ->
List[Directions]:
    """Search the node that has the lowest combined cost and heuristic
    first."""
    "*** YOUR CODE HERE ***"

    priority_queue = util.PriorityQueue()
    visited = {}
    start_state = problem.getStartState()
    if problem.isGoalState(start_state):
        return []
    priority_queue.push((start_state, [], 0),
    heuristic(start_state, problem))
    # here we set the priority of the start state as the heuristic

    while not priority_queue.isEmpty():
        current_state, path, current_cost = priority_queue.pop()
        if problem.isGoalState(current_state):
            return path
        # if the current state is already visited, if our current
        cost is <= than the previous one
        # it means that we found a cheaper way
        if current_state in visited and visited[current_state] <=
        current_cost:
            continue
        visited[current_state] = current_cost

        successors = problem.getSuccessors(current_state)
        for succ in successors:
            succ_state, succ_action, succ_cost = succ
            cost_until_succ = current_cost + succ_cost
            # if the state has not been visited or it can be reached
            # at a lower cost than previously recorded
            # proceed to calculate the total cost and push the state onto
            the priority queue.
            if succ[0] not in visited or visited[succ_state] >
            cost_until_succ:
                total_cost = cost_until_succ +
                heuristic(succ_state, problem)
                priority_queue.push((succ_state, path + [succ_action],
                cost_until_succ), total_cost)
    util.raiseNotDefined()
```

A.5 Eat all Dots and Avoid Ghosts

The implementations started from the Eat all Dots problem using the ClosestDotSearchAgent as showed in the documentation of the Pacman project, Q8: Suboptimal search. We modified this class as follows:

```
class ClosestDotSearchAgent(SearchAgent):
...
    def findPathToClosestDot(self, gameState: pacman.GameState):
        """
        Returns a path (a list of actions) to the closest dot,
        starting from gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)
        "*** YOUR CODE HERE ***"

        return search.breadthFirstSearchAvoidGhosts(problem, gameState)
```

We also modified the isGoalState method of the AnyFoodSearchProblem to check if certain state contains food.

```
class AnyFoodSearchProblem(PositionSearchProblem):
...
    def isGoalState(self, state: Tuple[int, int]):
        """
        The state is Pacman's position. Fill this in with a goal test
        that will complete the problem definition.
        """
        x, y = state
        "*** YOUR CODE HERE ***"

        if state in self.food.asList():
            return True
        else:
            return False
```

The next method that needed to be implemented is the modified breadth first search.

```
def breadthFirstSearchAvoidGhosts(problem: SearchProblem,
gameState: pacman.GameState) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    queue = util.Queue()
    visited = []

    ghostPositions = gameState.getGhostPositions()
    print("GHOST" + str(ghostPositions))
    ghostAvoidanceDistance = 2
    startState = problem.getStartState()
    if problem.isGoalState(problem.getStartState()):
```

```
    return []
queue.push((problem.getStartState(),[]))
while not queue.isEmpty():
    currentState, path = queue.pop()
    if problem.isGoalState(currentState):
        # if a dot is found, check if you can make a move towards it
        nextState = getNextSqaure(startState, path[0])
        tooCloseToGhost = False
        for ghostPositon in ghostPositions:
            if util.manhattanDistance(nextState, ghostPositon) <
            ghostAvoidanceDistance:
                tooCloseToGhost = True
        if tooCloseToGhost:
            # get search for the next dot that is closest
            continue
        return [path[0]]

    visited.append(currentState)
    successors = problem.getSuccessors(currentState)
    for succ in successors:
        if succ[0] not in visited:
            queue.push((succ[0], path + [succ[1]]))
            visited.append(succ[0])

# if there is no available action towards a dot do the first
available move
successors = problem.getSuccessors(startState)
for succ in successors:
    tooCloseToGhost = False
    for ghostPosition in ghostPositions:
        if util.manhattanDistance(succ[0], ghostPosition) <
        ghostAvoidanceDistance:
            print(succ[0])
            tooCloseToGhost = True
    if tooCloseToGhost:
        continue
    else:
        return [succ[1]]
return []
```