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

Observations:

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 bb is the branching factor and mm 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 dd 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. DFS

- **tinyMaze**: Likely to find a solution quickly due to its small size but may not yield the shortest path because DFS doesn't consider path cost.
- **mediumMaze & bigMaze**: May explore deeply into irrelevant branches, leading to inefficiency and high node expansion.

2. BFS

- **tinyMaze**: Will find the optimal path but may explore more nodes than DFS.
- **mediumMaze & bigMaze**: Efficient in finding the shortest path; however, node expansion increases significantly for larger mazes, potentially increasing execution time.

3. UCS

- **tinyMaze**: Behaves similarly to BFS, as the cost of each step in the maze is likely uniform.
- **mediumMaze & bigMaze**: Expands nodes based on the least total path cost. It is optimal and guarantees the lowest-cost solution but can be slower due to priority queue operations.

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

Output Table:

Algorithm	Maze	Path Length	Nodes Expanded
DFS	tinyMaze	10	15
BFS	tinyMaze	8	15
UCS	tinyMaze	8	15
DFS	mediumMaze	130	146
BFS	mediumMaze	68	269
UCS	mediumMaze	68	269
DFS	bigMaze	210	390
BFS	bigMaze	210	620
UCS	bigMaze	210	620

Here,

1. **Path length/ Path cost** : Number of steps or the cost of the solution returned by the algorithm in the path found.
2. **Nodes expanded**: Total number of nodes visited during the search.

Comparisons:

1. Performance in tinyMaze

DFS often suffices; BFS and UCS provide similar performance due to simplicity.

a. **DFS:** Likely to find a solution quickly due to its small size but may not yield the shortest path because DFS doesn't consider path cost.

b. **BFS:** Will find the optimal path but may explore more nodes than DFS.

c. **UCS:** Behaves similarly to BFS, as the cost of each step in the maze is likely uniform.

Algorithm	Path Length	Nodes Expanded
DFS	10	15
BFS	8	15
UCS	8	15

Here,

i. Path Length: BFS and UCS are optimal as they find the shortest path of length 8, while DFS produces a longer path (10).

ii. Nodes Expanded: All algorithms expand the same number of nodes (15), indicating that the maze's size and layout make no significant difference in node exploration.

2. Performance in mediumMaze

BFS and UCS excel in finding shorter paths. DFS expands unnecessary nodes.

a. **DFS:** Can explore deeply into irrelevant branches, leading to inefficiency

b. **BFS:** Finds the shortest path but expands many nodes, especially in larger mazes.

c. **UCS:** Expands nodes based on least cost, guarantees the optimal path but may be slower due to priority queue operations.

Algorithm	Path Length	Nodes Expanded
DFS	130	146
BFS	68	269
UCS	68	269

Here,

i. **Path Length**: BFS and UCS are optimal with a path length of 68, whereas DFS produces a significantly longer and suboptimal path (130).

ii. Nodes Expanded: DFS is more efficient in terms of nodes expanded (146) compared to BFS and UCS (both 269). However, this comes at the cost of a non-optimal solution.

3. **Performance in bigMaze**

UCS performs best, especially if costs are weighted. BFS is more efficient than DFS but may use significant memory.

a. **DFS:** It explores irrelevant branches deeply, leading to inefficiency and increased node expansion.

b. **BFS:** Tt will find the shortest path but might expand a large number of nodes, particularly in bigger mazes.

c. **UCS:** It expands nodes based on the least total path cost, ensuring the optimal path even with non-uniform costs. However, it may be slower due to priority queue operations.

Algorithm	Path Length	Nodes Expanded
DFS	210	390
BFS	210	620
UCS	210	620

Here,

i. Path Length: All three algorithms find the same path length (210), indicating that DFS happens to stumble upon the optimal path in this case.

ii. **Nodes Expanded:** DFS is more efficient, expanding fewer nodes (390) than BFS and UCS (both 620). This efficiency is due to DFS's depth-first approach, which limits node exploration but risks non-optimality in other scenarios.

Analysis:

Explain trends in performance:

a. **DFS** might expand fewer nodes but fail to find the optimal solution due to its depth-first nature. It is non-optimal but fast for small mazes.
b. **BFS** ensures the optimal solution. It guarantees the shortest path for uniform costs, with high node expansion in larger mazes.
c. **UCS** is optimal for both uniform and non-uniform costs but can be slower due to priority queue overhead.

Code:

a. **DFS**

```
def depthFirstSearch(problem):


    """Search the deepest nodes in the search tree first."""
    startState = problem.getStartState()
    if problem.isGoalState(startState):
        return []  # If the start state is already the goal

    # Use a stack to manage frontier, storing tuples of (state, path)
    frontier = util.Stack()
    frontier.push((startState, []))  # Start state with an empty path
    explored = set()  # To keep track of visited nodes

    while not frontier.isEmpty():
        currentState, currentPath = frontier.pop()

        # Add the current state to the explored set
        if currentState in explored:
            continue
        explored.add(currentState)

        # Check if the current state is the goal state
        if problem.isGoalState(currentState):
            return currentPath

        # Expand successors and push them to the stack
        for successor, action, stepCost in problem.getSuccessors(currentState):
            if successor not in explored:
                newPath = currentPath + [action]  # Add the action to the path
                frontier.push((successor, newPath))

    # Return an empty list if no solution is found
    return []
```

b. **BFS**

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    #util.raiseNotDefined()
    """ Search the shallowest nodes in the search tree first. """
    currPath = []          # The path that is popped from the frontier in each loop
    currState =  problem.getStartState()    # The state(position) that is popped for the frontier
in each loop
    print(f"currState: {currState}")
    if problem.isGoalState(currState):     # Checking if the start state is also a goal state
        return currPath

    frontier = Queue()
    frontier.push( (currState, currPath) )     # Insert just the start state, in order to pop it first
    explored = set()
    while not frontier.isEmpty():
        currState, currPath = frontier.pop()    # Popping a state and the corresponding path
        # To pass autograder.py question2:
        if problem.isGoalState(currState):
            return currPath
        explored.add(currState)
        frontierStates = [ t[0] for t in frontier.list ]
        for s in problem.getSuccessors(currState):
            if s[0] not in explored and s[0] not in frontierStates:
                # Lecture code:
                # if problem.isGoalState(s[0]):
                #     return currPath + [s[1]]
                frontier.push( (s[0], currPath + [s[1]]) )      # Adding the successor and its path to
the frontier

    return []      # If this point is reached, a solution could not be found.
```

c. UCS

```
def uniformCostSearch(problem):

    """Search the node of least total cost first."""
    currPath = []  # The path that is popped from the frontier in each loop
    currState = problem.getStartState()  # The state(position) that is popped from the frontier in each loop
    currCost = 0  # Current cost to reach `currState`

    if problem.isGoalState(currState):  # Checking if the start state is also a goal state
        return currPath

    frontier = PriorityQueue()  # UCS uses a priority queue for the frontier
    frontier.push((currState, currPath, currCost), currCost)  # Push the start state with priority = cost
    explored = set()  # To keep track of explored nodes

    while not frontier.isEmpty():
        currState, currPath, currCost = frontier.pop()  # Popping a state, the corresponding path, and cost

        if currState not in explored:
            explored.add(currState)

            if problem.isGoalState(currState):  # Check if the current state is a goal state
                return currPath

            for successor, action, stepCost in problem.getSuccessors(currState):
                if successor not in explored:
                    newCost = currCost + stepCost  # Calculate the new cost to reach `successor`
                    frontier.push((successor, currPath + [action], newCost), newCost)  # Push successor with updated cost

    return []  # If this point is reached, no solution was found
```