East West University

Department of CSE

Course Code and Name: **CSE366 Artificial Intelligence**

Section : **02**

Pacman

Semester and Year: **Fall 2024**

Name of Student, ID

Md. Bayzid Ohin (2021-3-60-288)

Course Instructor information:

Dr. Raihan Ul Islam
Associate Professor
Department of Computer Science & Engineering

Pacman Search Algorithms Report

Introduction

This report investigates the implementation and performance of three search algorithms – Depth-First Search (DFS), Breadth-First Search (BFS), and Uniform Cost Search (UCS) – within the Pacman environment. The algorithms were tested on different maze layouts to analyze their efficiency in terms of execution time, nodes explored, and the cost of the path found.

Overview of the Algorithms

1. **Depth-First Search (DFS):** DFS explores as far as possible along a branch before backtracking. It uses a stack to manage the frontier.

2. **Breadth-First Search (BFS):** BFS explores all nodes at the present depth level before moving on to nodes at the next depth level. It employs a queue to manage the frontier.

3. **Uniform Cost Search (UCS):** UCS expands the node with the lowest total cost first. It uses a priority queue to manage the frontier.

Implementation of the Code

Below are the implementations of the three algorithms as used in this project:

Depth-First Search (DFS):

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.
    """
    from util import Stack
    import time

    # Initialize timer
    start_time = time.time()

    # Initialize the frontier with the start state
    start_state = problem.getStartState()
    if problem.isGoalState(start_state):  # Check if the start state is the goal
```

```
    return []

frontier = Stack()  # Use a stack for DFS
frontier.push((start_state, []))  # (state, path)

explored = set()  # A set to track explored states
nodes_explored = 0

while not frontier.isEmpty():
    # Pop the most recently added state and path
    state, path = frontier.pop()

    nodes_explored += 1  # Increment node counter

    if state in explored:  # Skip states we've already explored
        continue

    # Mark the state as explored
    explored.add(state)

    # Check if the current state is a goal state
    if problem.isGoalState(state):
        end_time = time.time()
        time_taken_ms = (end_time - start_time) * 1000
        print(f"Time taken: {time_taken_ms:.2f} ms")
        print(f"Nodes explored: {nodes_explored}")
        return path  # Return the path that leads to the goal

    # Get successors of the current state
    for successor, action, step_cost in problem.getSuccessors(state):
        if successor not in explored:
            # Push the successor to the frontier with the updated path
            frontier.push((successor, path + [action]))

# No solution found
end_time = time.time()
time_taken_ms = (end_time - start_time) * 1000
print(f"Time taken: {time_taken_ms:.2f} ms")
print(f"Nodes explored: {nodes_explored}")
return []
```

Breadth-First Search (BFS):

```
def breadthFirstSearch(problem):
    """
    Search the shallowest nodes in the search tree first.
    """
    from util import Queue
    import time

    # Initialize timer
    start_time = time.time()

    # Initialize the frontier with the start state
    start_state = problem.getStartState()
    if problem.isGoalState(start_state):  # Check if the start state is the goal
        return []

    frontier = Queue()
    frontier.push((start_state, []))  # (state, path)

    explored = set()  # A set to track explored states
    nodes_explored = 0

    while not frontier.isEmpty():
        # Pop the shallowest state and path
        state, path = frontier.pop()

        nodes_explored += 1  # Increment node counter

        if state in explored:  # Skip states we've already explored
            continue

        # Mark the state as explored
        explored.add(state)

        # Check if the current state is a goal state
        if problem.isGoalState(state):
            end_time = time.time()
            time_taken_ms = (end_time - start_time) * 1000
            print(f"Time taken: {time_taken_ms:.2f} ms")
            print(f"Nodes explored: {nodes_explored}")
            return path  # Return the path that leads to the goal

        # Get successors of the current state
        for successor, action, step_cost in problem.getSuccessors(state):
            if successor not in explored:
```

```
            # Push the successor to the frontier with the updated path
            frontier.push((successor, path + [action]))

    # No solution found
    end_time = time.time()
    time_taken_ms = (end_time - start_time) * 1000
    print(f"Time taken: {time_taken_ms:.2f} ms")
    print(f"Nodes explored: {nodes_explored}")
    return []
```

Uniform Cost Search (UCS):

```
def uniformCostSearch(problem):
    """
    Search the node of least total cost first.
    """
    from util import PriorityQueue
    import time

    # Initialize timer
    start_time = time.time()

    # Initialize the frontier with the start state
    start_state = problem.getStartState()
    if problem.isGoalState(start_state):  # If the start state is the goal
        return []

    frontier = PriorityQueue()  # Priority queue for UCS
    frontier.push((start_state, []), 0)  # Push (state, path) with priority = 0
    explored = set()  # Track explored states
    nodes_explored = 0

    while not frontier.isEmpty():
        # Pop the state with the lowest cost
        state, path = frontier.pop()

        nodes_explored += 1  # Increment node counter

        if state in explored:
            continue

        explored.add(state)  # Mark as explored
```

```
        # Check if it's the goal state
        if problem.isGoalState(state):
            end_time = time.time()
            time_taken_ms = (end_time - start_time) * 1000
            print(f"Time taken: {time_taken_ms:.2f} ms")
            print(f"Nodes explored: {nodes_explored}")
            return path

        # Add successors to the frontier
        for successor, action, step_cost in problem.getSuccessors(state):
            if successor not in explored:
                new_path = path + [action]
                total_cost = problem.getCostOfActions(new_path)
                frontier.push((successor, new_path), total_cost)

    # No solution found
    end_time = time.time()
    time_taken_ms = (end_time - start_time) * 1000
    print(f"Time taken: {time_taken_ms:.2f} ms")
    print(f"Nodes explored: {nodes_explored}")
    return []
```

Experimental Results

The following table summarizes the results of the experiments conducted using DFS, BFS, and UCS on three different maze layouts:

Algorithm	Maze	Time Taken (ms)	Nodes Explored	Path Cost	Score
DFS	mediumMaze	8	147	130	380
DFS	bigMaze	9	391	210	300
DFS	tinyMaze	1.01	16	10	500
BFS	mediumMaze	8	276	68	442
BFS	bigMaze	14	621	210	300
BFS	tinyMaze	1	17	8	502
UCS	mediumMaze	13.03	276	68	442
UCS	bigMaze	47	621	210	300
UCS	tinyMaze	1.01	17	8	502

Observations

1. **DFS:**
    - Performed efficiently on smaller mazes like tinyMaze but explored unnecessary nodes in larger mazes.
    - Path cost was higher compared to BFS and UCS in some cases.
2. **BFS:**
    - Explored more nodes than DFS but found shorter paths.
    - Performed consistently across all maze sizes.
3. **UCS:**
    - Found optimal paths similar to BFS but incurred higher computational time due to the priority queue.
    - Performed well in finding paths with minimal costs.

Conclusion

- DFS is suitable for smaller problems or when memory usage is a concern.
- BFS ensures shorter paths but may consume more memory.
- UCS is ideal for cost-sensitive scenarios but is computationally expensive.

Recommendations

- Use UCS or BFS when optimal solutions are required.
- DFS can be employed for quick, approximate solutions in smaller environments.
- Further experiments with heuristic-based searches like A* may provide insights into more efficient solutions.

References

- Berkeley AI Search Project
- Berkeley AI Project Overview