VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report
Lab 01: Search Algorithms Assignment

Course name: Fundamentals of Artificial Intelligence

CSC14003_22CLC09

Students:
Nguyen Ho Dang Duy
22127085
22CLC09

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

Ho Chi Minh city, June 2024

Contents

1 Information page

- Full name: Nguyen Ho Dang Duy
- Student ID: 22127085
- Class: 22CLC09
- Link to source code on GitHub: Lab01-Search_algorithms
- Link to screen recording for algorithm: Drive 22127085_Sorting

2 Completion Level of Each Requirement

2.1 Implementation of 5 core algorithms

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

2.2 Implementation of additional algorithms

- Depth-Limited Search (DLS): 10
- Iterative Deepening Search (IDS): 10
- Bidirectional Search: 10

3 Test cases

The red node is the starting node, the green node is the ending node.

Input 01

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

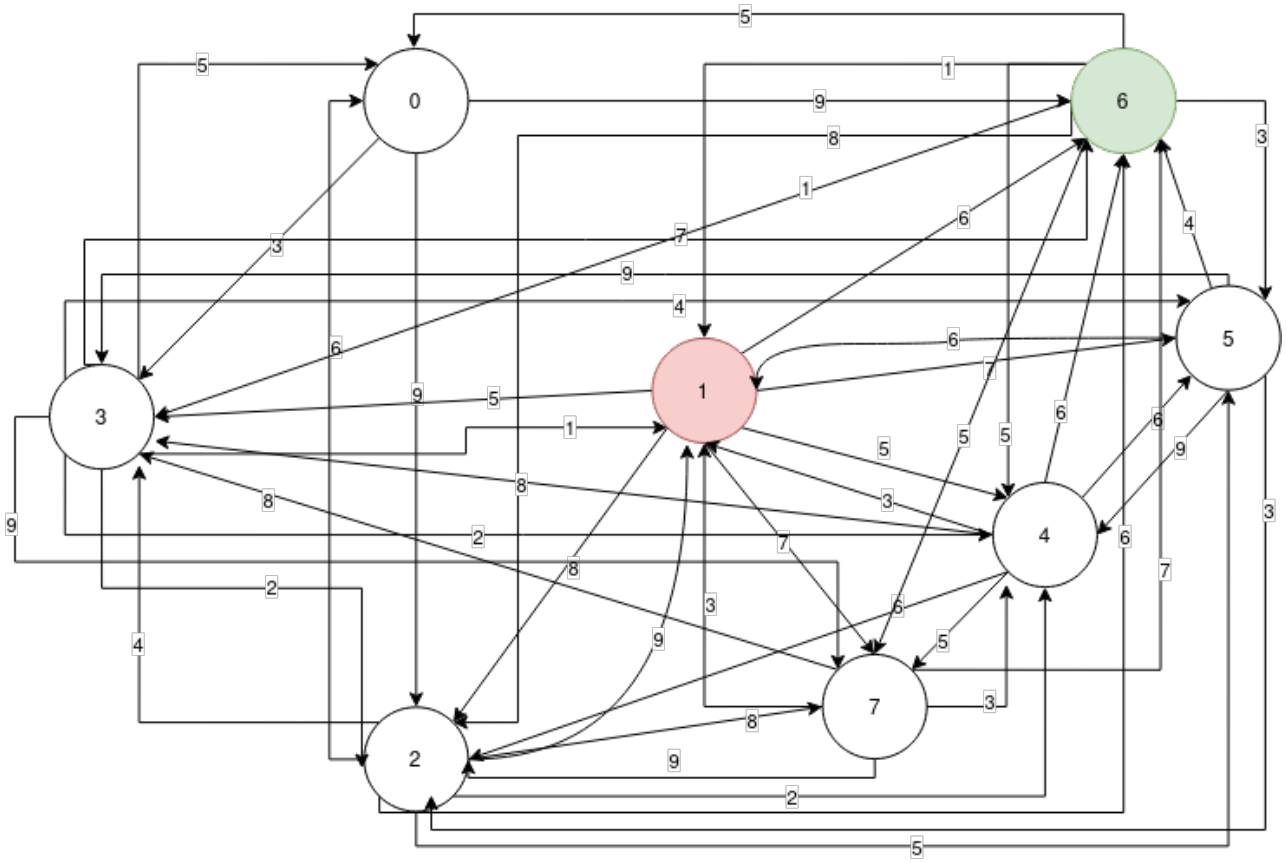

Figure 1: Graph for input 01

Input 02

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

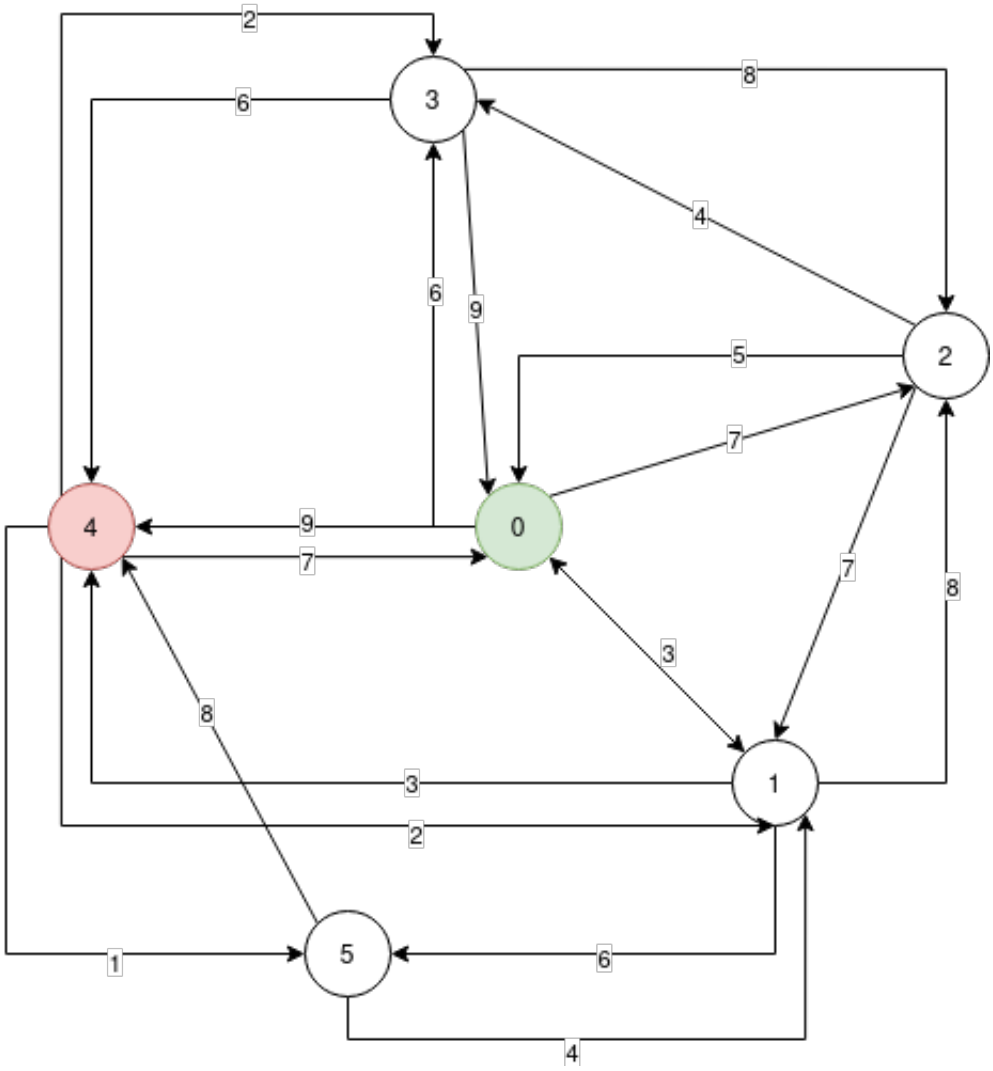

Figure 2: Graph for input 02

Input 03

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

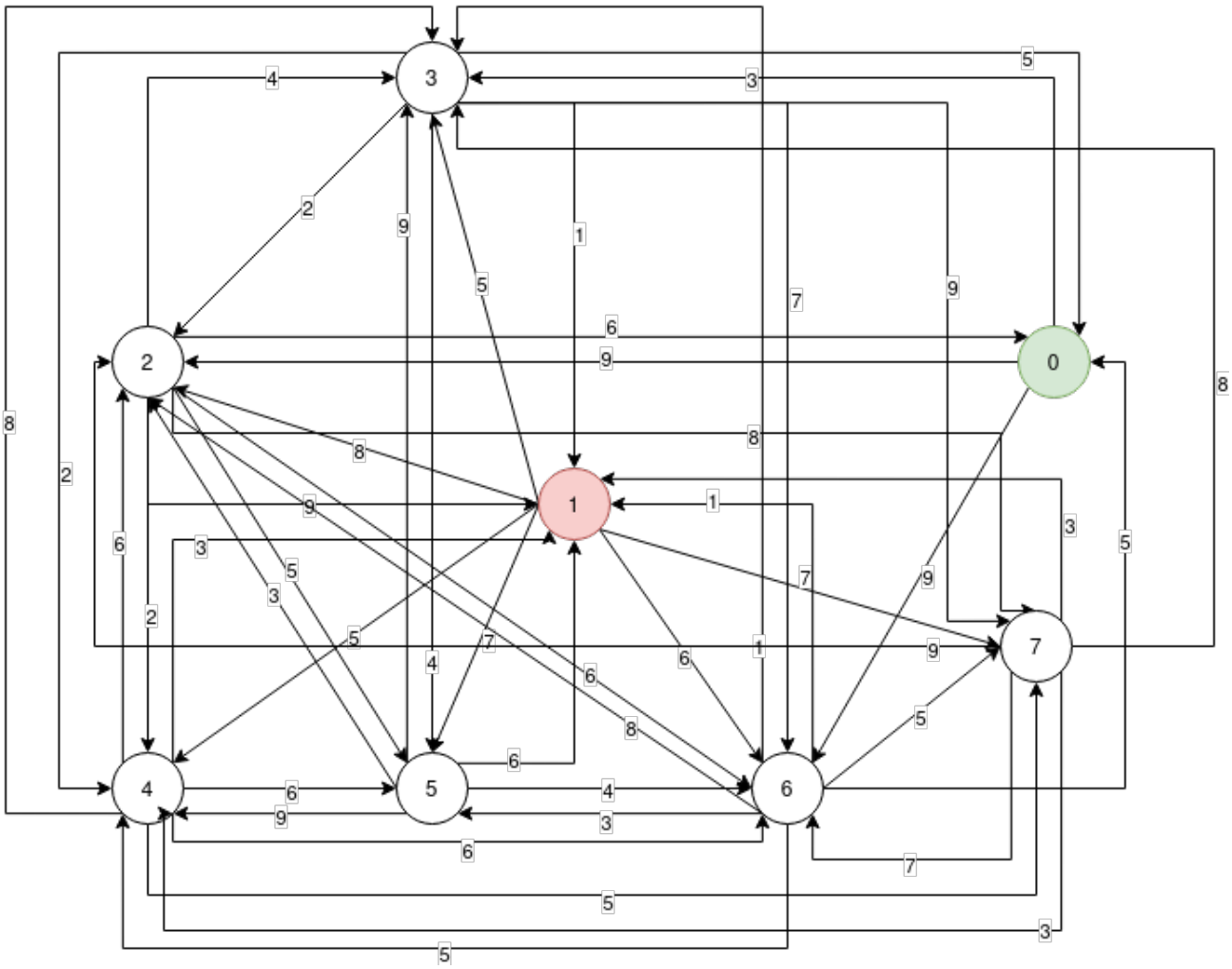

Figure 3: Graph for input 03

Input 04

```
0 8
0 2 1 0 0 1 0 0 0
0 0 1 0 0 0 2 0 0
0 0 0 2 2 1 0 5 0
0 0 0 0 1 0 0 0 2
0 0 0 0 0 0 0 2 2
0 0 0 3 0 0 0 0 0
0 0 0 0 0 0 0 2 1
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

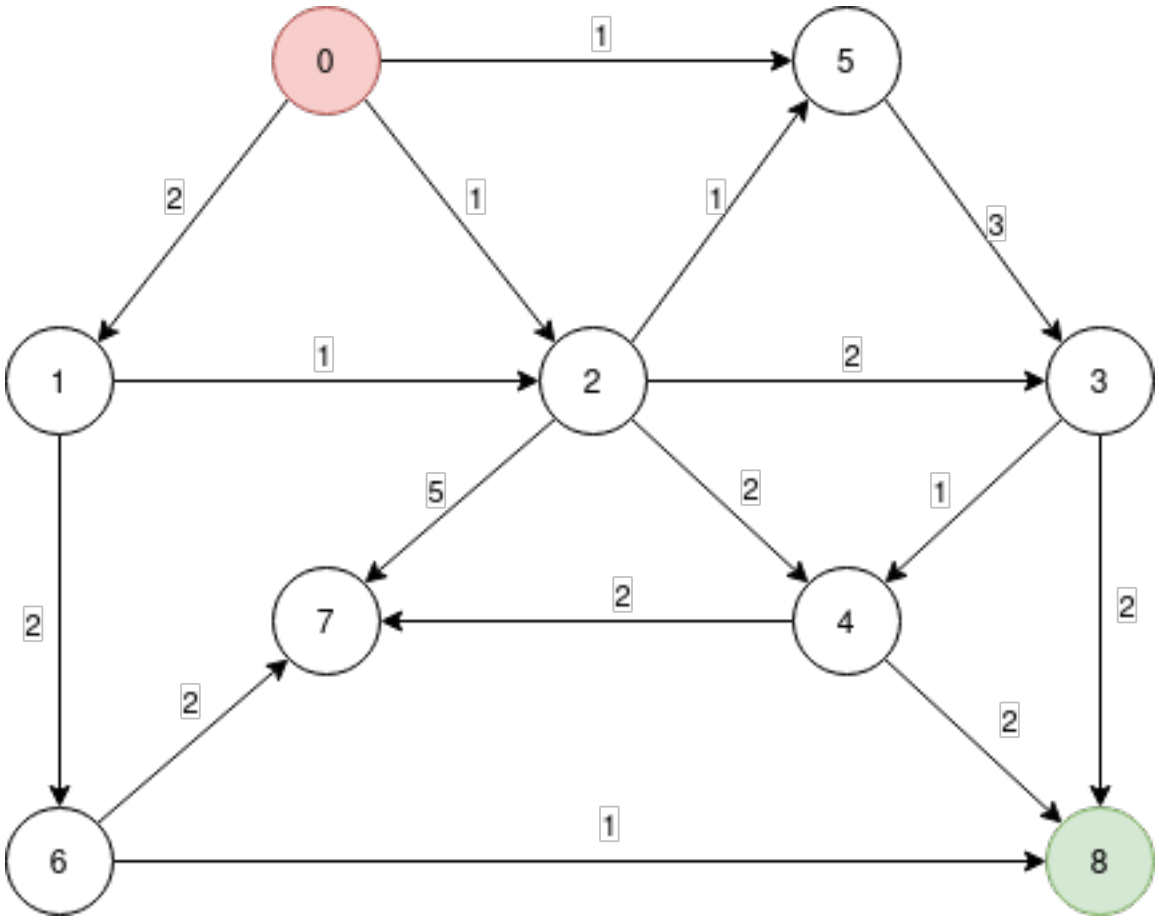

Figure 4: Graph for input 04

Input 05

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

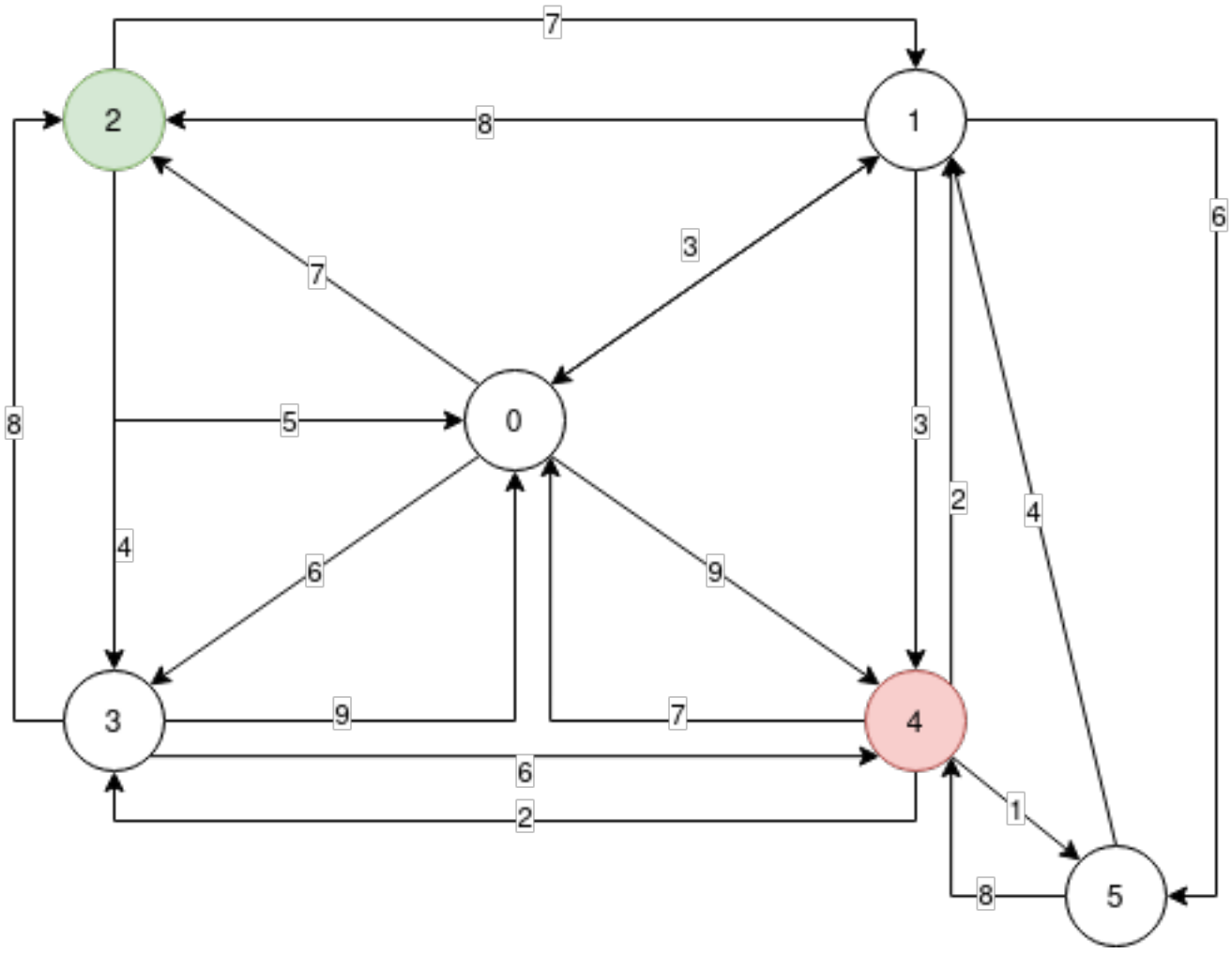

Figure 5: Graph for input 05

4 BFS algorithm

4.1 Concepts

Breath First Search (BFS) is an algorithm for traversing or searching tree or graph data structures that explores all the vertices in a tree or graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. [1]

4.2 Complexity

- **Time Complexity:** O(V+E) where V is the number of vertices and E is the number of edges in the graph. This complexity arises because each vertex and each edge will be explored in the worst case.
- **Space Complexity:** O(V) where V is the number of vertices. This space is required for the queue that stores the vertices that need to be explored.

4.3 Properties

- **Completeness:** BFS is complete, meaning it will always find a solution if one exists.
- **Optimality:** BFS is optimal if all edge weights are equal because it will always find the shortest path in terms of the number of edges.
- **Memory Usage:** BFS can use a lot of memory, especially for wide graphs, because it stores all nodes at the present level before moving on to the next level.

4.4 Implemented BFS Description

4.4.1 Path Construction Function

```
Function construct_path(visited, end)
    Initialize path as an empty list
    Set current to end node

    While current is not None
        Insert current at the beginning of path
        Set current to visited[current]

    Return visited and path
End Function
```

This function takes a **visited** dictionary and an **end** node to construct the path from the start node to the end node. The **visited** dictionary holds each node as a key and the node from which it was visited as the value. Starting from the **end** node, it backtracks through the **visited** nodes until it reaches the start node, thus constructing the path.

4.4.2 BFS Algorithm

```
Function BFS(matrix, start, end)
    Initialize frontier as a queue
    Enqueue start node into frontier
    Initialize visited dictionary with start node as key and None as value

    While frontier is not empty
        Dequeue current node from frontier

        If current node is equal to end node
            Call construct_path(visited, end)
            Return visited and path

        For each neighbor in matrix[current]
            If matrix[current][neighbor] is 1 (indicating an edge) And neighbor
   is not in visited
                Set visited[neighbor] to current
                Enqueue neighbor into frontier

    Return visited and an empty path
End Function
```

This BFS implementation begins by initializing the **frontier** with the **start** node and marking it as visited. It then enters a loop where it explores each node's neighbors. If it finds the **end** node, it constructs and returns the path using the **construct_path** function. If a neighbor node is found and hasn't been visited, it is added to the **frontier** and marked as visited. The loop continues until all nodes in the **frontier** have been explored.

4.5 Conclusion

The BFS algorithm is effective for finding the shortest path in unweighted graphs. The path construction function efficiently backtracks from the destination node to the start node to produce the final path. The time and space complexities are crucial considerations, particularly for large graphs, where BFS's space complexity can become a limiting factor due to the breadth-first nature of the search.

5 DFS algorithm

5.1 Concepts

Depth-First Search (DFS) is a fundamental algorithm used to traverse or search through tree or graph data structures. The algorithm starts at a given node (referred to as the 'root' in tree structures) and explores as far as possible along each branch before backtracking. This method ensures that all nodes are visited by diving deep into each branch of the graph.level of neighbors. [2]

5.2 Complexity

- **Time Complexity:** O(V + E), where V is the number of vertices and E is the number of edges in the graph. Each vertex and edge is explored once in the worst case.
- **Space Complexity:** O(V), where V is the number of vertices. This space is required to store the recursion stack (for recursive implementation) or the stack used to keep track of nodes to visit next (for iterative implementation).

5.3 Properties

- **Completeness:** DFS is not guaranteed to be complete in infinite graphs; it might get stuck going down one infinite path.
- **Optimality:** DFS is not guaranteed to find the shortest path in an unweighted graph. It will find a path, but it might not be the shortest.
- **Memory Usage:** DFS generally requires less memory than BFS. The memory usage is proportional to the depth of the deepest path from the start node.

5.4 Implemented DFS Description

```
Function DFS(matrix, start, end)
    Initialize frontier as a stack with (start, None)
    Initialize visited dictionary as empty
    Initialize path list as empty

    While frontier is not empty
        Pop current node and its predecessor from frontier
        Set visited[current] to predecessor

        While path is not empty And last element of path is not visited[current]
            Remove last element from path
        Append current to path

        If current is equal to end node
            Print visited and path
            Return visited and path

        For each neighbor from last to first in matrix[current]
```

```
            If matrix[current][neighbor] is not 0 And neighbor is not in visited
                Append (neighbor, current) to frontier

    Print visited
    Return visited and path
End Function
```

To solve this problem, I've used non-recursion DFS [3]. This implementation of DFS uses an iterative approach with an explicit stack (**frontier**) to manage the nodes to be explored. The **visited** dictionary keeps track of each node and its predecessor, enabling the reconstruction of the path from the **start** node to the **end** node.

5.5 Conclusion

The DFS algorithm is effective for traversing or searching deep into a graph. The iterative implementation using an explicit stack ensures controlled and manageable space complexity. While not always optimal for finding the shortest path, DFS is invaluable for applications requiring exhaustive exploration of graph branches and is especially memory efficient compared to BFS for deep graphs.

6 UCS algorithm

6.1 Concepts

Uniform-Cost Search (UCS) is a search algorithm used to traverse or search through a weighted graph. It is a variant of **Dijkstra's algorithm** and is used to find the lowest cost path between a start node and a goal node. UCS explores nodes in order of their cumulative cost from the start node.

6.2 Complexity

- **Time Complexity:** O((V + E) log V), where V is the number of vertices and E is the number of edges. This complexity arises due to the priority queue operations required for efficiently selecting the next node to explore.[4]
- **Space Complexity:** O(V), where V is the number of vertices. This space is used for the priority queue and storage of cost and visited information.

6.3 Properties

- **Completeness:** UCS is complete, meaning it will always find a solution if one exists, given that edge costs are non-negative
- **Optimality:** UCS is optimal; it finds the lowest cost path if all edge costs are non-negative.
- **Memory Usage:** UCS requires significant memory to store the priority queue and track the costs and visited nodes, making it less efficient for very large graphs.

6.4 Implemented UCS Description

```
Function UCS(matrix, start, end)
    Initialize path as an empty list
    Initialize visited dictionary with start node as key and None as value
    Initialize priority queue pq
    Put (0, start) into pq
    Initialize cost dictionary with start node as key and 0 as value

    While pq is not empty
        Get node with lowest cost from pq
        Set current_cost and node

        If node is equal to end node
            Break from loop

        For each neighbor and weight in matrix[node]
            If weight is not 0
                Calculate new_cost as current_cost + weight
                If neighbor is not in cost or new_cost < cost[neighbor]
                    Set cost[neighbor] to new_cost
                    Put (new_cost, neighbor) into pq
                    Set visited[neighbor] to node
```

```
22
23      If end node is in visited
24          Set node to end
25          While node is not None
26              Insert node at beginning of path
27              Set node to visited[node]
28
29      Return visited and path
30  End Function
```

This UCS implementation uses a priority queue to manage the frontier of nodes to be explored based on their cumulative cost from the start node. The **visited** dictionary tracks the nodes and their predecessors, enabling path reconstruction once the end node is reached.

6.5 Conclusion

The UCS algorithm is efficient for finding the least cost path in weighted graphs with non-negative edge weights. It guarantees an optimal solution and is complete. However, it requires significant memory for large graphs due to the storage needs of the priority queue and cost information. UCS is particularly useful in scenarios where the cost of traversing edges varies and finding the least expensive path is crucial.

7 GBFS algorithm

7.1 Concepts

- **Greedy Best First Search (GBFS)** is an informed search algorithm where the evaluation function is strictly equal to the heuristic function, disregarding the edge weights in a weighted graph because only the heuristic value is considered. In order to search for a goal node it expands the node that is closest to the goal as determined by the heuristic function. [5]
- **Heuristic Function** h in this context is defined as the edge weight between nodes in the graph. It estimates the cost from the current node to the goal node based on direct edge weights. GBFS selects nodes for expansion based solely on this estimated cost, prioritizing nodes that are believed to be closer to the goal.

7.2 Complexity

- **Time Complexity:** O(V + E) in the best case and O(V$\hat{2}$) in the worst case where V is the number of vertices and E is the number of edges. The actual performance depends heavily on the heuristic.
- **Space Complexity:** O(V) due to the storage of the priority queue and visited nodes.

7.3 Properties

- **Completeness:** GBFS is complete in finite state spaces but may not be complete in infinite state spaces due to the potential for encountering loops or cycles.
- **Optimality:** GBFS is not generally optimal because it does not consider the total path cost but relies solely on the heuristic function to guide the search.
- **Heuristic-Driven:** The performance and results of GBFS depend heavily on the quality of the heuristic used. In this case, using edge weights as the heuristic provides a straightforward estimate of path cost.

7.4 Implemented GBFS Description

```
Function GBFS(matrix, start, end)
    Initialize path as an empty list
    Initialize visited as a dictionary with start node as key and None as value
    Initialize pq as a priority queue
    pq.enqueue((0, start))  // Initialize with start node and heuristic value 0

    while pq is not empty
        current_heuristic, node = pq.dequeue()

        if node = end
            Break out of the loop

        for each neighbor, weight in matrix[node]
            if weight is not 0 and neighbor not in visited
```

```
                // Use edge weight as heuristic in this context
                heuristic_value = weight
                pq.enqueue((heuristic_value, neighbor))
                visited[neighbor] = node

    if end in visited
        // Reconstruct path from end node to start node using visited dictionary
        node = end
        while node is not None
            path.insert(0, node)
            node = visited[node]

    Return visited, path
End Function
```

This GBFS implementation uses a priority queue to manage the frontier of nodes based on their heuristic values, which in this case are the edge weights. The **visited** dictionary tracks the nodes and their predecessors, enabling path reconstruction once the end node is reached.

7.5 Conclusion

Greedy Best First Search (GBFS) is a heuristic-driven algorithm that prioritizes nodes based on their estimated cost to the goal. While it can be faster than exhaustive search methods, it is not guaranteed to find the shortest path or even any path if one exists. The choice of heuristic greatly affects its performance and outcome. In this case, using edge weights as the heuristic provides a straightforward estimation of path cost, but it does not ensure optimality or completeness.

8 A* algorithm

8.1 Concepts

A* Search is an informed search algorithm that combines the benefits of both Breadth-First Search (BFS) and Greedy Best-First Search (GBFS). It uses a heuristic function to guide the search towards the goal node while also considering the actual cost incurred from the start node to the current node. [5] **Heuristic Function** h estimates the cost from the current node to the goal node. The heuristic function hh used in this implementation calculates the Euclidean distance between nodes based on their positions in a 2D space.

8.2 Complexity

- **Time Complexity:** In the best case, where the heuristic is perfect and guides the search efficiently, the time complexity is close to O(d), where d is the shortest path length. In the worst case, it can be exponential due to the nature of exploring multiple paths.
- **Space Complexity:** O(V) due to the priority queue and visited nodes storage.

8.3 Properties

- **Completeness:** A* is complete in finite graphs without infinite costs, meaning it will always find a solution if one exists, given enough time and memory.
- **Optimality:** When the heuristic function is admissible (never overestimates the true cost) and consistent (satisfies the triangle inequality), A* guarantees finding the shortest path from the start node to the goal node.
- **Efficiency:** A* Search efficiently explores paths by prioritizing nodes with lower combined costs (current cost from start + heuristic estimate to goal), ensuring it considers promising paths early in the search.
- **Heuristic Influence:** The effectiveness of A* heavily relies on the quality of the heuristic function. In the provided implementation, the Euclidean distance between nodes' positions serves as a heuristic, providing an estimate of direct distance which guides the search towards the goal.

8.4 Implemented A* Description

```
Function Astar(matrix, start, end, pos)
    Initialize path as an empty list
    Initialize visited as a dictionary with start node as key and None as value
    Initialize pq as a priority queue with (0, start)  // Start with start node
   and heuristic value 0
    Initialize cost as a dictionary with start node and cost 0

    while pq is not empty
        current_cost, node = pq.dequeue()

```

```
        if node == end
            Break out of the loop

        for each neighbor, weight in matrix[node]
            if weight is not 0
                new_cost = cost[node] + weight
                if neighbor not in cost or new_cost < cost[neighbor]
                    cost[neighbor] = new_cost
                    priority = new_cost + heuristic(neighbor, end, pos)
                    pq.enqueue((priority, neighbor))
                    visited[neighbor] = node

        // Sorting the priority queue for visualization purposes
        frontier = sort pq.queue by priority

    if end in visited
        // Reconstruct path from end node to start node using visited dictionary
        node = end
        while node is not None
            path.insert(0, node)
            node = visited[node]

    Return visited, path
End Function
```

This A* Search implementation uses a priority queue (**pq**) to manage nodes based on their cumulative cost from the start node (cost) plus the heuristic estimate (heuristic). Nodes with lower combined costs are explored first, aiming to reach the goal node efficiently.

8.5 Conclusion

A* Search algorithm remains a cornerstone in pathfinding algorithms due to its balance between completeness, optimality, and efficiency. By intelligently combining the cost to reach a node and a heuristic estimate of the remaining cost to the goal, A* continues to be instrumental in various domains where efficient route planning is essential. Its adaptability to different heuristic functions allows it to be tailored to specific problem domains, making it a versatile tool in computational intelligence and problem-solving applications.

9 Comparison of UCS and A* Algorithms:

Uniform Cost Search (UCS) and A* Search are both fundamental algorithms in the field of pathfinding, each offering distinct advantages based on their approach to exploring paths in graphs. This comparison highlights their differences in terms of strategy, efficiency, optimality, and practical applications.

Features	Uniform Cost Search (UCS)	A* Search
Approach	Explores nodes in order of their cumulative path cost from the start node, expanding the least-cost path first	Combines path cost and heuristic value to evaluate nodes based on the sum of the path cost and heuristic estimate.
Strategy	Guarantees finding the shortest path in terms of path cost when all edge weights are non-negative.	Focuses on more promising paths guided by the heuristic function.
Optimality	Optimal when all edge costs are positive.	Optimal if the heuristic function is admissible and consistent.
Completeness	Complete and will find a solution if one exists in finite graphs.	Complete and will find a solution if one exists, given an admissible and consistent heuristic.
Efficiency	Can become inefficient in graphs with varying edge costs or complex topologies.	More efficient in practice due to heuristic guidance, suitable for large and complex graphs.
Memory Usage	Requires less memory, storing path costs and explored nodes.	Requires additional memory for storing the priority queue and visited nodes, more memory-intensive in larger graphs.
Heuristic Function	Does not use a heuristic function. We can view UCS as a function f(n) = g(n) where g(n) is a path cost. We can call the UCS algorithm is the special case of the A* algorithm where h(n) is equal to 0 [8]	Relies on a heuristic function to estimate the cost from the current node to the goal; quality of the heuristic impacts efficiency and optimality. The total cost function for A* is f(n) = g(n) + h(n)
Applications	Suitable for scenarios with uniform or unknown edge costs, such as basic pathfinding or unweighted graphs.	Ideal for real-time applications like navigation systems, robotics, and game AI, where the heuristic function can accurately estimate the cost to the goal.

Table 1: Comparison of UCS and A* Algorithms

10 DLS algorithm

10.1 Concepts

Depth-Limited Search (DLS) is a variant of Depth-First Search (DFS) that limits the depth of the search to a predetermined limit. This method helps prevent the algorithm from diving too deep into the search space, which can be beneficial in infinite or excessively large graphs. This report provides an overview of the DLS algorithm, its concepts, properties, complexity, and applications. To run DLS in this project, use this command line argument:

```
    python main.py <input_file_path> dls <time_delay>(optional)
```

10.2 Complexity

- **Time Complexity:** O(b^l), where b is the branching factor (the average number of children per node) and l is the depth limit.
- **Space Complexity:** O(b*l), which is linear in terms of the depth limit and branching factor. This is due to the nature of depth-first search, which only needs to store the nodes on the current path.

10.3 Properties

- **Completeness:** DLS is not complete if the depth limit is less than the depth of the shallowest goal node. It will fail to find a solution if the goal is beyond the depth limit.
- **Optimality:** DLS is not optimal. It does not guarantee finding the least-cost path to a goal if there are multiple paths.
- **Space Efficiency:** DLS is more space-efficient compared to breadth-first search (BFS) as it requires less memory, only needing to store the current path up to the depth limit.

10.4 Implemented DLS Description

```
Function DLS(matrix, start, end, limit):
    // Initialize the path and visited dictionary
    path = []
    visited = {start: None}

    // Recursive function for depth-limited search
    function recursive_dls(node, depth):
        if depth > limit:
            return False
        if node == end:
            return True
        for neighbor in range(len(matrix[node])):
            if matrix[node][neighbor] != 0 and neighbor not in visited:
                visited[neighbor] = node
                if recursive_dls(neighbor, depth + 1):
                    path.insert(0, neighbor)
```

```
                    return True
                // If not successful, backtrack
                visited.pop(neighbor)
        return False

    // Start the recursive depth-limited search
    print("Starting DLS with limit", limit)
    if recursive_dls(start, 0):
        path.insert(0, start)
    else:
        print("No path found within depth limit.")

    print("path:", path)
    print("visited:", visited)
    return visited, path
```

10.5 Conclusion

Depth-Limited Search (DLS) is an effective algorithm for problems where the search depth can be restricted. While it is not guaranteed to be complete or optimal, its space efficiency and ability to limit exploration to a specific depth make it useful in various practical applications. Understanding its properties and limitations is crucial for effectively applying DLS to appropriate search problems.

11 IDS algorithm

11.1 Concepts

Iterative Deepening Search (IDS) is a powerful search algorithm that combines the depth-wise exploration of Depth-First Search (DFS) with the level-wise completeness of Breadth-First Search (BFS). By using Depth-Limited Search (DLS) iteratively with increasing depth limits, IDS guarantees completeness and optimality while maintaining space efficiency.
To run IDS in this project, use this command line argument

```
    python main.py <input_file_path> ids <time_delay>(optional)
```

11.2 Complexity

- **Time Complexity:** O(b^d), where b is the branching factor and d is the depth of the goal node
- **Space Complexity:** O(d), where d is the depth of the goal node

11.3 Properties

- **Completeness:** IDS is complete; it guarantees finding the goal node if it exists within the search space.
- **Optimality:** IDS is optimal; it always finds the shallowest (or least costly) goal node first.

11.4 Implemented IDS Description

```
Function DLS_for_IDS(matrix, node, goal, limit, visited):
    if node is equal to goal:
        return True
    if limit <= 0:
        return False
    for each neighbor in range(len(matrix[node])):
        if matrix[node][neighbor] != 0 and neighbor not in visited:
            visited[neighbor] = node
            if DLS(matrix, neighbor, goal, limit - 1, visited):
                return True
            visited.pop(neighbor)  # Backtrack
    return False

Function IDS(matrix, start, goal):
    maxDepth = len(matrix)
    for depth from 0 to maxDepth:
        visited = {start: None}
        if DLS_for_IDS(matrix, start, goal, depth, visited):
            path = []
            node = goal
            while node is not None:
                path.insert(0, node)
                node = visited[node]
```

```
            return visited, path
    return {}, []
```

While the DLS function performs a depth-limited search in a general context, DLS_for_IDS is tailored specifically for use within the Iterative Deepening Search (IDS) algorithm. It manages depth limits, frontier nodes, and state management in a way that aligns with the iterative nature of IDS, ensuring efficient and effective exploration of nodes up to increasing depth levels. This approach enhances algorithmic clarity, modularity, and adherence to the requirements of IDS for exploring large and unknown search spaces.
The IDS algorithm repeatedly applies DLS with increasing depth limits until the goal node is found. This process ensures that all nodes are explored in a depth-first manner up to each depth limit, ultimately covering the entire search space in a manner similar to BFS.

11.5 Conclusion

Iterative Deepening Search (IDS) is a robust algorithm that effectively balances the depth-wise exploration of Depth-First Search (DFS) with the level-wise completeness of Breadth-First Search (BFS). By using Depth-Limited Search (DLS) iteratively, IDS ensures both completeness and optimality while maintaining space efficiency. Its ability to handle large and unknown search spaces makes it a valuable tool in various graph traversal and search applications.

12 Bidirectional Search

12.1 Concepts

Bidirectional Search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. It runs two simultaneous searches: one forward from the initial state, and one backward from the goal, stopping when the two meet. This approach is particularly effective in reducing the search space compared to traditional BFS or DFS, especially in large graphs. [12]
To run Bidirectional Search in this project, use this command line argument

```
python main.py <input_file_path> bidirect <time_delay>(optional)
```

12.2 Complexity

- **Time Complexity:** The time complexity of Bidirectional Search is typically less than traditional BFS or DFS. In the worst case, the complexity can be expressed as $O(b^{d/2})$ where b is the branching factor of the graph and d is the distance between the start and end nodes. [11]
- **Space Complexity**: $O(b^{d/2})$

12.3 Properties

- **Completeness:** If a path exists between the start and end nodes, Bidirectional Search will find it, making it a complete algorithm.
- **Optimality:** Given the nature of BFS, the path found by Bidirectional Search is optimal in terms of the number of edges.
- **Space Complexity:** The algorithm requires space for maintaining two frontiers and two visited dictionaries, which is generally manageable unless the graph is extremely large.
- **Efficiency:** Bidirectional Search is more efficient than traditional BFS or DFS for finding shortest paths between two nodes.

12.4 Implemented Bidirection Search Description

12.4.1 Path Construction Function

```
Function construct_path(src_visited, dest_visited, intersecting_node):
    Initialize path as empty list

    # From source to intersection
    current = intersecting_node
    While current is not None:
        Append current to path
        current = src_visited[current]
    Reverse path

    # From intersection to destination
```

```
    current = dest_visited[intersecting_node]
    While current is not None:
        Append current to path
        current = dest_visited[current]

    Return path
```

This helper function constructs the path from the **start** node to the **end** node using the intersecting node where the two searches met. Main idea:

- Build the path by traversing back from the intersecting node to the start node using **src_visited**.
- Then traverse from the intersecting node to the end node using **dest_visited**.
- Combine these two parts to form the complete path.

12.4.2 Format visited dictionaries

```
Function format_visited(src_visited, dest_visited):
    Merge src_visited and dest_visited
    Return merged result
```

This helper function combines the visited dictionaries from both the source and destination searches into a single string representation.

12.4.3 Bidirectional Search Algorithm

```
Function bidirectional_search(matrix, start, end):
    If start == end:
        Return format "start: None\nend: None", [start]

    Initialize src_queue with start
    Initialize dest_queue with end
    Initialize src_visited with {start: None}
    Initialize dest_visited with {end: None}

    While src_queue and dest_queue are not empty:
        Perform BFS step from src_queue
        If intersection found:
            visited = format_visited(src_visited, dest_visited)
            path = construct_path(src_visited, dest_visited, intersection)
            Return visited, path

        Perform BFS step from dest_queue
        If intersection found:
            visited = format_visited(src_visited, dest_visited)
            path = construct_path(src_visited, dest_visited, intersection)
            Return visited, path

    Return format_visited(src_visited, dest_visited), []
```

This is the main function implementing the Bidirectional Search algorithm. It initializes the search and coordinates the bidirectional traversal until the path is found or all nodes are exhausted. Main idea:

- Initialize two search frontiers: **src_queue** starting from **start** and **dest_queue** starting from **end**.
- Maintain two visited dictionaries: **src_visited** for the source-side search and **dest_visited** for the destination-side search.
- Perform BFS alternately from both the source and the destination until the search frontiers intersect or both are exhausted.
- If the search frontiers meet, construct the path using the intersecting node.

12.5 Conclusion

Bidirectional Search is a powerful technique for finding shortest paths in graphs by simultaneously exploring from both the start and end nodes. It offers significant improvements in efficiency over traditional BFS, making it particularly suitable for large-scale graph traversal problems where performance is crucial. By leveraging two search fronts and merging results at the intersection, Bidirectional Search strikes a balance between completeness and optimality in pathfinding algorithms.

References

[1] Improve, G. (2012, March 20). Breadth first search or BFS for a graph. GeeksforGeeks. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[2] Improve, G. (2012a, March 15). Depth first search or DFS for a graph. GeeksforGeeks. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

[3] Dragon, D. (2021, June 8). Depth-First Search, without recursion - David dragon. Medium. https://david9dragon9.medium.com/depth-first-search-without-recursion-b8827065d2b6

[4] andrew1234. (2019, March 25). Uniform-cost search (Dijkstra for large graphs). GeeksforGeeks. https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

[5] Sahil_Kumar, Christine_Yang. (n.d.). Greedy Best-First Search. Codecademy. Retrieved June 22, 2024, from https://www.codecademy.com/resources/docs/ai/search-algorithms/greedy-best-first-search

[6] A* Search algorithm. (2016, June 16). GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm/

[7] Wikipedia contributors. (2024, June 6). A* search algorithm. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=A*_search_algorithmoldid=1227493899

[8] Pranjalpanchawate. (2023, September 7). 'Uniform Cost Search (UCS) is special case of A* algorithm. Medium. https://medium.com/@pranjalpanchawate/uniform-cost-search-ucs-is-special-case-of-a-algorithm-ca7f828c62fe

[9] Singh, S. (2022, April 13). Depth Limited Search. OpenGenus IQ: Learn Algorithms, DL, System Design. https://iq.opengenus.org/depth-limited-search/

[10] Improve, G. (2016, May 19). Iterative deepening search(IDS) or iterative deepening depth first search(IDDFS). GeeksforGeeks. https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/

[11] Bidirectional search. (2017, June 11). GeeksforGeeks. https://www.geeksforgeeks.org/bidirectional-search/

[12] Wikipedia contributors. (2023, August 13). Bidirectional search. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Bidirectional_search&oldid=1170094757

Contributors

1. Chat GPT: https://chatgpt.com/
2. GitHub Copilot: https://github.com/features/copilot