East West University

Lab Report 03

Course Title: Artificial Intelligence

Semester: Fall 2024

Course code: CSE366

Section:01

Submitted to:

Dr. Raihan Ul Islam

Associate Professor

Department of Computer Science and Engineering

Submitted by:

Name: Rawnak
ID: 2020-1-60-263

Submission Date: 30- 11 - 2024

<u>Observations:</u>

Depth-First Search (DFS):

- **Performance**: DFS explores deeper paths first. While it can find solutions quickly for small mazes, it often explores inefficient paths in larger mazes.
- **Nodes expanded**: DFS can expand many nodes due to its nature of backtracking when it hits dead ends.
- **Solution length**: The solution may not be optimal because DFS does not consider cost.
- **Time and space complexity**:
  - **Time:** O(b^m), where b is the branching factor and m is the maximum depth.
  - **Space:** O(bm) for the recursive stack.

Breadth-First Search (BFS):

- **Performance**: BFS explores all nodes at the current depth before moving deeper. It guarantees the shortest path if costs are uniform.
- **Nodes expanded**: BFS generally expands more nodes compared to DFS but avoids revisiting explored nodes.
- **Solution length**: Always finds the shortest path for unweighted graphs or mazes.
- **Time and space complexity**:
  - **Time:** O(b^(d+1)), where d is the shallowest depth of the solution.
  - **Space:** O(b^(d+1)), as it stores all nodes at the current depth.

Uniform Cost Search (UCS):

- **Performance**: UCS considers the cost and guarantees an optimal solution. It is slower than BFS and DFS in small mazes but outperforms them in large, weighted mazes.
- **Nodes expanded**: UCS explores fewer nodes than DFS or BFS for cost-based problems.
- **Solution length**: Always optimal, considering weights or costs.
- **Time and space complexity**:
  - **Time:** O(b^{C^* / \e}), where C^* is the cost of the optimal solution and ϵ\epsilon is the smallest step cost.
  - **Space:** O(b^{C^* / \e}), similar to time complexity.

Expected Performance:

1. Breadth-First Search (BFS)

- **TinyMaze**: BFS will efficiently find the optimal path with minimal memory usage.
- **MediumMaze**: BFS may still perform well but will expand many nodes, leading to higher memory usage.
- **BigMaze**: BFS's performance will degrade due to exponential growth in memory and execution time.

2. Depth-First Search (DFS)

- **TinyMaze**: DFS will quickly find a path, but it may not be optimal.
- **MediumMaze**: DFS will expand fewer nodes compared to BFS but may find a non-optimal path.
- **BigMaze**: DFS is likely to perform faster and use less memory, but the path quality will depend on the maze structure.

3. Uniform-Cost Search (UCS)

- **TinyMaze**: UCS will efficiently find the shortest-cost path, similar to BFS for uniform costs.
- **MediumMaze**: UCS may take longer than BFS but will find an optimal path if costs are non-uniform.
- **BigMaze**: UCS will expand many nodes, leading to long execution times and high memory usage.

General Recommendations:

- **TinyMaze**: DFS or UCS for faster performance; BFS for guaranteed optimality.
- **MediumMaze**: BFS if memory allows; UCS for guaranteed optimal cost.
- **BigMaze**: DFS for speed and low memory usage; UCS if optimality is critical.

Commands to test and run code:

 # Run BFS

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

 # Run UCS

python pacman.py -l bigMaze -p SearchAgent -a fn=ucs -z .5
python pacman.py -l tinyMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

 # Run DFS

python pacman.py -l bigMaze -p SearchAgent -a fn=dfs -z .5
python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs

Table :

Algorithm	Maze Type	Path Found	Execution Time (s)	Nodes Expanded
BFS	TinyMaze	8	0.034571 seconds	15
DFS	TinyMaze	10	0.021522 seconds	15
UCS	TinyMaze	8	0.020995 seconds	15
BFS	MediumMaze	210	0.025733 seconds	620
DFS	MediumMaze	130	0.183040 seconds	146
UCS	MediumMaze	68	0.267070 seconds	269
BFS	BigMaze	210	1.055383 seconds	620
DFS	BigMaze	210	0.758171 seconds	390
UCS	BigMaze	210	1.052238 seconds	620

Here is the comparison:

1. TinyMaze

Algorithm	Path Found	Execution Time (s)	Nodes Expanded
BFS	8	0.034571	15
DFS	10	0.021522	15
UCS	8	0.020995	15

- **Key Observations**:
    - All algorithms expanded the same number of nodes (15).
    - UCS was the fastest with the least execution time (0.020995 seconds).
    - DFS found a slightly longer path with 10 compared to BFS and UCS both with 8.

2. MediumMaze

Algorithm	Path Found	Execution Time (s)	Nodes Expanded
BFS	210	0.025733	620
DFS	130	0.18304	146
UCS	68	0.26707	269

- **Key Observations**:
    - BFS found the longest path with 210, while UCS found the shortest path with cost 68.
    - BFS had the fastest execution time (0.025733 seconds), followed by DFS, and UCS was the slowest.
    - BFS expanded the most nodes (620), while DFS expanded the least nodes (146).

3. BigMaze

Algorithm	Path Found	Execution Time (s)	Nodes Expanded
BFS	210	1.055383	620
DFS	210	0.758171	390
UCS	210	1.052238	620

- **Key Observations**:
  - All algorithms found the same path length with cost 210.
  - DFS was the fastest (0.758171 seconds), while UCS and BFS had similar execution times (~1.05 seconds).
  - DFS expanded fewer nodes (390) compared to BFS and UCS (both 620).

General Insights:

1. **Efficiency**:
   - DFS is generally faster for larger mazes, though it may not always yield the optimal solution (as seen in MediumMaze).
   - UCS and BFS have comparable execution times for BigMaze, but UCS often finds optimal solutions.
2. **Path Optimality**:
   - UCS consistently finds optimal paths (shortest), as it is designed for cost-based exploration.
   - BFS also finds the shortest path in cases with uniform cost but may take more time as maze size increases.
   - DFS may find suboptimal paths as it prioritizes depth over optimality.
3. **Scalability**:
   - DFS is more scalable in terms of nodes expanded for larger mazes, making it computationally cheaper.
   - BFS and UCS require significant node expansions, especially in larger mazes.

This comparison highlights trade-offs between time efficiency, optimality, and resource usage for the algorithms in different maze scenarios.

Code:

a. **DFS**

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.
    """
    start_time = time.time()  # Start the timer

    startState = problem.getStartState()  # Get the start state of
the problem
    if problem.isGoalState(startState):  # Check if the start state
is already the goal state
        print("Execution time: {:.6f} seconds".format(time.time() -
start_time))
        return []

    frontier = util.Stack()  # Initialize the frontier using a stack
    frontier.push((startState, []))  # Push the start state and an
empty path onto the stack
    explored = set()  # Set to keep track of explored states

    while not frontier.isEmpty():
        currentState, currentPath = frontier.pop()  # Pop a state
and its path from the stack
        if currentState in explored:  # Skip if the state has
already been explored
            continue
        explored.add(currentState)  # Mark the state as explored
        if problem.isGoalState(currentState):  # Check if the
current state is the goal state
            print("Execution time: {:.6f}
seconds".format(time.time() - start_time))
            return currentPath
        for successor, action, stepCost in
problem.getSuccessors(currentState):  # Explore successors
            if successor not in explored:
                newPath = currentPath + [action]  # Create a new
path including the current action
```

```
                frontier.push((successor, newPath))  # Push the successor and its path onto the stack

    print("Execution time: {:.6f} seconds".format(time.time() - start_time))
    return []
```

b. BFS

```
def breadthFirstSearch(problem):
    """
    Search the shallowest nodes in the search tree first.
    """
    start_time = time.time()  # Start the timer
    currState = problem.getStartState()  # Get the start state of the problem
    if problem.isGoalState(currState):  # Check if the start state is already the goal state
        print("Execution time: {:.6f} seconds".format(time.time() - start_time))
        return []
    frontier = util.Queue()  # Initialize the frontier using a queue
    frontier.push((currState, []))  # Push the start state and an empty path onto the queue
    explored = set()  # Set to keep track of explored states
    while not frontier.isEmpty():
        currState, currPath = frontier.pop()  # Pop a state and its path from the queue
        if problem.isGoalState(currState):  # Check if the current state is the goal state
            print("Execution time: {:.6f} seconds".format(time.time() - start_time))
            return currPath
        explored.add(currState)  # Mark the state as explored
        frontierStates = [t[0] for t in frontier.list]  # Get all states currently in the frontier
        for s in problem.getSuccessors(currState):  # Explore successors
            if s[0] not in explored and s[0] not in frontierStates: # Check if the successor is unexplored
```

```
                frontier.push((s[0], currPath + [s[1]]))  # Push the
successor and its path onto the queue

    print("Execution time: {:.6f} seconds".format(time.time() -
start_time))
    return []
```

c. **UCS**

```
def uniformCostSearch(problem):
    """
    Search the node of least total cost first.
    """
    start_time = time.time()  # Start the timer

    currState = problem.getStartState()  # Get the start state of
the problem
    if problem.isGoalState(currState):  # Check if the start state
is already the goal state
        print("Execution time: {:.6f} seconds".format(time.time() -
start_time))
        return []

    frontier = util.PriorityQueue()  # Initialize the frontier using
a priority queue
    frontier.push((currState, [], 0), 0)  # Push the start state,
empty path, and cost 0 onto the priority queue
    explored = set()  # Set to keep track of explored states

    while not frontier.isEmpty():
        currState, currPath, currCost = frontier.pop()  # Pop a
state, its path, and cost from the priority queue
        if currState not in explored:  # Check if the state has
already been explored
            explored.add(currState)  # Mark the state as explored
            if problem.isGoalState(currState):  # Check if the
current state is the goal state
```

```
                print("Execution time: {:.6f}
seconds".format(time.time() - start_time))
                return currPath
            for successor, action, stepCost in
problem.getSuccessors(currState):  # Explore successors
                if successor not in explored:
                    newCost = currCost + stepCost  # Calculate the
new cost to reach the successor
                    frontier.push((successor, currPath + [action],
newCost), newCost)  # Push the successor onto the priority queue

    print("Execution time: {:.6f} seconds".format(time.time() -
start_time))
    return []
```