Vietnam National University, Ho Chi Minh City

University of Science

Faculty of Information Technology

Lab Report

Lab: Searching

Course: Introduction to Artificial Intelligence

Author:
22127390 - Nguyen Van le Ba Thanh - 22CLC08

Instructors:
Mr. Bui Tien Len

July 14, 2024

Contents

1 Acknowledgement

Without the help and guidance of many individuals and from many different sources, this lab would not have been possible to complete.

First, I want to thank the developers and contributors of the Numpy, Tracemalloc, Time, and heapq libraries. The documentation available on their website is very helpful in getting acquainted and applying to install the necessary functions and algorithms for this project.

I also want to thank my teachers and instructors for their advice and lectures with the necessary knowledge throughout this subject. My understanding of this project was greatly enhanced by the clear explanations and openness to questions.

I especially want to thank ChatGPT, a large language model, for helping gather information and summarize information about search strategies, how they work, and their characteristics.

2 Completeness

No.	Details	Complete
1	Implement BFS	100%
2	Implement DFS	100%
3	Implement UCS	100%
4	Implement IDS	100%
5	Implement GBFS	100%
6	Implement A*	100%
7	Implement Hill-climbing	100%
8	Generate at least 5 test cases for all algorithm with different attributes. Describe them in the experiment section of your report.	100%
9	Report your algorithm, experiment with some reflection or comments	100%

Table 1: Completeness of tasks

3 Introduction

In artificial intelligence (AI), finding the best answers often comes with challenging problem environments. AI systems use search algorithms as their compass to navigate these spaces methodically and find states that are considered the goal state. Search strategies can divided into three categories: informed search, uninformed search, and local search.

This report discusses the ideas of multiple search strategies and how to implement them. Uninformed search, also referred to as blind search, operates without any knowledge. These algorithms systematically explore the state space and evaluate all available options. Informed search, in contrast, leverages domain knowledge in the form of heuristics. A heuristic is an approximate function that estimates the cost or distance to reach the goal state from a given state. Local search is used on problems that can be understood as finding a solution that maximizes a criterion among some candidate solutions. Then this report will also give some test cases and performance metrics for these algorithms.

The following sections are organized as follows: Section 4 with ideas and descriptions of search strategies. Section 5 describes the system workflow and helper function. Section 6 gives results with specific test cases and comments on these results. Conclusions will be in Section 7.

4 Search Strategies Descripton

The search strategies implemented in this lab have their characteristics and use different ideas to implement, which can categorized as follows.

- **Uniformed Search:**
  - Breath-first search (BFS)
  - Depth-first search (DFS)
  - Uniform-cost search (UCS)
  - Iterative deepening search (IDS)
- **Informed Search:**
  - Greedy best-first search (GBFS)
  - Graph-search A* (A*)
- **Local Search:**
  - Hill-climbing search (HC) variant

In this section, I will introduce each search strategies begin with a uniform search and each search strategy will based on the following outline:

1. Brief description of the search strategy and how it works with an illustrative image
2. Detail explanation about implementing search strategy in Python
3. Determine strategy performance in time and space complexity along with its properties.

4.1 Breath-first search (BFS)

1. **General description:**

   Breadth-first search (BFS) is a graph traversal algorithm that explores all the neighbors of a node at a level before moving to the next level. This systematic exploration ensures that all reachable nodes are visited in the order of their distance from the starting node.

   Here are some key points in this search strategy:

   - **Level-by-Level Exploration:** BFS prioritizes visiting all nodes at the current level before moving to their unvisited neighbors at the next level.
   - **Queue based:** BFS uses a queue data structure to maintain the order of nodes to be explored. Nodes are added to the front of the queue and processed in the order they were added (First-In-First-Out).

   Let us understand the working of the algorithm with the help of the following example made by AIMA book [1, Fig. 3.8, p. 76].

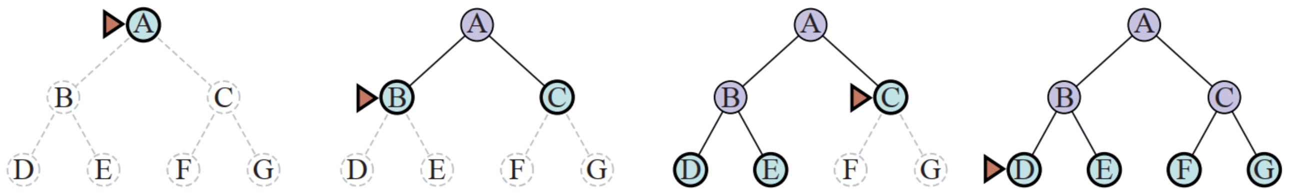


Figure 1: Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

2. **Implementation description:**

   (a) **Input arguments:**

   - arr: This is the adjacency matrix representing the graph. An element arr[i][j] is non-zero if there is an edge from node i to node j.
   - source: The starting node for the BFS.
   - destination: The target node for which we need to find the path.

   (b) **Initialization:**

   - path: An empty list that will store the path from the source to the destination if one is found.
   - visited: A dictionary where the key is a visited node and the value is its predecessor node. It starts with the source node having no predecessor (None).

- queue: A list used to implement the queue for BFS. It initializes with the source node.
- num_nodes: The number of nodes in the graph, derived from the size of the adjacency matrix.

(c) **BFS exploration:**

BFS iterates through the following steps as long as the queue is not empty:

1: Dequeue a node from the queue and assigns it to variable named current_node with "current_node = queue.pop(0)".

2: Check if the current_node is the destination node with if "current_node == destination:". If true, the loop breaks, meaning the path has been found. Else continue to step 3.

3: Iterates through all potential neighbors of the current_node with for neighbor in range(num_nodes):.

4: Checks if there is an edge between current_node and neighbor and if the neighbor has not been visited yet using if "arr[current_node][neighbor] != 0 and neighbor not in visited:",

4.1: If true, marks the neighbor as visited and records the current node as its predecessor with "visited[neighbor] = current_node",

4.2: Use "queue.append(neighbor)" to enqueue neighbor node into the queue for the next iteration.

(d) **Path construction:**

1: Checks if the destination node has been visited with "if destination in visited:". If true, a path exists.

2: Begin from the destination node, then use "while current_node is not None:" to continue backtracking from the destination to the source using the visited dictionary.

2.1: Use "path.append(current_node)" to appends the current node to the path.

2.1: Then moves to the predecessor of the current node with "current_node = visited[current_node]"

3: Finally use "path.reverse()" to reverse the path to get it in the correct order from source to destination.

3. **Strategy evaluation**

   (a) **Time complexity**

   The time complexity of the BFS algorithm in this implementation can be analyzed as follows: [2]

   - Initializing the visited dictionary and queue takes $\theta(1)$ time.
   - The while loop runs until the queue is empty. In the worst case, all nodes and edges are explored.
   - For each node, the algorithm checks all its neighbors, leading to $\theta(V)$ node explorations and $\theta(V^2)$ adjacency matrix checks (since the adjacency matrix size is (VxV)

   Thus, the overall time complexity is $\theta(V^2)$ where V is the number of vertices (nodes).

   (b) **Space Complexity**

   The space complexity of the BFS algorithm includes:

   - The visited dictionary, which stores each node once: $\theta(V)$
   - The queue, which in the worst case may contain all nodes: $\theta(V)$
   - The path list, which in the worst case contains all nodes: $\theta(V)$

   Thus, the overall space complexity is $\theta(V)$

   (c) **Properties** [1, Fig. 3.15, p. 84]

   - Completeness: BFS is complete, meaning it will always find a path if one exists.
   - Optimality: BFS is not optimal in general, meaning it won't find the shortest path.

4.2 Depth-first search (DFS)

1. **General description:**

   Depth-first search (DFS) is a graph traversal and search algorithm that explores as far as possible along each branch before backtracking and exploring other branches. Unlike BFS, which explores all neighbors at a level first, DFS explores a single branch until it reaches a dead end and then goes back to explore other branches.

   Here are some key points in this search strategy:

   - **Depth-first Exploration:** DFS prioritizes exploring a single branch until it reaches a dead end (no unvisited neighbors) and then backtracks to explore other branches. This ”depth-first” exploration allows it to find solutions quickly in some cases, especially when the goal is likely located deep within a branch.

   - **Stack-based:** DFS uses a stack data structure to maintain the order of nodes to be explored. Nodes are added to the front of the stack and processed in the reverse order they were added (Last-In-First-Out)

   Let us understand the working of the algorithm with the help of the following example made by AIMA book [1, Fig. 3.11, p. 79].

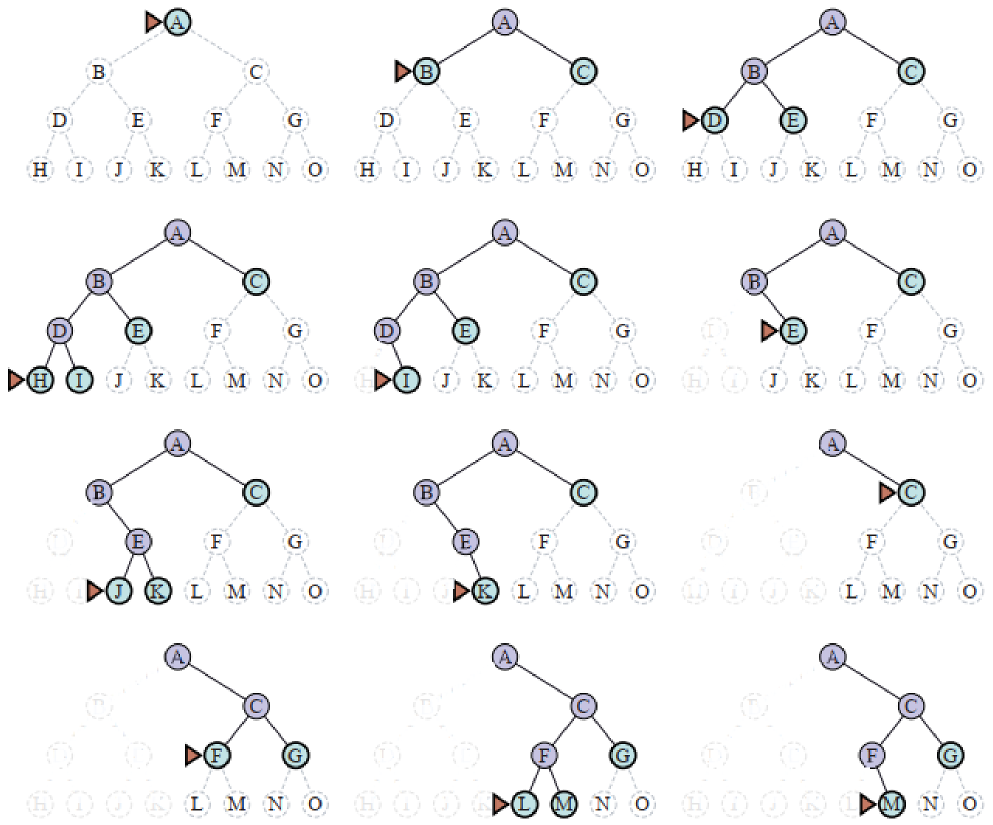


Figure 2: Depth-first search on a binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

2. **Implementation description:**

   (a) **Input arguments:**

   - arr: This is the adjacency matrix representing the graph. An element arr[i][j] is non-zero if there is an edge from node i to node j.
   - source: The starting node for the DFS.
   - destination: The target node for which we need to find the path.

   (b) **Initialization:**

   - path: An empty list that will store the path from the source to the destination if one is found.
   - visited: A dictionary where the key is a visited node and the value is its predecessor node. It starts with the source node having no predecessor (None).
   - stack: A list used to implement the stack for DFS. It starts with the source node.
   - num_nodes: The number of nodes in the graph, derived from the size of the adjacency matrix.

   (c) **DFS exploration:**

   DFS iterates through the following steps as long as the stack is not empty:

   1: Pop a node from the stack and assigns it to variable named current_node with "current_node = stack.pop()."

   2: Check if the current_node is the destination node with if "current_node == destination:". If true, the loop breaks, meaning the path has been found. Else continue to step 3.

   3: Iterates through all potential neighbors of the current_node with for neighbor in range(num_nodes):.

   4: Checks if there is an edge between current_node and neighbor and if the neighbor has not been visited yet using if "arr[current_node][neighbor] != 0 and neighbor not in visited:"

   4.1: If true, marks the neighbor as visited and records the current node as its predecessor with "visited[neighbor] = current_node"

   4.2: Use "stack.append(neighbor)" to push neighbor node into the stack for the next iteration.

(d) **Path construction:**

1: Checks if the destination node has been visited with "if destination in visited:". If true, a path exists.

2: Begin from the destination node, then use "while current_node is not None:" to continue backtracking from the destination to the source using the visited dictionary.

2.1: Use "path.append(current_node)" to appends the current node to the path.

2.1: Then moves to the predecessor of the current node with "current_node = visited[current_node]"

3: Finally use "path.reverse()" to reverse the path to get it in the correct order from source to destination.

3. **Strategy evaluation**

(a) **Time complexity**

The time complexity of the DFS algorithm in this implementation can be analyzed as follows: [3]

- Initializing the visited dictionary and stack takes $\theta(1)$ time.
- The while loop runs until the stack is empty. In the worst case, all nodes and edges are explored.
- For each node, the algorithm checks all its neighbors, leading to $\theta(V)$ node explorations and $\theta(V^2)$ adjacency matrix checks (since the adjacency matrix size is (VxV)

Thus, the overall time complexity is $\theta(V^2)$ where V is the number of vertices (nodes).

(b) **Space Complexity**

The space complexity of the DFS algorithm includes: [3]

- The visited dictionary, which stores each node once: $\theta(V)$
- The stack, which in the worst case may contain all nodes: $\theta(V)$
- The path list, which in the worst case contains all nodes: $\theta(V)$

Thus, the overall space complexity is $\theta(V)$

(c) **Properties** [1, Fig. 3.15, p. 84]

- Completeness: DFS is not guaranteed to be complete, as it may get stuck in cycles or explore paths that do not lead to the goal.
- Optimality: DFS is not optimal, as it does not guarantee the shortest path.

4.3 Uniform-cost search (UCS)

1. **General description:**

   Uniform-cost search (UCS) explores graphs by expanding nodes from the start to the goal based on edge costs. It finds the lowest-cost path, essential when step costs vary for optimal solutions.

   Here are some key points in this search strategy:

   - **Optimality:** UCS makes a goal test to a node when being selected for expansion, this test is added in case a better path is found to a node currently on the frontier.
   - **Priority_queue based:** UCS uses a priority queue to store nodes, where the priority determines the total cost incurred so far to reach that node. Nodes with lower total costs have higher priority in the queue to ensure that exploration paths with lower cumulative costs first.

   Let us understand the working of the algorithm with the help of the following example made by AIMA book [1, Fig. 3.10, p. 78].

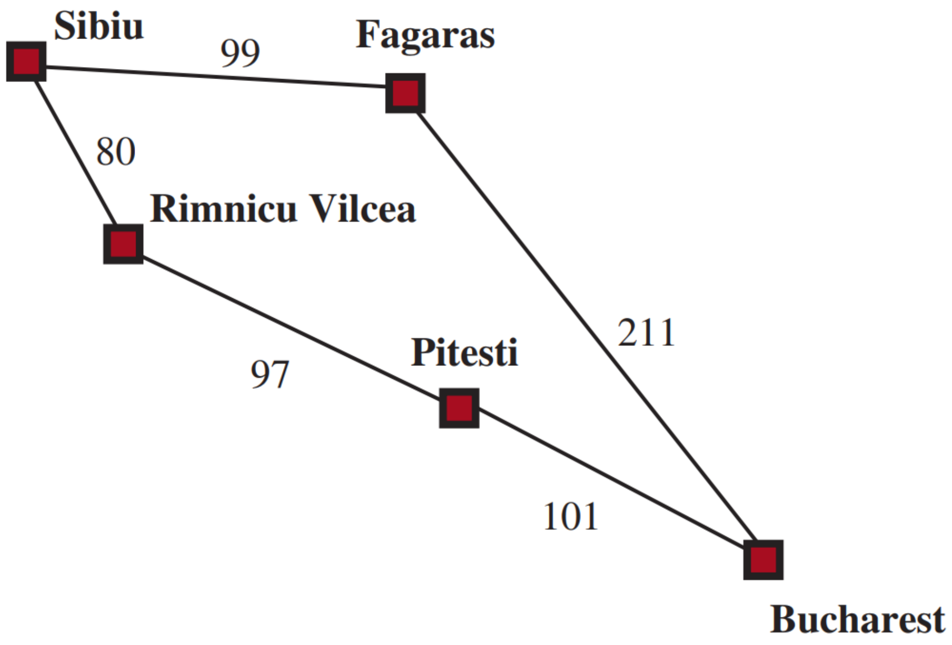


Figure 3: Part of the Romania state space, selected to illustrate uniform-cost search

Based on figure 3, the successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177. The least cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310. Bucharest is the goal, but the algorithm tests for goals only when it expands a node, not when it generates a node, so it has not yet detected that this is a path to the goal.

The algorithm continues on, choosing Pitesti for expansion next and adding a second path to Bucharest with cost 80 + 97 + 101=278. It has a lower cost, so it replaces the previous path in reached and is added to the frontier. It turns out this node now has the lowest cost, so it is considered next, found to be a goal, and returned. [1, p. 78]

2. **Implementation description:**

   (a) **Input arguments:**

   - arr: This is the adjacency matrix representing the graph. An element arr[i][j] is non-zero if there is an edge from node i to node j.
   - source: The starting node for the UCS.
   - destination: The target node for which we need to find the path.

   (b) **Initialization:**

   - path: An empty list that will store the path from the source to the destination if one is found.
   - visited: A dictionary where the key is a visited node and the value is its predecessor node. It starts with the source node having no predecessor (None).
   - costs: A dictionary to store the cost to reach each node from the source. It starts with the source node having a cost of 0.
   - num_nodes: The number of nodes in the graph, derived from the size of the adjacency matrix.
   - priority_queue: A priority queue initialized with the source node and its cost (0). The priority queue ensures that the node with the lowest cost is explored first.

   (c) **UCS exploration:**

   UCS iterates through the following steps as long as the priority_queue is not empty:

   1: Dequeue the node with the lowest cost from the priority queue with "current_cost, current = heappop(priority_queue)".

   2: Check if the current_node is the destination node with if "current_node == destination:". If true, the loop breaks, meaning the path has been found. Else continue to step 3.

   3: Iterates through all potential neighbors of the current_node with "for neighbor in range(num_nodes):".

   4: Checks if there is an edge between current_node and neighbor using if "arr[current_node] [neighbor] != 0". If true, then calculates the total cost to reach the neighbor with "total_cost = current_cost + arr[current][neighbor]".

5: Use "if neighbor not in costs or total_cost ¡ costs[neighbor]:" to check if the neighbor has not been visited or if the new path to the neighbor has a lower cost.

5.1: If true, updates the cost to reach the neighbor with "costs[neighbor] = total_cost".

5.2: Marks the neighbor as visited and records the current node as its predecessor with "visited[neighbor] = current_node"

5.3: Use "heappush(priority_queue, (total_cost, neighbor))" to push the neighbor onto the priority queue with its total cost.

(d) **Path construction:**

1: Checks if the destination node has been visited with "if destination in visited:". If true, a path exists.

2: Begin from the destination node, then use "while current_node is not None:" to continue backtracking from the destination to the source using the visited dictionary.

2.1: Use "path.append(current_node)" to appends the current node to the path.

2.1: Then moves to the predecessor of the current node with "current_node = visited[current_node]"

3: Finally use "path.reverse()" to reverse the path to get it in the correct order from source to destination.

3. **Strategy evaluation**

(a) **Time complexity and Space complexity**

The time complexity and space complexity of the UCS algorithm in this implementation is $\theta(b^{(1+C)/\varepsilon})$ [4]

(b) **Properties** [1, Fig. 3.15, p. 84]

- Completeness: UCS is complete, as it explores all nodes reachable from the source, ensuring that the destination is found if a path exists.
- Optimality: UCS is optimal, as it guarantees finding the lowest-cost path to the destination.

4.4 Iterative deepening search (IDS)

1. **General description:**

   - **Depth-limited search (DLS):**
     - DLS is a variant of DFS with a predetermined depth limit.
     - It explores the nodes of the graph to a specified depth and stops further exploration beyond this limit.
     - DLS can be used within IDS to perform searches up to the current depth limit.
   - **Iterative deepening search (IDS):**
     - IDS combines the benefits of Depth-First Search (DFS) and Breadth-First Search (BFS).
     - It performs a series of depth-limited searches, each with an increasing depth limit.
     - IDS is complete (will find a solution if one exists) and optimal (finds the shallowest goal node).
     - The space complexity of IDS is linear in the depth of the search tree, making it memory-efficient.

   Let us understand the working of the algorithm with the help of the following example made by AIMA book [1, Fig. 3.13, p. 82].

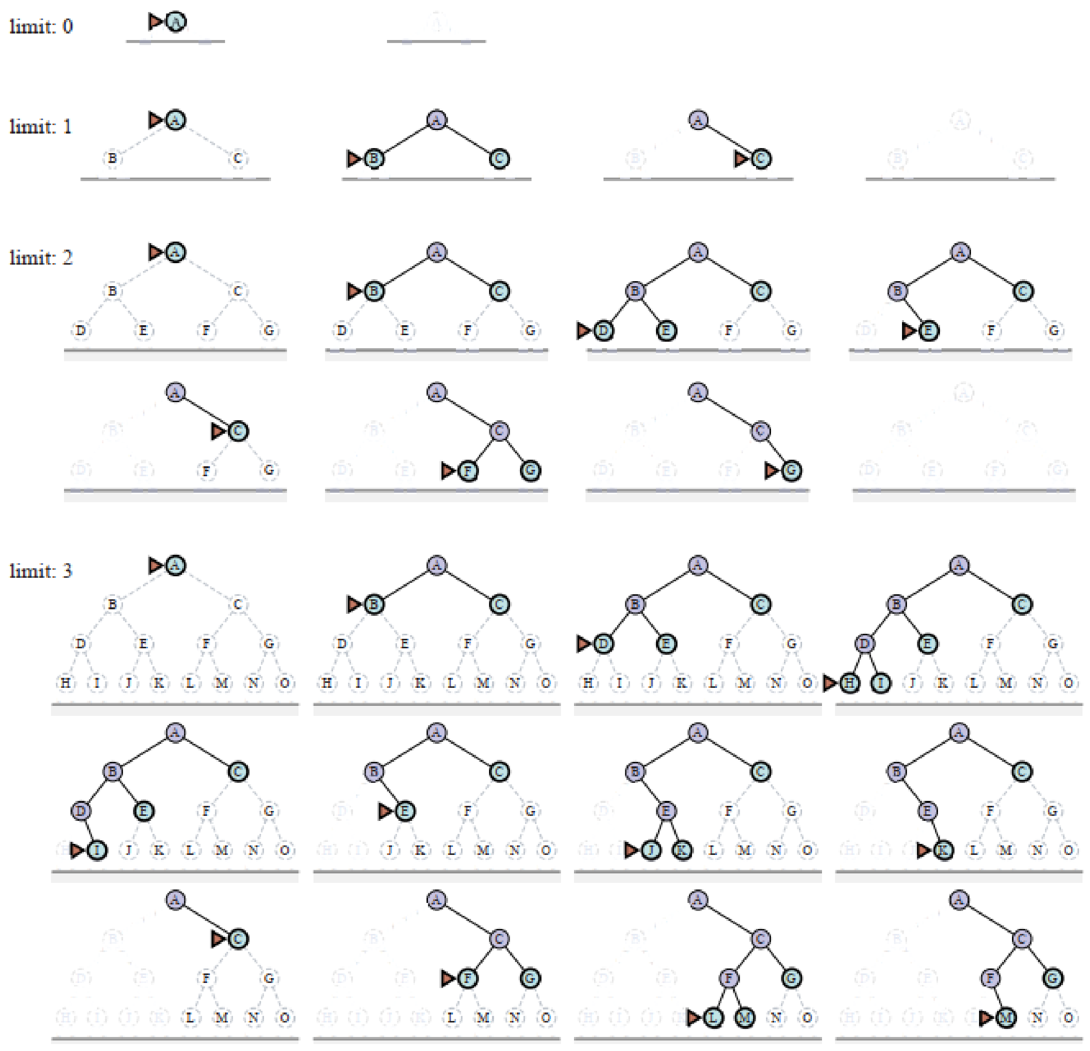


Figure 4: Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

2. **Implementation description for DLS:**

   (a) **Input arguments:**

      - arr: This is the adjacency matrix representing the graph. An element arr[i][j] is non-zero if there is an edge from node i to node j.
      - source: The starting node for the DLS.
      - destination: The target node for which we need to find the path.
      - depth_limit: The maximum depth to explore during the search.

(b) **Initialization:**

- path: An empty list that will store the path from the source to the destination if one is found.
- visited: A dictionary where the key is a visited node and the value is its predecessor node. It starts with the source node having no predecessor (None).
- stack: A stack initialized with a tuple containing the source node and its depth (0). This stack will be used for the iterative depth-limited search.

(c) **DLS exploration:**

DLS iterates through the following steps as long as the stack is not empty:

1: Pops the node and its depth from the stack with ”current_node, current_depth = stack.pop()”.

2: Check if the current depth is less than the depth limit. If true, continues check if the current node is the destination with ”If current_depth == destination:”. If true, then begin from the destination node use ”while current node is not None:” continue backtracking from the destination to the source using the visited dictionary.

  2.1: Use ”path.append(current node)” to appends the current node to the path.

  2.2: Then moves to the predecessor of the current node with ”current node = visited[current node]”

  2.3: Finally use ”path.reverse()” to reverse the path to get it in the correct order from source to destination.

3: Iterates through all potential neighbors of the current_node with for neighbor in range(num_nodes):.

4: Checks if there is an edge between current_node and neighbor and if the neighbor has not been visited yet using if ”arr[current_node][neighbor] != 0 and neighbor not in visited:”

  4.1: If true, marks the neighbor as visited and records the current node as its predecessor with ”visited[neighbor] = current_node”

  4.2: Use ”stack.append((neighbor, current_depth + 1))” to push neighbor node into the stack with an incremented depth.

3. **Implementation description for IDS:**

   (a) **Input arguments:**

   - arr: This is the adjacency matrix representing the graph. An element arr[i][j] is non-zero if there is an edge from node i to node j.
   - source: The starting node for the IDS.
   - destination: The target node for which we need to find the path.

   (b) **Initialization:**

   - path: An empty list that will store the path from the source to the destination if one is found.
   - visited: A dictionary where the key is a visited node and the value is its predecessor node. It starts with the source node having no predecessor (None).
   - depth_limit: Initially set to 0, will increment in each iteration of IDS.

   (c) **IDS exploration:**

   IDS iterates through the following steps as long as the stack is not empty:

   1: Performs an infinite loop, calling dls with increasing depth limits.

   2: If a path is found (path is not empty), it returns the visited dictionary and the path.

4. **Strategy evaluation**

   (a) **Time complexity** As IDS called DLS repeatedly until the result is found, the time complexity of DLS and IDS algorithm in this implementation can be analyzed as follows:

   - Initializing the visited dictionary and stack takes $\theta(1)$ time.
   - The while loop runs until the stack is empty. In the worst case, all nodes and edges are explored.
   - For each node, the algorithm checks all its neighbors, leading to $\theta(V)$ node explorations and $\theta(V^2)$ adjacency matrix checks (since the adjacency matrix size is (VxV)

   Thus, the overall time complexity is $\theta(V^2)$ where V is the number of vertices (nodes).

(b) **Space Complexity**

The space complexity of the IDS and DLS algorithm includes:

- The visited dictionary, which stores each node once: $\theta(V)$
- The stack, which in the worst case may contain all nodes: $\theta(V)$
- The path list, which in the worst case contains all nodes: $\theta(V)$

Thus, the overall space complexity is $\theta(V)$

(c) **Properties** [5]

- Completeness: IDS is not guaranteed to be complete, when the branching factor b is infinite and DLS will only complete if predetermined depth limit l < depth.
- Optimality: IDS is not optimal, as it does not guarantee the shortest path. For DLS algorithm it is not optimal when predetermined depth limit l > depth.

4.5 Greedy best-first search (GBFS)

1. **General description:**

   Greedy Best-First Search (GBFS) is a graph search algorithm that prioritizes exploring nodes that appear closest to the goal based on a heuristic function. It starts at a specified vertex and iteratively visits the unvisited neighbor with the most promising heuristic value. This exploration continues until the destination node is found or all reachable nodes have been explored.

   Here are some key points in this search strategy:

   - **Heuristic Function:** This function estimates the cost of reaching the goal from a given node. A lower heuristic value indicates a node is likely closer to the goal.
   - **Prioritization:** GBFS prioritizes exploring nodes with lower heuristic values, aiming to find the goal efficiently.

   Let us understand the working of the algorithm with the help of the following example made by AIMA book [1, Fig. 3.17, p. 86].

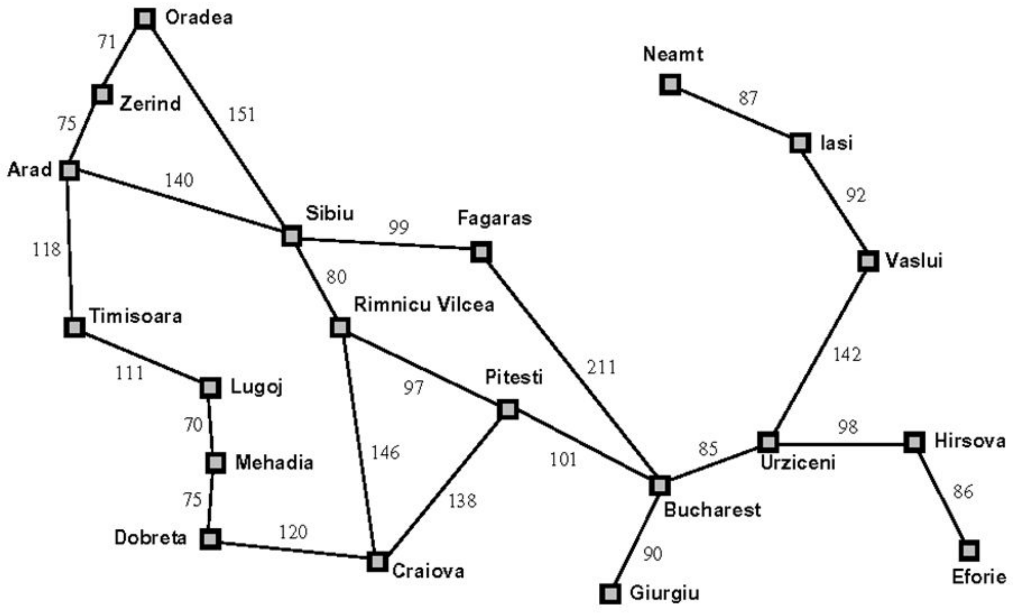


Figure 5: Map of Romania

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 6: Values of straight-line distances heuristic to Bucharest.

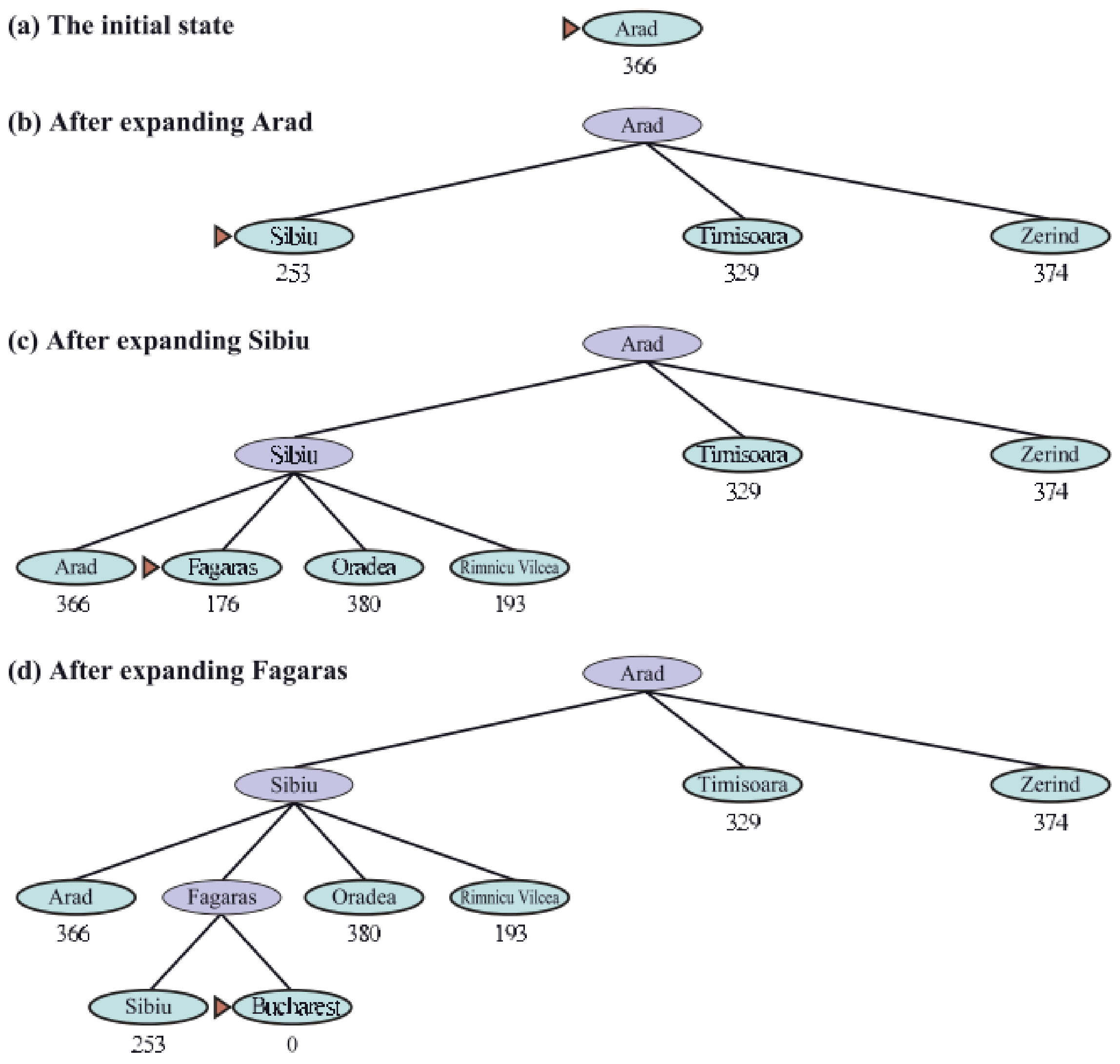


Figure 7: Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic. Nodes are labeled with their h-values.

2. **Implementation description:**

   (a) **Input arguments:**

   - arr: This is the adjacency matrix representing the graph. An element arr[i][j] is non-zero if there is an edge from node i to node j.
   - source: The starting node for the GBFS.
   - destination: The target node for which we need to find the path.
   - heuristic: An array representing heuristic values from each node to the goal node.

   (b) **Initialization:**

   - path: An empty list to store the found path.
   - visited: A dictionary to track visited nodes and their predecessors, initialized with the source node having no predecessor (None).
   - num_nodes: The number of nodes in the graph, derived from the size of the adjacency matrix.
   - priority_queue: A priority queue initialized with the source node and its heuristic value. The priority queue ensures that the node with the lowest heuristic value is explored first.

   (c) **GBFS exploration:**

   GBFS iterates through the following steps as long as the priority_queue is not empty:

   1: Dequeue the node with the lowest cost from the priority queue with "current_heuristic, current = heappop(priority_queue)".

   2: Check if the current_node is the destination node with if "current_node == destination:". If true, the loop breaks, meaning the path has been found. Else continue to step 3.

   3: Iterates through all potential neighbors of the current_node with "for neighbor in range(num_nodes):".

   4: Checks if there is an edge between current_node and neighbor and if the neighbor has not been visited yet using if "arr[current_node][neighbor] != 0 and neighbor not in visited:"

   4.1: If true, marks the neighbor as visited and records the current node as its predecessor with "visited[neighbor] = current_node"

   4.2: Use "heappush(priority_queue, (heuristic[neighbor], neighbor))" to push the neighbor onto the priority queue with its heuristic value.

(d) **Path construction:**

1: Checks if the destination node has been visited with "if destination in visited:". If true, a path exists.

2: Begin from the destination node, then use "while current_node is not None:" to continue backtracking from the destination to the source using the visited dictionary.

2.1: Use "path.append(current_node)" to appends the current node to the path.

2.1: Then moves to the predecessor of the current node with "current_node = visited[current_node]"

3: Finally use "path.reverse()" to reverse the path to get it in the correct order from source to destination.

3. **Strategy evaluation**

(a) **Time complexity**

The time complexity of the GBFS algorithm in this implementation can be analyzed as follows: [6]

- Initializing the visited dictionary and priority_queue takes $\theta(1)$ time.
- The while loop runs until the priority_queue is empty. In the worst case, all nodes and edges are explored.
- For each node, the algorithm checks all its neighbors, leading to $\theta(V)$ node explorations and $\theta(V^2)$ adjacency matrix checks (since the adjacency matrix size is (VxV).
- Using a priority queue (heap) as mentioned before in UCS strategy evaluation, takes $\theta(\log V)$ time for each insertion and extraction operation.

Thus, the overall time complexity is $\theta((V^2)\log V)$ where V is the number of vertices (nodes).

(b) **Space Complexity**

The space complexity of the GBFS algorithm includes:

- The visited dictionary, which stores each node once: $\theta(V)$
- The stack, which in the worst case may contain all nodes: $\theta(V)$
- The path list, which in the worst case contains all nodes: $\theta(V)$

Thus, the overall space complexity is $\theta(V)$

(c) **Properties** [7, p. 14]

- Completeness: GBFS is not complete, as it may get stuck in loops or dead ends if the heuristic function is not well-designed.
- Optimality: GBFS is not optimal, as it only considers the heuristic function and does not guarantee the shortest path.

4.6 Grapth search A* (A*)

1. **General description:**

A* search strategy is a graph search algorithm that combines informed search with efficient exploration. It leverages a heuristic function to estimate the remaining cost to reach the goal from a particular node. The heuristic function, with the actual cost incurred so far, guides the search strategy to the most promising nodes, aiming for the shortest path to the destination.

Here are some key points in this search strategy:

- **Heuristic Function:** A* relies on a heuristic function h(n) that estimates the cost of reaching the goal from a given node n. A lower heuristic value indicates the node is likely closer to the goal.
- **f-score:** A* calculates an f-score f(n) for each node, which combines the actual cost (g(n)) of reaching that node from the starting point and the heuristic estimate h(n). with f(n) = g(n) + h(n)

Let us understand the working of the algorithm with the help of the following example made by AIMA book [1, Fig. 3.18, p. 87].

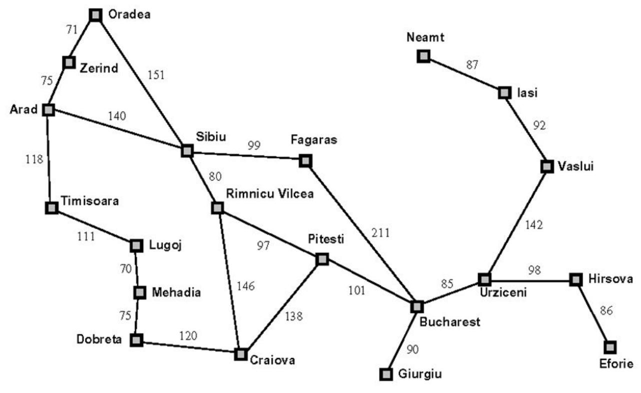


Figure 8: Map of Romania

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 9: Values of straight-line distances heuristic to Bucharest.

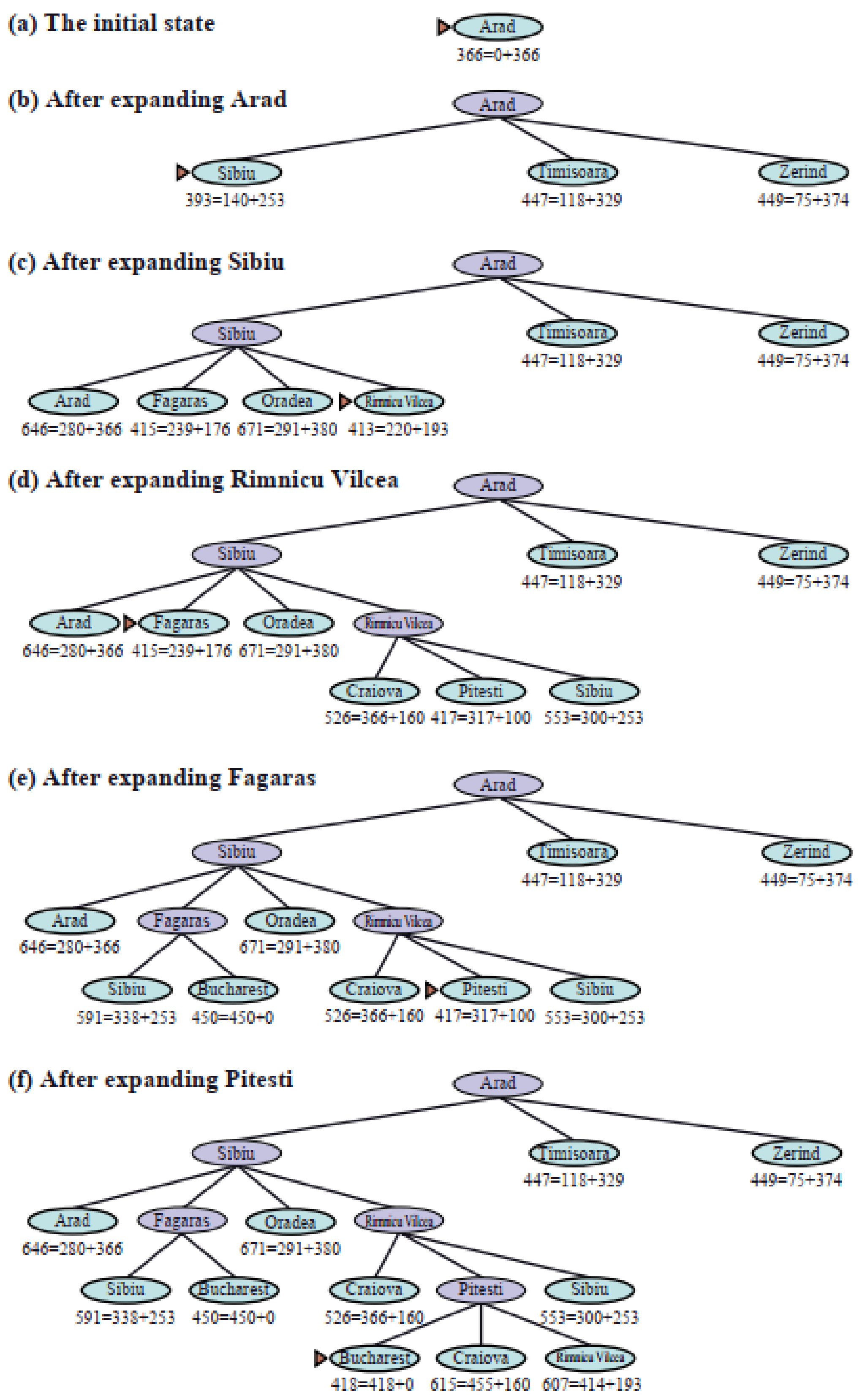


Figure 10: Stages in A* search for Bucharest. Nodes are labeled with f = g + h.

2. **Implementation description:**

   (a) **Input arguments:**

   - arr: This is the adjacency matrix representing the graph. An element arr[i][j] is non-zero if there is an edge from node i to node j.
   - source: The starting node for the A*.
   - destination: The target node for which we need to find the path.
   - heuristic: An array representing heuristic values from each node to the goal node.

   (b) **Initialization:**

   - path: An empty list to store the found path.
   - visited: A dictionary to track visited nodes and their predecessors, initialized with the source node having no predecessor (None).
   - costs: A dictionary to store the cost to reach each node from the source, including the heuristic value. It starts with the source node having its heuristic value.
   - num_nodes: The number of nodes in the graph, derived from the size of the adjacency matrix.
   - priority_queue: A priority queue initialized with the source node and its heuristic value. The priority queue ensures that the node with the lowest estimated total cost is explored first.

   (c) **A* exploration:**

   A* iterates through the following steps as long as the priority_queue is not empty:

   1: Dequeue the node with the lowest cost from the priority queue with "current_cost, current = heappop(priority_queue)".

   2: Check if the current_node is the destination node with if "current_node == destination:". If true, the loop breaks, meaning the path has been found. Else continue to step 3.

   3: Iterates through all potential neighbors of the current_node with "for neighbor in range(num_nodes):".

   4: Check if there is an edge to a neighbor "arr[current][neighbor] != 0". If true, then calculates the total cost to reach the neighbor:

   4.1: total_cost = current_cost - heuristic[current] + arr[current][neighbor] + heuristic[neighbor]. This total cost includes the cost to reach the current node, the cost of the edge to the neighbor, and the heuristic value for the neighbor.

5: Check if the neighbor has not been visited or the new path to the neighbor has a lower total cost.

   5.1: If true, updates the cost to reach the neighbor with ”costs[neighbor] = total_cost”.

   5.2: Marks the neighbor as visited and records the current node as its predecessor with ”visited[neighbor] = current”

   5.3: Use ”heappush(priority_queue, (total_cost, neighbor))” to push the neighbor onto the priority queue with its total cost.

(d) **Path construction:**

- After the A* exploration is complete, the code checks if the destination node was visited.
- Check if the destination node was visited, then a path exists from the source node to the destination node. If true, constructs the path by backtracking through the predecessor nodes stored in the visited dictionary. If not, return an empty path
- Then use reverse method for path to get the correct order since we are using backtracking and return the path.

3. **Strategy evaluation**

(a) **Time complexity**

The time complexity of the A* algorithm in this implementation can be analyzed as follows: [6]

- Initializing the visited dictionary and priority_queue takes $\theta(1)$ time.
- The while loop runs until the priority_queue is empty. In the worst case, all nodes and edges are explored.
- For each node, the algorithm checks all its neighbors, leading to $\theta(V)$ node explorations and $\theta(V^2)$ adjacency matrix checks (since the adjacency matrix size is (VxV).
- Using a priority queue (heap) as mentioned before in UCS strategy evaluation, takes $\theta(\log V)$ time for each insertion and extraction operation.

Thus, the overall time complexity is $\theta(V^2 \log V)$ where V is the number of vertices (nodes).

(b) **Space Complexity**

The space complexity of the A* algorithm includes:

- The visited dictionary, which stores each node once: $\theta(V)$
- The priority_queue, which in the worst case may contain all nodes: $\theta(V)$
- The path list, which in the worst case contains all nodes: $\theta(V)$
- The costs dictionary, which stores the cost for each node: $\theta(V)$

Thus, the overall space complexity is $\theta(V)$

(c) **Properties** [7, p. 23]

- Completeness: A* is complete, as it explores all nodes reachable from the source, ensuring that the destination is found if a path exists.
- Optimality: A* is optimal, as it guarantees finding the lowest-cost path to the destination when the heuristic function is admissible (never overestimates the true cost).

4.7 Hill-climbing search (HC)

1. **General description:**

Hill Climbing (HC) is a heuristic search algorithm used for mathematical optimization problems. It aims to find a path from a starting node to a destination node in a graph by iteratively selecting the neighboring node with the lowest heuristic value. The process continues until the destination node is reached or no better neighbors are found.

Here are some key points in this search strategy:

- **Heuristic Function:** This function estimates the cost of reaching the goal from a given node. A lower heuristic value indicates a node is likely closer to the goal.
- **Greedy Approach:** HC always moves to the neighbor with the lowest heuristic value, aiming to find the goal efficiently.
- **Local Maximum:** The algorithm may stop at a local maximum if no neighbor has a lower heuristic value than the current node.

2. **Implementation description:**

   (a) **Input arguments:**

   - arr: This is the adjacency matrix representing the graph. An element arr[i][j] is non-zero if there is an edge from node i to node j.
   - source: The starting node for the HC.
   - destination: The target node for which we need to find the path.
   - heuristic: An array representing heuristic values from each node to the goal node.

   (b) **Initialization:**

   - visited: A dictionary to track visited nodes and their predecessors, initialized with the source node having no predecessor (None).
   - path: An empty list to store the found path.
   - current: The current node being explored, initially set to the source node.

   (c) **HC exploration:** HC iterates through the following steps until the destination node is found or no better neighbors are available:

   1: Constructs a list of tuples containing each neighbor and its heuristic value if there is an edge from the current node to the neighbor with ”neighbors = [(neighbor, heuristic[neighbor]) for neighbor, connected in enumerate(arr[current]) if connected != 0]” by using enumerate(arr[current]) will return the index and the value corresponding to that index.

   2: Checks if there are no neighbors. If there are no neighbors, the loop breaks, meaning no path is found.

   3: Selects the neighbor with the smallest heuristic value using the min function then assign the value to next_node: ”next_node = min(neighbors, key=lambda x: x[1])[0]”.

   4: Use ”if heuristic[next_node] ¿= heuristic[current]:” to check if the selected neighbor’s heuristic value is not less than the current node’s heuristic value, the loop breaks to avoid getting stuck in a local minimum.

   5: Marks the neighbor as visited, sets the current node as its predecessor, and updates the current node to the selected neighbor.

(d) **Path construction:**

1: Checks if the destination node has been visited with "if destination in visited:". If true, a path exists.

2: Begin from the destination node, then use "while current_node is not None:" to continue backtracking from the destination to the source using the visited dictionary.

2.1: Use "path.append(current_node)" to appends the current node to the path.

2.1: Then moves to the predecessor of the current node with "current_node = visited[current_node]"

3: Finally use "path.reverse()" to reverse the path to get it in the correct order from source to destination.

3. **Strategy evaluation**

(a) **Time complexity**

The time complexity of the HC algorithm in this implementation is $\theta(\infty)$ [8]

(b) **Space Complexity**

The space complexity of the HC algorithm includes:

- The visited dictionary, which stores each node once: $\theta(V)$
- The path list, which in the worst case contains all nodes: $\theta(V)$

Thus, the overall space complexity is $\theta(V)$ [8]

(c) **Properties**

- Completeness: HC is not complete, as it may get stuck in a local minimum and fail to find the goal even if a path exists. [8]
- Optimality: HC is not optimal, as it does not guarantee finding the lowest-cost path to the destination. [8]

5 System description

In this section, I will give details about my system including helper function and the system flow from reading input file to writing to output file.

Here are some library that were used in this lab:

- Numpy: is used for creating and storing adjacency matrix and heuristic array
- tracemalloc: is used for monitoring the peak memory usage of each algorithm
- time: is used for capture the execution time for each algorithm
- heapq: is used to perform priority_queue operation such as enqueue and dequeue.

5.1 readInputData function

The readInputData function reads input data from a specified file and returns the starting node, goal node, adjacency matrix, and heuristic values. Here's a detailed breakdown of the code:

- **Input arguments:**
  - file_path: A string representing the file path of the input file.
- **File Reading and Initial Processing:**
  - The file at file_path is opened in read mode.
  - The first line is read to get the number of nodes in the graph, which is then converted to an integer and stored in numberOfNodes.
- **Reading Start and Goal Nodes:**
  - The second line is read, which contains the start and goal nodes.
  - The line is split into two parts using space as the delimiter.
  - startNode is assigned the integer value of the first part.
  - goalNode is assigned the integer value of the second part, with any trailing newline character removed.

- **Reading the Adjacency Matrix:**
  - First an adjacency matrix of size numberOfNodes x numberOfNodes is initialized with zeros using NumPy.
  - A loop iterates through the next numberOfNodes lines to read the adjacency matrix.
  - Each line is read into temp.
  - The trailing newline character is removed, and the line is converted to a NumPy array of floats using np.fromstring with space as the delimiter.
  - The resulting vector is added to the corresponding row in the matrix.
- **Reading Heuristic Values:**
  - The next line, which contains the heuristic values, is read into temp.
  - The line is converted to a NumPy array of floats using np.fromstring with space as the delimiter.

Finally, the function returns the startNode, goalNode, adjacency matrix, and heuristic values.

5.2 System workflow

1. The system prompts the user for the input file path, extracts the file name for output naming and read the input data using readInputData.
2. Prepares an output file on the same directory of input_file, named output_<input_file_name>.txt and clears any existing content.
3. Loops through each algorithm (from 0 to 6), measuring memory and execution time. Executes the corresponding search algorithm based on algoChoice and collect memory usage and execution time for each algorithm.
4. Constructs a string representing the path found. Writes the algorithm name, found path (or -1 if no path), execution time, and memory usage to the output file.
5. Finally, closes the output file.

5.3 System review

In my opinion, the implemented system has some advantages and disadvantages:

- **Advantages**
  - Flexible Input Handling: By reading the adjacency matrix and heuristics from a file, the system can easily handle different graphs and scenarios without modify the code.
  - Detailed Output: The output file provides a clear summary of each algorithm's performance and the paths found, make it easy for analysis.
  - Performance Measurement: The inclusion of time and memory tracking provides valuable insights into the efficiency of each algorithm. This is crucial for understanding the trade-offs between different search strategies.
- **Disadvantages**
  - Scalability Concerns: While the system works well for small to medium-sized graphs, the use of simple data structures (e.g., lists for queues and stacks) might not be efficient for very large graphs.
  - Hill-Climbing Limitations: The hill-climbing algorithm implemented is a simple variant that can easily get stuck in local optimal. More advanced variants (e.g., simulated annealing or random restart) could be explored for better performance.

6 Evaluation

In this section, I will give details about my test cases used in this lab and the performance metrics of the algorithms when running on these test cases, then make a detailed review and evaluation of these metrics. The remaining section will be formatted as follows: 6.1 input file format, 6.2 provides test cases and their illustration, 6.3 performance metrics of algorithms for each test case and evaluation for them, and 6.4 will summarize and conclude with search strategies performance.

6.1 Input file format

As mentioned before in search strategies, for this lab I will use adjacent matrix to illustrate the actual graph, which is stored in a text file and formatted as follow:

- The first line contains the number of nodes in the graph.
- The second line contains two integers representing the start and goal nodes.
- The subsequent lines contain the adjacency matrix of the graph.
- The last line contains the heuristic weights for each node (for algorithms that use heuristics).

6.2 Test cases

6.2.1 Test case 1

```
6
0 5
0 1 0 0 5 0
1 0 3 0 0 0
0 3 0 1 0 1
0 0 1 0 0 1
5 0 0 0 0 1
0 0 1 1 1 0
6 3 9 2 4 0
```

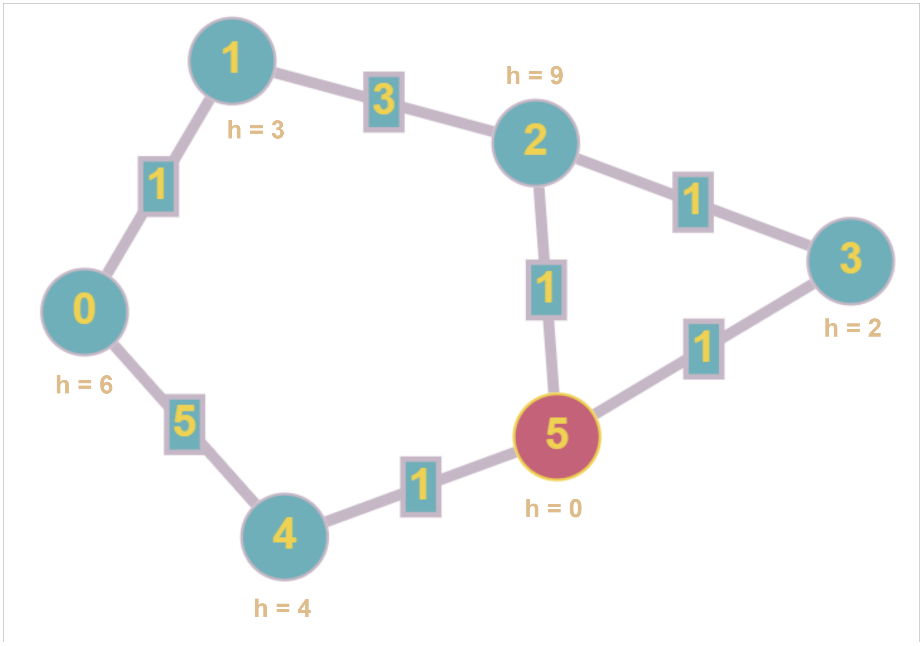


Figure 11: Test case 1 input file and its illustrative graph

6.2.2 Test case 2

```
7
0 6
0 3 2 0 0 0 0
3 0 0 5 0 10 0
2 0 0 1 4 0 0
0 5 1 0 1 2 0
0 0 4 1 0 0 3
0 10 0 2 0 0 4
0 0 0 0 3 4 0
7 9 5 4 3 2 0
```

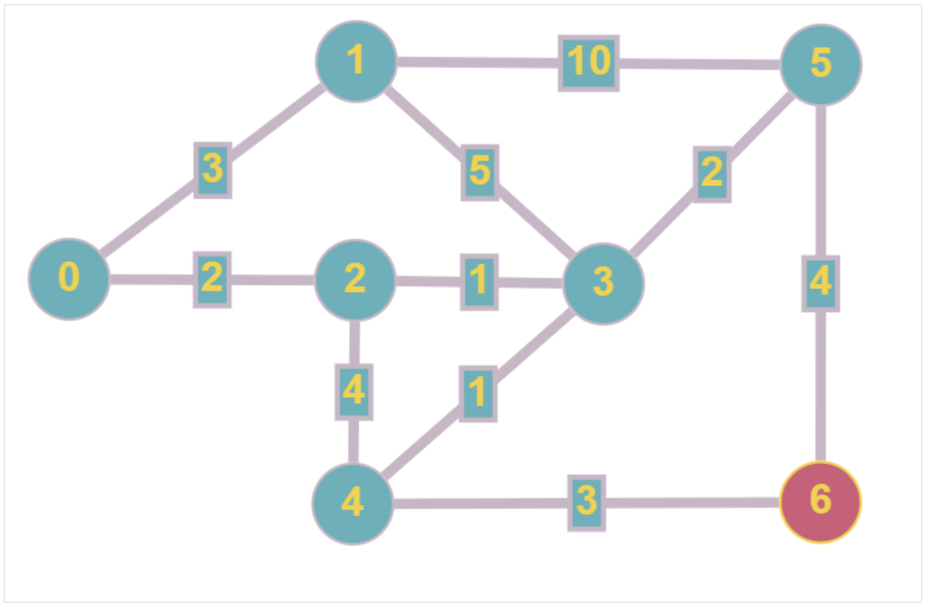


Figure 12: Test case 2 input file and its illustrative graph

6.2.3 Test case 3

```
8
0 7
0 4 5 0 0 7 0 0
4 0 2 5 6 0 0 11
5 2 0 3 0 0 0 0
0 5 3 0 1 0 0 0
0 6 0 1 0 8 0 6
7 0 0 0 8 0 5 0
0 0 0 0 0 5 0 3
0 11 0 0 6 0 3 0
8 5 1 4 3 7 5 0
```

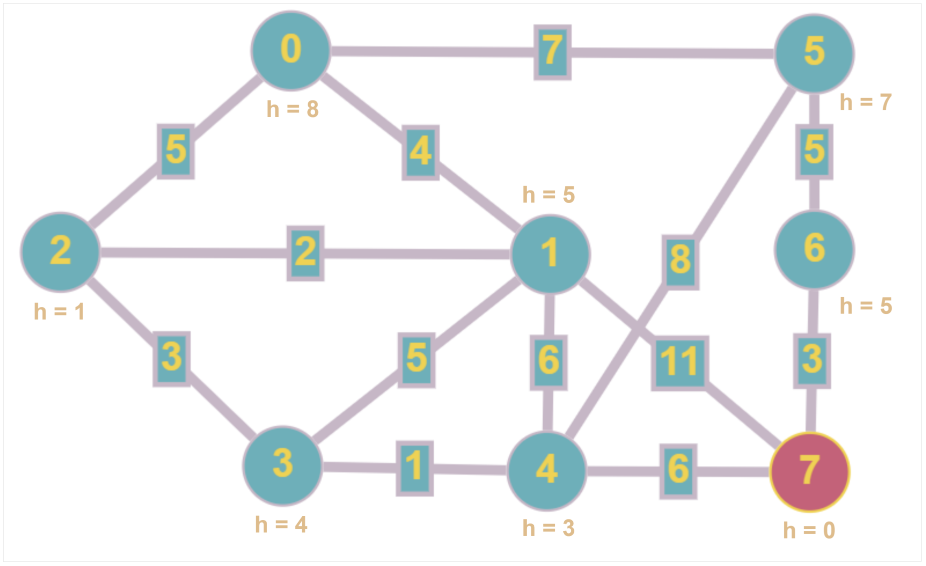


Figure 13: Test case 3 input file and its illustrative graph

6.2.4 Test case 4

```
9
0 4
0 0 2 0 0 0 7 8 8
0 0 0 2 0 0 0 4 4
2 0 0 9 0 0 0 0 0
0 2 9 0 0 1 4 0 9
0 0 0 0 0 8 0 0 0
0 0 0 1 8 0 0 0 0
7 0 0 4 0 0 0 0 6
8 4 0 0 0 0 0 0 0
8 4 0 9 0 0 6 0 0
8 4 3 3 0 2 9 1 7
```

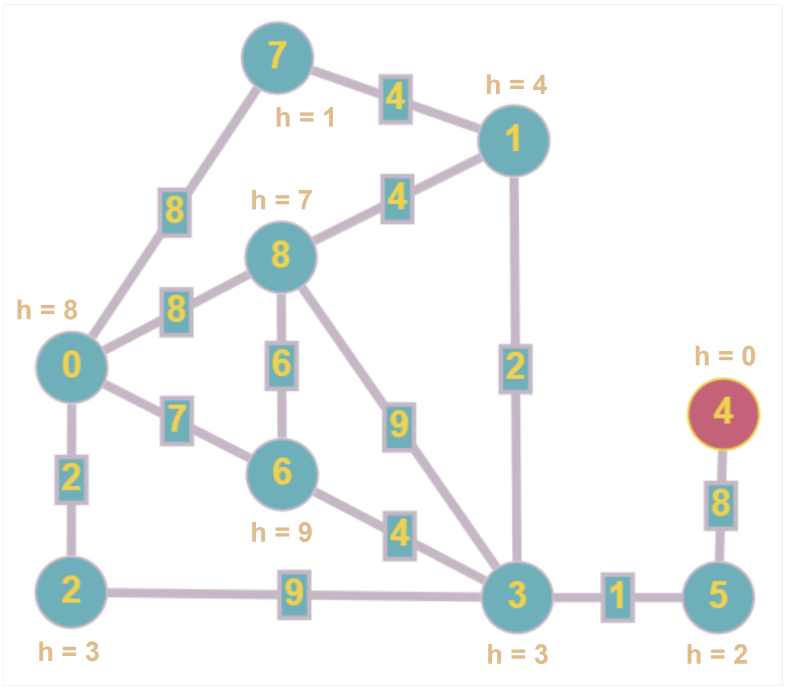


Figure 14: Test case 4 input file and its illustrative graph

6.2.5 Test case 5

```
10
3 8
0 3 7 0 0 2 0 0 0 0
3 0 4 0 0 0 0 0 0 1
7 4 0 5 0 0 0 0 0 0
0 0 5 0 8 0 0 0 0 0
0 0 0 8 0 1 4 0 0 0
2 0 0 0 1 0 3 0 0 6
0 0 0 0 4 3 0 5 0 0
0 0 0 0 0 0 5 0 7 4
0 0 0 0 0 0 0 7 0 3
0 1 0 0 0 6 0 4 3 0
10 6 3 4 5 1 9 7 0 8
```

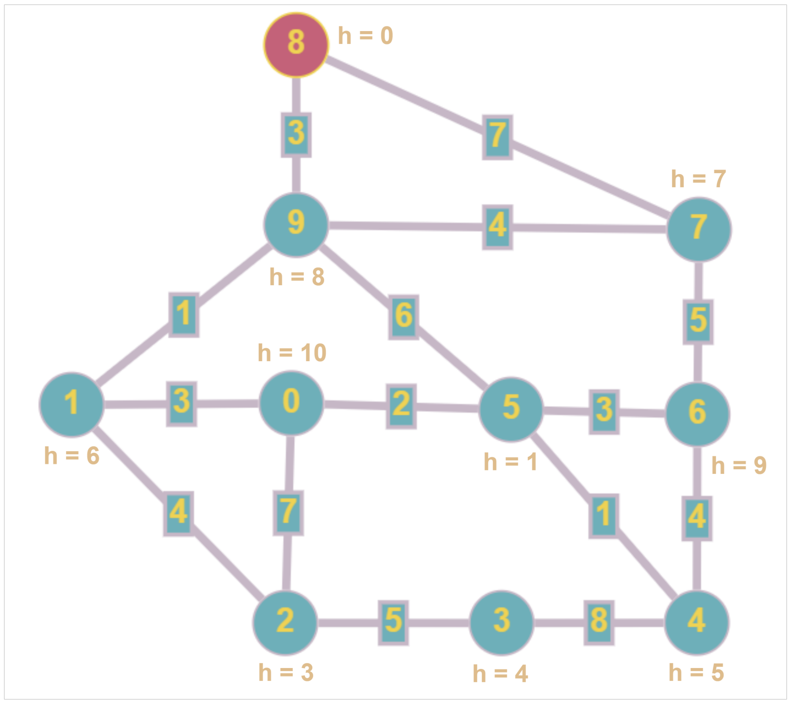
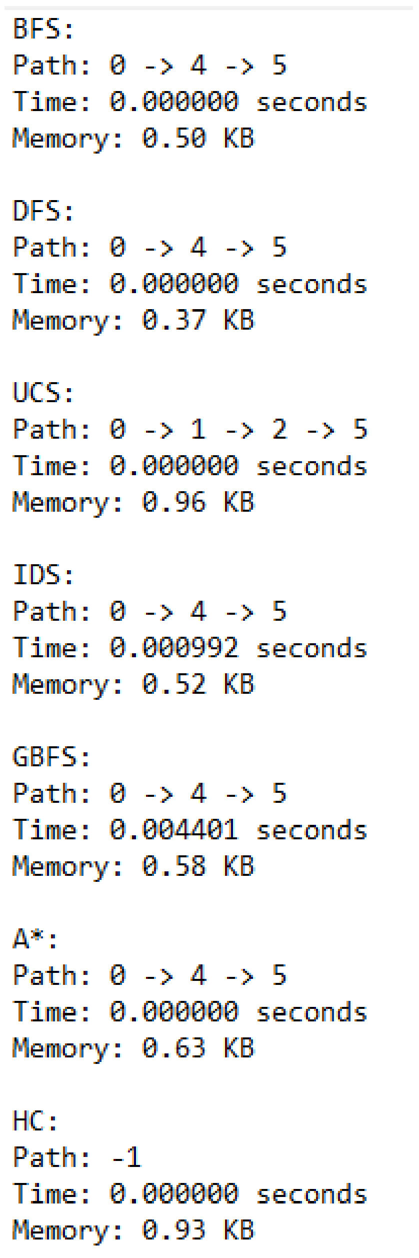


Figure 15: Test case 5 input file and its illustrative graph

6.3 Performance metrics

6.3.1 Test case 1



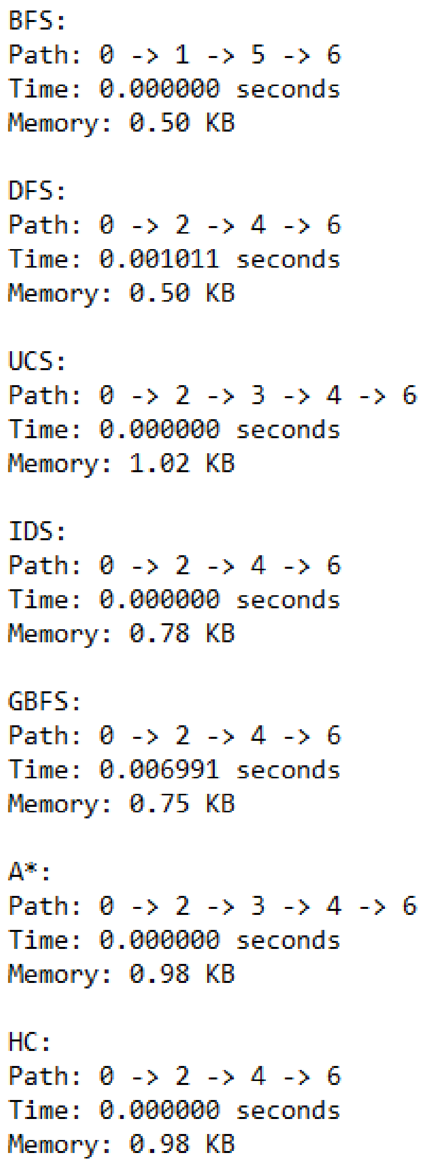
```
BFS:
Path: 0 -> 4 -> 5
Time: 0.000000 seconds
Memory: 0.50 KB

DFS:
Path: 0 -> 4 -> 5
Time: 0.000000 seconds
Memory: 0.37 KB

UCS:
Path: 0 -> 1 -> 2 -> 5
Time: 0.000000 seconds
Memory: 0.96 KB

IDS:
Path: 0 -> 4 -> 5
Time: 0.000992 seconds
Memory: 0.52 KB

GBFS:
Path: 0 -> 4 -> 5
Time: 0.004401 seconds
Memory: 0.58 KB

A*:
Path: 0 -> 4 -> 5
Time: 0.000000 seconds
Memory: 0.63 KB

HC:
Path: -1
Time: 0.000000 seconds
Memory: 0.93 KB
```

Figure 16: Result of test case 1

Based on the results on Figure 16, we can extract some information as follow:

1. Path Found:
   - BFS, DFS, IDS, GBFS, and A* all found the same path $0 \rightarrow 4 \rightarrow 5$.
   - UCS found a different path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 5$, indicating that it considers the cost of edges and found a less direct but potentially cheaper path.
   - Hill Climbing (HC) failed to find a path (Path: -1). Because when expanding node 1 it is at a local maximum (node 1 with a heuristic of 3) and cannot find any unvisited neighbors with a lower heuristic value. As a result, the destination node (5) is never reached.
2. Time Efficiency:
   - Most algorithms, except for IDS and GBFS, reported negligible computation time (0.000000 seconds).
   - IDS took slightly longer (0.000992 seconds), and GBFS took the longest (0.004401 seconds).
3. Memory Usage:
   - DFS used the least memory (0.37 KB).
   - BFS (0.50 KB), IDS (0.52 KB), GBFS (0.58 KB), and A* (0.63 KB) had moderate memory usage.
   - UCS used the most memory (0.96 KB), which is expected given that UCS keeps track of path costs.
   - Hill Climbing (HC) also used significant memory (0.93 KB), despite failing to find a path.

6.3.2 Test case 2



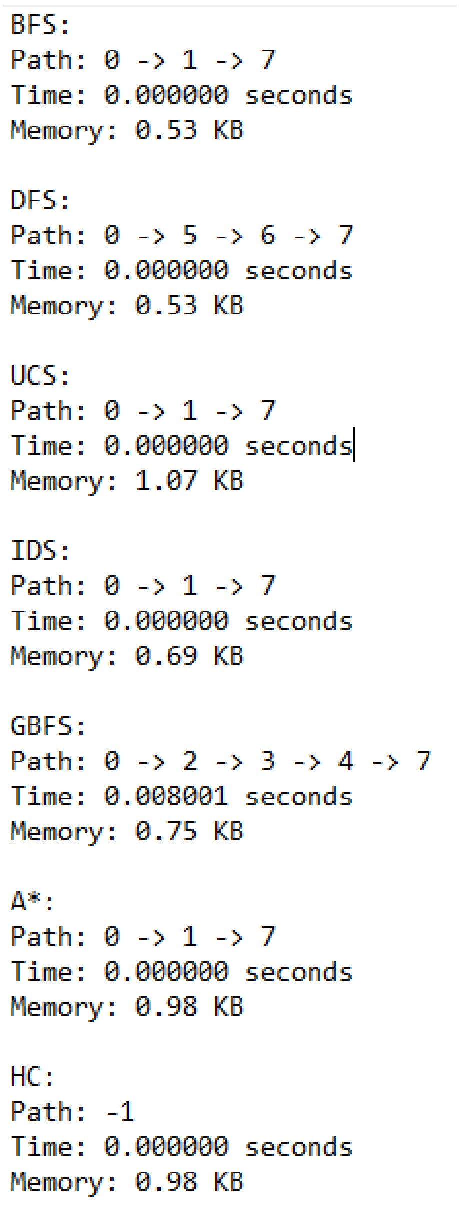
```
BFS:
Path: 0 -> 1 -> 5 -> 6
Time: 0.000000 seconds
Memory: 0.50 KB

DFS:
Path: 0 -> 2 -> 4 -> 6
Time: 0.001011 seconds
Memory: 0.50 KB

UCS:
Path: 0 -> 2 -> 3 -> 4 -> 6
Time: 0.000000 seconds
Memory: 1.02 KB

IDS:
Path: 0 -> 2 -> 4 -> 6
Time: 0.000000 seconds
Memory: 0.78 KB

GBFS:
Path: 0 -> 2 -> 4 -> 6
Time: 0.006991 seconds
Memory: 0.75 KB

A*:
Path: 0 -> 2 -> 3 -> 4 -> 6
Time: 0.000000 seconds
Memory: 0.98 KB

HC:
Path: 0 -> 2 -> 4 -> 6
Time: 0.000000 seconds
Memory: 0.98 KB
```

Figure 17: Result of test case 2

Based on the results on Figure 17, we can extract some information as follow:

1. Path Found:
   - BFS found a path: $0 \to 1 \to 5 \to 6$.
   - DFS, IDS, GBFS, and HC found the same path $0 \to 2 \to 4 \to 6$
   - UCS and A* found a path $0 \to 2 \to 3 \to 4 \to 6$ indicating that it considers the cost of edges and found a less direct but potentially cheaper path.
2. Time Efficiency:
   - Most algorithms reported negligible computation time (0.000000 seconds), except DFS (0.001011 seconds) and GBFS (0.006991 seconds), which took slightly longer.
   - The time differences, although small, indicate variations in algorithmic efficiency and implementation.
3. Memory Usage:
   - Memory usage varied among the algorithms, with DFS and BFS using the least (0.50 KB).
   - UCS used the most memory (1.02 KB), as it maintains path costs.
   - A* and HC also had higher memory usage (0.98 KB), reflecting their need to store additional heuristic information.
   - IDS and GBFS had moderate memory usage (0.78 KB and 0.75 KB respectively).

6.3.3 Test case 3



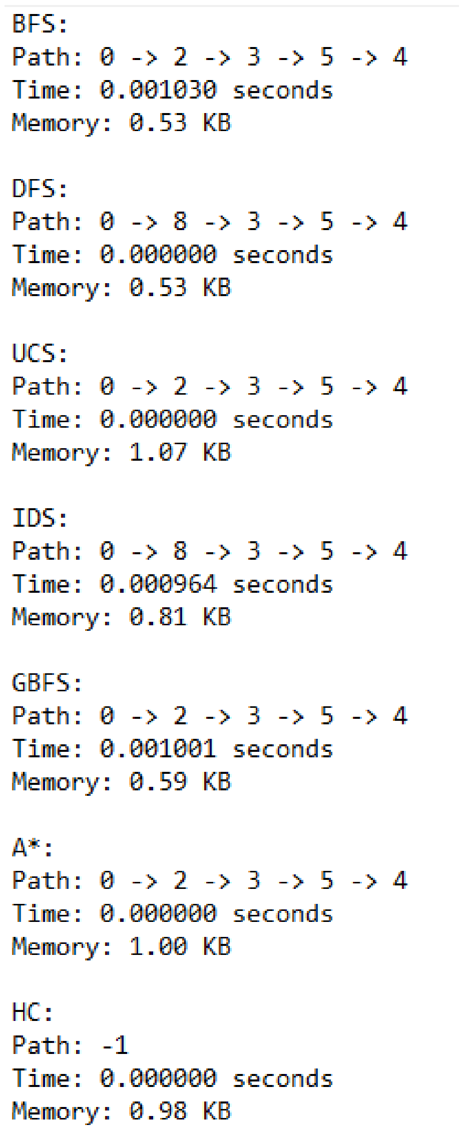
```
BFS:
Path: 0 -> 1 -> 7
Time: 0.000000 seconds
Memory: 0.53 KB

DFS:
Path: 0 -> 5 -> 6 -> 7
Time: 0.000000 seconds
Memory: 0.53 KB

UCS:
Path: 0 -> 1 -> 7
Time: 0.000000 seconds
Memory: 1.07 KB

IDS:
Path: 0 -> 1 -> 7
Time: 0.000000 seconds
Memory: 0.69 KB

GBFS:
Path: 0 -> 2 -> 3 -> 4 -> 7
Time: 0.008001 seconds
Memory: 0.75 KB

A*:
Path: 0 -> 1 -> 7
Time: 0.000000 seconds
Memory: 0.98 KB

HC:
Path: -1
Time: 0.000000 seconds
Memory: 0.98 KB
```

Figure 18: Result of test case 3

Based on the results on Figure 18, we can extract some information as follow:

1. Path Found:
   - BFS, UCS, IDS, and A* all found the same path: $0 \to 1 \to 7$.
   - DFS found a different path: $0 \to 5 \to 6 \to 7$, indicating its depth-first nature, which can lead to different paths that are not necessarily the shortest.
   - GBFS found a different path: $0 \to 2 \to 3 \to 4 \to 7$, highlighting its heuristic-driven approach that does not always lead to the shortest path.
   - HC failed to find a path, returning Path: -1.
2. Time Efficiency:
   - Most algorithms reported negligible computation time (0.000000 seconds), except GBFS which took longer (0.008001 seconds).
3. Memory Usage:
   - Memory usage varied among the algorithms, with BFS and DFS using the least (0.53 KB).
   - UCS used the most memory (1.07 KB), which is expected given it maintains the cost of paths.
   - A* and HC had higher memory usage (0.98 KB), reflecting their need to store additional heuristic information.
   - IDS and GBFS had moderate memory usage (0.69 KB and 0.75 KB respectively).

6.3.4 Test case 4



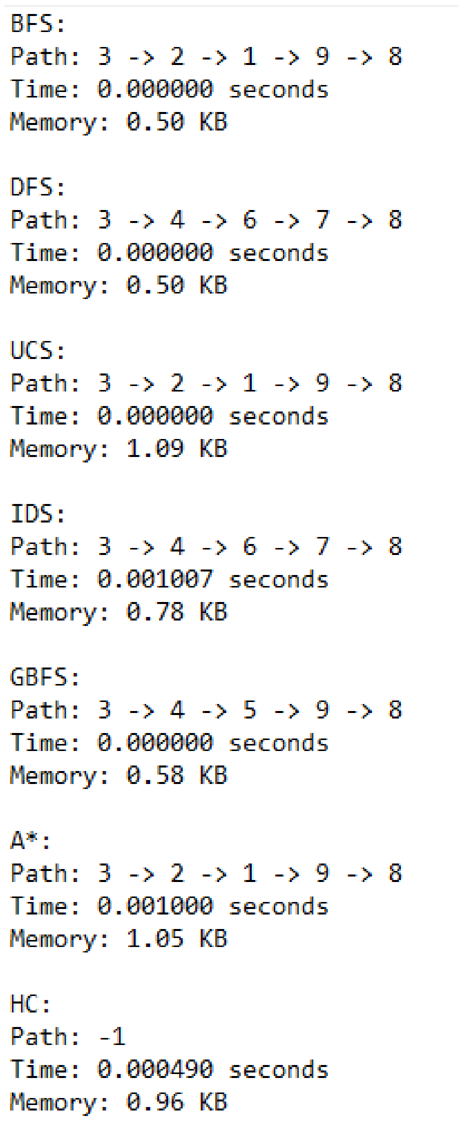
```
BFS:
Path: 0 -> 2 -> 3 -> 5 -> 4
Time: 0.001030 seconds
Memory: 0.53 KB

DFS:
Path: 0 -> 8 -> 3 -> 5 -> 4
Time: 0.000000 seconds
Memory: 0.53 KB

UCS:
Path: 0 -> 2 -> 3 -> 5 -> 4
Time: 0.000000 seconds
Memory: 1.07 KB

IDS:
Path: 0 -> 8 -> 3 -> 5 -> 4
Time: 0.000964 seconds
Memory: 0.81 KB

GBFS:
Path: 0 -> 2 -> 3 -> 5 -> 4
Time: 0.001001 seconds
Memory: 0.59 KB

A*:
Path: 0 -> 2 -> 3 -> 5 -> 4
Time: 0.000000 seconds
Memory: 1.00 KB

HC:
Path: -1
Time: 0.000000 seconds
Memory: 0.98 KB
```

Figure 19: Result of test case 4

Based on the results on Figure 19, we can extract some information as follow:

1. Path Found:
   - BFS, UCS, IDS, and A* all found the same path: $0 \to 2 \to 3 \to 5 \to 4$.
   - DFS: Found a different path:$0 \to 8 \to 3 \to 5 \to 4$, illustrating its depth-first nature which can lead to different paths that are not necessarily the shortest.
   - GBFS found the same path as BFS, UCS, IDS, and A* path: $0 \to 2 \to 3 \to 5 \to 4$, highlighting its heuristic-driven approach that does not always lead to the shortest path.
   - HC failed to find a path, returning Path: -1.
2. Time Efficiency:
   - Most algorithms reported negligible computation time (0.000000 seconds), except BFS (0.001030 seconds), IDS (0.000964 seconds), and GBFS (0.001001 seconds), which took slightly longer.
3. Memory Usage:
   - BFS and DFS used the least memory (0.53 KB).
   - UCS used the most memory (1.07 KB), which is expected given it maintains the cost of paths.
   - A* and HC had high memory usage (1.00 KB) and (0.98 KB), reflecting their need to store additional heuristic information.
   - IDS (0.81 KB) and GBFS (0.59 KB) had moderate memory usage.

6.3.5 Test case 5



```
BFS:
Path: 3 -> 2 -> 1 -> 9 -> 8
Time: 0.000000 seconds
Memory: 0.50 KB

DFS:
Path: 3 -> 4 -> 6 -> 7 -> 8
Time: 0.000000 seconds
Memory: 0.50 KB

UCS:
Path: 3 -> 2 -> 1 -> 9 -> 8
Time: 0.000000 seconds
Memory: 1.09 KB

IDS:
Path: 3 -> 4 -> 6 -> 7 -> 8
Time: 0.001007 seconds
Memory: 0.78 KB

GBFS:
Path: 3 -> 4 -> 5 -> 9 -> 8
Time: 0.000000 seconds
Memory: 0.58 KB

A*:
Path: 3 -> 2 -> 1 -> 9 -> 8
Time: 0.001000 seconds
Memory: 1.05 KB

HC:
Path: -1
Time: 0.000490 seconds
Memory: 0.96 KB
```

Figure 20: Result of test case 5

Based on the results on Figure 20, we can extract some information as follow:

1. Path Found:
   - BFS and UCS found the same path: $3 \to 2 \to 1 \to 9 \to 8$, indicating their ability to find the shortest or lowest-cost path.
   - DFS and IDS found a different path: $3 \to 4 \to 6 \to 7 \to 8$, showcasing DFS's depth-first nature which can lead to different, non-optimal paths.
   - GBFS found another path: $3 \to 4 \to 5 \to 9 \to 8$, highlighting its heuristic-driven approach that does not always yield the shortest path.
   - A* found the same path as BFS and UCS $3 \to 2 \to 1 \to 9 \to 8$, emphasizing its optimality when heuristics are accurate.
   - HC: Failed to find a path, returning Path: -1.
2. Time Efficiency:
   - Most algorithms reported negligible computation time (0.000000 seconds), except IDS (0.001007 seconds), A* (0.001000 seconds), and HC (0.000490 seconds), which took slightly longer.
3. Memory Usage:
   - BFS and DFS used the least memory (0.50 KB).
   - UCS used the most memory (1.09 KB), which is expected due to maintaining detailed path cost information.
   - A* (1.05 KB) and HC (0.96 KB) had high memory usage, reflecting their need to store additional heuristic information.
   - IDS (0.78 KB) and GBFS (0.58 KB) had moderate memory usage.

6.4 Conclusion on search strategies performance

After observing the performance metrics of test cases above, In this sub section I have drawn some insights for seven search strategies that is implemented in this lab.

- **Path Found:**
  - Consistent search strategies (BFS, UCS, A*): These algorithms consistently find the optimal or shortest path. They demonstrated reliability in locating the correct path in different scenarios.
  - Variable search strategies (DFS, IDS, GBFS):
    - DFS and IDS: Often find different paths due to their depth-first nature, which can lead to suboptimal solutions.
    - GBFS: Relies heavily on heuristics, leading to paths that are not always the shortest.
  - Failure search strategie (HC): Frequently failed to find a path, indicating its susceptibility to getting stuck in local minima.
- **Time Efficiency**
  - Fast Algorithms: BFS, DFS, UCS, GBFS generally reported negligible computation time.
  - Slightly Slower Algorithms: IDS and A* occasionally took slightly longer due to additional computations for depth limitations and heuristic evaluations.
  - Unreliable Performance (HC): Time efficiency varied, but even when fast, it failed to consistently find paths.
- **Memory Usage**
  - Memory-Efficient: BFS and DFS consistently used the least memory, making them suitable for environments with limited resources.
  - Memory-Intensive: UCS and A* required more memory due to maintaining detailed cost and heuristic information.
  - Moderate Memory Usage: IDS and GBFS had moderate memory requirements, balancing performance and resource usage.
  - High Memory with Limited Success (HC): Despite using considerable memory, it frequently failed to find a path.

7 Conclusion

We have numerous algorithms in artificial intelligence that we can use to address problems. Selecting an appropriate search algorithm is contingent upon the particular demands of an issue. The Best Course with Enough Resources: When computing resources are available, the lowest-cost pathways can be found most effectively with A* and UCS. Memory-Constrained Environments: Because BFS and DFS use less memory, they are better options. Balanced Performance: IDS and GBFS offer a decent compromise between performance and memory consumption. Complex Pathfinding: Because of its unpredictability and propensity to become trapped in local minima, HC is generally not advised.

References

[1] S. Russel and P. Norvig, Artificial intelligence: A Modern approach., 4th ed. Prentice Hall, 2021.

[2] “Time Complexity of breadth first search with adjacency matrix representation?,” Stack Overflow. https://stackoverflow.com/questions/20767396/time-complexity-of-breadth-first-search-with-adjacency-matrix-representation (accessed date: 7/7/2024)

[3] “DFS,” www.thealgorists.com.https://www.thealgorists.com/Algo/DFS (accessed date: 7/7/2024)

[4] “algorithm - Time complexity of uniform-cost search,” Stack Overflow. https://stackoverflow.com/questions/19204682/time-complexity-of-uniform-cost-search (accessed date: 8/7/2024)

[5] Bui Tien Len, Uninformed search, lec03-uninformed-search.pdf, 2021 (accessed Date: 8/7/2024)

[6] S. Sakaue and T. Oki, “Sample Complexity of Learning Heuristic Functions for Greedy-Best-First and A* Search,” arXiv.org, May 23, 2022. https://arxiv.org/abs/2205.09963 (accessed 10/7/2024).

[7] Bui Tien Len, Informed search, lec03-informed-search.pdf, 2021 (accessed Date: 8/7/2024)

[8] “HillClimbing (Orbital),” symbolaris.com. https://symbolaris.com/orbital/Orbital-doc/api/orbital/algorithm/template/HillClimbing.html (10/7/2024).