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: 22127154 – Nguyễn Gia Huy

Ho Chi Minh City – 2024

Table of contents

1. Student information

- Student ID: 22127154
- Full name: Nguyen Gia Huy
- 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 different attributes. Describe them in the experiment section	100%
9	Report algorithm, experiment with some reflection or comments	100%

3. Detailed algorithm description

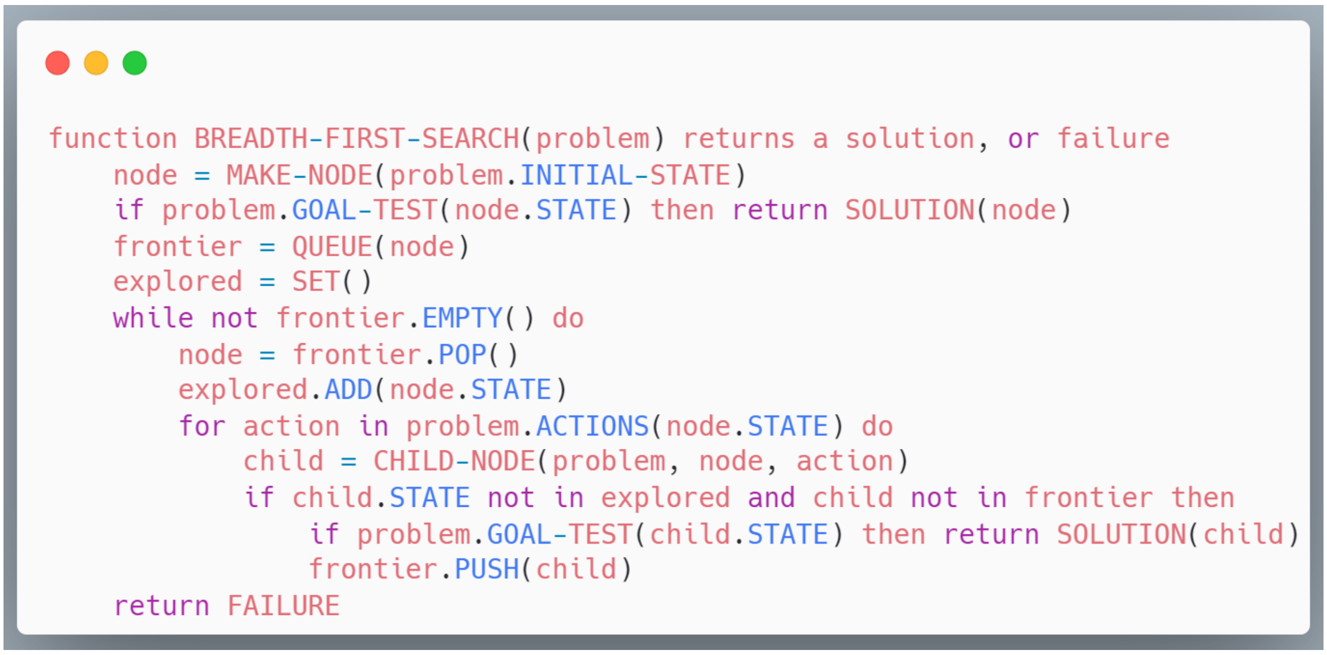
In this lab, we will implement the following search algorithms:

- 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.

3.1. Breadth-first search (BFS)

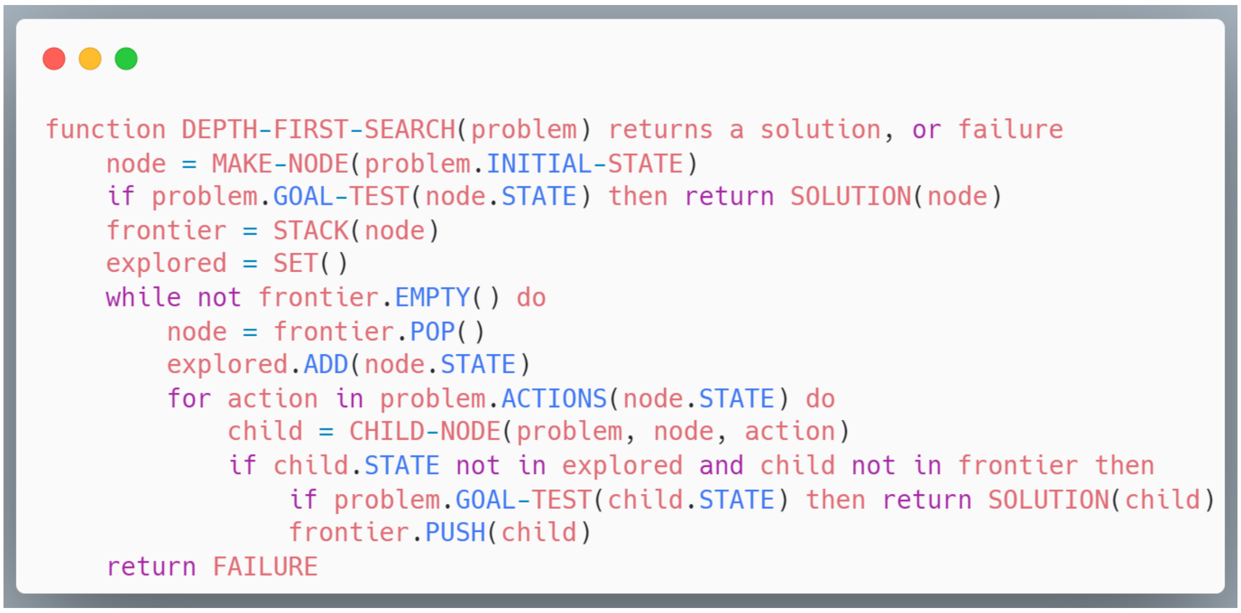
Breadth-first search is a simple graph traversal algorithm that starts at the root node and explores all the neighboring nodes. Then, it moves to the next level of nodes and explores their neighbors, and so on. The algorithm uses a queue to keep track of the nodes to be explored. The algorithm is implemented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



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node = MAKE-NODE(problem.INITIAL-STATE)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier = QUEUE(node)
    explored = SET()
    while not frontier.EMPTY() do
        node = frontier.POP()
        explored.ADD(node.STATE)
        for action in problem.ACTIONS(node.STATE) do
            child = CHILD-NODE(problem, node, action)
            if child.STATE not in explored and child not in frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier.PUSH(child)
    return FAILURE
```

3.2. Depth-first search (DFS)

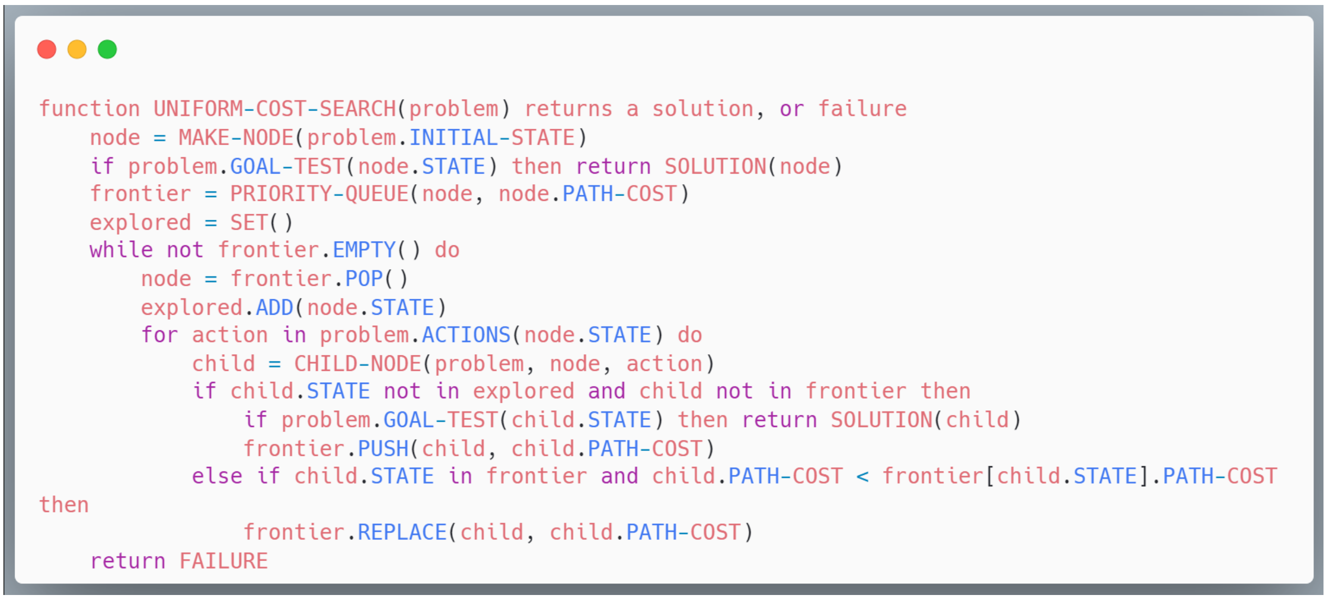
Depth-first search is another graph traversal algorithm that starts at the root node and explores as far as possible along each branch before backtracking. The algorithm uses a stack to keep track of the nodes to be explored. The algorithm is implemented as follows:



```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
    node = MAKE-NODE(problem.INITIAL-STATE)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier = STACK(node)
    explored = SET()
    while not frontier.EMPTY() do
        node = frontier.POP()
        explored.ADD(node.STATE)
        for action in problem.ACTIONS(node.STATE) do
            child = CHILD-NODE(problem, node, action)
            if child.STATE not in explored and child not in frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier.PUSH(child)
    return FAILURE
```

3.3 Uniform-cost search (UCS)

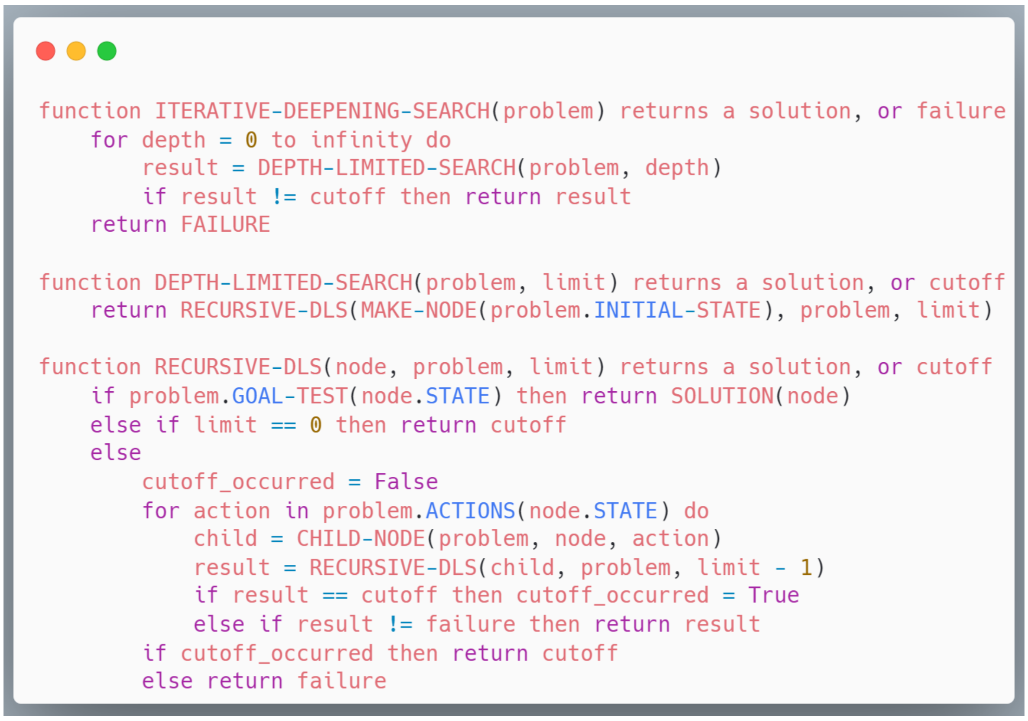
Uniform-cost search is a variant of Dijkstra's algorithm that finds the shortest path from the start node to the goal node. The algorithm uses a priority queue to keep track of the nodes to be explored. The algorithm is implemented as follows:



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node = MAKE-NODE(problem.INITIAL-STATE)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier = PRIORITY-QUEUE(node, node.PATH-COST)
    explored = SET()
    while not frontier.EMPTY() do
        node = frontier.POP()
        explored.ADD(node.STATE)
        for action in problem.ACTIONS(node.STATE) do
            child = CHILD-NODE(problem, node, action)
            if child.STATE not in explored and child not in frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier.PUSH(child, child.PATH-COST)
            else if child.STATE in frontier and child.PATH-COST < frontier[child.STATE].PATH-COST
then
                frontier.REPLACE(child, child.PATH-COST)
    return FAILURE
```

3.4. Iterative deepening search (IDS)

Iterative deepening search is a combination of depth-first search and breadth-first search that uses a depth-first search strategy with a depth limit. The algorithm is implemented as follows:



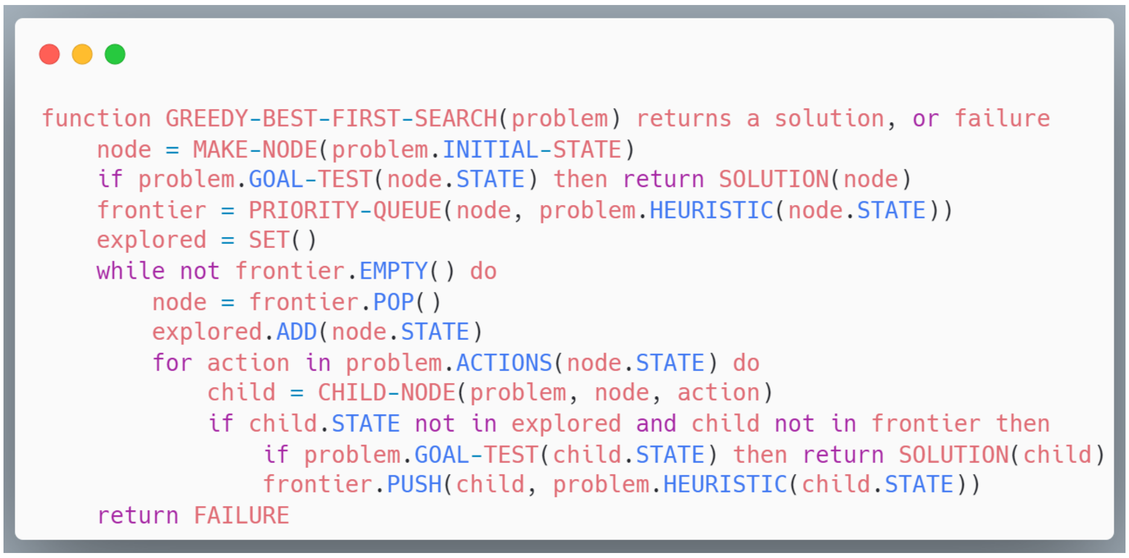
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to infinity do
        result = DEPTH-LIMITED-SEARCH(problem, depth)
        if result != cutoff then return result
    return FAILURE

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if limit == 0 then return cutoff
    else
        cutoff_occurred = False
        for action in problem.ACTIONS(node.STATE) do
            child = CHILD-NODE(problem, node, action)
            result = RECURSIVE-DLS(child, problem, limit - 1)
            if result == cutoff then cutoff_occurred = True
            else if result != failure then return result
        if cutoff_occurred then return cutoff
        else return failure
```

3.5. Greedy best-first search (GBFS)

Greedy best-first search is a graph traversal algorithm that uses a heuristic function to determine the best node to explore next. The algorithm uses a priority queue to keep track of the nodes to be explored. The algorithm is implemented as follows:



```
function GREEDY-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    node = MAKE-NODE(problem.INITIAL-STATE)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier = PRIORITY-QUEUE(node, problem.HEURISTIC(node.STATE))
    explored = SET()
    while not frontier.EMPTY() do
        node = frontier.POP()
        explored.ADD(node.STATE)
        for action in problem.ACTIONS(node.STATE) do
            child = CHILD-NODE(problem, node, action)
            if child.STATE not in explored and child not in frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier.PUSH(child, problem.HEURISTIC(child.STATE))
    return FAILURE
```

3.6. A* search

A* search is a graph traversal algorithm that uses a combination of the cost to reach a node and the estimated cost to reach the goal node to determine the best node to explore next. The algorithm uses a priority queue to keep track of the nodes to be explored. The algorithm is implemented as follows:

```
function A*-SEARCH(problem) returns a solution, or failure
    node = MAKE-NODE(problem.INITIAL-STATE)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier = PRIORITY-QUEUE(node, node.PATH-COST + problem.HEURISTIC(node.STATE))
    explored = SET()
    while not frontier.EMPTY() do
        node = frontier.POP()
        explored.ADD(node.STATE)
        for action in problem.ACTIONS(node.STATE) do
            child = CHILD-NODE(problem, node, action)
            if child.STATE not in explored and child not in frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier.PUSH(child, child.PATH-COST + problem.HEURISTIC(child.STATE))
            else if child.STATE in frontier and child.PATH-COST + problem.HEURISTIC(child.STATE) <
            frontier[child.STATE].PATH-COST + problem.HEURISTIC(frontier[child.STATE].STATE) then
                frontier.REPLACE(child, child.PATH-COST + problem.HEURISTIC(child.STATE))
    return FAILURE
```

3.7. 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:

```
function HILL-CLIMBING(problem) returns a solution, or failure
    current = MAKE-NODE(problem.INITIAL-STATE)
    while True do
        neighbor = MAX-VALUE(problem.NEIGHBORS(current.STATE))
        if problem.HEURISTIC(neighbor.STATE) <= problem.HEURISTIC(current.STATE) then return FAILURE
        else if problem.GOAL-TEST(neighbor.STATE) then return SOLUTION(neighbor)
        current = neighbor
```

4. Implementation and experimentation

This section describes the implementation of the search algorithms and presents the experimental results on 5 test cases. The implementation details, test cases, and results are presented below.

4.1 Test cases

I have implemented the search algorithms in Python and tested them on 5 test cases.

The input file contains information about the graph and weights, formatted as follows:

- 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).

Note that the graph can be either directed or undirected.

I used *Visualgo* generate the visual representation graphs for the test cases.

I have referenced test cases and solutions for running the algorithm and checking its correctness here.

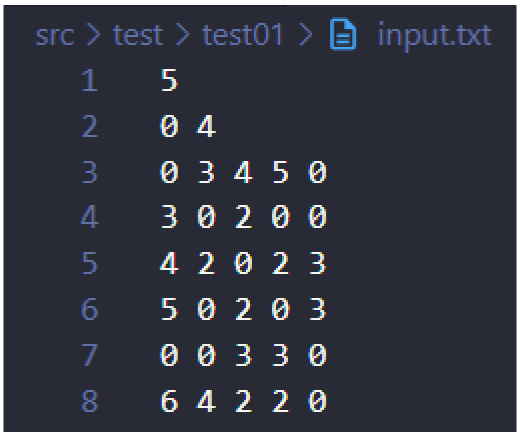
The details of the test cases are as follows:

4.1.1 Test case 1: Simple Weighted Undirected Graph with Admissible Heuristics

- Initial state: 0

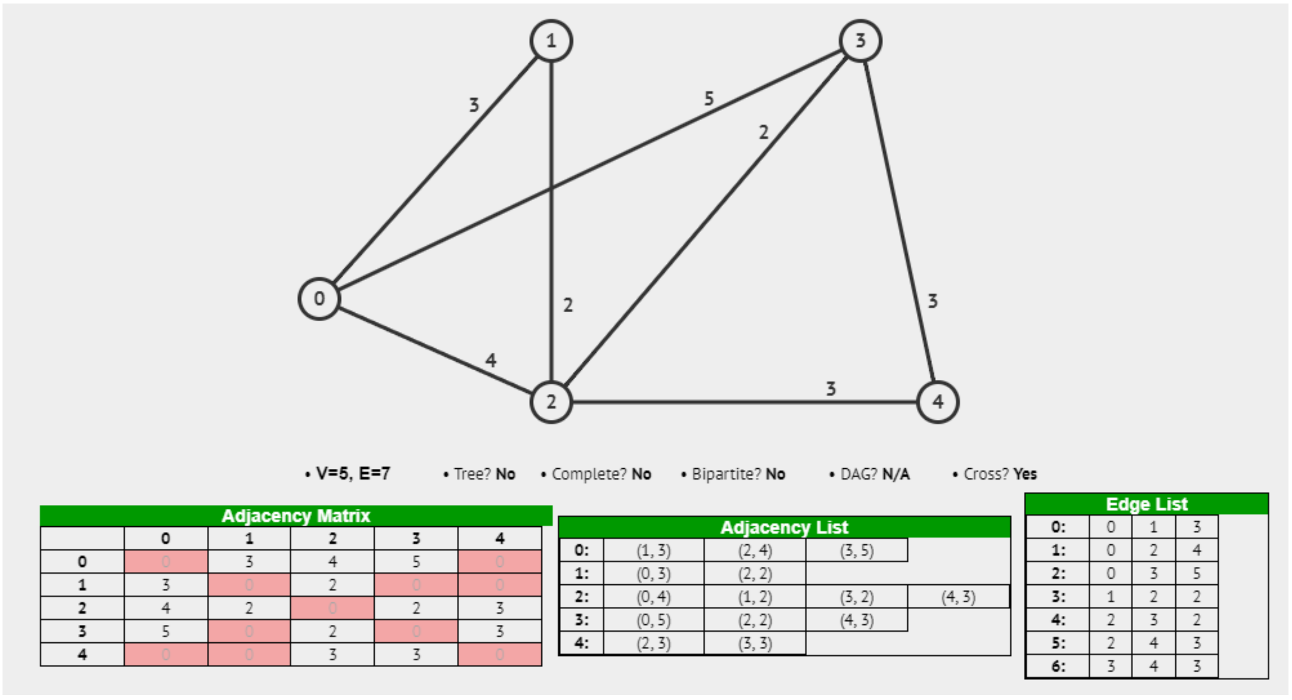
- Goal state: 4

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



```
5
0 4
0 3 4 5 0
3 0 2 0 0
4 2 0 2 3
5 0 2 0 3
0 0 3 3 0
6 4 2 2 0
```

Figure 1: Input file of test case 1



• V=5, E=7 • Tree? No • Complete? No • Bipartite? No • DAG? N/A • Cross? Yes

Adjacency Matrix

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

Adjacency List

0:	(1, 3)	(2, 4)	(3, 5)	
1:	(0, 3)	(2, 2)		
2:	(0, 4)	(1, 2)	(3, 2)	(4, 3)
3:	(0, 5)	(2, 2)	(4, 3)	
4:	(2, 3)	(3, 3)		

Edge List

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

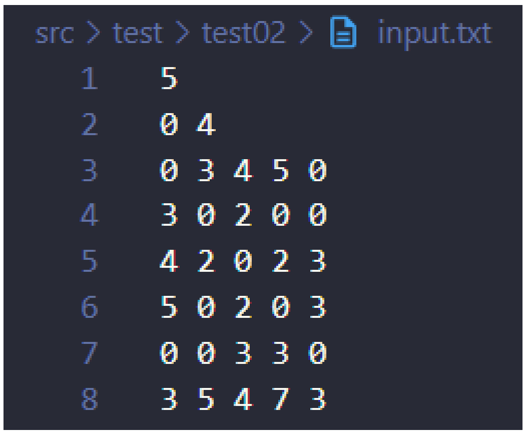
Figure 2: Detailed illustration graph of test case 1

4.1.2 Test case 2: Simple Weighted Undirected Graph with Inadmissible Heuristics

- Initial state: 0

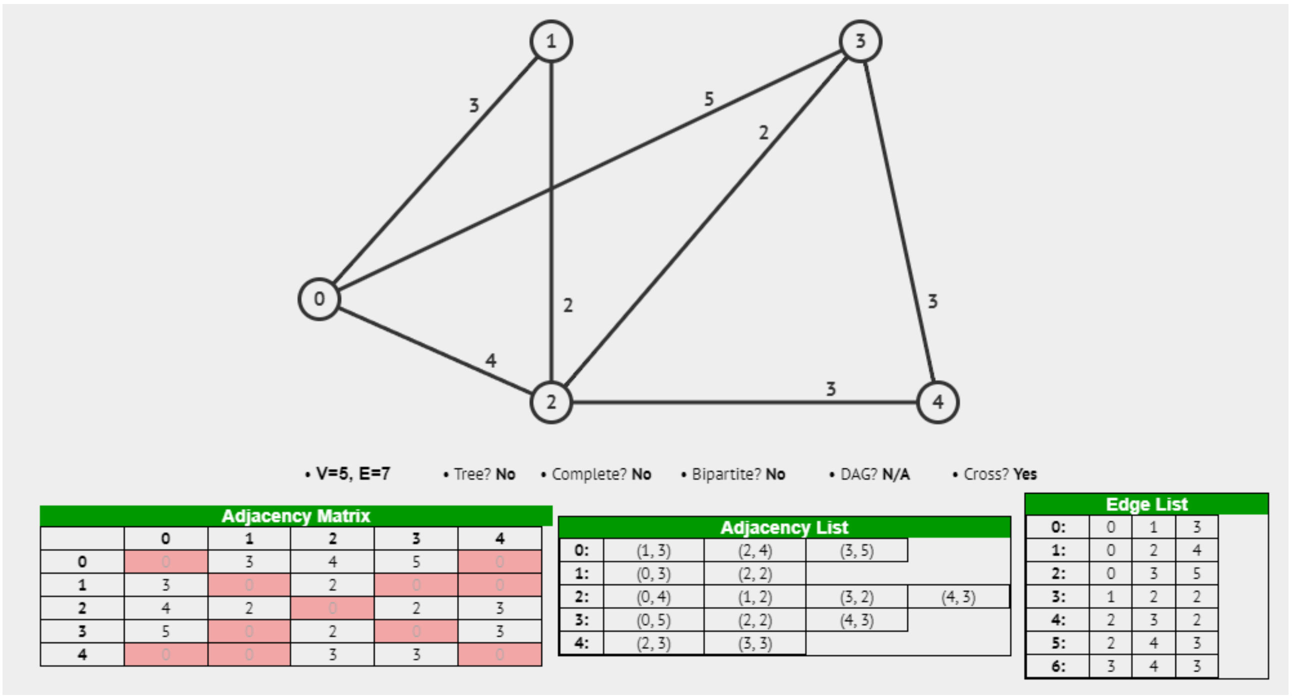
- Goal state: 4

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



```
5
0 4
0 3 4 5 0
3 0 2 0 0
4 2 0 2 3
5 0 2 0 3
0 0 3 3 0
3 5 4 7 3
```

Figure 3: Input file of test case 2



Adjacency Matrix

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

Adjacency List

0:	(1, 3)	(2, 4)	(3, 5)	
1:	(0, 3)	(2, 2)		
2:	(0, 4)	(1, 2)	(3, 2)	(4, 3)
3:	(0, 5)	(2, 2)	(4, 3)	
4:	(2, 3)	(3, 3)		

Edge List

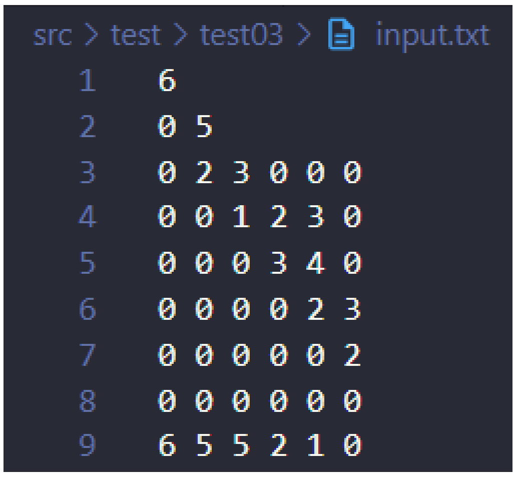
0:	0	1	3
1:	0	2	4
2:	0	3	5
3:	1	2	2
4:	2	3	2
5:	2	4	3
6:	3	4	3

Figure 4: Detailed illustration graph of test case 2

4.1.3 Test case 3: Simple Weighted Directed Graph with Admissible Heuristics

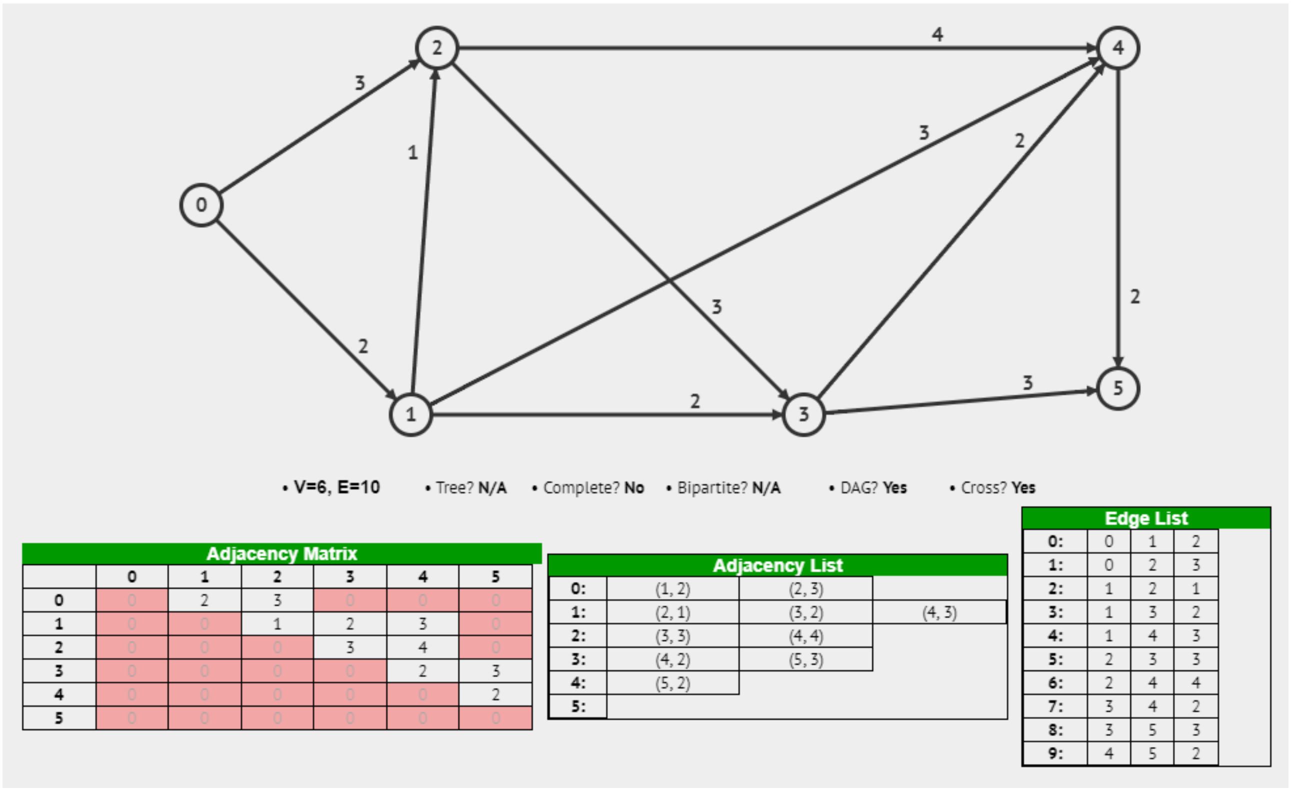
- Initial state: 0
- Goal state: 5

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


src > test > test03 > input.txt

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

Figure 5: Input file of test case 3



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

Adjacency Matrix

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

Adjacency List

0:	(1, 2)	(2, 3)	
1:	(2, 1)	(3, 2)	(4, 3)
2:	(3, 3)	(4, 4)	
3:	(4, 2)	(5, 3)	
4:	(5, 2)		
5:			

Edge List

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

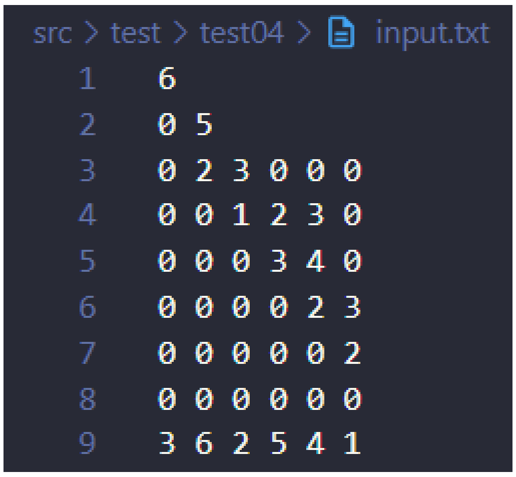
Figure 6: Detailed illustration graph of test case 3

4.1.4 Test case 4: Simple Weighted Undirected Graph with Inadmissible Heuristics

- Initial state: 0

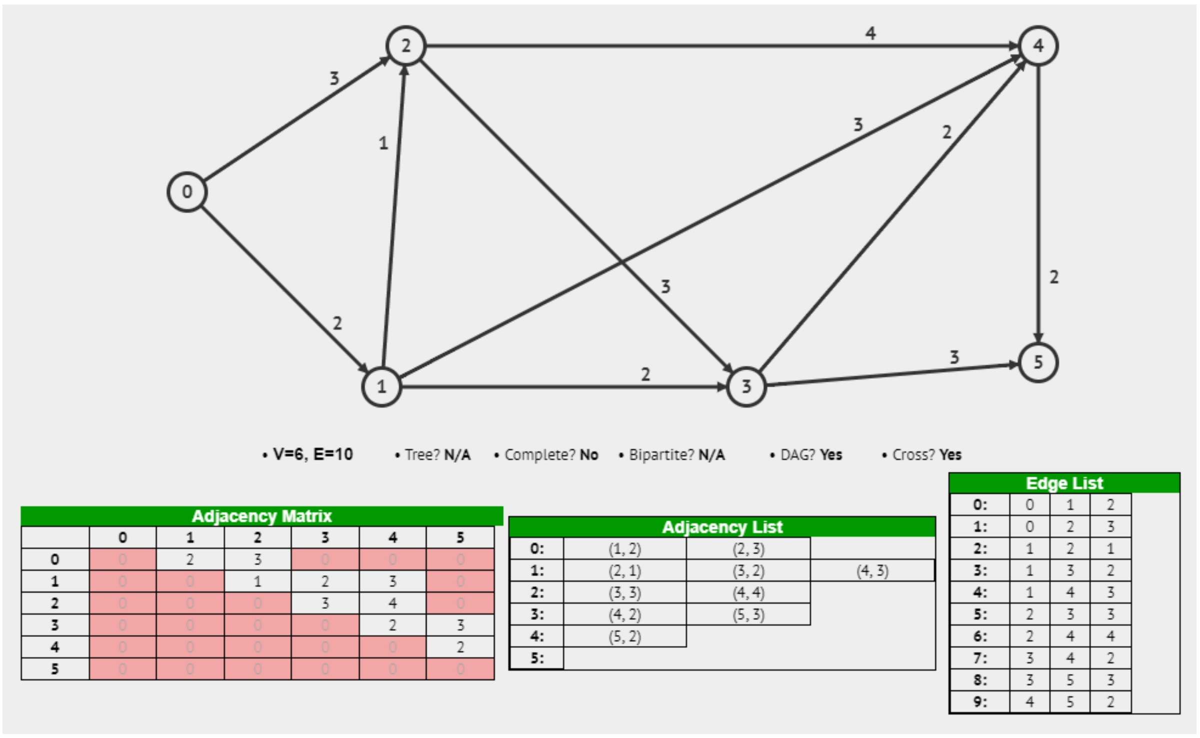
- Goal state: 5

Vertex	0	1	2	3	4	5
Heuristic	7	5	9	6	3	5



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

Figure 7: Input file of test case 4



Adjacency Matrix

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

Adjacency List

0:	(1, 2)	(2, 3)	
1:	(2, 1)	(3, 2)	(4, 3)
2:	(3, 3)	(4, 4)	
3:	(4, 2)	(5, 3)	
4:	(5, 2)		
5:			

Edge List

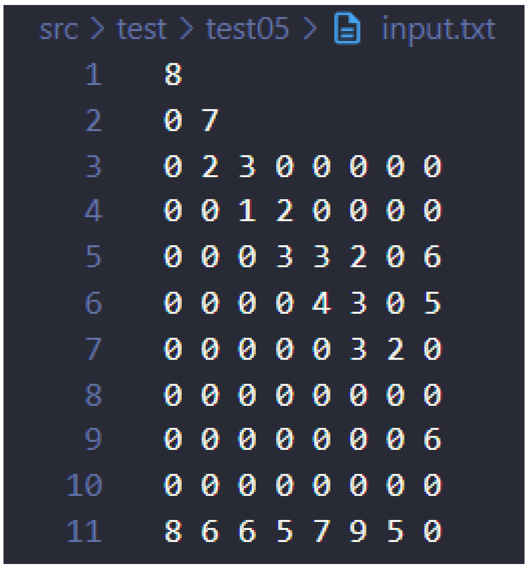
0:	0	1	2
1:	0	2	3
2:	1	2	1
3:	1	3	2
4:	1	4	3
5:	2	3	3
6:	2	4	4
7:	3	4	2
8:	3	5	3
9:	4	5	2

Figure 8: Detailed illustration graph of test case 4

4.1.5 Test case 5: Complex Graph with Admissible Heuristics

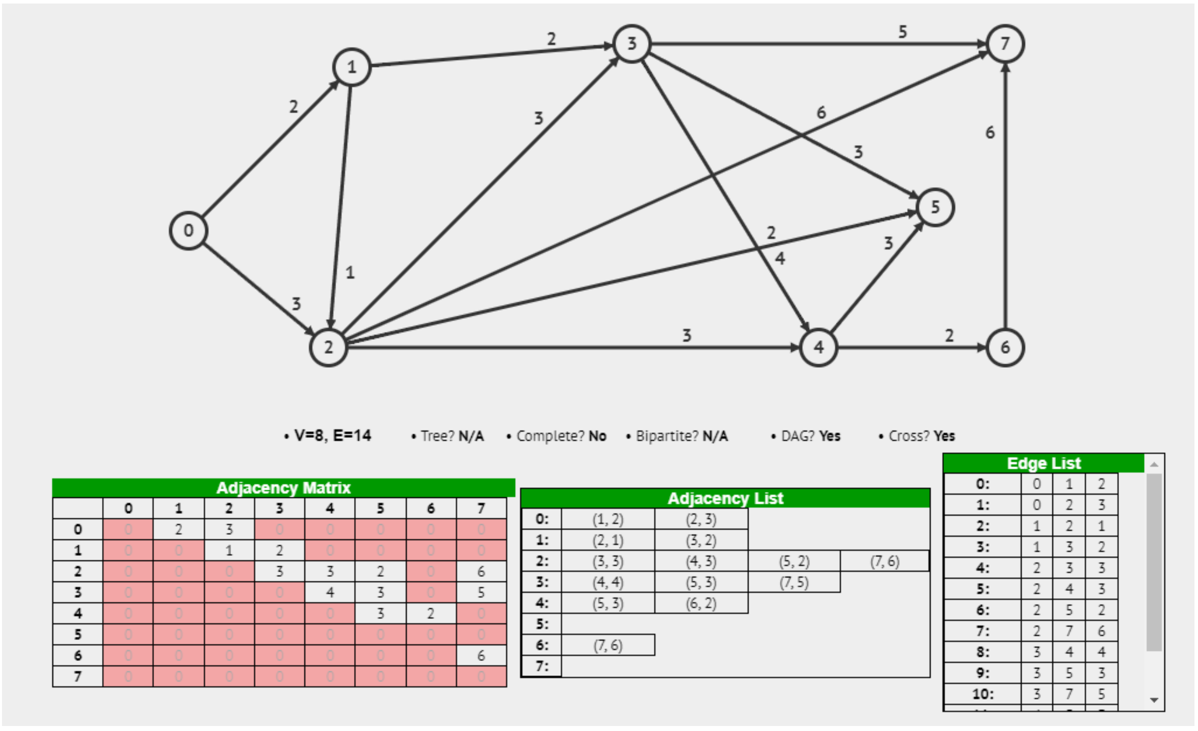
- Initial state: 0
- Goal state: 7

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



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

Figure 9: Input file of test case 5



Adjacency Matrix

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

Adjacency List

0:	(1, 2)	(2, 3)		
1:	(2, 1)	(3, 2)		
2:	(3, 3)	(4, 3)	(5, 2)	(7, 6)
3:	(4, 4)	(5, 3)	(7, 5)	
4:	(5, 3)	(6, 2)		
5:				
6:	(7, 6)			
7:				

Edge List

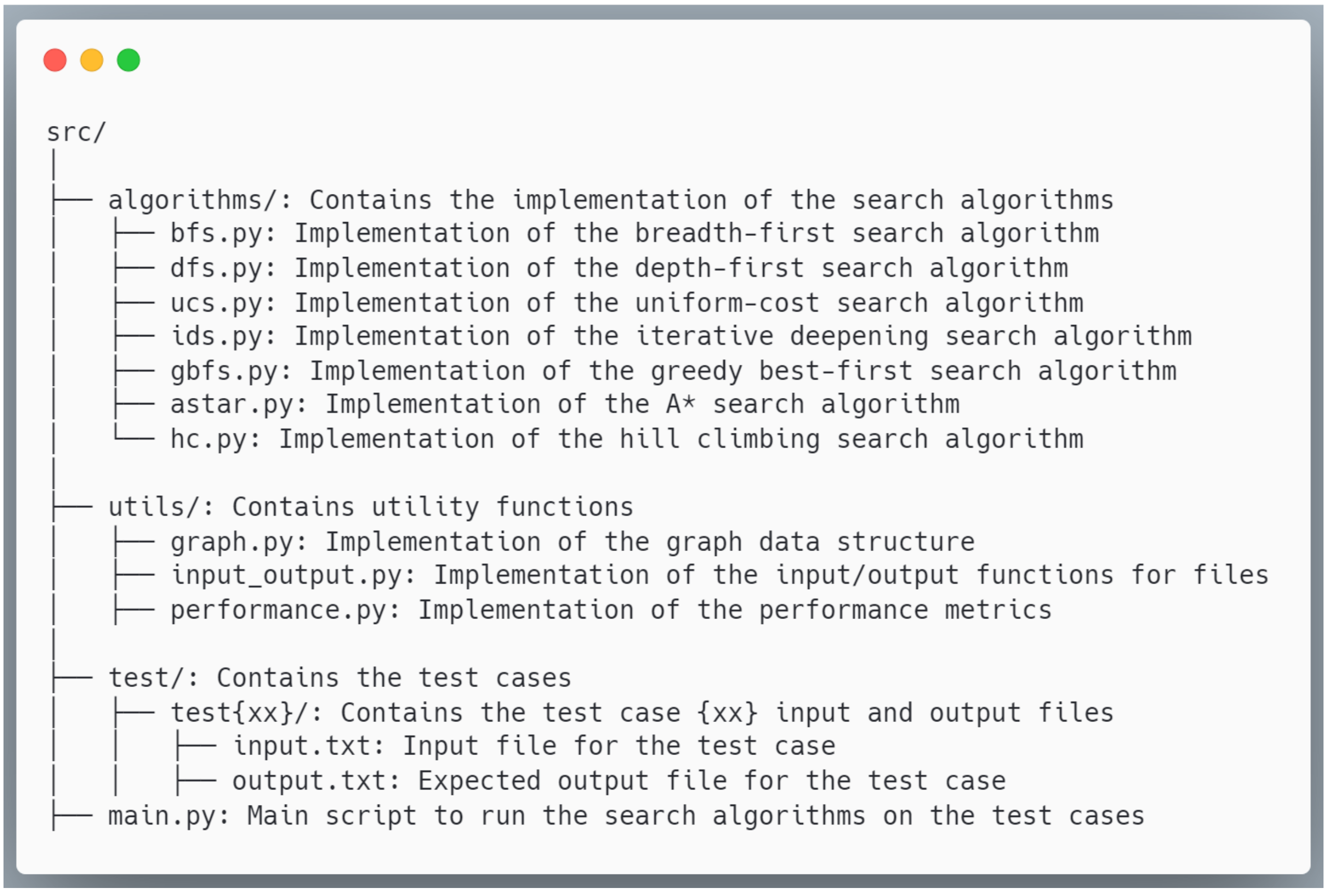
0:	0	1	2
1:	0	2	3
2:	1	2	1
3:	1	3	2
4:	2	3	3
5:	2	4	3
6:	2	5	2
7:	2	7	6
8:	3	4	4
9:	3	5	3
10:	3	7	5

Figure 10: Detailed illustration graph of test case 5

4.2. Implementation

I used Python to implement the search algorithms. The complete implementation and detailed documentation of code can be found in the src folder.

Here is the structure of the src folder:



```
src/
|
├── algorithms/: Contains the implementation of the search algorithms
|    ├── bfs.py: Implementation of the breadth-first search algorithm
|    ├── dfs.py: Implementation of the depth-first search algorithm
|    ├── ucs.py: Implementation of the uniform-cost search algorithm
|    ├── ids.py: Implementation of the iterative deepening search algorithm
|    ├── gbfs.py: Implementation of the greedy best-first search algorithm
|    ├── astar.py: Implementation of the A* search algorithm
|    └── hc.py: Implementation of the hill climbing search algorithm
|
├── utils/: Contains utility functions
|    ├── graph.py: Implementation of the graph data structure
|    ├── input_output.py: Implementation of the input/output functions for files
|    ├── performance.py: Implementation of the performance metrics
|
├── test/: Contains the test cases
|    ├── test{xx}/: Contains the test case {xx} input and output files
|    |    ├── input.txt: Input file for the test case
|    |    ├── output.txt: Expected output file for the test case
├── main.py: Main script to run the search algorithms on the test cases
```

4.2.1 External libraries

I used the following external libraries in the implementation:

- ***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
- ***tracemalloc***: Used for measuring the memory usage of the algorithms

4.3 Experimental results

I tested the search algorithms on the 5 test cases and measured the execution time and memory usage. The results are presented in the following tables:

4.3.1 Execution time

The execution time (second) of the search algorithms on the test cases is as follows:

Test case	**Nodes**	**Edges**	**Admissible Heuristic**	**BFS**	**DFS**	**UCS**	**IDS**	**GBFS**	**A***	**HC**
1	5	7	Yes	0.0015	0.0	0.0	0.0	0.0	0.0	0.0
2	5	7	No	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	6	10	Yes	0.0	0.0	0.01	0.0	0.0	0.0	0.0
4	6	10	No	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	8	14	Yes	0.0	0.0	0.0	0.0	0.0	0.0	0.0

4.3.2 Memory usage

The memory usage (KB) of the search algorithms on the test cases is as follows:

Test case	**Nodes**	**Edges**	**Admissible Heuristic**	**BFS**	**DFS**	**UCS**	**IDS**	**GBFS**	**A***	**HC**
1	5	7	Yes	2.0	0.91	1.02	0.59	0.56	1.24	0.36
2	5	7	No	1.75	0.73	0.48	0.48	0.50	0.43	0.30
3	6	10	Yes	1.97	0.88	1.10	0.55	0.46	0.69	0.36
4	6	10	No	1.97	0.88	1.10	0.55	0.52	0.55	0.36
5	8	14	Yes	2.27	0.76	1.29	0.46	0.52	0.67	0.36

4.3.3 Discussion

The results show that the search algorithms have different performance characteristics in terms of execution time, memory usage, and correctness. The following observations can be made:

- **Execution time**: Because the number of nodes and edges in the graph is small, the execution time of the algorithms is relatively low and not significantly different. However, the algorithms that use heuristics (GBFS, A*) tend to be faster than the other algorithms because they can make more informed decisions about which nodes to explore next. The hill climbing search algorithm has the highest execution time because it only considers the neighboring nodes without any heuristic information.

- **Memory usage**: The memory usage of the algorithms is also relatively low due to the small size of the graph. The hill climbing search algorithm has the lowest memory usage because it only needs to store the current node and its neighbors. The algorithms that use heuristics (GBFS, A*) have higher memory usage because they need to store additional information about the estimated cost to reach the goal node. The algorithms such as DFS, IDS and GBFS have higher memory usage because they need to store the frontier nodes in a stack or queue. The UCS and A* algorithms have the higher memory usage because they need to store the priority queue of nodes to be explored based on the path cost and heuristic value respectively. The BFS algorithm has the highest memory usage because it needs to store all the nodes in the frontier queue.

- **Correctness**: All the algorithms except the hill climbing search algorithm are complete and optimal. The hill climbing search algorithm is not complete because it can get stuck in a local maximum and not reach the goal state. The hill climbing search algorithm is also not optimal because it only considers the neighboring nodes and may not find the shortest path to the goal state.

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

5. Conclusion

In this lab, we implemented and tested several search algorithms, including breadth-first search, depth-first search, uniform-cost search, iterative deepening search, greedy best-first search, A* search, and hill climbing search. We presented the detailed algorithm descriptions, the implementation details, and the experimental results on 5 test cases. The results show that the algorithms have different performance characteristics in terms of execution time and memory usage. The choice of algorithm depends on the specific problem and the desired trade-offs between time and space complexity.

6. References

- Lecture slides and materials from the course.
- GeeksforGeeks. (2023, March 22). Search algorithms in AI. GeeksforGeeks. https://www.geeksforgeeks.org/search-algorithms-in-ai/
- GeeksforGeeks. (2023b, April 20). *Introduction to hill climbing artificial intelligence*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/