VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

Search

COURSE NAME: Fundamentals of Artificial Intelligence

Student:
Vo Thanh Nghia (22127295)

Lecturer:
Nguyen Thi Thu Hang
Pham Trong Nghia
Bui Duy Dang

June 22, 2024

Contents

1 Information

1.1 Student Information

Student ID	Full name	Email
22127295	Vo Thanh Nghia	vtnghia22@clc.fitus.edu.vn

2 Abstract

This report details the implementation and evaluation of five fundamental graph search algorithms: Breadth First Search (BFS), Depth First Search (DFS), Uniform-Cost Search (UCS), Greedy Best First Search (GBFS), and A*. Developed within the provided student_functions.py framework, the project adheres to specific guidelines and constraints. The report includes a self-assessment of the project's completeness, an exploration of the underlying theories, a comparative analysis of UCS and A* algorithms, and the implementation of additional search algorithms for extra credit. This work demonstrates a comprehensive understanding of various graph search strategies, their theoretical foundations, and practical applications.

3 Completion Level of Each Requirement

3.1 Self Assessment

3.1.1 Implementation of 5 core algorithms

- Breadth First Search (BFS): 10
- Depth First Search (DFS): 10
- Uniform-Cost Search (UCS):
- Greedy Best First Search (GBFS): 10
- A* Search: 10

3.1.2 Implementation of additional algorithms

- Iterative Deepening Search (IDS): 10
- Bidirectional Search (BDS): 10
- A* with Manhattan Distance: 10

3.2 5 test cases for each algorithm

3.2.1 test case 1

4 Basic Theories

4.1 Breadth First Search (BFS) Algorithm

Concepts

- **Introduction:** BFS is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.
- **Traversal Method:** BFS uses a queue data structure to keep track of nodes to be explored. It starts from a source node and explores all its neighboring nodes level by level.

Pseudo Code

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
    node←NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier←a FIFO queue, with node as an element
    reached← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node←POP(frontier)
        for each child in EXPAND(problem, node) do
        s←child.STATE
        if problem.IS-GOAL(s) then return child
        if s is not in reached then
            add s to reached
            add child to frontier
return failure
```

Complexity

- **Time Complexity:** $O(V + E)$, where V is the number of vertices and E is the number of edges. This is because each vertex and each edge is processed once.
- **Space Complexity:** $O(V)$, as additional space is required for the queue and the visited set.

Properties

- **Completeness:** BFS is complete, meaning it will always find a solution if one exists, given that the graph is finite.
- **Optimality:** BFS is optimal if all edges have the same weight or no weights. It finds the shortest path in an unweighted graph.
-
- **Traversal Type:** UCS explores nodes with the lowest path cost first.

4.2 Depth First Search (DFS) Algorithm

Concepts

- **Introduction:** Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It traverses deeper into the graph, exploring nodes in depth-first order.
- **Traversal Method:** DFS uses a recursion (or stack) to keep track of nodes to be explored. It starts from a source node and explores as far as possible along each branch before backtracking.

Pseudo Code

```
function RECURSIVE-DFS(node, problem) returns a solution node or failure
    if problem.IS-GOAL(node.STATE) then return node

    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if s is not visited then
            mark s as visited
            result ← RECURSIVE-DFS(child, problem)
            if result != failure then return result

    return failure

function DEPTH-FIRST-SEARCH(problem) returns a solution node or failure
    node ← NODE(problem.INITIAL)
    mark node.STATE as visited
    return RECURSIVE-DFS(node, problem)
```

Complexity

- **Time Complexity:** $O(V + E)$, where V is the number of vertices and E is the number of edges. In the worst case, DFS explores all vertices and edges of the graph.
- **Space Complexity:** $O(V)$, as additional space is required for the stack and the visited set.

Properties

- **Completeness:** DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.
- **Optimality:** DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.
- **Traversal Type:** Recursive DFS is a depth-first traversal.

4.3 Uniform-Cost Search (UCS)

Concepts

- **Introduction:** Uniform-Cost Search (UCS) is a graph search algorithm that finds the lowest cost path from a starting node to a goal node in a weighted graph.
- **Traversal Method:** UCS uses a priority queue ordered by path cost to expand nodes with the lowest cumulative cost.

Pseudo Code

```
function BEST-FIRST-SEARCH(problem,f) returns a solution node or failure
    node←NODE(STATE=problem.INITIAL)
    frontier←a priority queue ordered by f , with node as an element
    reached←a lookup table, with one entry with key problem.INITIAL and value node
    while not IS-EMPTY(frontier) do
        node←POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            s←child.STATE
            if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
                reached[s]←child
                add child to frontier
    return failure
function EXPAND(problem, node) yields nodes
    s←node.STATE
    for each action in problem.ACTIONS(s) do
        s' ←problem.RESULT(s, action)
        cost←node.PATH-COST + problem.ACTION-COST(s, action,s')
        yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
    return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Complexity

- **Time Complexity:** $O((V+E)\log V)$, where V is the number of vertices and E is the number of edges. UCS uses a priority queue, and each edge and vertex may be processed multiple times.
- **Space Complexity:** $O(V)$, as additional space is required for the priority queue and the visited set.

Properties

- **Completeness:** UCS is complete, as it will always find a solution if one exists, given non-negative edge costs.
- **Optimality:** UCS is optimal; it finds the lowest cost path if all edge costs are non-negative.
- **Traversal Type:** UCS explores nodes with the lowest path cost first.

4.4 Greedy Best First Search (GBFS) Algorithm

Concepts

- **Introduction:** Greedy Best First Search (GBFS) is a graph search algorithm that expands the most promising node chosen according to a heuristic function. It does not guarantee optimal solutions but is often efficient in practice for certain types of problems.
- **Traversal Method:** GBFS uses a priority queue ordered by the heuristic value to expand nodes with the most promising estimated cost.

Pseudo Code

function BEST-FIRST-SEARCH(problem, f) was mentioned in Section 4.3.

```
function GREEDY-BEST-FIRST-SEARCH(problem) returns a solution node or failure
    return BEST-FIRST-SEARCH(problem, HEURISTIC)
```

Complexity

- **Time Complexity:** The worst-case time complexity of GBFS is $O(|V|)$, where V is the number of vertices. With a good heuristic function, however, the complexity can be substantially reduced, potentially reaching $O(b^m)$ on certain problems, where b is the branching factor and m is the maximum depth of the search.
- **Space Complexity:** GBFS has a space complexity of $O(|V|)$, primarily due to the priority queue and the visited set.

Properties

- **Completeness:** GBFS is complete in finite state spaces but may not be complete in infinite state spaces due to the potential for encountering loops or cycles.
- **Optimality:** GBFS is not generally optimal because it does not consider the total path cost but relies solely on the heuristic function to guide the search.
- **Traversal Type:** GBFS explores nodes based on the heuristic value, prioritizing nodes that appear to be closest to the goal according to the heuristic.

4.5 A* Search Algorithm

Concepts

- **Introduction:** A* Search is a graph search algorithm that combines the advantages of both Dijkstra's algorithm and greedy best-first search. It uses both the actual cost from the start node (g-value) and an estimated cost to the goal node (h-value) to guide the search.
- **Traversal Method:** A* Search uses a priority queue ordered by the sum of g-value and h-value (f-value) to expand nodes with the lowest estimated total cost first.

Pseudo Code

function BEST-FIRST-SEARCH(problem, f) was mentioned in Section 4.3.

```
function A-STAR-SEARCH(problem) returns a solution node or failure
    return BEST-FIRST-SEARCH(problem, g+h)
```

Complexity

- **Time Complexity:** In the worst case, A* Search has a time complexity of $O(b^d)$, where b is the branching factor of the graph and d is the depth of the optimal solution. However, with a good heuristic function, it often performs much better.
- **Space Complexity:** A* Search has a space complexity of $O(|V|)$, where V is the number of vertices, due to the priority queue and the visited set.

Properties

- **Completeness:** A* Search is complete if a solution exists, given that the branching factor is finite and all edge costs are non-negative.
- **Optimality:** A* Search is optimal if the heuristic function $h(s)$ is admissible (never overestimates the true cost to reach the goal) and consistent (satisfies the triangle inequality).
- **Traversal Type:** A* Search explores nodes based on the f-value, which combines the g-value (cost from start to current node) and the h-value (estimated cost from current node to goal).

5 Comparison of UCS and A* Algorithms

	Uniform-Cost Search (UCS)	**A* Search**
Heuristic Function	Does not use a heuristic function. It purely considers the path cost from the start node to the current node ($g(n)$).	Incorporates a heuristic function ($h(n)$) in addition to the path cost. The total cost function for A* is $f(n) = g(n) + h(n)$, where $h(n)$ is an estimate of the cost from the current node to the goal.
Cost Evaluation	Chooses the node with the minimum path cost ($g(n)$) among nodes that have not been visited but have a neighbor that has been visited.	Chooses the node with the minimum combined cost ($g(n)+h(n)$) among nodes in the frontier.
Optimality	Guaranteed to find an optimal solution if the heuristic used in A* is admissible (never overestimates the true cost).	
Time and Space Complexity	Have the same time and space complexity in the worst case. However, A* can be more efficient in practice due to the heuristic guiding the search more effectively.	
Completeness	Complete, meaning if there is a solution, both algorithms will find it.	
Practical Benefit	Does not benefit from heuristic guidance and may explore more nodes than necessary.	Allows the use of a heuristic, which in non-worst-case scenarios can significantly reduce the number of nodes explored, providing a large practical benefit.

Table 1: Comparison of UCS and A* Algorithms