UNIVERSITY OF SCIENCE, VNU-HCMC

FALCUTY INFORMATION TECHNOLOGY

CSC14003 - Introduction to Artificial Intelligence

Lab 1

Searching

Instructors: Nguyễn Trần Duy Minh, Nguyễn Ngọc Thảo,
Nguyễn Thanh Tình, Nguyễn Hải Đăng

Class: 22CLC06

Student: 22127210 – Phạm Anh Khôi

Ho Chi Minh City – 2024

Contents

1. Student information

- Student ID: 22127210
- Full name: Phạm Anh Khôi
- Class: 22CLC06

2. Self-evaluation

No.	Details	Completion Rate (%)
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 algorithms with differentattributes. Describe them in the experiment section	100%
9	Report algorithm, experiment with some reflection or comments	100%

3. Detailed algorithm description

In this lab, I will implement the following search algorithms:

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Uniform-cost search (UCS)
- Iterative deepening search (IDS)
- Greedy best-first search (GBFS)
- A* search
- Hill climbing search

The following sections will describe the algorithms in detail.

Class detail:

```
class Graph:
    def __init__(self, num_nodes, start, goal, adjacency_matrix, heuristic_weights):
        self.num_nodes = num_nodes
        self.start = start
        self.goal = goal
        self.adjacency_matrix = adjacency_matrix
        self.heuristic_weights = heuristic_weights
```

3.1. Breadth-first search (BFS)

Breadth-first search is a simple graph traversal algorithm that starts at the root node and explores all theneighboring nodes. Then, it moves to the next level of nodes and explores their neighbors, and so on. The algorithm uses a queue to keep track of the nodes to be explored. The algorithm is implemented as follows:

```
def bfs(self):
    visited = [False] * self.num_nodes
    queue = deque([(self.start, [self.start])])
    while queue:
        current, path = queue.popleft()
        if current == self.goal:
            return path
        for neighbor, weight in enumerate(self.adjacency_matrix[current]):
            if weight > 0 and not visited[neighbor]:
                visited[neighbor] = True
                queue.append((neighbor, path + [neighbor]))
    return []
```

3.1. Depth-first search (DFS)

Depth-first search is a graph traversal algorithm that begins at the root node and explores as deeply as possible along each branch before backtracking. It employs a stack to manage the nodes awaiting exploration. The algorithm is typically implemented as follows:

```
def dfs(self):
    visited = [False] * self.num_nodes
    stack = [(self.start, [self.start])]
    while stack:
        current, path = stack.pop()
        if current == self.goal:
            return path
        for neighbor, weight in enumerate(self.adjacency_matrix[current]):
            if weight > 0 and not visited[neighbor]:
                visited[neighbor] = True
                stack.append((neighbor, path + [neighbor]))
    return []
```

3.3 Uniform-cost search (UCS)

Uniform-cost search is an algorithm similar to Dijkstra's algorithm, designed to find the shortest path from a start node to a goal node. It achieves this by using a priority queue to manage nodes based on their path costs from the start node. The implementation of the algorithm proceeds as follows:

```
def ucs(self):
    priority_queue = [(0, self.start, [self.start])]
    visited = [False] * self.num_nodes
    while priority_queue:
        current_cost, current_node, path = heapq.heappop(priority_queue)
        if current_node == self.goal:
            return path, current_cost
        if not visited[current_node]:
            visited[current_node] = True
            for neighbor, weight in enumerate(self.adjacency_matrix[current_node]):
                if weight > 0:
                    heapq.heappush(priority_queue, (current_cost + weight, neighbor, path + [neighbor]))
    return [], float('inf')
```

3.4. Iterative deepening search (IDS)

Iterative deepening search combines the strategies of depth-first search and breadth-first search by gradually increasing the depth limit of a depth-first search approach. The algorithm is implemented as follows:

```
def ids(self, max_depth=50):
    def dfs_limited(node, path, depth):
        if depth == 0 and node == self.goal:
            return path
        if depth > 0:
            for neighbor, weight in enumerate(self.adjacency_matrix[node]):
                if weight > 0 and neighbor not in path:
                    result = dfs_limited(neighbor, path + [neighbor], depth - 1)
                    if result:
                        return result
        return None

    for depth in range(max_depth):
        result = dfs_limited(self.start, [self.start], depth)
        if result:
            return result
    return []
```

3.5. Greedy best-first search (GBFS)

Greedy best-first search is a graph traversal algorithm that prioritizes nodes based solely on a heuristic function, aiming to explore nodes that appear closest to the goal. It utilizes a priority queue to manage nodes awaiting exploration. The algorithm's implementation typically follows these steps:

```
def gbfs(self):
    priority_queue = [(self.heuristic_weights[self.start], self.start, [self.start])]
    visited = [False] * self.num_nodes
    while priority_queue:
        _, current_node, path = heapq.heappop(priority_queue)
        if current_node == self.goal:
            return path
        if not visited[current_node]:
            visited[current_node] = True
            for neighbor, weight in enumerate(self.adjacency_matrix[current_node]):
                if weight > 0:
                    heapq.heappush(priority_queue, (self.heuristic_weights[neighbor], neighbor, path + [neighbor]))
    return []
```

3.4. A* search

A* search is a graph traversal algorithm that combines the cost to reach a node with an estimated cost to reach the goal node to decide the next node to explore. This approach allows it to find the most efficient path. It employs a priority queue to manage the nodes to be explored. The implementation of the algorithm proceeds as follows:

```
def a_star(self):
    open_set = [(self.heuristic_weights[self.start], 0, self.start, [self.start])]
    heapq.heapify(open_set)
    g_score = [float('inf')] * self.num_nodes
    g_score[self.start] = 0

    while open_set:
        _, current_g, current_node, path = heapq.heappop(open_set)

        if current_node == self.goal:
            return path, g_score[self.goal]

        for neighbor, weight in enumerate(self.adjacency_matrix[current_node]):
            if weight > 0:
                tentative_g_score = current_g + weight
                if tentative_g_score < g_score[neighbor]:
                    g_score[neighbor] = tentative_g_score
                    f_score = tentative_g_score + self.heuristic_weights[neighbor]
                    heapq.heappush(open_set, (f_score, tentative_g_score, neighbor, path + [neighbor]))
    return [], float('inf')
```

3.5. Hill climbing search

Hill climbing search is a local search algorithm that starts at the initial state and moves to the neighboring state with the highest heuristic value. The algorithm is implemented as follows:

```
def hill_climbing(self):
    current = self.start
    path = [current]
    while current != self.goal:
        # Get neighbors of the current node
        neighbors = [(neighbor, self.heuristic_weights[neighbor])
                     for neighbor, weight in enumerate(self.adjacency_matrix[current])
                     if weight > 0]
        if not neighbors:
            return [-1]
        # Select the neighbor with the lowest heuristic value
        next_node = min(neighbors, key=lambda x: x[1])[0]

        if self.heuristic_weights[next_node] >= self.heuristic_weights[current]:
            break
        path.append(next_node)
        current = next_node
    if current != self.goal:
        return [-1]
    return path
```

4. Implementation and experimentation

In this section, I implement the detail of our search algorithms and showcase the results obtained from 5 distinct test cases. The following subsections provide an overview of our approach and the outcomes achieved.

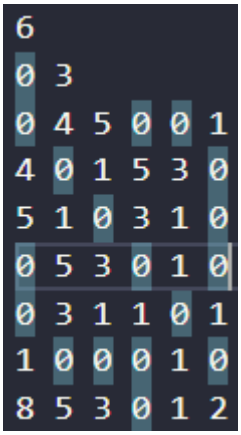
4.1 Test cases

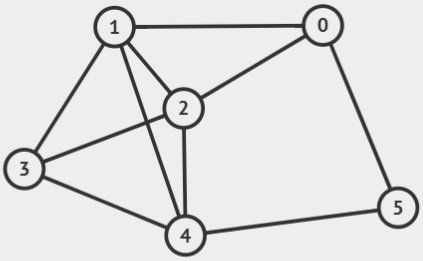
- Graph Representation:
  - The graph is represented using an adjacency matrix.
  - The first line specifies the number of nodes in the graph.
  - The second line contains two integers representing the start and goal nodes.
  - The subsequent lines provide the adjacency matrix, which describes the connections between nodes.
- Heuristic Weights:
  - The last line contains heuristic weights for each node (used by algorithms that incorporate heuristics).
- Visual Representation:
  - I've used Visualgo to generate visual representations of the graphs for your test cases.
- Test Cases and Solutions:
  - I've referenced test cases and solutions to verify the correctness of your algorithm.

4.1.1 Test case 1

Start: 0
Goal: 3

Vertex	1	2	3	4	5	6
Heuristic	8	5	3	0	1	2





• **V=6, E=10** • Tree? **No** • Complete? **No** • Bipartite? **No** • DAG? **N/A** • Cross? **Yes**

Adjacency Matrix

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

Adjacency List

0:	1	2	5	
1:	0	2	3	4
2:	0	1	3	4
3:	1	2	4	
4:	1	2	3	5
5:	0	4		

Edge List

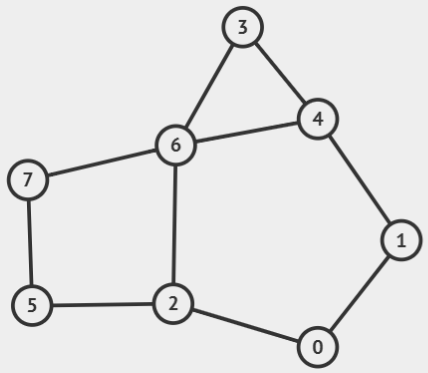
0:	0	1
1:	0	2
2:	0	5
3:	1	2
4:	1	3
5:	1	4
6:	2	3
7:	2	4
8:	3	4
9:	4	5

4.1.2 Test case 2

Start: 0
Goal: 7

Vertex	1	2	3	4	5	6	7	8
Heuristic	6	7	5	2	3	4	1	0

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



• **V=8, E=10** • Tree? **No** • Complete? **No** • Bipartite? **No** • DAG? **N/A** • Cross? **No**

Adjacency Matrix

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

Adjacency List

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

Edge List

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

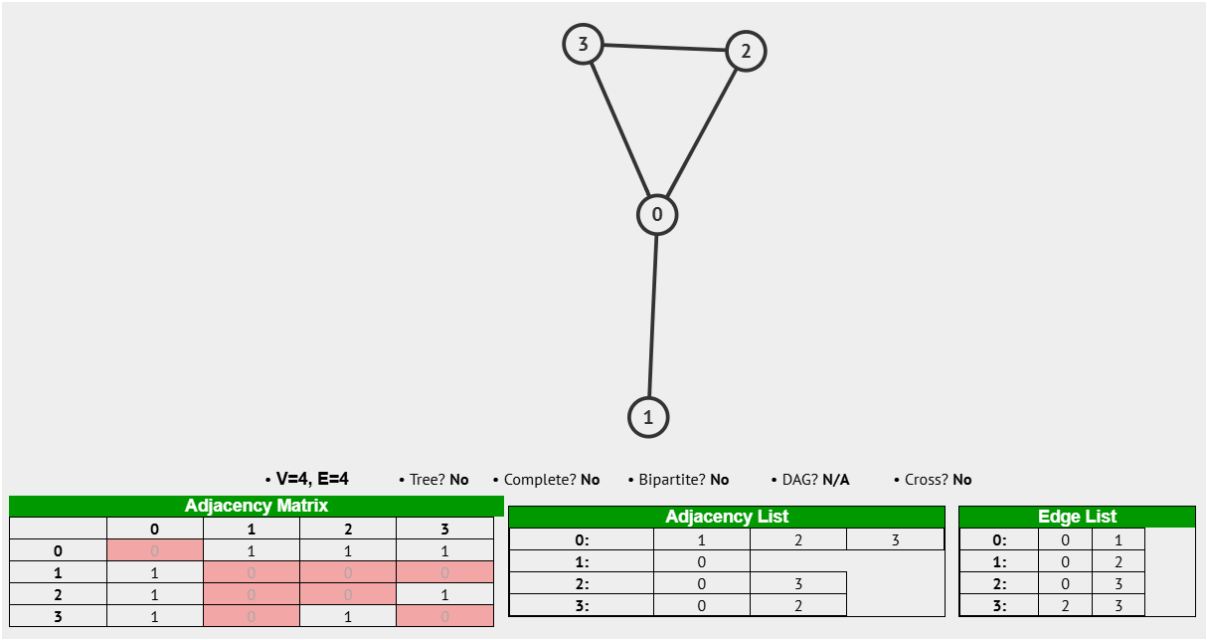
4.1.3 Test case 3

Start: 0
Goal: 3

Vertex	1	2	3	4
Heuristic	10	5	8	12

graph_input_2.txt

```
4
0 3
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
10 5 8 12
```



• **V=4, E=4** • Tree? **No** • Complete? **No** • Bipartite? **No** • DAG? **N/A** • Cross? **No**

Adjacency Matrix

	0	1	2	3
0	0	1	1	1
1	1	0	0	0
2	1	0	0	1
3	1	0	1	0

Adjacency List

0:	1	2	3
1:	0		
2:	0	3	
3:	0	2	

Edge List

0:	0	1
1:	0	2
2:	0	3
3:	2	3

4.1.4 Test case 4

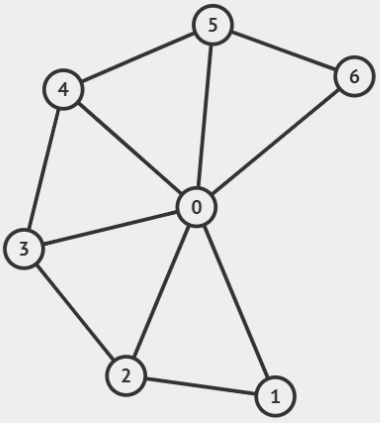
Start: 1
Goal: 6

Vertex	1	2	3	4	5	6	7
Heuristic	1	2	3	4	5	6	7

graph_input_4.txt

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

```



• V=7, E=11 • Tree? No • Complete? No • Bipartite? No • DAG? N/A • Cross? No

Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	1	1	1	1
1	1	0	1	0	0	0	0
2	1	1	0	1	0	0	0
3	1	0	1	0	1	0	0
4	1	0	0	1	0	1	0
5	1	0	0	0	1	0	1
6	1	0	0	0	0	1	0

Adjacency List

0:	1	2	3	4	5	6
1:	0	2				
2:	0	1	3			
3:	0	2	4			
4:	0	3	5			
5:	0	4	6			
6:	0	5				

Edge List

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

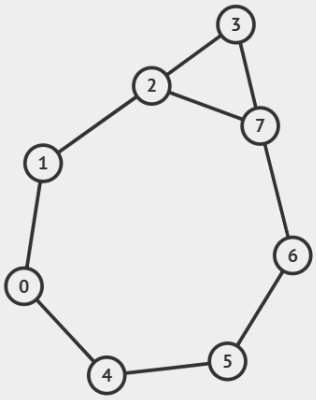
4.1.5 Test case 5

Start: 2
Goal: 6

Vertex	1	2	3	4	5	6	7	8
Heuristic	3	2	5	6	4	1	2	7

graph_input_3.txt

```
1   8
2   2 6
3   0 1 0 0 1 0 0 0
4   1 0 1 0 0 0 0 0
5   0 1 0 1 0 0 0 1
6   0 0 1 0 0 0 0 1
7   1 0 0 0 0 1 0 0
8   0 0 0 0 1 0 1 0
9   0 0 0 0 0 1 0 1
10  0 0 1 1 0 0 1 0
11  3 2 5 6 4 1 2 7
```



• V=8, E=9 • Tree? **No** • Complete? **No** • Bipartite? **No** • DAG? **N/A** • Cross? **No**

Adjacency Matrix

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

Adjacency List

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

Edge List

0:	0	1
1:	0	4
2:	1	2
3:	2	3
4:	2	7
5:	3	7
6:	4	5
7:	5	6
8:	6	7

4.2 Results

4.2.1 Execution time

Test case	BFS	DFS	UCS	IDS	GBFS	A*	HC
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.001	0.0	0.0	0.0	0.0	0.0

Note: Because graphs are small, so execution time is negligible.

4.2.2 Memory usage

The memory usage (KB)

Test Case	BFS	DFS	UCS	IDS	GBFS	A*	Hill Climbing
1	0.08	0.09	0.05	0.08	0.09	0.05	0.12
2	0.09	0.09	0.05	0.09	0.09	0.05	0.12
3	0.07	0.07	0.05	0.07	0.08	0.05	0.06
4	0.08	0.10	0.05	0.08	0.11	0.05	0.06
5	0.08	0.08	0.05	0.08	0.10	0.05	0.06

4.2.3 Discussion

The study reveals that various search algorithms exhibit distinct behaviors concerning execution time, memory usage, and correctness. Here are the key observations:

- **Execution Time:**
  - Due to the small graph size (few nodes and edges), execution times for all algorithms are relatively low.
  - Heuristic-based algorithms (such as GBFS and A*) tend to be faster. They make informed decisions about which nodes to explore next.
  - The hill climbing search algorithm, lacking heuristics, has the highest execution time.
- **Memory Usage:**
  - Again, the small graph size contributes to low memory usage across algorithms.
  - The hill climbing search algorithm has the lowest memory requirements. It only stores the current node and its neighbors.
  - Heuristic-based algorithms (GBFS, A*) consume more memory due to additional information about estimated costs.
  - DFS, IDS, and GBFS store frontier nodes in a stack or queue.
  - UCS and A* algorithms maintain priority queues based on path cost and heuristic value, respectively.
  - BFS, storing all nodes in the frontier queue, has the highest memory usage.
- **Correctness:**
  - All algorithms except hill climbing are complete and optimal.
  - Hill climbing can get stuck in local maxima and fail to reach the goal state.
  - It is also suboptimal since it only considers neighboring nodes.
- **Implementation:**

```
22127210
├── Test_case
│   ├── graph_input_1.txt     // Input data for graph processing (1st dataset)
│   ├── graph_input_2.txt     // Input data for graph processing (2nd dataset)
│   ├── graph_input_3.txt     // Input data for graph processing (3rd dataset)
│   ├── graph_input_4.txt     // Input data for graph processing (4th dataset)
│   ├── graph_input_5.txt     // Input data for graph processing (5th dataset)
│   └── graph_output.txt      // Output data generated from graph processing
├── report                    // Project reports
└── src                       // Source code files
```

- **External libraries:**
  - collections: Used for implementing the queue and stack data structures
  - heapq: Used for implementing the priority queue data structure
  - time: Used for measuring the execution time of the algorithms
  - sys: Used for measuring the memory usage

In practice, the choice of algorithm depends on the specific problem and the desired trade-offs between time and space complexity

5 Conclution

- I implemented and tested various search algorithms, including BFS, DFS, UCS, IDS, GBFS, A*, and hill climbing.
- Detailed algorithm descriptions and implementation specifics were provided.
- Experimental results highlighted differences in execution time and memory usage, emphasizing the need to choose algorithms based on problem specifics and trade-offs.

6 References

- Lecture slides and materials from the course.