VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report

Lab 1: Searching

Subject: Fundamentals of Artificial Intelligence

Instructor:

Nguyễn Ngọc Thảo

Lab Instructor:

Nguyễn Thanh Tình

Hồ Thị Thanh Tuyến

Student:

Nguyễn Thanh Nam

Ho Chi Minh City, July 2024

Contents

1 Information page

- Full name: Nguyễn Thanh Nam
- ID: 22127286
- Class: 22CLC10

2 Requirements

Details	Rate
Implement BFS correctly.	10%
Implement DFS correctly.	10%
Implement UCS correctly.	10%
Implement IDS correctly.	10%
Implement GBFS correctly.	10%
Implement A* correctly.	10%
Implement Hill-climbing correctly.	10%
Generate at least 5 test cases for all algorithms. Describe them in the experiment section of your report.	10%
Report your algorithm, experiment with some reflection or comments.	20%

Table 1: Completion rate

3 Algorithm description

3.1 Breadth-First search (BFS)

3.1.1 Concepts

Breadth-First Search (BFS) is a graph traversal algorithm that systematically explores a graph by visiting all the vertices at a given level before moving on to the next level. It starts from a starting vertex, enqueues it into a queue, and marks it as visited. Then, it dequeues a vertex from the queue, visits it, and enqueues all its unvisited neighbors into the queue. This process continues until the queue is empty. [3]

3.1.2 Complexity

- **Time Complexity:** $\mathcal{O}(b^d)$, where b is the branching factor of the graph and d (distance) is the number of edges between the start node and the nodes you are interested in finding.
- **Space Complexity:** $\mathcal{O}(b^d)$.

3.1.3 Evaluation

- **Completeness:** Yes, since BFS exhaustively searches all possible paths level by level, it is guaranteed to find the shortest path to the goal if one exists, making it complete.
- **Optimal:** Yes if costs are all uniform.

3.1.4 Implementation

Algorithm 1 Breadth-First Search (BFS)

```
function BFS(start, end)
    if start = end then
        return [start]                                         ▷ Return single-path node
    end if
    Initialize the visited list with False values
    Mark the start node as visited
    Initialize the queue with the start node and its path
    while the queue is not empty do
        Pop the first element from the queue, setting it as the current node and path
        Retrieve and sort the neighbors of the current node
        for each neighbor in the sorted list of neighbors do
            if neighbor is not visited then
                Mark the neighbor as visited
                if neighbor is end then
                    return the current path concatenated with the neighbor
                end if
                Append the neighbor and the updated path to the queue
            end if
        end for
    end while
    return −1                                                  ▷ No path found
end function
```

Initially, the BFS function checks if the start and end nodes are the same, returning a list containing only the start node if true. It marks the start node as visited and uses a queue to track nodes to explore, with each entry consisting of a node and the path taken to reach it. While the queue is not empty, it dequeues a node-path pair, checks its neighbors (sorted for consistent ordering), and extends the path by appending each unvisited neighbor, marking them as visited. If an extended path reaches the end node, it returns the complete path. If no path is found after exploring all nodes, it returns -1.

3.2 Tree-Search Depth-First Search (DFS)

3.2.1 Concepts

Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. [4]

3.2.2 Complexity

- **Time Complexity:** $\mathcal{O}(b^d)$, where every state has b successors and the solution is at depth d.
- **Space Complexity:** $\mathcal{O}(bd)$ for implicit graphs without elimination of duplicate nodes.

3.2.3 Evaluation

- **Completeness:** Yes if loops prevented.
- **Optimal:** No, the "leftmost" solution, regardless of depth or cost.

3.2.4 Implementation

Algorithm 2 Depth-First Search (DFS)

```
function DFS(start, end)
    if start = end then
        return [start]                                        ▷ Return single-node path
    end if
    Initialize the stack with the start node and its path: stack ← [(start, [start])]
    while the stack is not empty do
        Pop the vertex and its path from the top of the stack: (vertex, path) ← stack.pop()
        Retrieve and sort the neighbors of vertex in reverse order
        for each neighbor in the sorted list of neighbors do
            if neighbor is not in path then
                if neighbor is end then
                    return path + [neighbor]
                end if
                Push (neighbor, path + [neighbor]) onto the stack
            end if
        end for
    end while
    return −1                                                 ▷ No path found
end function
```

The DFS function use a non-recursive approach. The benefits of using an iterative version of DFS extend beyond not exceeding recursion limits. It also makes DFS fit in better with other algorithms. Initially, it begins by checking if the start and end nodes are the same, returning a list containing only the start node if true. It initializes a stack with the start node and its path. While the stack is not empty, it pops a node-path pair, explores its neighbors sorted in reverse order (for consistent behavior), and extends the path by appending each unvisited neighbor to the stack. If a neighbor leads to the end node, it returns the complete path. If no path is found after exploring all nodes, it returns -1, indicating failure to find a path. [2]

3.3 Uniform-Cost search (UCS)

3.3.1 Concepts

Uniform-Cost Search (UCS) is a variant of Dijikstra's algorithm. Here, instead of inserting all vertices into a priority queue, we insert only the source, then one by one insert when needed. In every step, we check if the item is already in the priority queue (using the visited array). If yes, we perform the decrease key, else we insert it. This variant of Dijkstra is useful for infinite graphs and that graph which are too large to represent in memory. Uniform-Cost Search is mainly used in Artificial Intelligence. [7]

3.3.2 Complexity

- **Time Complexity:** $\mathcal{O}\left(b^{1+\lceil \frac{C^*}{\epsilon} \rceil}\right)$ (Let C* be the cost of the optimal solution, and $\epsilon > 0$ be the lower bound of the cost of each action).
- **Space Complexity:** $\mathcal{O}\left(b^{1+\lceil \frac{C^*}{\epsilon} \rceil}\right)$.

3.3.3 Evaluation

- **Completeness:** Yes (assume that the best solution has a finite cost and minimum arc cost is positive).
- **Optimal:** Yes, it always finds the least-cost path to the goal.

3.3.4 Implementation

Algorithm 3 Uniform-Cost Search (UCS)

```
function UCS(start, end)
    if start = end then
        return ([start], 0)                ▷ Return single-node path with zero cost
    end if
    Initialize visited as a boolean dictionary to track visited nodes
    Initialize queue as a priority queue
    Enqueue the start node with zero cost: queue.put((0, [start]))
    while queue is not empty do
        Dequeue the path with the lowest cost: (cost, path) ← queue.get()
        Get the last node in the current path: node ← path[−1]
        if node = end then
            return path, cost              ▷ Return the path and total cost if the end node is reached
        end if
        if not visited[node] then
            Mark the node as visited: visited[node] ← true
            for each neighbour in self.graph[node] do
                if neighbor not in path then
                    Create a new path by extending the current path to the neighbor: new_path ← path + [neighbour]
                    Calculate the new cost by adding the cost of the edge to the current cost: new_cost ← cost + self.cost[(node, neighbour)]
                    Enqueue the new path with the new cost: queue.put((new_cost, new_path))
                end if
            end for
        end if
    end while
    return (−1, −1)                        ▷ No path found
end function
```

The UCS function begins by checking if the start and end nodes are the same, returning the start node and a cost of 0 if true. Using a priority queue, it explores paths, always expanding the least-cost path first. Each entry in the queue consists of the current cost and the path taken to reach a node. When the end node is reached, the function returns the path and its total cost. Nodes are marked as visited to prevent re-expansion. If no path is found, it returns -1 for both the path and the cost.

3.4 Iterative Deepening Search (IDS)

3.4.1 Concepts

Iterative Deepening Search (IDS) is an iterative searching technique that combines the advantages of both DFS and BFS. While searching a particular node in a graph representation BFS requires lots of space thus increasing the space complexity and the DFS takes a little more time thus this search strategy has much time complexity and also DFS does not always find the cheapest path. To overcome all these drawbacks of DFS and BFS, IDS is implemented. [1, 6]

3.4.2 Complexity

- **Time Complexity:** $\mathcal{O}(b^d)$, where b is the branching factor and d is the depth of the shallowest solution (if there is a solution).
- **Space Complexity:** $\mathcal{O}(bd)$ (if there is a solution).

3.4.3 Evaluation

- **Completeness:** Yes when the branching factor if finite.
- **Optimal:** Yes if step cost is equals to 1.

3.4.4 Implementation

Algorithm 4 Depth-Limited Search (DLS)

```
function DLS(start, end, limit, path ← [])
    if start = end then
        return True                              ▷ Return True if the start node is the end node
    end if
    if limit ≤ 0 then
        return False                             ▷ Return False if the depth limit is reached
    end if
    for each neighbour in self.graph[start] do
        if neighbour not in path then
            Add neighbour to path
            if DLS(neighbour, end, limit − 1, path) then
                return path                      ▷ Return the path if a path is found within the depth limit
            end if
            Remove neighbour from path
        end if
    end for
    return False                                 ▷ Return False if no path is found
end function
```

The DLS function works recursively, checking if the current node is the end node and returning True if it is. If the depth limit is reached, it returns False. Otherwise, it explores each neighbor of the current node that is not already in the path, adding the neighbor to the path and recursively calling DLS with a decremented depth limit. If a valid path is found within the limit, it returns the path; otherwise, it backtracks by removing the last node from the path.

Algorithm 5 Iterative Deepening Search (IDS)

```
function IDS(start, end)
    if start = end then
        return [start]                           ▷ Return single-node path
    end if
    for each depth from 0 to self.n − 1 do
        Perform Depth-Limited Search with the current depth: path ← DLS(start, end, depth)
        if path then
            return [start] + path                ▷ Return the path if a path is found within the current depth limit
        end if
    end for
    return −1                                    ▷ No path found
end function
```

The IDS function starts with a depth of 0 and increases the limit incrementally until it finds a valid path or exhausts the search space. If the start and end nodes are the same, it returns a list containing only the start node. For each depth, IDS calls DLS and checks if a path is found. If DLS returns a path, IDS prepends the start node to the path and returns it. This approach combines the space efficiency of depth-first search with the completeness of breadth-first search.

3.5 Greedy Best-First Search (GBFS)

3.5.1 Concepts

Greedy Best-First Search (GBFS) is an AI search algorithm that attempts to find the most promising path from a given starting point to a goal. It prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path. The algorithm works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached. [5]

3.5.2 Complexity

- **Time Complexity:** $\mathcal{O}(b^m)$, reduced substantially with a good heuristic, on certain problems reaching $\mathcal{O}(bm)$.
- **Space Complexity:** $\mathcal{O}(bm)$, all nodes are kept in memory.

3.5.3 Evaluation

- **Completeness:** No, GBFS may get stuck forever because it only considers the heuristic value, which estimates the cost to reach the goal, and ignores the actual path cost. If the heuristic misguides it, GBFS can repeatedly explore paths that seem promising but don't lead to the goal, potentially looping indefinitely without finding a solution.
- **Optimal:** No, because it only considers the heuristic value, potentially missing the least-cost path to the goal.

3.5.4 Implementation

Algorithm 6 Greedy Best-First Search (GBFS)

```
function GBFS(start, end)
    if start = end then
        return [start]                                  ▷ Return single-node path
    end if
    Initialize the visited set with the start node: visited ← {start}
    Initialize the queue and push the start node with its heuristic value
    Initialize the trace dictionary for reconstructing the path
    while the queue is not empty do
        Pop the node with the lowest heuristic value from the queue: current ← heapq.heappop(pq)
        Set currentNode to the node value of current
        if currentNode = end then
            Initialize an empty list for the path
            while currentNode ≠ start do
                Append currentNode to the path
                Update currentNode to its parent from the trace dictionary
            end while
            Append start to the path
            Reverse the path
            return path
        end if
        for each neighbor in the neighbors of currentNode do
            if neighbor is not in visited then
                Push the neighbor to the queue with its heuristic value
                Add neighbor to the visited set
                Update the trace dictionary to record the parent of neighbor
            end if
        end for
    end while
    return −1                                           ▷ No path found
end function
```

Initially, the GBFS begins by checking if the start and end nodes are the same, returning a list containing only the start node if true. Using a priority queue (min-heap), it explores nodes, always selecting the node with the lowest heuristic value. Nodes are marked as visited and their predecessors are tracked in the trace dictionary. When the end node is reached, the function reconstructs and returns the path by backtracking through the trace dictionary. If no path is found, the function returns -1. [12]

3.6 Graph-Search A* (A*)

3.6.1 Concepts

Graph-Search A* (A*) is the advanced form of the BFS algorithm (Breadth-first search), which searches for the shorter path first than, the longer paths. It is a complete as well as an optimal solution for solving path and grid problems. [11]

3.6.2 Complexity

- **Time Complexity:** $\mathcal{O}(b^d)$, depends on the heuristic.
- **Space Complexity:** $\mathcal{O}(b^d)$, where d is the depth of the solution (the length of the shortest path) and b is the branching factor (the average number of successors per state), as it stores all generated nodes in memory. [13]

3.6.3 Evaluation

- **Completeness:** Yes if all step costs exceed some finite ϵ and if b is finite (assume that all action costs are at least ϵ greater than 0).
- **Optimal:** Yes, with conditions on heuristic being used.

3.6.4 Implementation

Algorithm 7 Graph-Search A* (A*)

```
function A_STAR(start, end)
    if start = end then
        return [start]                                   ▷ Return single-node path
    end if
    Initialize open_list with the start node
    Initialize closed_list as empty
    Initialize g[start] to 0 and parents[start] to start
    while open_list is not empty do
        Set n to the node in open_list with the lowest g[v] + heuristics[v]
        if n = end then
            Initialize path as an empty list
            while parents[n] ≠ n do
                Append n to path
                Update n to parents[n]
            end while
            Append start to path and reverse it
            return path
        end if
        for each neighbor m of n do
            weight ← cost[(n, m)]
            if m is not in open_list and m is not in closed_list then
                Add m to open_list
                Update parents[m] to n and g[m] to g[n] + weight
            else if g[m] > g[n] + weight then
                Update g[m] to g[n] + weight and parents[m] to n
                if m is in closed_list then
                    Move m to open_list
                end if
            end if
        end for
        Move n from open_list to closed_list
    end while
    return −1                                            ▷ No path found
end function
```

The A_STAR function initializes open_list with the start node and closed_list as empty. The g dictionary keeps track of the cost from the start node to each node, while parents records the parent of each node for path reconstruction. The function repeatedly selects the node from open_list with the lowest combined cost (g[node] + heuristic[node]). If this node is the end node, it reconstructs and returns the path. Otherwise, it explores its neighbors, updating their costs and parent relationships as necessary. Nodes are moved from open_list to closed_list once fully explored. If the end node is not reachable, the function returns -1.

3.7 Hill-Climbing (HC) variant

3.7.1 Concepts

Hill-Climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value. [8]

3.7.2 Evaluation

- **Completeness:** No, Hill Climbing algorithm may get stuck.
- **Optimal:** No, it is a local search algorithm that makes incremental improvements by continuously moving towards higher values in the search space.

3.7.3 Implementation

Algorithm 8 Hill Climbing (HC)

```
function HC(start, end)
    if start = end then
        return [start]                                        ▷ Return single-node path
    end if
    Set current_node to the tuple (start, heuristic value of start)
    Initialize path with current_node[0]
    while True do
        Initialize next_node to False
        for each neighbor in self.graph[current_node[0]] do
            if neighbor is not in path then
                if heuristic value of neighbor is less than current_node[1] then
                    Set next_node to True
                    Update current_node to (neighbor, heuristic value of neighbor)
                end if
            end if
        end for
        if not next_node then
            return −1                                         ▷ No better neighbor found
        end if
        if current_node[0] = end then
            Append current_node[0] to path
            return path
        end if
        Append current_node[0] to path
    end while
end function
```

Initially, the HC function checks if the start and end nodes are the same, returning a list containing only the start node if true. Starting from the current node (initially the start node), it looks for the neighbor with the lowest heuristic value that has not been visited. If such a neighbor is found, it moves to that neighbor, updating the path and cost accordingly. If the end node is reached, the path is returned. If no better neighbor is found, indicating a local minimum or plateau, the function returns -1, signifying failure to find a path. This algorithm does not guarantee finding the optimal path, as it may get stuck in local minima.

4 Program details

4.1 Library

- sys: Provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
- time: Provides various time-related functions.
- tracemalloc: This module is used for tracing memory allocations in Python programs.
- collections: Provides alternative data structures to the built-in ones like deque (a double-ended queue) and defaultdict (a dictionary with default values for non-existent keys).
- heapq: The property of this data structure in Python is that each time the smallest heap element is popped(min-heap).
- queue: Provides priority queue data structure, where elements are stored in the queue and retrieved in ascending order of their priority.

4.2 Usage

Use this command to run the program:

```
python search.py {input file}
```

or

```
python3 search.py {input file}
```

5 Test cases

5.1 Test case 1

- **Start node:** 0.
- **End node:** 9.

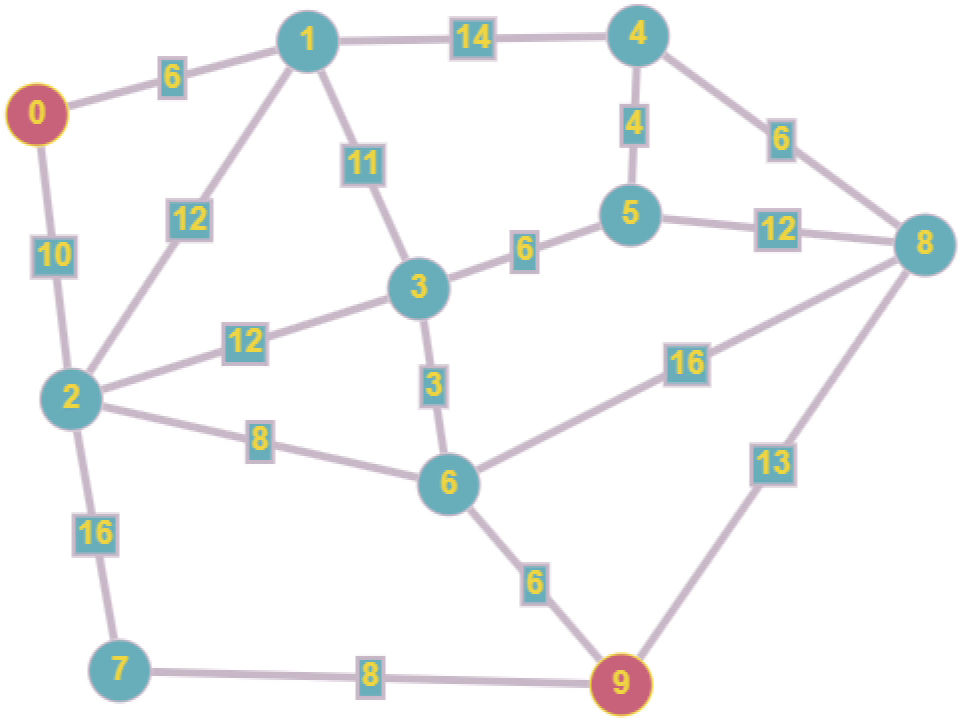

Figure 1: Test case 1

Vertex	Heuristic
0	10
1	5
2	6
3	4
4	15
5	5
6	8
7	1
8	10
9	0

Table 2: Heuristic table (Test case 1)

5.1.1 Result

Algorithm	Path return	Time (second)	Memory usage (KB)
Breadth-First Search	$0 \rightarrow 2 \rightarrow 6 \rightarrow 9$	0	0.4453125
Depth-First Search	$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 8 \rightarrow 9$	0	0.6953125
Uniform-Cost Search	$0 \rightarrow 2 \rightarrow 6 \rightarrow 9$	0	6.1015625
Iterative Deepening Search	$0 \rightarrow 2 \rightarrow 6 \rightarrow 9$	0	0.25
Greedy Best-First Search	$0 \rightarrow 2 \rightarrow 7 \rightarrow 9$	0	2.609375
Graph-Search A*	$0 \rightarrow 2 \rightarrow 6 \rightarrow 9$	0	2.046875
Hill-Climbing	-1	0	0.125

Table 3: Result of Test case 1

5.1.2 Explanation

- **BFS:**
  - Initialization:
    - Start node = 0, end node = 9
    - visited = [True, False, False, False, False, False, False, False, False, False]
    - queue = [(0, [0])]
  - First iteration:
    - Dequeue: (0, [0])
    - Neighbors of 0 (unvisited): [1, 2], sorted neighbors: [1, 2]
    - Enqueue (1, [0, 1]), (2, [0, 2])
    - visited = [True, True, True, False, False, False, False, False, False, False]
    - queue = [(1, [0, 1]), (2, [0, 2])]
  - Second iteration:
    - Dequeue: (1, [0, 1])
    - Neighbors of 1 (unvisited): [3, 4], sorted neighbors: [3, 4]
    - Enqueue (3, [0, 1, 3]), (4, [0, 1, 4])
    - visited = [True, True, True, True, True, False, False, False, False, False]
    - queue = [(2, [0, 2]), (3, [0, 1, 3]), (4, [0, 1, 4])]

- Third iteration:
  * Dequeue: (2, [0, 2])
  * Neighbors of 2 (unvisited): [6, 7], sorted neighbors: [6, 7]
  * Enqueue (6, [0, 2, 6]), (7, [0, 2, 7])
  * visited = [True, True, True, True, True, False, True, True, False, False]
  * queue = [(3, [0, 1, 3]), (4, [0, 1, 4]), (6, [0, 2, 6]), (7, [0, 2, 7])]
- Fourth iteration:
  * Dequeue: (3, [0, 1, 3])
  * Neighbors of 3 (unvisited): [5], sorted neighbors: [5]
  * Enqueue (5, [0, 1, 3, 5])
  * visited = [True, True, True, True, True, True, True, True, False, False]
  * queue = [(4, [0, 1, 4]), (6, [0, 2, 6]), (7, [0, 2, 7]), (5, [0, 1, 3, 5])]
- Fifth iteration:
  * Dequeue: (4, [0, 1, 4])
  * Neighbors of 4 (unvisited): [8], sorted neighbors: [8]
  * Enqueue (8, [0, 1, 4, 8])
  * visited = [True, True, True, True, True, True, True, True, True, False]
  * queue = (6, [0, 2, 6]), (7, [0, 2, 7]), (5, [0, 1, 3, 5]), (8, [0, 1, 4, 8])
- Sixth iteration:
  * Dequeue: (6, [0, 2, 6])
  * Neighbors of 6 (unvisited): [9], sorted neighbors: [9] → Return the path: [0, 2, 6, 9]
  * visited = [True, True, True, True, True, True, True, True, True, True]

- **DFS:**
  - Initial Stack: [(0, [0])]
    * Start from node 0.
  - First Iteration: Pop (0, [0])

    * Neighbors: 1, 2 (sorted in reverse order).
    * Stack: [(2, [0, 2]), (1, [0, 1])]
  - Second Iteration: Pop (1, [0, 1])
    * Neighbors: 2, 3, 4 (sorted in reverse order).
    * Stack: [(2, [0, 2]), (4, [0, 1, 4]), (3, [0, 1, 3]), (2, [0, 1, 2])]
  - Third Iteration: Pop (2, [0, 1, 2])
    * Neighbors: 6, 3, 7 (sorted in reverse order).
    * Stack: [(2, [0, 2]), (4, [0, 1, 4]), (3, [0, 1, 3]), (7, [0, 1, 2, 7]), (6, [0, 1, 2, 6]), (3, [0, 1, 2, 3])]
  - Fourth Iteration: Pop (3, [0, 1, 2, 3])
    * Neighbors: 5, 6 (sorted in reverse order).
    * Stack: [(2, [0, 2]), (4, [0, 1, 4]), (3, [0, 1, 3]), (7, [0, 1, 2, 7]), (6, [0, 1, 2, 6]), (6, [0, 1, 2, 3, 6]), (5, [0, 1, 2, 3, 5])]
  - Fifth Iteration: Pop (5, [0, 1, 2, 3, 5])
    * Neighbors: 4, 8 (sorted in reverse order).
    * Stack: [(2, [0, 2]), (4, [0, 1, 4]), (3, [0, 1, 3]), (7, [0, 1, 2, 7]), (6, [0, 1, 2, 6]), (6, [0, 1, 2, 3, 6]), (8, [0, 1, 2, 3, 5, 8]), (4, [0, 1, 2, 3, 5, 4])]
  - Sixth Iteration: Pop (4, [0, 1, 2, 3, 5, 4])
    * Neighbors: 8 (sorted in reverse order).
    * Stack: [(2, [0, 2]), (4, [0, 1, 4]), (3, [0, 1, 3]), (7, [0, 1, 2, 7]), (6, [0, 1, 2, 6]), (6, [0, 1, 2, 3, 6]), (8, [0, 1, 2, 3, 5, 8]), (8, [0, 1, 2, 3, 5, 4, 8])]
  - Seventh Iteration: Pop (8, [0, 1, 2, 3, 5, 4, 8])
    * Neighbors: 9 (sorted in reverse order).
    * Stack: [(2, [0, 2]), (4, [0, 1, 4]), (3, [0, 1, 3]), (7, [0, 1, 2, 7]), (6, [0, 1, 2, 6]), (6, [0, 1, 2, 3, 6]), (8, [0, 1, 2, 3, 5, 8]), (9, [0, 1, 2, 3, 5, 4, 8, 9])] → Return path: [0, 1, 2, 3, 5, 4, 8, 9]

- **UCS:**

Explored set	Priority Queue (Cost, Path)
{}	(0, [0])
{0}	(6, [0, 1]), (10, [0, 2])
{0, 1}	(10, [0, 2]), (17, [0, 1, 3]), (20, [0, 1, 4])
{0, 1, 2}	(17, [0, 1, 3]), (18, [0, 2, 6]), (20, [0, 1, 4]), (26, [0, 2, 7])
{0, 1, 2, 3}	(18, [0, 2, 6]), (20, [0, 1, 4]), (23, [0, 1, 3, 5]), (26, [0, 2, 7])
{0, 1, 2, 3, 6}	(20, [0, 1, 4]), (23, [0, 1, 3, 5]), (24, [0, 2, 6, 9]), (26, [0, 2, 7]), (34, [0, 2, 6, 8])
{0, 1, 2, 3, 6, 4}	(23, [0, 1, 3, 5]), (24, [0, 2, 6, 9]), (26, [0, 2, 7]), (26, [0, 1, 4, 8])
{0, 1, 2, 3, 6, 4, 5}	(24, [0, 2, 6, 9]), (26, [0, 2, 7]), (26, [0, 1, 4, 8])
{0, 1, 2, 3, 6, 4, 5, 9}	Return path: [0, 2, 6, 9]

Table 4: UCS

- **IDS:**
  - Depth = 0: 0
  - Depth = 1: 0, (1), (2)
  - Depth = 2: 0, 1, (2, 3, 4), 2, (1, 3, 6, 7)
  - Depth = 3: 0, 1, 2, (3, 6, 7), 3, (2, 5, 6), 4, (5, 8), 2, 1, (3, 4), 3, (1, 5, 6), 6, (3, 8, 9) → Return the path: [0, 2, 6, 9]
- **GBFS:**

Closed List	Queue (Current Heuristic, Path)
{}	(10, [0])
{0}	(5, [0, 1]), (6, [0, 2])
{0, 1}	(4, [0, 1, 3]), (6, [0, 2]), (15, [0, 1, 4])
{0, 1, 3}	(5, [0, 1, 3, 5]), (6, [0, 2]), (8, [0, 1, 3, 6]), (15, [0, 1, 4])
{0, 1, 3, 5}	(6, [0, 2]), (8, [0, 1, 3, 6]), (10, [0, 1, 3, 5, 8]), (15, [0, 1, 4])
{0, 1, 3, 5, 2}	(1, [0, 2, 7]), (8, [0, 1, 3, 6]), (10, [0, 1, 3, 5, 8]), (15, [0, 1, 4])
{0, 1, 3, 5, 2, 7}	Return path: [0, 2, 7, 9]

Table 5: GBFS

- **A*:**

Closed List	**Expand Node**	**Priority Queue (Cost, Path)**
{}	0	(11, [0, 1]), (16, [0, 2])
{0}	1	(16, [0, 2]), (21, [0, 1, 3]), (35, [0, 1, 4])
{0, 1}	2	(21, [0, 1, 3]), (26, [0, 2, 6]), (27, [0, 2, 7]), (35, [0, 1, 4])
{0, 1, 2}	3	(26, [0, 2, 6]), (27, [0, 2, 7]), (28, [0, 1, 3, 5]), (28, [0, 1, 3, 6]), (35, [0, 1, 4])
{0, 1, 2, 6}	6	(24, [0, 2, 6, 9]), (27, [0, 2, 7]), (28, [0, 1, 3, 5]), (28, [0, 1, 3, 6]), (35, [0, 1, 4])
{0, 1, 2, 6, 9}	9	(24, [0, 2, 6, 9]), (27, [0, 2, 7]), (28, [0, 1, 3, 5]), (28, [0, 1, 3, 6]), (35, [0, 1, 4]) → Return path: [0, 2, 6, 9]

Table 6: A*

- **Hill-Climbing:**

Path	**C**urrent Heuristic	**Neighbor (Node, Heuristic)**
[0]	10	(1, 5), (2, 6)
[0, 1]	5	(3, 4), (4, 15)
[0, 1, 3]	4	(2, 6), (5, 5), (6, 8) → Path return: -1 → No better neighbor found (Current Heuristic < All Neighbor's Heuristic)

Table 7: Hill Climbing

5.2 Test case 2

- **Start node:** 0.
- **End node:** 5.

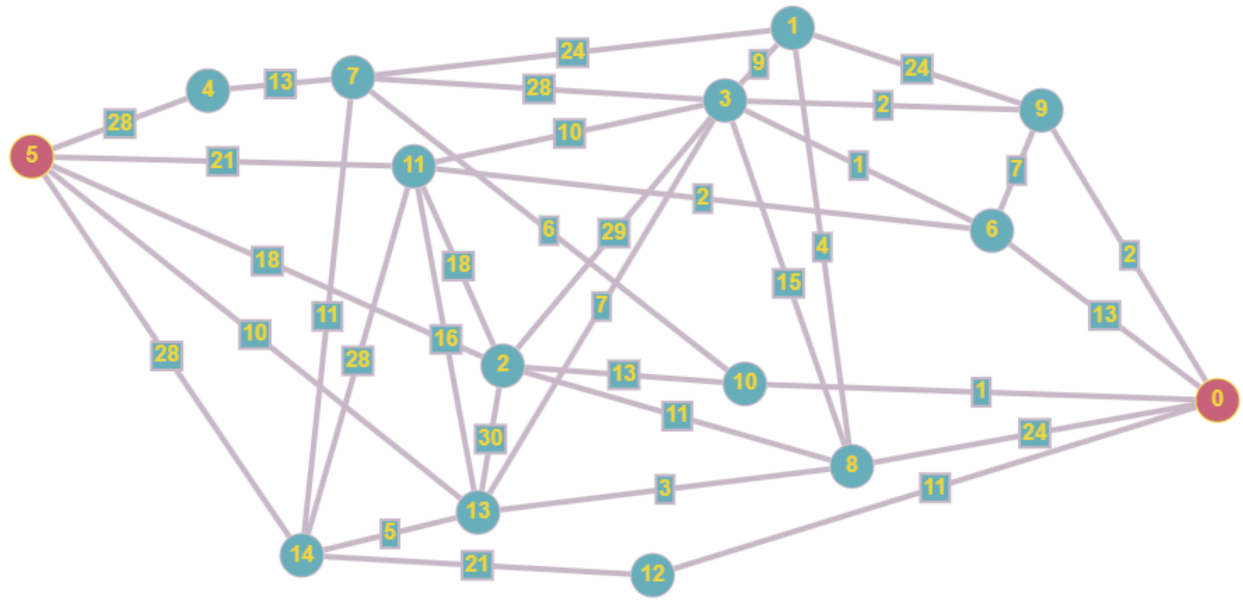

Figure 2: Test case 2

Vertex	Heuristic
0	15
1	5
2	9
3	17
4	2
5	0
6	12
7	4
8	8
9	9
10	10
11	16
12	13
13	17
14	16

Table 8: Heuristic table (Test case 2)

5.2.1 Result

Algorithm	Path return	Time (second)	Memory usage (KB)
Breadth-First Search	$0 \to 6 \to 11 \to 5$	0	0.8046875
Depth-First Search	$0 \to 6 \to 3 \to 1 \to 7 \to 4 \to 5$	0	1.4375
Uniform-Cost Search	$0 \to 9 \to 3 \to 13 \to 5$	0.0011556149	9.5390625
Iterative Deepening Search	$0 \to 6 \to 11 \to 5$	0	0.25
Greedy Best-First Search	$0 \to 8 \to 1 \to 7 \to 4 \to 5$	0	3.921875
Graph-Search A*	$0 \to 9 \to 3 \to 6 \to 11 \to 5$	0	2.59375
Hill-Climbing	$0 \to 8 \to 1 \to 7 \to 4 \to 5$	0	0.125

Table 9: Result of Test case 2

5.3 Test case 3

- **Start node:** 5.
- **End node:** 11.

Figure 3: Test case 3

Vertex	Heuristic
0	3
1	2
2	2
3	16
4	13
5	11
6	8
7	16
8	6
9	20
10	6
11	0
12	16
13	7
14	13

Table 10: Heuristic table (Test case 3)

5.3.1 Result

Algorithm	Path return	Time (second)	Memory usage (KB)
Breadth-First Search	$5 \to 14 \to 12 \to 11$	0	0.859375
Depth-First Search	$5 \to 10 \to 0 \to 6 \to 11$	0	0.6875
Uniform-Cost Search	$5 \to 10 \to 8 \to 9 \to 11$	0	8.9296875
Iterative Deepening Search	$5 \to 14 \to 12 \to 11$	0	0.25
Greedy Best-First Search	$5 \to 10 \to 0 \to 6 \to 11$	0	3.8984375
Graph-Search A*	$5 \to 10 \to 8 \to 3 \to 11$	0	2.59375
Hill-Climbing	-1	0	0.125

Table 11: Result of Test case 3

5.4 Test case 4

- **Start node:** 17.
- **End node:** 14.

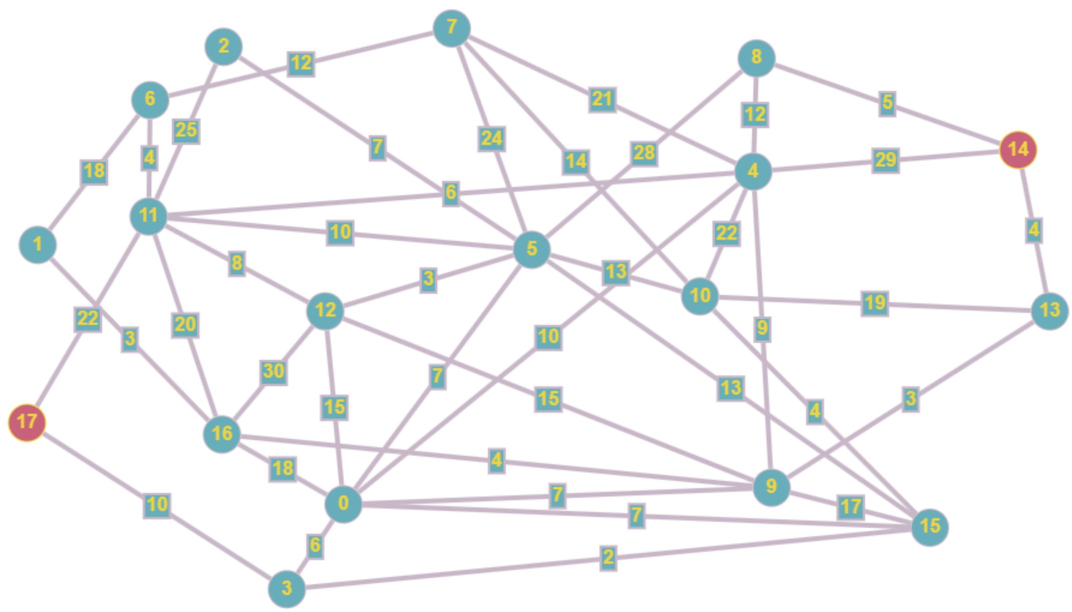

Figure 4: Test case 4

Vertex	Heuristic
0	4
1	11
2	17
3	15
4	4
5	8
6	12
7	10
8	5
9	3
10	19
11	15
12	18
13	14
14	0
15	13
16	5
17	16

Table 12: Heuristic table (Test case 4)

5.4.1 Result

Algorithm	Path return	Time (second)	Memory usage (KB)
Breadth-First Search	17 → 11 → 4 → 14	0	0.921875
Depth-First Search	17 → 3 → 0 → 4 → 14	0	0.4453125
Uniform-Cost Search	17 → 3 → 0 → 9 → 13 → 14	0.0011906624	11.59375
Iterative Deepening Search	17 → 11 → 4 → 14	0	0.25
Greedy Best-First Search	17 → 11 → 4 → 14	0	4.5390625
Graph-Search A*	17 → 3 → 0 → 9 → 13 → 14	0	2.59375
Hill-Climbing	17 → 11 → 4 → 14	0	0.125

Table 13: Result of Test case 4

5.5 Test case 5

- **Start node:** 0.
- **End node:** 16.

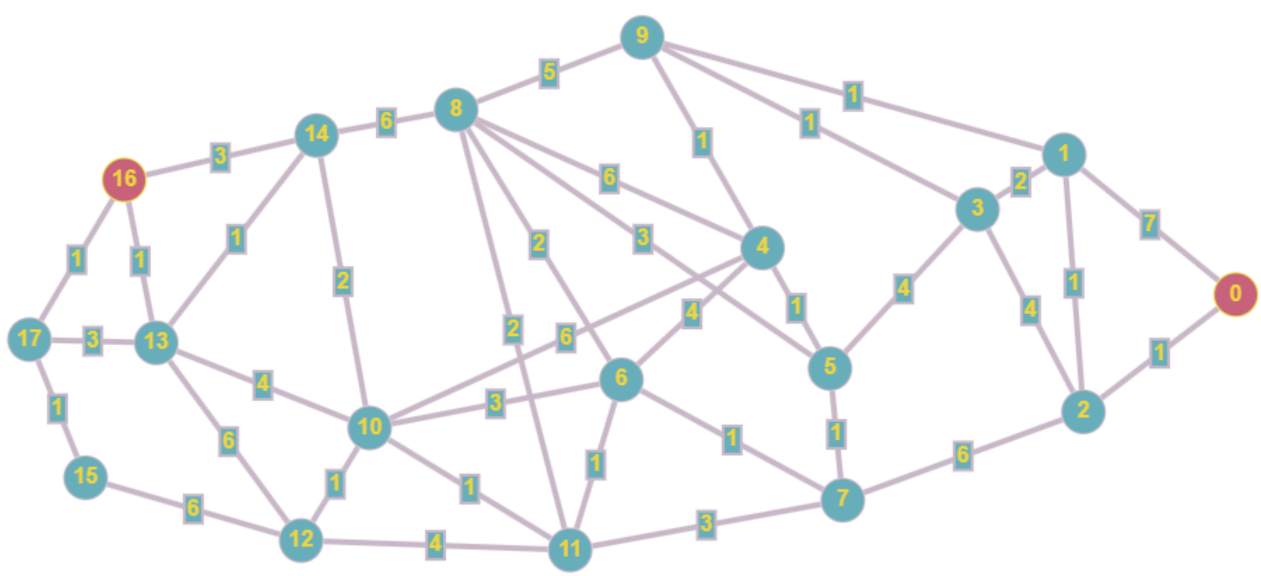

Figure 5: Test case 5

Vertex	Heuristic
0	10
1	7
2	9
3	11
4	5
5	10
6	3
7	8
8	6
9	3
10	12
11	16
12	18
13	6
14	14
15	5
16	0
17	1

Table 14: Heuristic table (Test case 5)

5.5.1 Result

Algorithm	Path return	Time (second)	Memory usage (KB)
Breadth-First Search	$0 \rightarrow 1 \rightarrow 9 \rightarrow 8 \rightarrow 14 \rightarrow 16$	0	0.6953125
Depth-First Search	$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 11 \rightarrow 8 \rightarrow 14 \rightarrow 16$	0	1.921875
Uniform-Cost Search	$0 \rightarrow 2 \rightarrow 1 \rightarrow 9 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 11 \rightarrow 10 \rightarrow 14 \rightarrow 13 \rightarrow 16$	0.001106739	8.328125
Iterative Deepening Search	$0 \rightarrow 1 \rightarrow 9 \rightarrow 8 \rightarrow 14 \rightarrow 16$	0	0.40625
Greedy Best-First Search	$0 \rightarrow 1 \rightarrow 9 \rightarrow 4 \rightarrow 10 \rightarrow 13 \rightarrow 16$	0	3.6328125
Graph-Search A*	$0 \rightarrow 2 \rightarrow 1 \rightarrow 9 \rightarrow 4 \rightarrow 10 \rightarrow 13 \rightarrow 16$	0	2.59375
Hill-Climbing	-1	0	0.125

Table 15: Result of Test case 5

6 Experiments

6.1 Comparison of BFS and DFS

Details	**BFS**	**DFS**
Space complexity	May be the whole search space.	Linear space.
Time complexity	Same, but BFS is always better than DFS in worst cases.	Same, but DFS is sometimes better on average (many goals, no loops, no infinite paths).
Memory usage	Less memory efficient than DFS as it has to store nodes of each layer before moving to the next layer.	DFS is memory efficient as it only needs to store the nodes on the path from the source node to the current node.
Optimal	BFS always finds the minimal path from the source node to the destination node.	DFS might not find the shortest path to a given node when there are multiple possible paths from the source node to the destination node.
In general	BFS is better if goal is not deep, if infinite paths, if many loops, if small search space.	DFS is better if many goals, not many loops, and it is much better in terms of memory.

Table 16: BFS vs. DFS [9]

6.2 Usage of BFS, DFS and IDS

- **BFS:**
  - When space is not an issue.
  - When we do care/want the closet answer to the root.
- **DFS:**
  - When you do not care if the answer is closet to the starting vertex/root.
  - When graph/tree is not very big/infinite.
- **IDS:**
  - When you want BFS, you do not have enough memory, and somewhat slower performance is accepted.
  - When you want both BFS and DFS.

6.3 Comparison of UCS and A*

- UCS and A* are effective for finding optimal paths but can be more memory and time-intensive.
- UCS is a special case of A*.
- UCS uses the evaluation function $f(n) = g(n)$, where $g(n)$ is the length of the path from the starting node to n, whereas A* uses the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ means the same thing as in UCS and $h(n)$, called the "heuristic" function, is an estimate of the distance from n to the goal node. In the A* algorithm, $h(n)$ must be admissible.
- UCS is a special case of A* which corresponds to having $h(n) = 0, \forall n$. A heuristic function h which has $h(n) = 0, \forall n$ is clearly admissible, because it always "underestimates" the distance to the goal, which cannot be smaller than 0, unless you have negative edges (assume that all edges are non-negative). So, indeed, UCS is a special case of A*, and its heuristic function is even admissible. [10]

6.4 GBFS

- GBFS can be fast but may not always find the most optimal path.

6.5 Hill Climbing

- Hill Climbing often fails to find a path due to getting stuck in local optima.

References

[1] Tanishka Dhondge. Depth First Iterative Deepening (DFID) Algorithm in Python. https://www.askpython.com/python/examples/depth-first-iterative-deepening-dfid. Accessed: 12.07.2024.

[2] David Dragon. Depth-First Search, without Recursion. https://david9dragon9.medium.com/depth-first-search-without-recursion-b8827065d2b6. Accessed: 12.07.2024.

[3] GeeksforGeeks. Breadth First Search or BFS for a Graph. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. Accessed: 12.07.2024.

[4] GeeksforGeeks. Depth First Search or DFS for a Graph. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/. Accessed: 12.07.2024.

[5] GeeksforGeeks. Greedy Best first search algorithm. https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/. Accessed: 12.07.2024.

[6] GeeksforGeeks. Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS). https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/. Accessed: 12.07.2024.

[7] GeeksforGeeks. Uniform-Cost Search (Dijkstra for large Graphs). https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/. Accessed: 12.07.2024.

[8] Javatpoint. Hill Climbing Algorithm in Artificial Intelligence. https://www.javatpoint.com/hill-climbing-algorithm-in-ai. Accessed: 12.07.2024.

[9] Vridhi Kamath. Iterative Deepening Search. https://iq.opengenus.org/iterative-deepening-search/. Accessed: 12.07.2024.

[10] nbro. How do I show that uniform-cost search is a special case of A*? https://ai.stackexchange.com/questions/9182/how-do-i-show-that-uniform-cost-search-is-a-special-case-of-a. Accessed: 12.07.2024.

[11] Python Pool. The Insider's Guide to A* Algorithm in Python. https://www.pythonpool.com/a-star-algorithm-python/. Accessed: 12.07.2024.

[12] NISHANT TIWARI. Understanding the Greedy Best-First Search (GBFS) Algorithm in Python. https://www.analyticsvidhya.com/blog/2024/06/understanding-the-greedy-best-first-search-gbfs-algorithm-in-python/. Accessed: 12.07.2024.

[13] Wikipedia. A* search algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm. Accessed: 12.07.2024.