Vietnam National University, Ho Chi Minh City
University Of Science

Introduction to Artificial Intelligent

Report Document

Project 1: Searching Algorithm

Advisors: Prof. Nguyễn Thanh Tình

HO CHI MINH CITY, JULY 2024

List of Members

Num.	Full Name	Student ID
1	Nguyễn Duy Ân	22127006

Self-evaluation

Num.	Details	Completion Rate
1	Implement DFS	100%
2	Implement BFS	100%
3	Implement UCS	100%
4	Implement IDS	100%
5	Implement GBFS	100%
6	Implement A*	100%
7	Implement Hill-climbing	100%
8	Generate 5 test cases for all algorithms with different attributes.	100%
9	Report the implementation and experiment of these algorithms	100%

Contents

1 Algorithm Description

In this initial section, we will provide an overview of various search algorithms, explaining how they function. We will also discuss their time and memory costs based on their complexity.

1.1 Breadth-First Search (BFS)

1. **Description** : Breadth-First Search is an uninformed search algorithm that begins at the root node and explores all nodes at the current level before proceeding to nodes at the next level. The level of a tree or graph can be determined by counting the number of descendants from the current node to the root node.

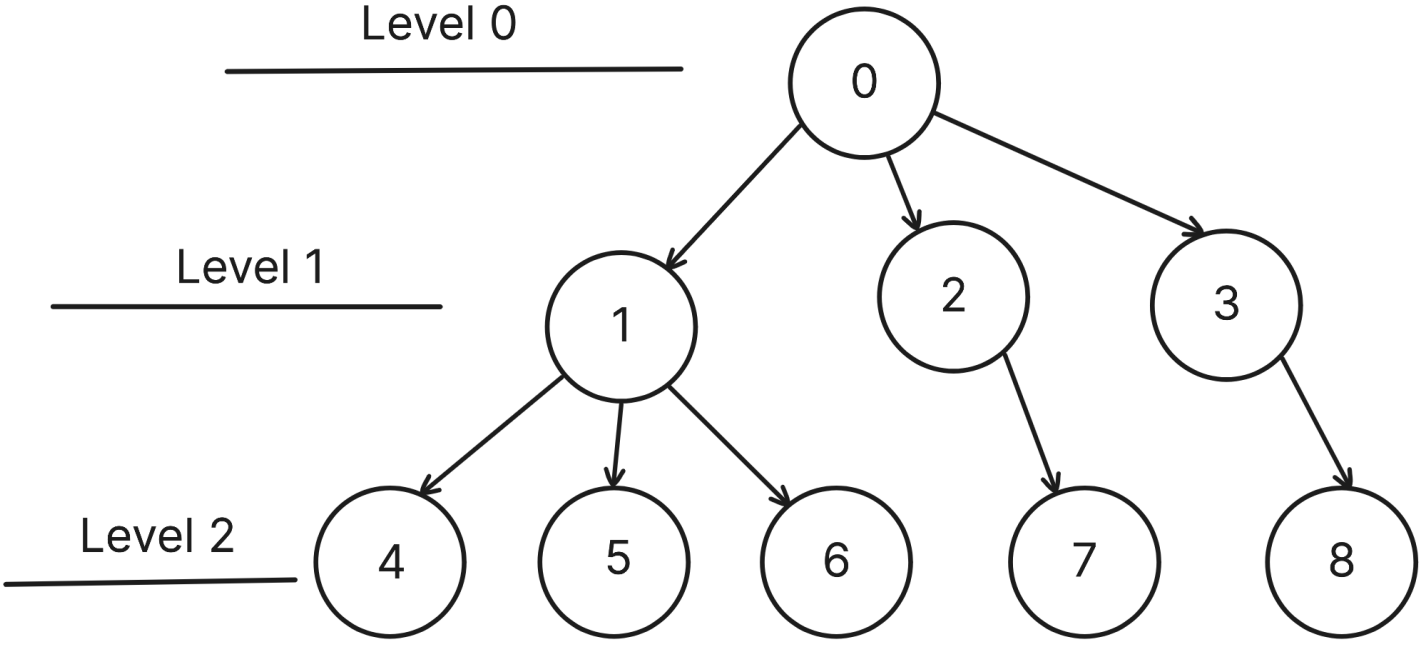


Figure 1: The tree graph with level

2. **Basic Implement**:

   - Breadth-First Search begins at the root node (source node) and requires a data structure to store generated nodes, typically a queue. A queue operates on a First In First Out (FIFO) principle, ensuring that nodes are processed in the order they were generated. This allows the algorithm to explore the lowest-level child nodes first. We also use a visited array to put all of visited node to that, and also to store the parent of each node, allowing us to trace the path from the source to the goal when the goal node is found.
   - The basic operations of Breadth-First Search can be describe in the following step:
     (a) Initialize the queue, push the root node into queue.
     (b) Create a main loop for traverse node, this loop will stop when there is no any node left in queue.
     (c) Expand a last node in a queue, we has a bunch of its children, if any of its children is a goal node, we return a path from root node to goal, otherwise we push node to queue and save the tracing path by the visited array.
     (d) Repeat until end of loop.
     (e) When we are out of the loop but have not meets a goal node yet, it means that there is no path from source to goal, we return -1 to announce it.

3. **Complexity** [1]:

- **Time Complexity**: If every nodes has b successors and the highest level is d, in the worst case, the goal node is in the highest level, so the level we have to traverse is d. So that, the number of nodes will be generate is $O(b^d)$
- **Space Complexity**: A space complexity base on the total space need for a queue and a visited array, in the worst case, we need to store all of node in queue, so that there will be $O(b^{d-1}$ in the visited array and $O(b^d)$ in the queue.
- **Conclusion** : A complexity of space is the big problem for BFS, if the graph is enormous, then we must to store big number of nodes in both queue and visited array. And also in the worst case, it seemly that we have to traverse to all node before reaching goal node, this is not optimal in practical for solving searching problem.

1.2 Depth-First Search (DFS)

1. **Description** : Depth-First Search is an uninformed search that will start as a root node and will go as far as possible to reach the highest level of the tree or graph. To perform this technique, Depth-First Search will using recursion or the stack data structure to explore all of a node's descendants before moving to its sibling.

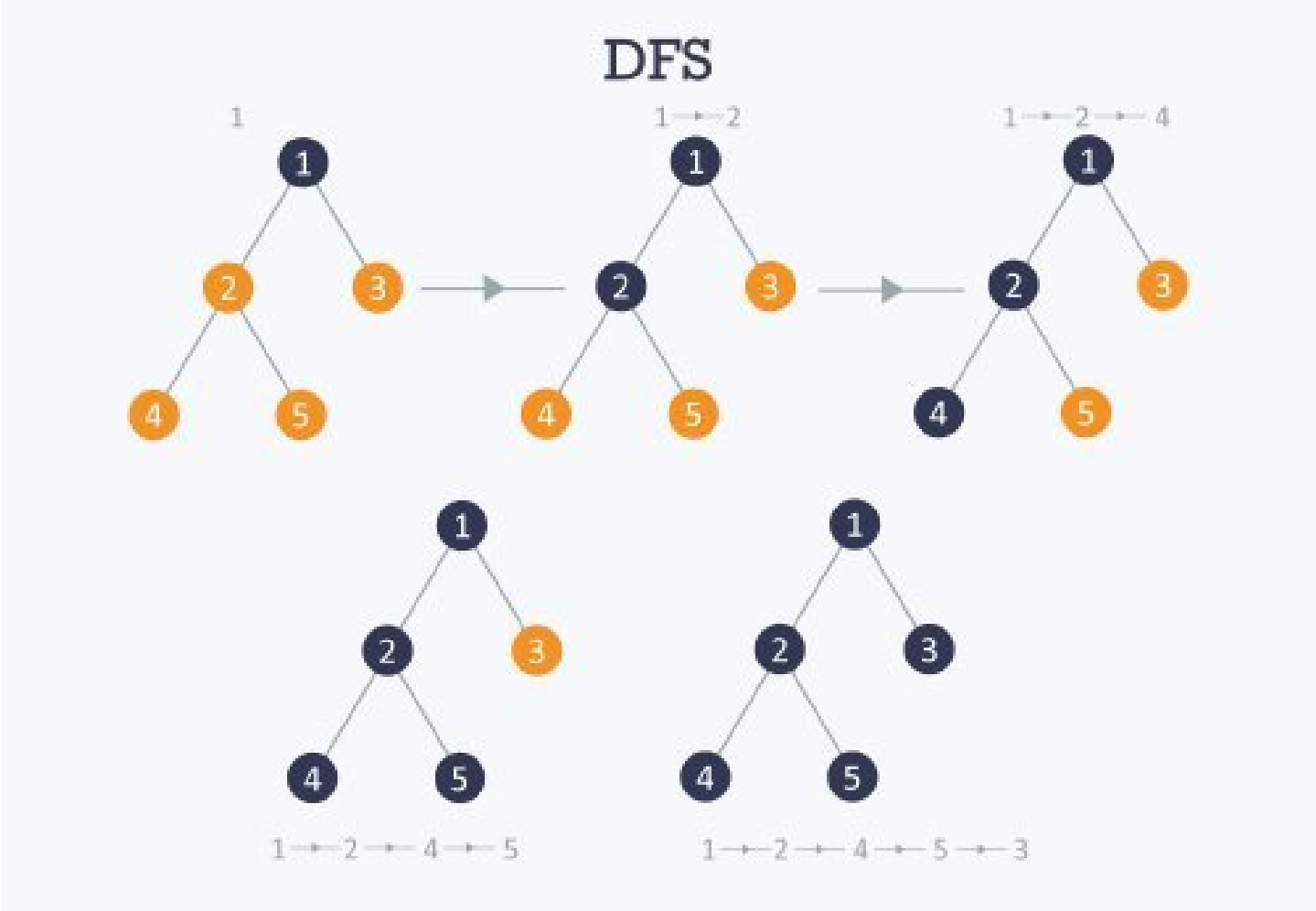


Figure 2: DFS traversal (source : Quora.com)

2. **Basic Implement**:

- Depth-First Search begins at a root node (source node), it will require a data structure to store a generated node, typically a stack. A stack operates on a Last In First Out (LIFO), ensuring the most recently generated nodes are processing first. This help the algorithms meets all the descendants of the node first before its siblings. We

also use the visited array to keep track all of visited nodes, allowing us to trace the path from source node to goal node once we reach the goal node.

- The basic operations of Depth-First Search can be described in the following steps:
  (a) Initialize the stack and push the source node onto the stack.
  (b) Create a main loop to traverse the nodes, which will stop if there is no node left in the stack.
  (c) At each node generated, we expand it and append all of its children to the stack, if any of its children is a goal node, the we return the path from source to goal. Otherwise, push it onto stack and traced by visited array.
  (d) Repeat the loop until there is no one node left
  (e) If the loop ends without finding the goal of the node, it means that there is no path from the source node to the goal node, return -1.

3. Complexity [1]

- **Time Complexity** : In the worst case, DFS may traverse all the nodes, so the time complexity will be $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree.
- **Space Complexity** : The space complexity is also $O(b^m)$, as the algorithms needs to store the nodes on the current path in stack.
- **Conclusion** : DFS is more memory - efficient than BFS if in the graph that has to traverse in the deepest level, DFS may not find the shortest path to the goal node and can get stuck in deep or infinite loops if the graph is in very deep branches.

1.3 Uniform Cost Search (UCS)

1. **Description** : Uniform Cost Search is an uninformed search algorithm that expands the node with the lowest path cost first. This algorithm is similar to BFS, but it considers the cost of each path, making it suitable for finding the shortest path in a weighted graph.

2. **Basic Implementation**:

- Uniform Cost Search starts at the root node (source node) and uses a priority queue to store generated nodes, where the priority is based on the path cost. This ensures that nodes are processed in order of their path cost, exploring the least costly paths first. We also use a visited array to track visited nodes and a parent array to store the parent of each node, allowing us to trace the path from the source to the goal when the goal node is found.
- The basic operations of Uniform Cost Search can be described in the following steps:
  (a) Initialize the priority queue and push the root node into the queue with a path cost of 0.
  (b) Create a main loop to traverse nodes, which stops when there are no nodes left in the priority queue.
  (c) Expand the node with the lowest path cost from the priority queue. If the node is the goal, return the path from the root node to the goal. Otherwise, generate its children and push them onto the priority queue with their corresponding path costs, updating the tracing path using the visited array.
  (d) Repeat until the loop ends.
  (e) If the loop ends without finding the goal node, it means there is no path from the source to the goal, so return -1 to indicate this.

3. **Complexity** [1]:

- **Time Complexity**: The time complexity of UCS is $O(b^{1+\lfloor C^/ \epsilon \rfloor})$, where C is the cost of the optimal solution, b is the branching factor, and ϵ is the minimum edge cost.

- **Space Complexity**: The space complexity is also $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ due to the storage requirements of the priority queue and the visited array.
- **Conclusion**: UCS is optimal and complete, ensuring that it finds the least costly path to the goal if one exists. However, it can be slow and memory-intensive for large graphs with many nodes and high branching factors.

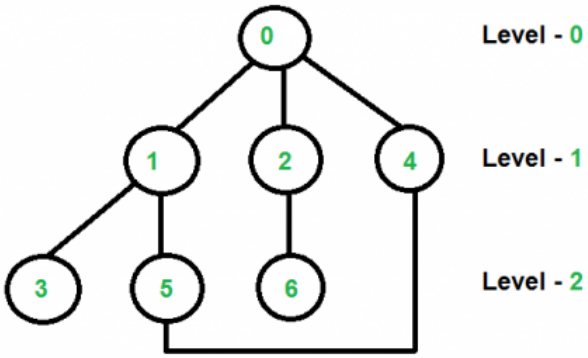
1.4 Depth-Limited Search (DLS)

1. **Description** : Depth-Limited Search is a variation of Depth-First Search that limits the depth of the search to a predetermined level l. This can prevent the algorithm from going down an infinite path in a graph with cycles or extremely deep branches.
2. **Basic Implementation**: [1]
   - Depth-Limited Search begins at the root node (source node) and uses a stack for node storage, similar to DFS. The main difference is the addition of a depth limit to prevent the search from exploring beyond a specified depth.
   - The basic operations of Depth-Limited Search can be described in the following steps:
     (a) Initialize the stack and push the root node onto the stack along with its depth (0).
     (b) Create a main loop to traverse nodes, which stops when there are no nodes left in the stack.
     (c) Expand the last node on the stack, generating its children. If any of its children is a goal node, return the path from the root node to the goal. Otherwise, push the node onto the stack if its depth is less than the depth limit and save the tracing path using the visited array.
     (d) Repeat until the loop ends.
     (e) If the loop ends without finding the goal node, it means there is no path from the source to the goal within the depth limit, so return -1 to indicate this.
3. **Complexity**:
   - **Time Complexity**: In the worst case, DLS may traverse all nodes within the depth limit, so the time complexity is $O(b^l)$, where b is the branching factor and l is the depth limit.
   - **Space Complexity**: The space complexity is $O(b^l)$, as the algorithm needs to store the nodes on the current path in the stack up to the depth limit.
   - **Conclusion**: DLS is memory-efficient and can avoid the issues of infinite paths, but it may not find a solution if the goal node is beyond the depth limit.

1.5 Iterative Deepening Search (IDS)

1. **Description** : Iterative Deepening Search combines the benefits of Depth-First Search and Breadth-First Search. It repeatedly applies Depth-Limited Search with increasing depth limits until the goal is found.

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

Figure 3: The tree graph with Iterative Deepening Search traversal (source : GeeksforGeeks)

2. **Basic Implementation**: [1]
   - Iterative Deepening Search performs a series of Depth-Limited Searches with increasing depth limits until the goal is found or all nodes are explored.
   - The basic operations of Iterative Deepening Search can be described in the following steps:
     (a) Initialize the depth limit to 0.
     (b) Perform a Depth-Limited Search with the current depth limit.
     (c) If the goal is found, return the path from the root node to the goal.
     (d) Increment the depth limit and repeat the Depth-Limited Search.
     (e) Repeat until the goal is found or all nodes are explored.
3. **Complexity**:
   - **Time Complexity**: The time complexity is $O(b^d)$, where b is the branching factor and d is the depth of the shallowest goal node. Each level is visited multiple times, but the overall complexity is dominated by the last level.
   - **Space Complexity**: The space complexity is $O(bd)$, as the algorithm needs to store the nodes on the current path in the stack for the deepest level.
   - **Conclusion**: IDS combines the memory efficiency of DFS with the completeness and optimality of BFS, making it suitable for many practical search problems.

1.6 Greedy Best-First Search

1. **Description** : Greedy Best-First Search is an informed search algorithm that expands the node that appears to be closest to the goal based on a heuristic function. It uses a priority queue to manage the nodes to be expanded, where the priority is determined by the heuristic function.

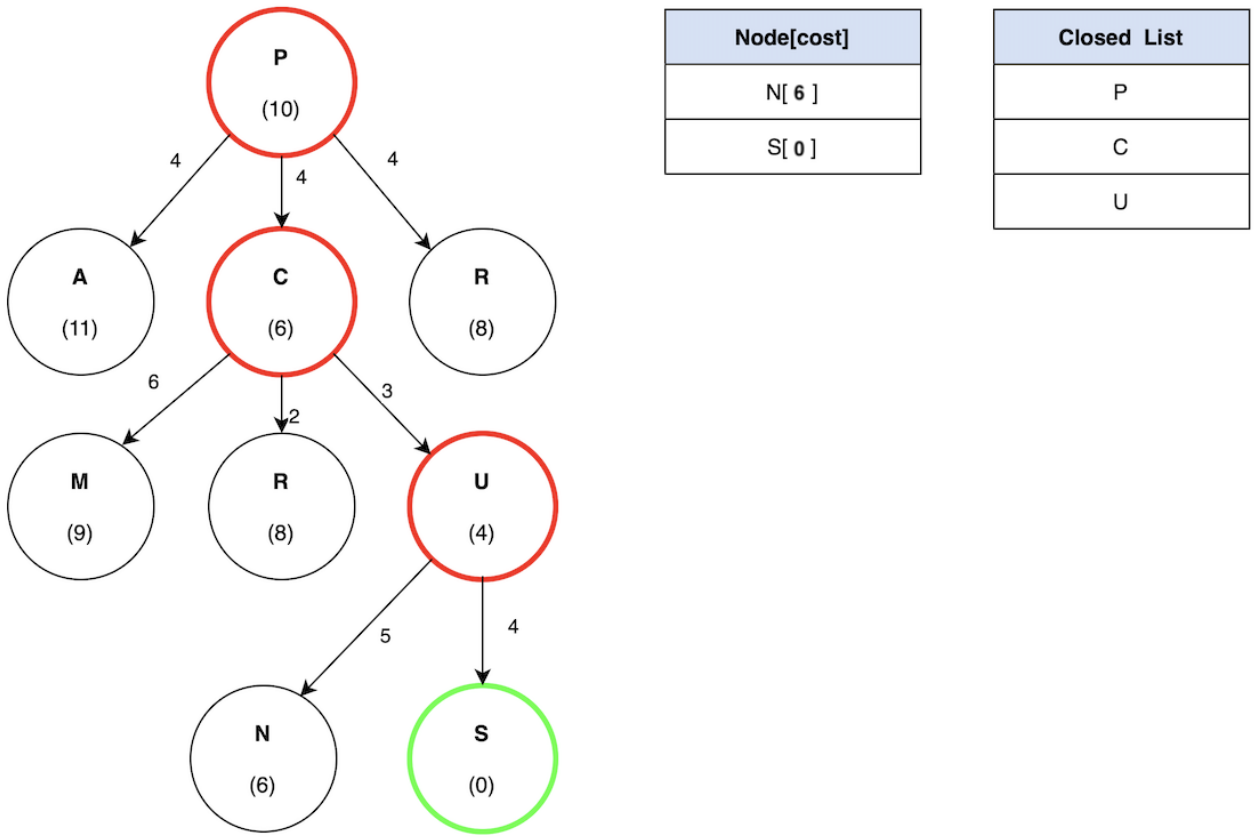


Figure 4: The tree graph with Greedy Best-First Search traversal (source: GeeksforGeeks)

2. **Basic Implementation**: [2]

   - Greedy Best-First Search starts at the root node and uses a priority queue to store generated nodes, prioritizing nodes based on the heuristic function.
   - The basic operations of Greedy Best-First Search can be described in the following steps:
     (a) Initialize the priority queue and push the root node into the queue with its heuristic value.
     (b) Create a main loop to traverse nodes, which stops when there are no nodes left in the priority queue.
     (c) Expand the node with the lowest heuristic value from the priority queue. If the node is the goal, return the path from the root node to the goal. Otherwise, generate its children and push them onto the priority queue with their heuristic values, updating the tracing path using the visited array.
     (d) Repeat until the loop ends.
     (e) If the loop ends without finding the goal node, it means there is no path from the source to the goal, so return -1 to indicate this.

3. **Complexity**

   - **Time Complexity**: The time complexity is $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree.
   - **Space Complexity**: The space complexity is $O(b^m)$, as the algorithm needs to store all nodes in the priority queue.

- **Conclusion**: Greedy Best-First Search is not optimal or complete, but it can be faster than uninformed search algorithms in practice. It heavily depends on the quality of the heuristic function.

1.7 Graph Search (A*)

1. **Description** : A* is an informed search algorithm that expands the node with the lowest combined cost of the path from the root to the node and the estimated cost from the node to the goal. It uses a priority queue to manage the nodes to be expanded, where the priority is determined by the sum of the path cost and the heuristic function.

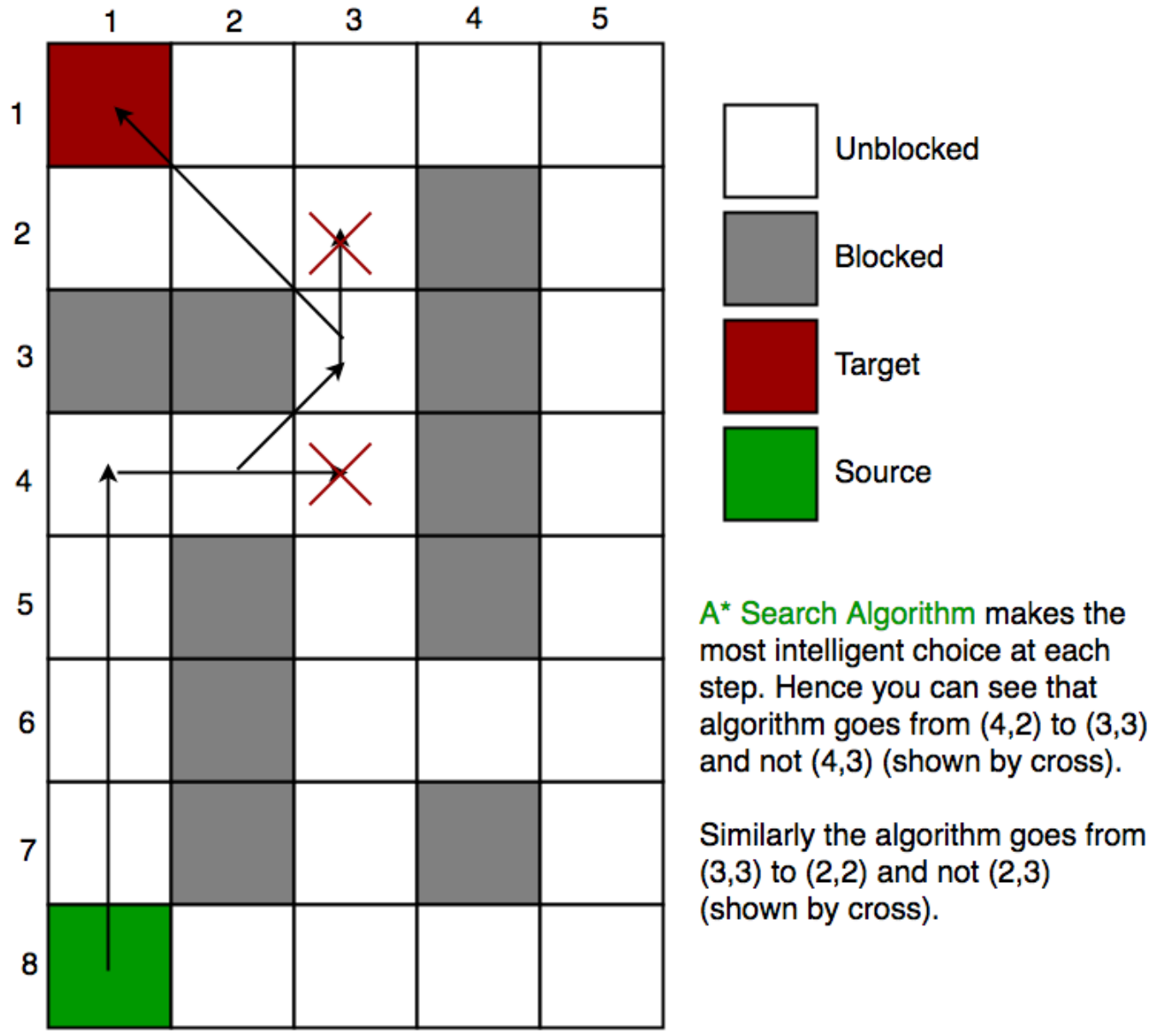


Figure 5: The tree graph with A* Search traversal (source: GeeksforGeeks)

2. **Basic Implementation**: [1]

- A* Search starts at the root node and uses a priority queue to store generated nodes, prioritizing nodes based on the sum of the path cost and the heuristic function.

- The basic operations of A* Search can be described in the following steps:
  (a) Initialize the priority queue and push the root node into the queue with its path cost plus heuristic value.
  (b) Create a main loop to traverse nodes, which stops when there are no nodes left in the priority queue.
  (c) Expand the node with the lowest combined cost from the priority queue. If the node is the goal, return the path from the root node to the goal. Otherwise, generate its children and push them onto the priority queue with their combined cost, updating the tracing path using the visited array.
  (d) Repeat until the loop ends.
  (e) If the loop ends without finding the goal node, it means there is no path from the source to the goal, so return -1 to indicate this.

3. **Complexity**:
   - **Time Complexity**: The time complexity is $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree.
   - **Space Complexity**: The space complexity is $O(b^m)$, as the algorithm needs to store all nodes in the priority queue.
   - **Conclusion**: A* is both optimal and complete, provided that the heuristic function is admissible and consistent. It is one of the most widely used search algorithms in AI.

1.8 Hill Climbing (First Choice)

1. **Description** : Hill Climbing is a local search algorithm that continuously moves in the direction of increasing value (or decreasing cost) to find the peak of the mountain (or the minimum cost). The First Choice variation randomly selects a successor and moves to it if it improves the evaluation function.

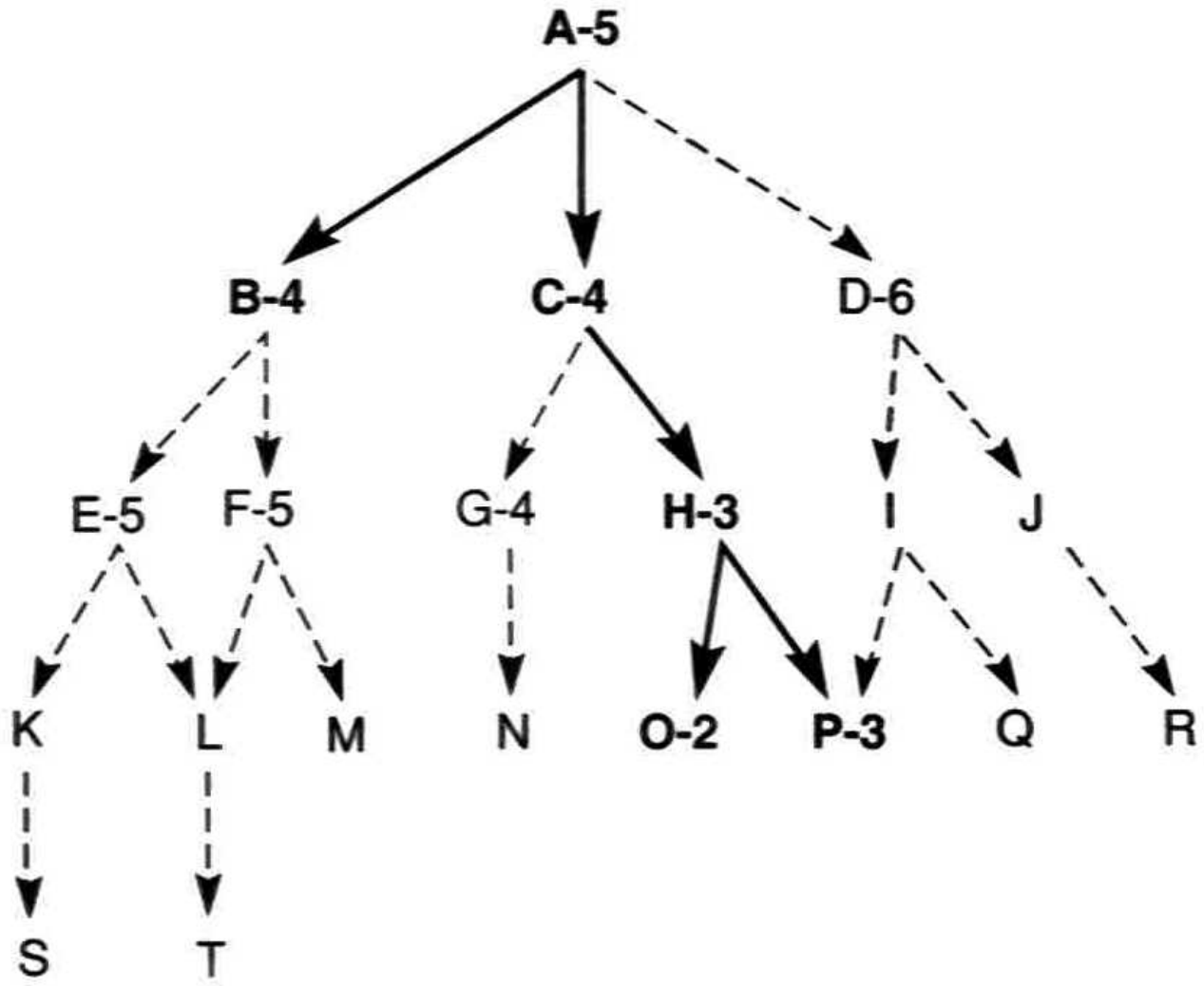


Figure 6: The tree graph with Hill Climbing (First Choice) traversal, (source : Mgformats)

2. **Basic Implementation**: [3]
   - Hill Climbing (First Choice) starts at the root node and randomly selects a successor, moving to it if it improves the evaluation function.
   - The basic operations of Hill Climbing (First Choice) can be described in the following steps:
     (a) Initialize the current node to the root node.
     (b) Create a loop to traverse nodes, which stops when no improvement can be made.
     (c) Randomly select a successor of the current node.
     (d) If the successor improves the evaluation function, move to the successor and update the current node.
     (e) Repeat until no improvement can be made.
     (f) If the loop ends without finding an improved node, return the current node as the result.
3. **Complexity** [1]:

- **Time Complexity**: The time complexity depends on the number of iterations and the number of successors generated at each step. It is generally $O(n)$, where n is the number of nodes.
- **Space Complexity**: The space complexity is $O(1)$, as the algorithm only needs to store the current node.
- **Conclusion**: Hill Climbing (First Choice) is simple and efficient for finding local optima, but it may get stuck in local optima and does not guarantee finding the global optimum.

2 Test Case Experiment

2.1 Test Case Description

To evaluate the performance and properties of various search algorithms, we present five distinct test cases. These cases are designed to highlight different aspects of the algorithms, such as handling unique paths, multiple paths, cycles, varying weights, and scalability. Each test case is detailed with the graph structure, including the number of nodes, start and goal nodes, adjacency matrix, and heuristic values where applicable. Below is a brief outline of each test case:
System Review

- System Name: Macbook Air Mid 2012
- Processor : 1,8 GHz Dual - Core Intel i5
- Graphics: Intel HD Graphics 4000 1536 MB
- RAM: 4gb RAM

Conventions : The runtime will be measured as second, and the defaul depth limit for DLS algorithm is 2.

1. **Test Case 1** : Small Graph
   Input

   ```
   4
   0 3
   0 1 1 0
   1 0 0 1
   1 0 0 1
   0 1 1 0
   3 2 1 0
   ```

   This test case will have this output as result :

   ```
   BFS
   Path: 0 -> 1 -> 3
   Runtime: 7.796287536621094e-05
   Memory usage: 2.09765625KB

   DFS
   Path: 0 -> 2 -> 3
   Runtime: 7.224082946777344e-05
   Memory usage: 0.982421875KB

   UCS
   Path: 0 -> 1 -> 3
   Runtime: 6.103515625e-05
   Memory usage: 1.123046875KB

   DLS
   Path: 0 -> 2 -> 3
   ```

```
Runtime: 3.1948089599609375e-05
Memory usage: 0.927734375KB

IDS
Path: 0 -> 2 -> 3
Runtime: 4.696846008300781e-05
Memory usage: 0.951171875KB

GBFS
Path: 0 -> 2 -> 3
Runtime: 3.314018249511719e-05
Memory usage: 1.185546875KB

ASTAR
Path: 0 -> 2 -> 3
Runtime: 4.9114227294921875e-05
Memory usage: 1.162109375KB

HC
Path: 0 -> 2 -> 3
Runtime: 0.0001049041748046875
Memory usage: 2.232421875KB
```

2. **Test Case: 2** : Medium - Sized graph with Multiple Paths
   Input

```
6
0 5
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 1 0 1
0 1 1 0 0 0
0 1 0 0 0 1
0 0 1 0 1 0
7 6 5 4 3 2
```

This test case will has this output as a result

```
BFS
Path: 0 -> 2 -> 5
Runtime: 4.315376281738281e-05
Memory usage: 0.826171875KB

DFS
Path: 0 -> 2 -> 5
```

```
Runtime: 2.7179718017578125e-05
Memory usage: 0.505859375KB

UCS
Path: 0 -> 2 -> 5
Runtime: 5.507469177246094e-05
Memory usage: 1.396484375KB

DLS
Path: 0 -> 2 -> 5
Runtime: 3.075599670410156e-05
Memory usage: 0.427734375KB

IDS
Path: 0 -> 2 -> 5
Runtime: 4.57763671875e-05
Memory usage: 0.474609375KB

GBFS
Path: 0 -> 2 -> 5
Runtime: 3.0994415283203125e-05
Memory usage: 0.685546875KB

ASTAR
Path: 0 -> 2 -> 5
Runtime: 4.029273986816406e-05
Memory usage: 1.216796875KB

HC
Path: -1
Runtime: 6.079673767089844e-05
Memory usage: 0.3671875KB
```

3. **Test Case 3: Graph with a cycle**
   Input

```
5
0 4
0 1 0 1 0
1 0 1 0 1
0 1 0 1 0
1 0 1 0 1
0 1 0 1 0
3 2 5 1 0
```

This test case will has this output as a result

```
BFS
```

```
Path: 0 -> 1 -> 4
Runtime: 2.7894973754882812e-05
Memory usage: 0.537109375KB

DFS
Path: 0 -> 3 -> 4
Runtime: 2.384185791015625e-05
Memory usage: 0.505859375KB

UCS
Path: 0 -> 1 -> 4
Runtime: 3.981590270996094e-05
Memory usage: 1.107421875KB

DLS
Path: 0 -> 3 -> 4
Runtime: 2.6941299438476562e-05
Memory usage: 0.427734375KB

IDS
Path: 0 -> 3 -> 4
Runtime: 4.00543212890625e-05
Memory usage: 0.474609375KB

GBFS
Path: 0 -> 3 -> 4
Runtime: 2.7179718017578125e-05
Memory usage: 0.685546875KB

ASTAR
Path: 0 -> 3 -> 4
Runtime: 3.314018249511719e-05
Memory usage: 0.638671875KB

HC
Path: 0 -> 1 -> 4
Runtime: 4.601478576660156e-05
Memory usage: 0.458984375KB
```

4. **Test Case 4** : Graph with different weights and heuristics.
   Input :

```
7
0 6
0 2 0 0 1 0 0
2 0 3 0 0 0 0
0 3 0 2 0 0 0
0 0 2 0 0 1 1
```

```
1 0 0 0 0 2 0
0 0 0 1 2 0 1
0 0 0 1 0 1 0
9 12 11 8 7 3 4
```

This test case will has this output as result

```
BFS
Path: 0 -> 4 -> 5 -> 6
Runtime: 6.389617919921875e-05
Memory usage: 0.875KB

DFS
Path: 0 -> 4 -> 5 -> 6
Runtime: 5.412101745605469e-05
Memory usage: 0.84375KB

UCS
Path: 0 -> 4 -> 5 -> 6
Runtime: 6.127357482910156e-05
Memory usage: 1.765625KB

DLS
Path: -1
Runtime: 2.6941299438476562e-05
Memory usage: 0.125KB

IDS
Path: 0 -> 4 -> 5 -> 6
Runtime: 7.295608520507812e-05
Memory usage: 0.8125KB

GBFS
Path: 0 -> 4 -> 5 -> 6
Runtime: 3.790855407714844e-05
Memory usage: 0.734375KB

ASTAR
Path: 0 -> 4 -> 5 -> 6
Runtime: 4.601478576660156e-05
Memory usage: 1.265625KB

HC
Path: -1
Runtime: 6.389617919921875e-05
Memory usage: 0.3671875KB
```

5. **Test Case 5** : Medium - Large Graph
   Input

```
10
0 9
0 1 0 0 1 0 0 0 0 0
1 0 1 0 0 1 0 0 0 0
0 1 0 1 0 0 1 0 0 0
0 0 1 0 1 0 0 1 0 0
1 0 0 1 0 0 0 0 1 0
0 1 0 0 0 0 1 0 0 1
0 0 1 0 0 1 0 1 0 0
0 0 0 1 0 0 1 0 1 0
0 0 0 0 1 0 0 1 0 1
0 0 0 0 0 1 0 0 1 0
15 14 13 12 11 10 9 8 5 3
```

   This test case will has this output as a result

```
BFS
Path: 0 -> 1 -> 5 -> 9
Runtime: 4.76837158203125e-05
Memory usage: 0.875KB

DFS
Path: 0 -> 4 -> 8 -> 9
Runtime: 3.361701965332031e-05
Memory usage: 0.84375KB

UCS
Path: 0 -> 1 -> 5 -> 9
Runtime: 9.298324584960938e-05
Memory usage: 1.734375KB

DLS
Path: -1
Runtime: 4.601478576660156e-05
Memory usage: 0.3671875KB

IDS
Path: 0 -> 4 -> 8 -> 9
Runtime: 7.700920104980469e-05
Memory usage: 0.8125KB

GBFS
```

```
Path: 0 -> 4 -> 8 -> 9
Runtime: 4.00543212890625e-05
Memory usage: 1.0234375KB

ASTAR
Path: 0 -> 4 -> 8 -> 9
Runtime: 4.9114227294921875e-05
Memory usage: 1.265625KB

HC
Path: 0 -> 1 -> 2 -> 6 -> 7 -> 8 -> 9
Runtime: 0.00015497207641601562
Memory usage: 1.083984375KB
```

2.2 Analysis

After running different search algorithm with 5 test cases, we have some analyses below:

1. BFS and DFS quickly find paths with minimal resources, but when memory usage become more critical, these algorithms start to show limitations, especially BFS will become impratical in the large graph.
2. Uniform Cost Search, A* and Greedy Best First Search perform most efficient through all of test.
3. Hill-Climbing and IDS show varying performances based on graph structure and heuristics
4. Hill-Climbing and DFS struggle with the increased complexity, highlighting their limitations in large graphs.

Conclusion:
Each algorithms has its unique strenghts and weekness, making them suitable for different types of problems. While BFS and DFS are fundamentals, UCS and A* will use for the weighted graphs. Greedy Best First Search and Hill-Climbing are fast but we need to care about the heuristic design. Iterative Deepening Search balance the extremes, providing flexibility and robustness in diverse scenarios. Select the algorithm will depend on the problem, graph structure and specific requirements for time and space optimality.

3 References

References

[1] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc, 2010.

[2] Prof. Dr. Malte Helmert. *A Case Study on the Search Topology of Greedy Best-First Search*. 2014.

[3] Kostadin Kratchanov, Emilia Golemanova, Tzanko Golemanov, and Tuncay Ercan. Non-procedural implementation of local heuristic search in control network programming. 01 2010.