Vietnam National University,
Ho Chi Minh City

University of Science

Faculty of Information Technology

Project 01: Ares's Adventure Report

CS14003 – Introduction to Artificial Intelligence

Ngo Nguyen The Khoa	23127065
Bui Minh Duy	23127040
Nguyen Le Ho Anh Khoa	23127211

March 3, 2025

Contents

1 Group Information

- **Subject:** Introduction to Artificial Intelligence.
- **Class:** 23CLC09.
- **Lecturer:** Bui Duy Dang, Le Nhut Nam.
- **Team members:**

No.	Fullname	Student ID	Email
1	Ngo Nguyen The Khoa	23127065	nntkhoa23@clc.fitus.edu.vn
2	Bui Minh Duy	23127040	bmduy23@clc.fitus.edu.vn
3	Nguyen Le Ho Anh Khoa	23127211	nlhakhoa23@clc.fitus.edu.vn

- **Tools:**
  - **Git**, **GitHub**: Source code version control.
  - **DrawIO**: UML drawing.
  - **CapCut**: Video editing.
  - **ChatGPT**, **Gemini** and **DeepSeek**.
  - **Visual Studio Code**: Code editing (Python, Latex).

2 Project Information

- **Name:** Ares.
- **Developing Environment:** Visual Studio Code (Windows).
- **Programming Language:** Python.
- **Libraries and Tools:**
  - **rye:** A comprehensive project and package management solution for Python.
  - **CustomTkinter:** GUI library.

3 Work assignment table

No.	Task Description	Assigned to
1	desc	member

4 Screenshots

4.1 Breadth First Search

Breadth First Search (BFS), as the name says, explores the search space in the increasing order of the depth and the costs of traveling from one state to another is assumed to be a positive number. Typically, this algorithm is often associated with the concept of stack and queue and pushing and popping from the stack.

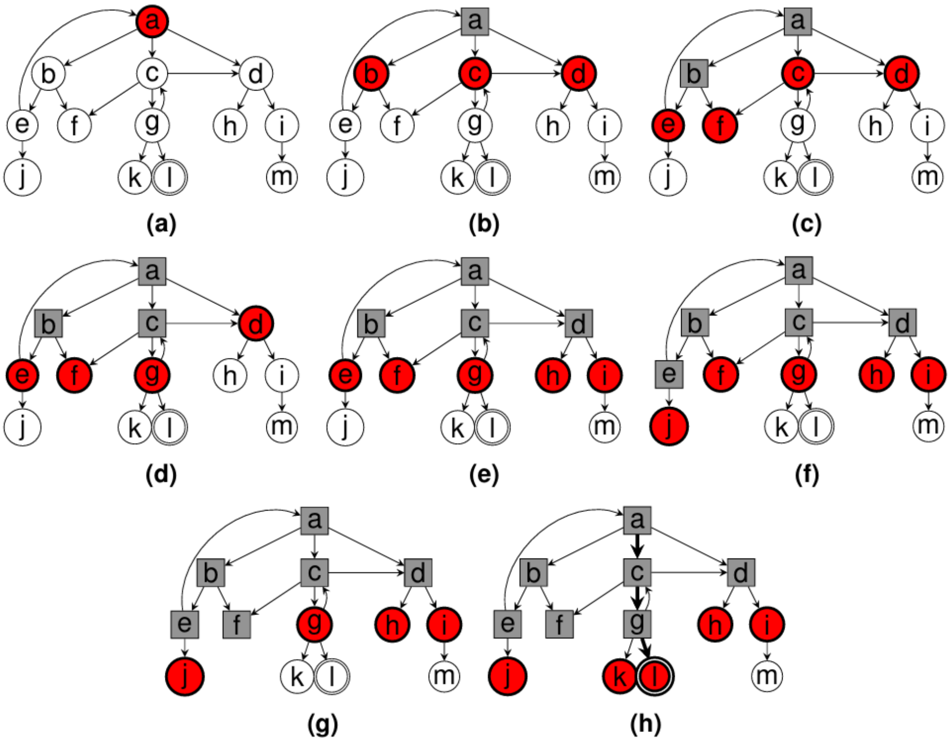

Figure 1: Breadth First Search

Pseudocode

Algorithm 1 Breadth First Search (*start, goal*)

```
queue ← [start]
while queue is not empty do
    node ← dequeue(queue)
    if node = goal then
        return path
    end if
    for all neighbor in valid moves do
        if neighbor not visited then
            mark neighbor as visited
            enqueue(queue, neighbor)
        end if
    end for
end while
return failure
```

Implementation

Time and Space Complexity

Time Complexity: $O(V+E)$, where V is the number of vertices and E is the number of edges. Each node and edge is processed once.

Space Complexity: $O(V)$ in the worst case, as the queue stores all nodes at the widest level of the graph.

4.2 Depth First Search

Depth First Search (DFS) is a special case of backtracking search algorithm. The search starts from the root and proceeds to the farthest node before backtracking. The difference between this and the backtracking is that this stops the search once a goal is reached and does not care if it is not minimum.

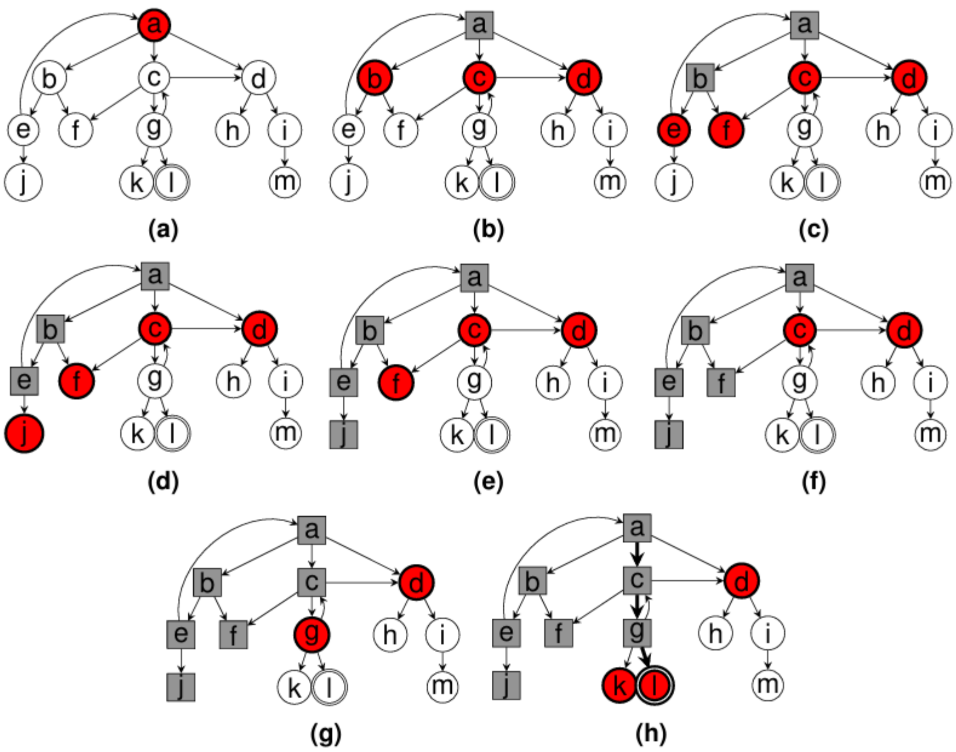

Figure 2: Depth First Search

Pseudocode

Algorithm 2 Depth First Search (*start, goal*)

```
stack ← [start]
while stack is not empty do
    node ← pop(stack)
    if node = goal then
        return path
    end if
    for all neighbor in valid moves do
        if neighbor not visited then
            mark neighbor as visited
            push(stack, neighbor)
        end if
    end for
end while
return failure
```

Implementation

Time and Space Complexity

4.3 Uniform Cost Search

For any search problem, Uniform Cost Search (UCS) is the better algorithm than the previous ones. The search algorithm explores in branches with more or less same cost. This consist of a priority queue where the path from the root to the node is the stored element and the depth to a particular node acts as the priority. UCS assumes all the costs to be non negative. While the DFS algorithm gives maximum priority to maximum depth, this gives maximum priority to the minimum cumulative cost.

Pseudocode

Algorithm 3 Uniform Cost Search (*start, goal*)

```
priority queue ← [(start, cost = 0)]
while priority queue is not empty do
    (node, cost) ← dequeue(priority queue)
    if node = goal then
        return path
    end if
    for all neighbor in valid moves do
        new cost ← cost + move cost
        if neighbor not visited or new cost < previous cost then
            mark neighbor as visited
            enqueue(priority queue, (neighbor, new cost))
        end if
    end for
end while
return failure
```

Implementation

Time and Space Complexity

4.4 A* Search with heuristic

A* algorithm is one of the popular technique used in path finding and graph traversals. This algorithm completely relies on heuristics for computing the future cost of a problem. This algorithm is equivalent to the uniform cost search with modified edge cost. This heuristics is chosen according to the case where the algorithm is implemented, thus emphasizing the importance of domain knowledge. This algorithm is consistent if the modified cost is greater than zero.

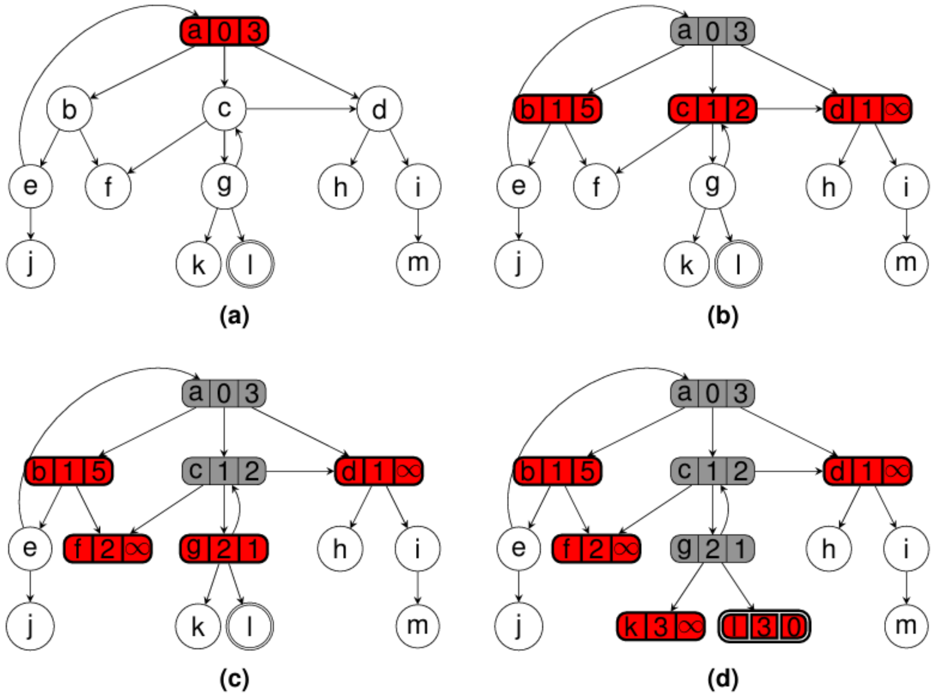

Figure 3: A* Algorithm

Pseudocode

Algorithm 4 A* Search (*start, goal, heuristic*)

priority queue ← [(start, cost = 0, estimated total cost = heuristic(start))]
while priority queue is not empty **do**
    (node, cost) ← dequeue(priority queue)
    if node = goal **then**
        return path
    end if
    for all neighbor in valid moves **do**
        new cost ← cost + move cost
        estimated total cost ← new cost + heuristic(neighbor)
        if neighbor not visited or new cost < previous cost **then**
            mark neighbor as visited
            enqueue(priority queue, (neighbor, new cost, estimated total cost))
        end if
    end for
end while
return failure

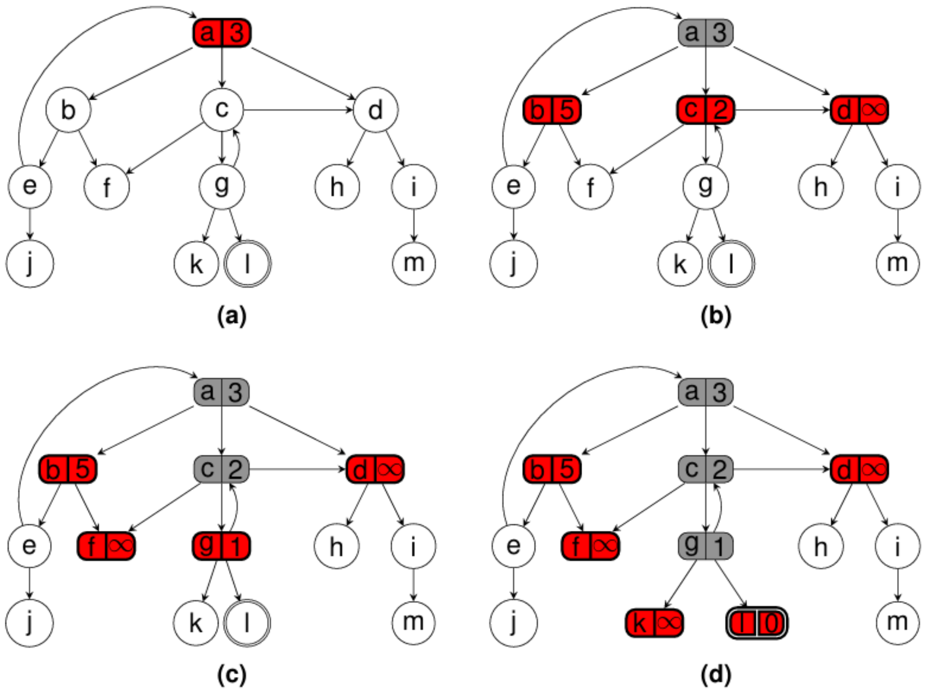

Figure 4: Greedy Best-First Search

Implementation

Time and Space Complexity

4.5 Greedy Best-First Search

Greedy Best-First Search is similar to A* algorithm. The difference is that the vertices in the priority queue are ordered only by the estimated remaining distance to the solution. It has to be noted that complete best first search is not optimal.

Pseudocode

Algorithm 5 Greedy Best-First Search (*start, goal, heuristic*)

```
priority queue ← [(start, heuristic(start))]
while priority queue is not empty do
    node ← dequeue(priority queue)
    if node = goal then
        return path
    end if
    for all neighbor in valid moves do
        if neighbor not visited then
            mark neighbor as visited
            enqueue(priority queue, (neighbor, heuristic(neighbor)))
        end if
    end for
end while
return failure
```

Time and Space Complexity

4.6 Dijkstra's Algorithm

Pseudocode

Algorithm 6 Dijkstra's Algorithm (*start, goal*)

```
priority queue ← [(start, cost = 0)]
distances[start] ← 0
while priority queue is not empty do
    (node, cost) ← dequeue(priority queue)
    if node = goal then
        return distances
    end if
    for all neighbor in valid moves do
        new cost ← cost + move cost
        if new cost < distances[neighbor] then
            distances[neighbor] ← new cost
            enqueue(priority queue, (neighbor, new cost))
        end if
    end for
end while
return distances
```

Implementation

Time and Space Complexity

4.7 Swarm Algorithm

Pseudocode

Implementation

Time and Space Complexity

5 References

1. Rye documentation