VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

June 15th, 2024

Introduction to Artificial Intelligence

Report project 1: Searching

Under the guidance of :

1. Mr.Nguyễn Thanh Tình
2. Mr.Phạm Trọng Nghĩa

Student:

Name : Nguyễn Hoàng Trung Kiên

ID: 22127478

Class: 22CLC08

Contents

I. Acknowledgement:

I would like to express my sincere gratitude to Mr. Nguyen Thanh Tinh, a lecturer in the Faculty of Information Technology at the University of Science, for providing me with the opportunity to complete this project. This project has been a significant milestone in my development, giving me the experience necessary to tackle more challenging projects in the future.

II. Self-evaluation:

No.	Task	Completion
1	Implement BFS	100%
2	Implement DFS	100%
3	Implement UCS	100%
4	Implement IDS	100%
5	Implement GBFS	100%
6	Implement A*	100%
7	Implement Hill-climbing	100%
8	Generate at least 5 test cases for all algorithm with different attributes	100%
9	Write report of algorithm	100%

Table 1 : Self-evaluation

III. Algorithm explanation:

1. Breadth-first Search (BFS):

Breadth-first search (BFS) is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. [1]

In my implementation, BFS stops when the goal node is generated, not when it is expanded.

Detailed description:

Initialization:

- Create a start node represents the source.
- Frontier is the double-ended queue (deque) used to store the nodes that need to be explored.
- Start node is put to the frontier.

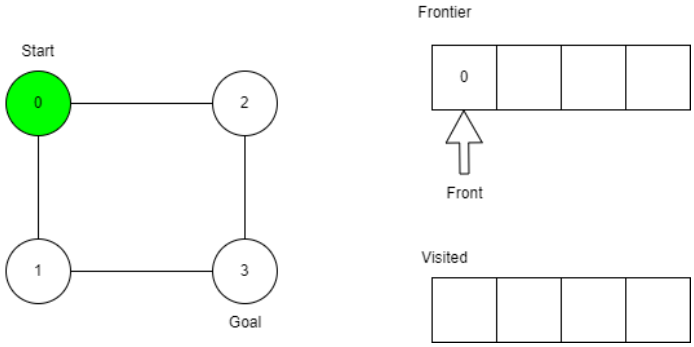


Figure 1 : Initialization (BFS)

Exploration:

- While the frontier still contains node(s):
    - The first node is removed from the front of frontier for exploration.
    - The algorithm checks if the current node's state matches the goal state. If it does, it returns the dictionary of visited nodes and the solution path. If it doesn't, continue the loop.
    - The node is added to visited, also the visited dictionary stores the parent's state of that node to trace back the path later. (If the node has no parent it stores 'None')
    - Then the algorithm enters another loop for retrieving neighbors from the current node's state:
        - It checks if the neighbor state is not already visited and not already in the frontier, if it is, a new child node is created.

- The algorithm checks again if this child node is the goal state. If it is, it returns the visited dictionary and the solution path. If not, the child node is added to the frontier for future exploration.

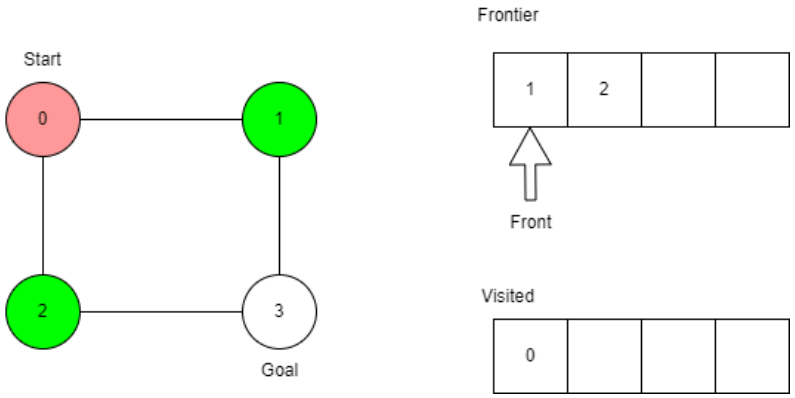


Figure 2 : Add neighbors to frontier and mark the visited node (BFS)

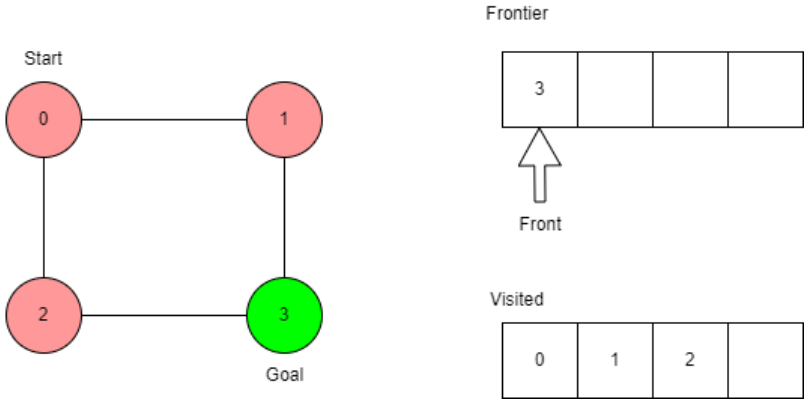


Figure 3 : Goal node is generated to the frontier and the algorithm stops (BFS)

- Finally, if there is no path the algorithm returns visited nodes and path = -1.

2. Depth-first Search (DFS):

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 node in the case of a graph) and explores as far as possible along each branch before backtracking. [2]

In my implementation, DFS stops when the goal node is generated, not when it is expanded.

Detailed description:

Initialization:

- Create a start node represents the source.
- Frontier is the stack used to store the nodes that need to be explored.
- Start node is put to the frontier.

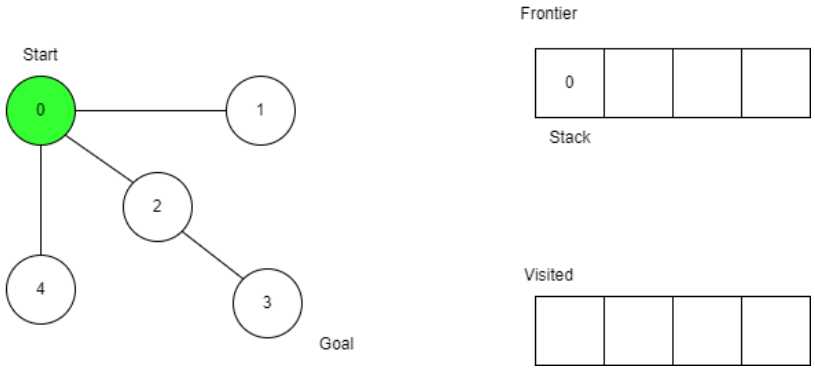


Figure 4 : Initialization (DFS)

Exploration:

- While the frontier still contains node(s):
    - The first node is popped out from stack(frontier)
    - The algorithm checks if the current node's state matches the goal state. If it does, it returns the dictionary of visited nodes and the solution path. If it doesn't, continue the loop.
    - If the node hasn't been visited, The node is added to visited, also the visited dictionary stores the parent's state of that node to trace back the path later. (If the node has no parent it stores 'None') and the algorithm enters another loop for retrieving neighbors from the current node's state:
        - It checks if the neighbor state is not already visited, if it isn't, the child node is added to the stack.

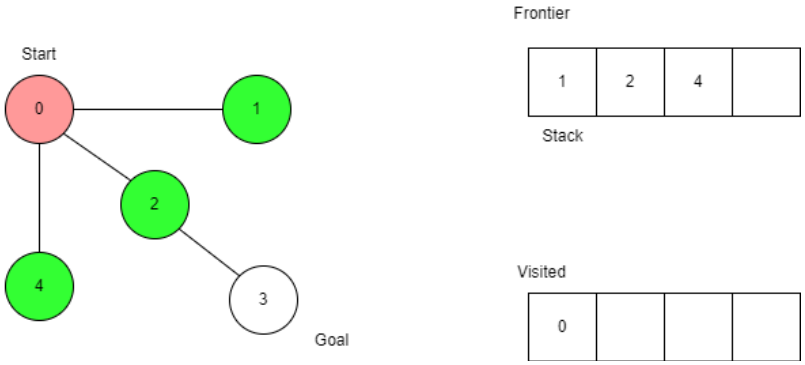


Figure 5 : Add child nodes to stack and mark the current node as visited (DFS)

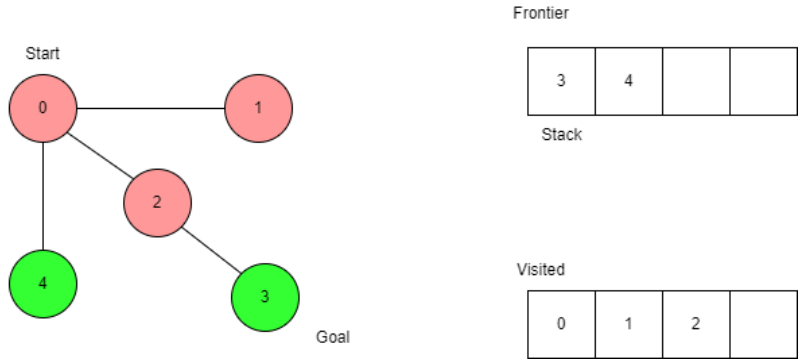


Figure 6 : Goal node is generated to the frontier and the algorithm stops (BFS)

- Finally, if there is no path the algorithm returns visited nodes and path = -1.

3. Uniform-cost Search (UCS):

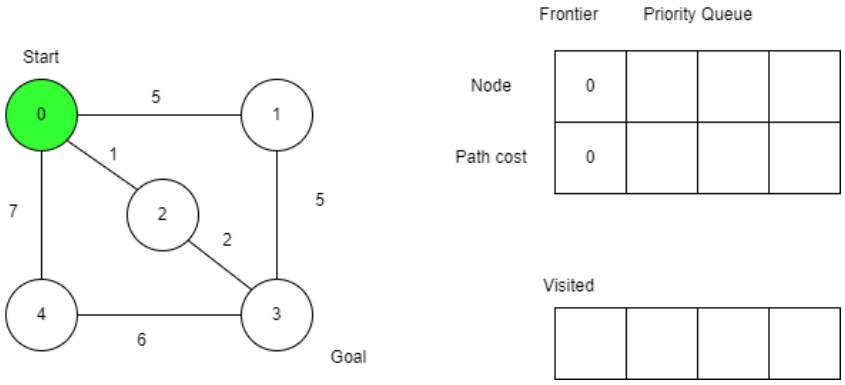
Uniform-Cost Search is similar to Dijkstra's algorithm. In this algorithm from the starting state, we will visit the adjacent states and will choose the least costly state then we will choose the next least costly state from the all un-visited and adjacent states of the visited states, in this way we will try to reach the goal state (note we won't continue the path through a goal state), even if we reach the goal state we will continue searching for other possible paths(if there are multiple goals). We will keep a priority queue that will give the least costly next state from all the adjacent states of visited states.[3]

In my implementation, DFS stops when the goal node is expanded.

Detailed description:

Initialization:

- Create a priority queue (min-heap) to store nodes to be explored. The heapq module is used to maintain the priority queue.
- Start node is generated with a path cost of 0 and pushed onto the priority queue



(frontier).

Figure 7 : Initialization (UCS)

Exploration:

- While the frontier still contains node(s):
  - The node with the lowest path cost is popped from the priority queue.
  - The algorithm checks if the current node is goal, if it is, the node is added to the visited dictionary, and the function returns the visited dictionary and the path found.
  - If the current node's state is not in the visited dictionary, it is added along with its parent state.
  - Then the algorithm enters another loop for retrieving neighbors from the current node's state:

- If the neighbor is not in the visited dictionary, the cost to move to this neighbor is obtained. Child node is created with the updated path cost. The child node is pushed onto the priority queue (frontier).

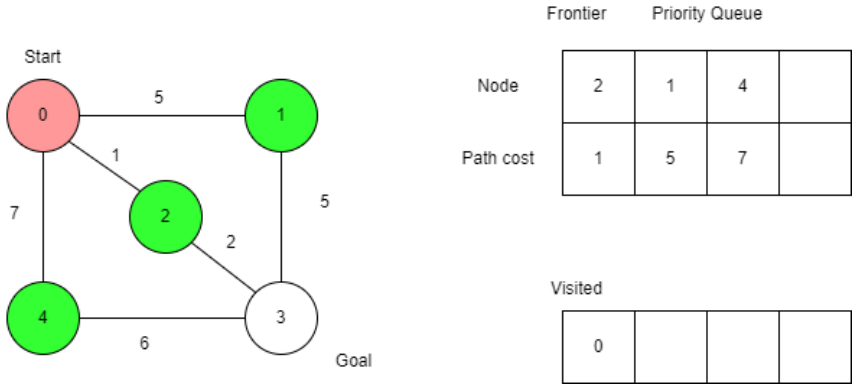


Figure 8 : Add neighbors to frontier with the following path cost and mark the visited node (UCS)

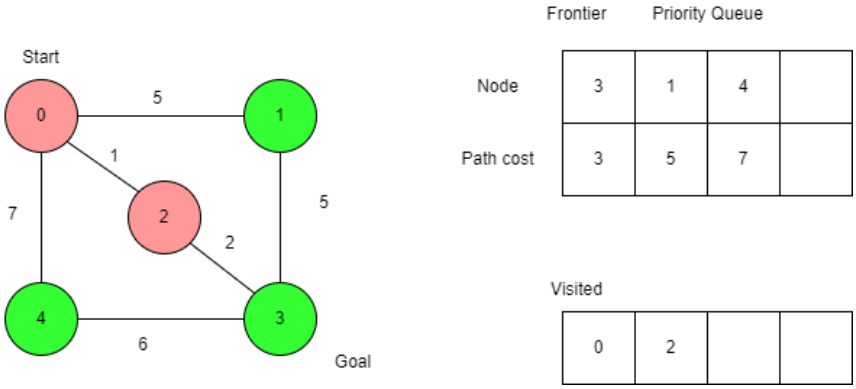


Figure 9 : Pop the front of the priority queue, mark it as visited and enqueue it's child node (UCS)

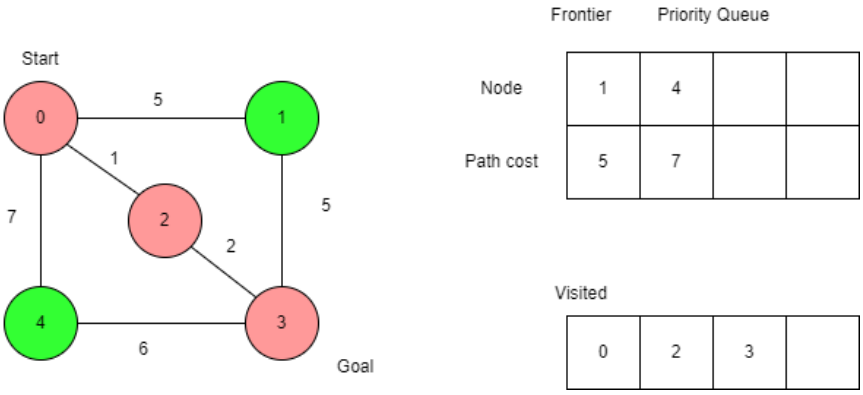


Figure 10 : The child node is goal so mark it as visited and the algorithm stops (UCS)

- Finally, if there is no path the algorithm returns visited nodes and path = -1.

4. Iterative Deepening Search (IDS):

Iterative deepening search or more specifically iterative deepening depth-first search (IDS or IDDFS) is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found.[4]

Detailed description:

Depth-limited search (DLS):

- Depth-limited search explores nodes up to a specified depth limit

Initialization:

- Start node is generated with a path cost of 0

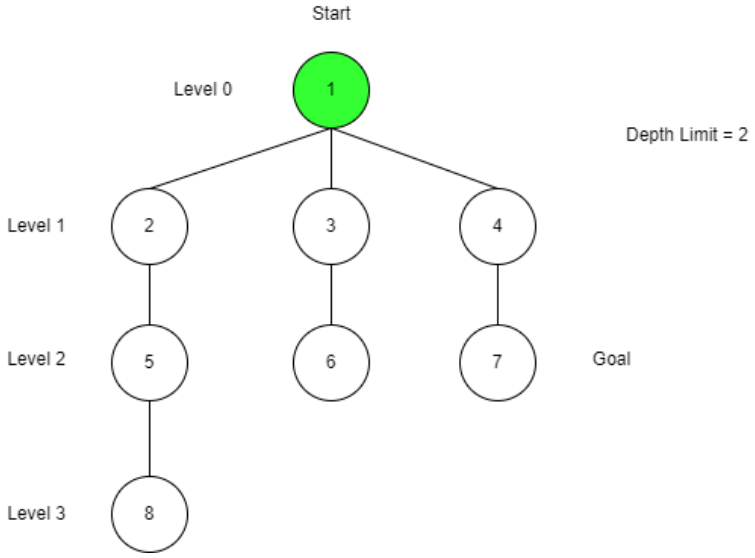


Figure 11 : Initialization (DLS)

Exploration:

- If the current node is the goal, extend the path and return True. If the depth limit is reached, return false.
- In the recursive case:
    - Mark the current node as visited.
    - The algorithm enters another loop for retrieving neighbors from the current node's state:
        - If the neighbor is not visited, child node is generated.
        - Recursive case is called with the child node and reduced depth.
        - If a path is found, return True.
- Start the recursive deep-limited search from the source node, the algorithm returns visited nodes and the corresponding path. Finally, if there is no path the algorithm returns visited nodes and path = -1.

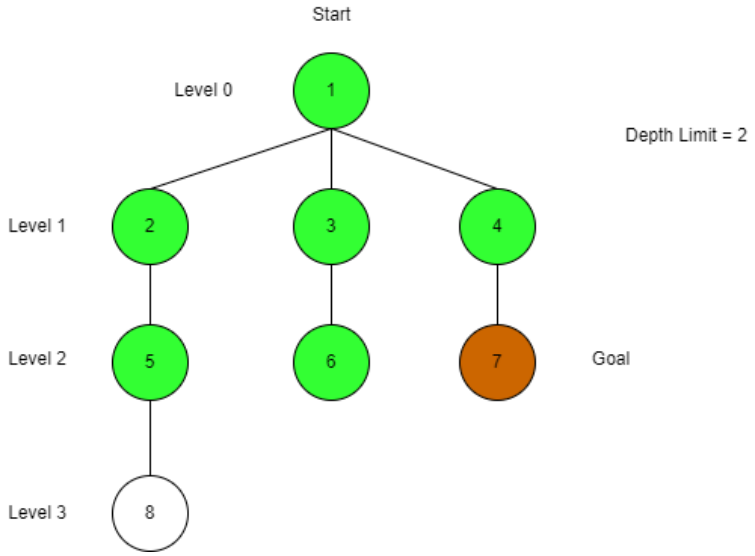


Figure 12: Recursively expand the visited node until the goal is reached or the depth limited is reached(DLS)

Iterative Deepening Search(IDS):

- Iterative Deepening Search performs an iterative deepening search by repeatedly calling Deep-limited Search with increasing depth limits.

Exploration:

- The algorithm loops over increasing depth limits from 1 to the number of nodes in the graph.
- For each depth limit, call Deep-limited Search.
- If the path is found, the algorithm returns visited nodes and the corresponding path. Finally, if there is no path the algorithm returns visited nodes and path = -1.

5. Greedy Best-first Search (GBFS):

Greedy Best-First Search is an AI search algorithm that attempts to find the most promising path from a given starting point to a goal. 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]

In my implementation, GBFS stops when the goal node is expanded.

Detailed description:

Initialization:

- Create a priority queue (min-heap) to store nodes to be explored.
- Start node is created with its heuristic cost and pushed onto the priority queue (frontier).

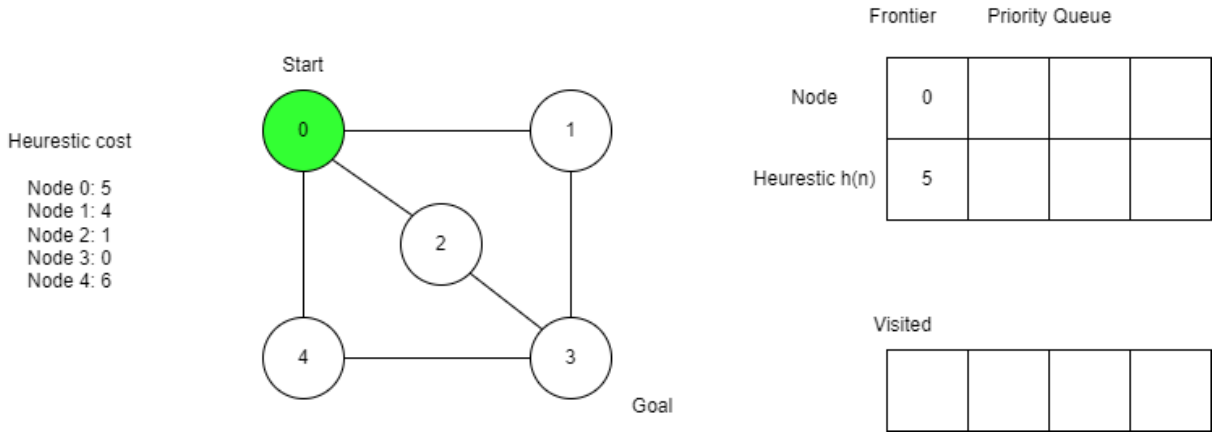


Figure 13 : Initialization(GBFS)

Exploration:

- While the frontier still contains node(s):
    - The node with the lowest heuristic cost is popped from the priority queue.
    - Checks if the current node's state is the goal. If it is, the function returns the visited dictionary and the path found.
    - The current node's state is marked as visited and added to the visited dictionary along with its parent state.
    - Then the algorithm enters another loop for retrieving neighbors from the current node's state if the neighbor is not in the visited dictionary and not already in the frontier:
        - Child node is generated with its heuristic cost and the current node as its parent.
        - If the child node is the goal, the function returns the visited dictionary and the path.
        - Otherwise, the child node is pushed onto the priority queue (frontier).

- Finally, if there is no path the algorithm returns visited nodes and path = -1.

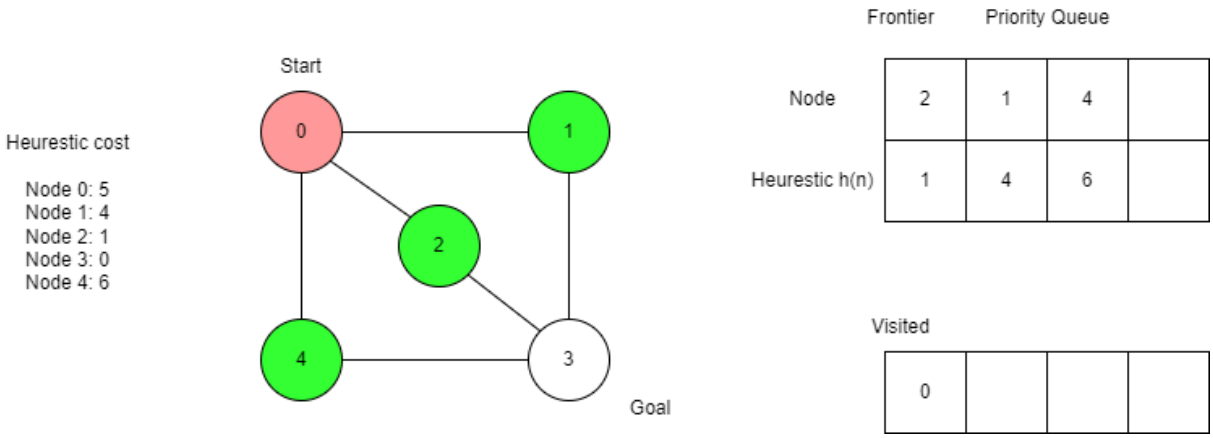


Figure 14 : Add neighbors to frontier with the following heurestic value and mark the visited node (GBFS)

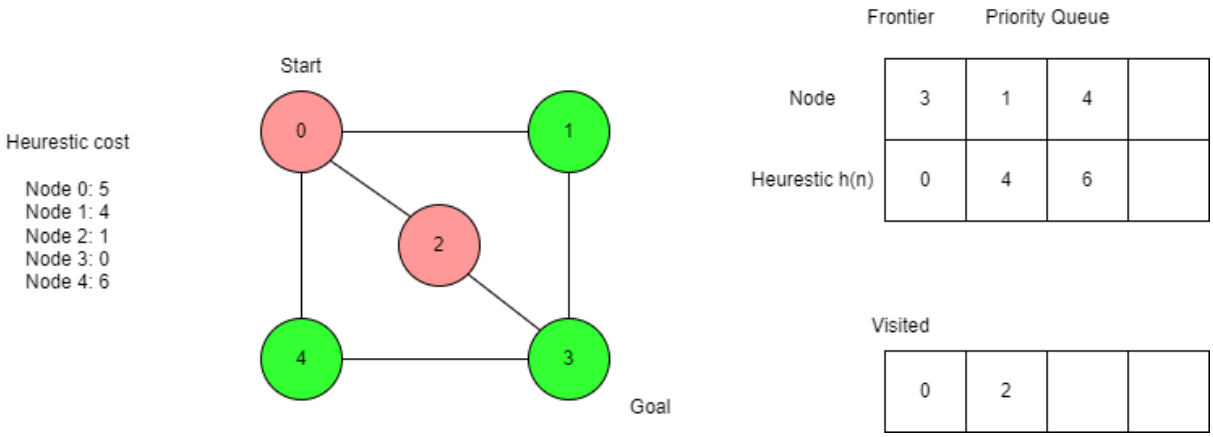


Figure 15 : Pop the front of the priority queue, mark it as visited and enqueue it's child node. Goal node is generated to the frontier and the algorithm stops (GBFS)

6. A* search:

A* Search is an informed best-first search algorithm that efficiently determines the lowest cost path between any two nodes in a directed weighted graph with non-negative edge weights. This algorithm is a variant of Dijkstra's algorithm. A slight difference arises from the fact that an evaluation function is used to determine which node to explore next. [6]

Detailed description:

Initialization:

- Create a priority queue (min-heap) to store nodes to be explored.
- Start node is created with its heuristic cost and pushed onto the priority queue(frontier).

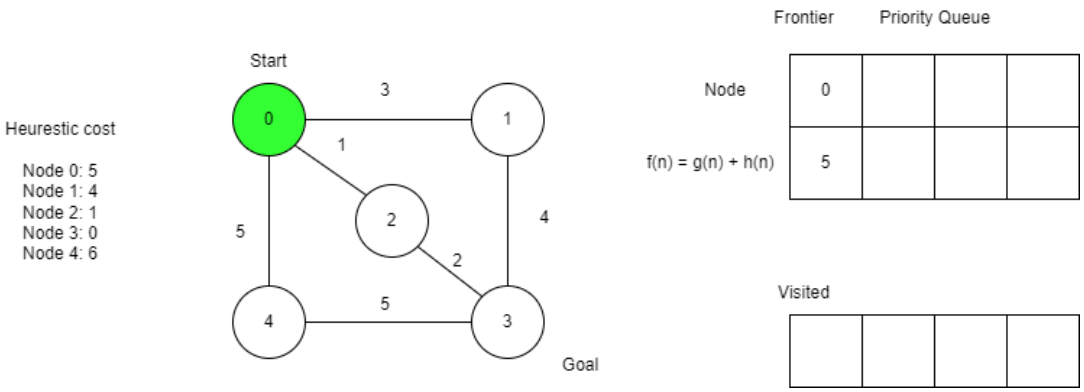


Figure 16 : Initialization (A*)

Exploration:

- While the frontier still contains node(s):
    - The node with the lowest f(n) = g(n) + h(n) cost is popped from the priority queue. (g(n) represents the cost to get to node n, h(n) represents the estimated cost to arrive at the goal node from node n).
    - Checks if the current node's state is the destination. If it is, the function returns the visited dictionary and the path found.
    - The current node's state is marked as visited and added to the visited dictionary along with its parent state.
    - Then the algorithm enters another loop for retrieving neighbors from the current node's state if the neighbor is not in the visited dictionary and not already in the frontier:
        - The cost to move to this neighbor is obtained.
        - The g(n) is calculated as the path cost to the current node plus the cost to the neighbor.
        - The f(n) is calculated as the g(n) plus the heuristic value of the neighbor.
        - Child node is generated with the g(n) and the current node as its parent.

- The child node is pushed onto the priority queue (frontier) with the f(n).

- Finally, if there is no path the algorithm returns visited nodes and path = -1.

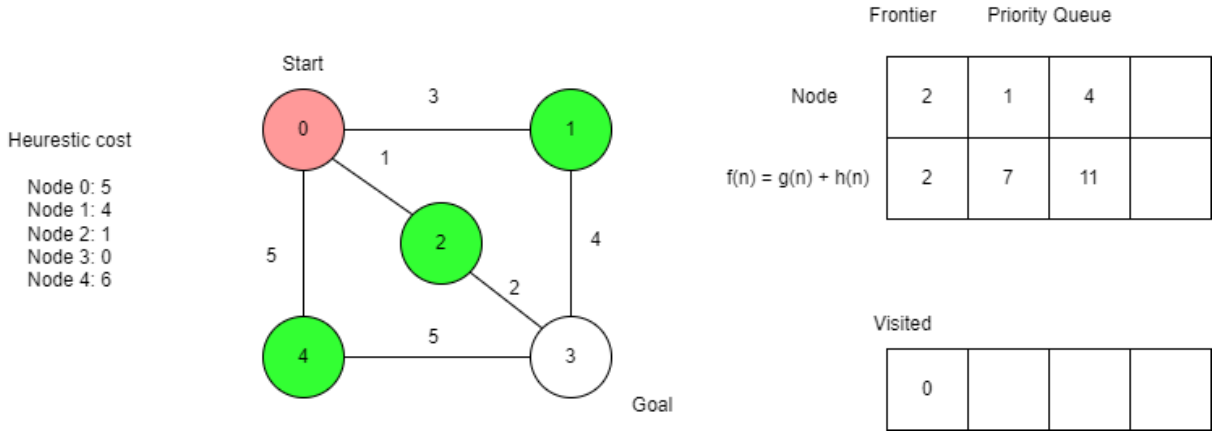


Figure 17 : Add neighbors to frontier with the following f(n) value and mark the visited node (A*)

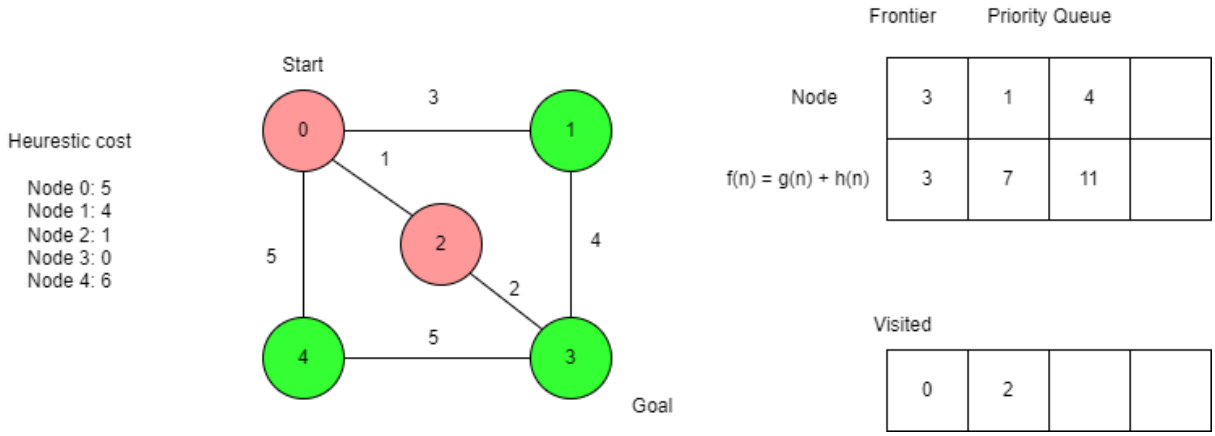


Figure 18 : Pop the front of the priority queue, mark it as visited and enqueue it's child node. (A*)

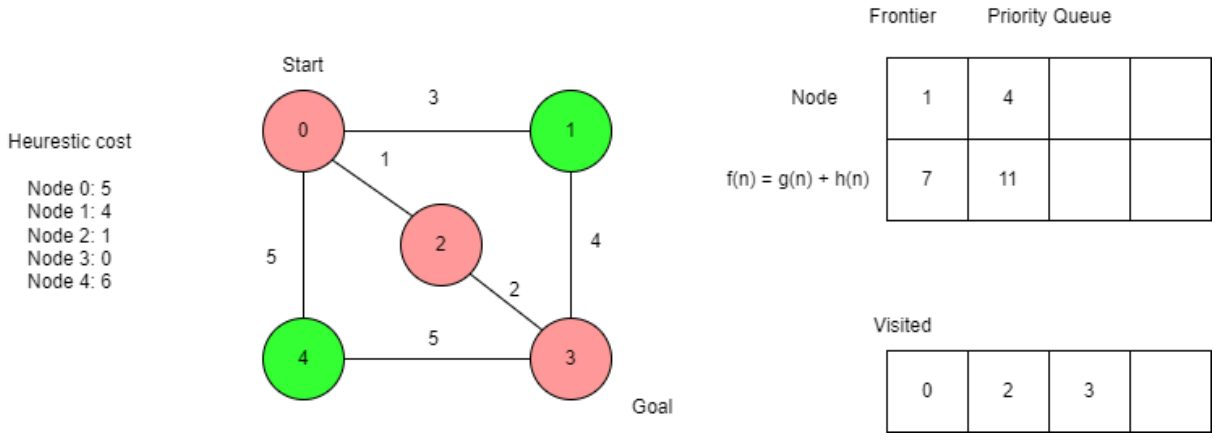


Figure 19 : The child node is goal so mark it as visited and the algorithm stops (A*)

7. Hill-climbing Search (HC):

Hill climbing is a simple optimization algorithm used in Artificial Intelligence (AI) to find the best possible solution for a given problem. It belongs to the family of local search algorithms and is often used in optimization problems where the goal is to find the best solution from a set of possible solutions. In Hill Climbing, the algorithm starts with an initial solution and then iteratively makes small changes to it in order to improve the solution. These changes are based on a heuristic function that evaluates the quality of the solution. The algorithm continues to make these small changes until it reaches a local maximum, meaning that no further improvement can be made with the current set of moves. [7]

Detailed description:

Initialization:

- Create the current node being explored, initialized to the start node with its heuristic cost.
- Mark the start node as visited.

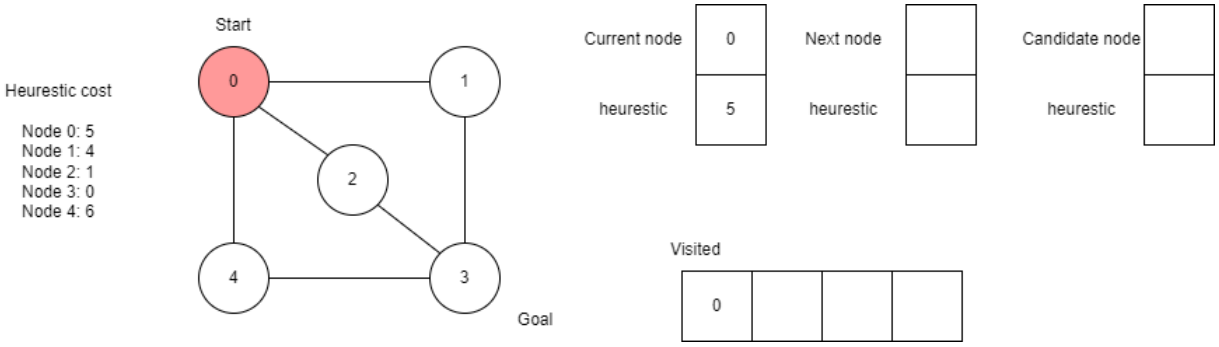


Figure 20 : Initialization (HC)

Exploration:

- While the neighbor(s) can be found:
    - Retrieving the neighbors.
    - Next node is initialized to None.
    - For each neighbors:
        - A candidate node is created representing the neighbor.
        - If the candidate node is the goal, the function returns the visited dictionary and the path found by the solution function.
        - If next node is None or the candidate node has a lower heuristic cost than next node, update next node.
        - If next node has a lower heuristic cost than current node:
            - Move to next node by setting it as current node.
            - Mark next node as visited.
        - If no better neighbor is found, the loop breaks.
- If the current node is the goal, return the path and the visited dictionary. If the goal is not reached, return -1.

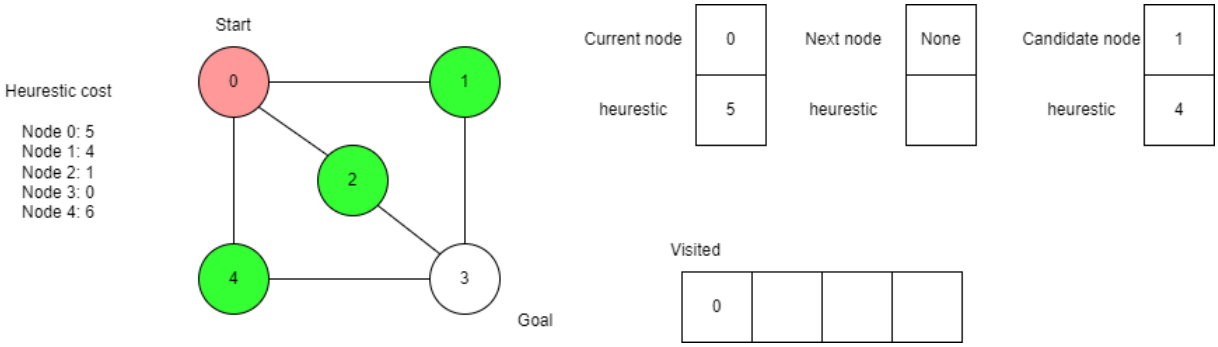


Figure 21 : Retrieving neighbors, initialize next node as None and candidate node as the representing neighbor (HC)

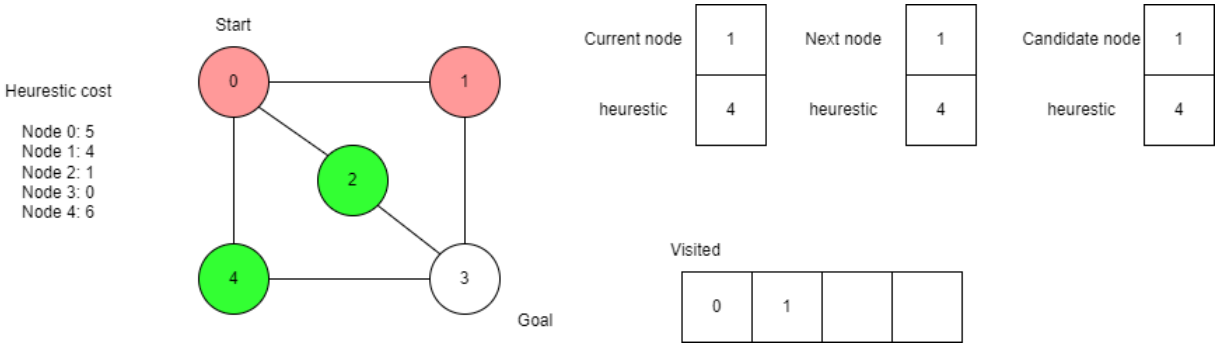


Figure 22 : Since next node is None and current node's heurestic is greater than candidate's node heurestic, current node and next node is replaced with candidate node and mark the correspoding node as visited (HC)

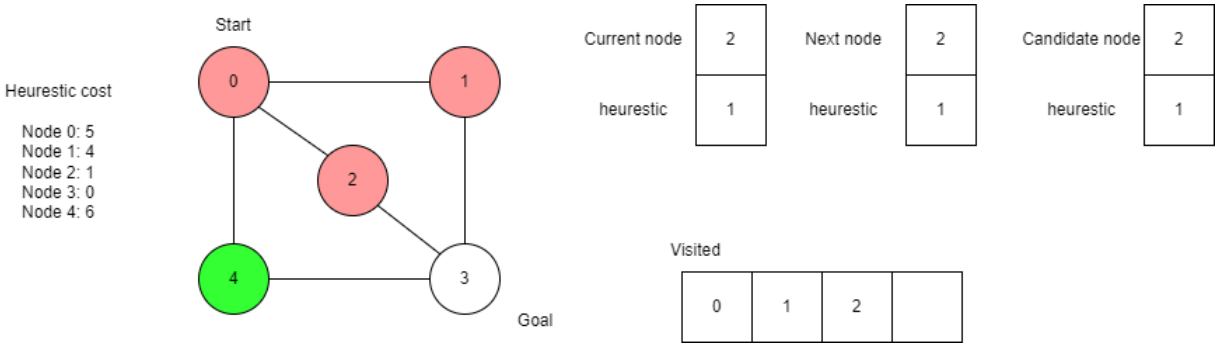


Figure 23 : Retrieving the next neighbor whose heurestic is lower than the current one and mark it as visited (HC)

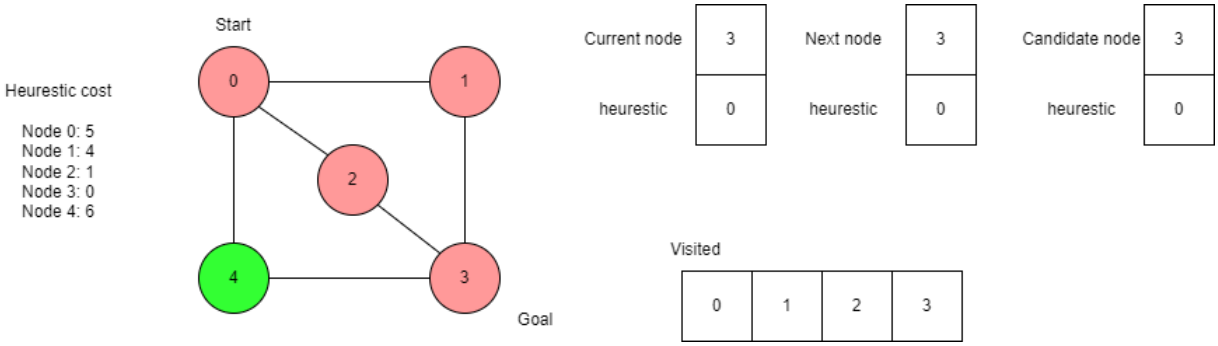


Figure 24 : The algorithms stop when the goal is reached (HC)

IV. Test cases and performance metrics:

1. Test case 1:

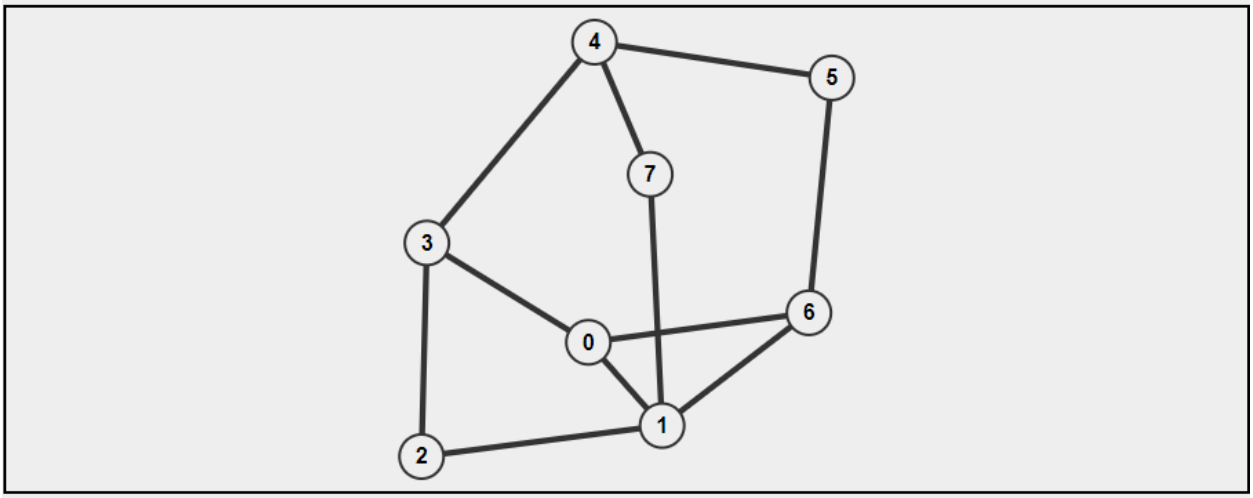


Figure 25 : Test case 01 (Non-weighted undirected graph)

Input	Heurestic	Start	Goal
8 2 5 0 1 0 1 0 0 1 0 1 0 1 0 0 0 1 1 0 1 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 5 7 10 8 5 0 6 3	Node 0: 5 Node 1: 7 Node 2: 10 Node 3: 8 Node 4: 5 Node 5: 0 Node 6: 6 Node 7: 3	2	5

Table 2 : Test case 01

Result:

```
BFS :
Path: 2 -> 1 -> 6 -> 5
Time: 0.1596000001882203 ms
Memory: 4.96875 KB
DFS :
Path: 2 -> 3 -> 4 -> 7 -> 1 -> 6 -> 5
Time: 0.1262999999198655 ms
Memory: 2.4716796875 KB
UCS :
Path: 2 -> 1 -> 6 -> 5
Time: 0.19419999989622738 ms
Memory: 3.5888671875 KB
IDS :
Path: 2 -> 1 -> 0 -> 6 -> 5
Time: 0.2936000000772765 ms
Memory: 6.9873046875 KB
GBFS :
Path: 2 -> 1 -> 7 -> 4 -> 5
Time: 0.13249999983599992 ms
Memory: 1.9326171875 KB
Astar :
Path: 2 -> 1 -> 6 -> 5
Time: 0.14740000005986076 ms
Memory: 2.3623046875 KB
HC :
Path: -1
Time: 0.07920000007288763 ms
Memory: 1.4560546875 KB
```

Figure 26 : Result of test case 01

Comment:

- Fastest runtime: Hill-climbing
- Slowest runtime: IDS
- Largest memory usage: IDS
- Lowest memory usage: Hill-climbing
- Explain: Hill-climbing has the fastest runtime and lowest memory usage because it stucks when traversing path. Meanwhile, IDS has the slowest runtime and largest memory usage due to traversing all nodes at each depth levels using recursive method.

2. Test case 2:

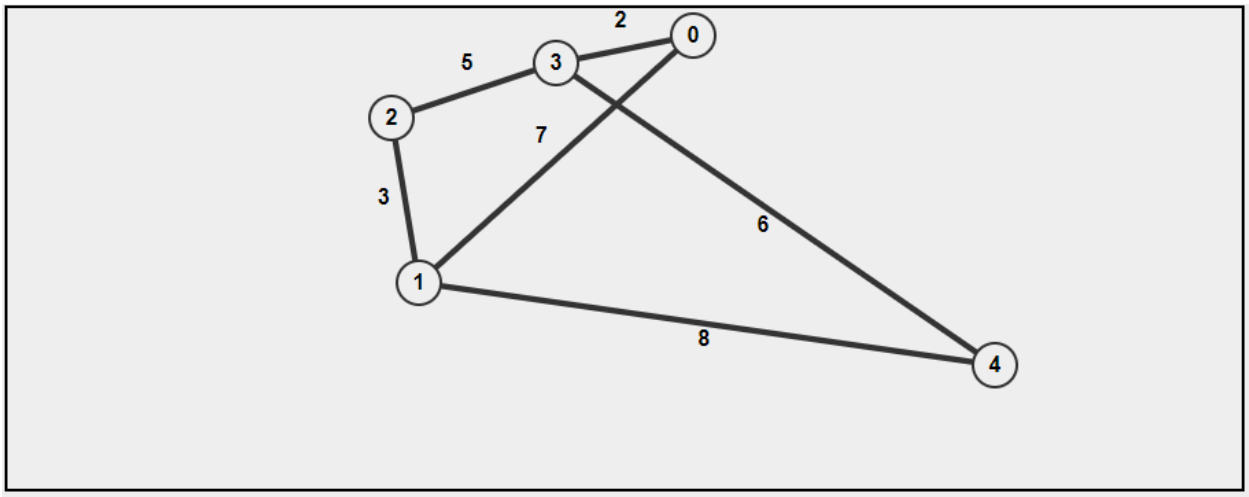


Figure 27 : Test case 02 (Weighted undirected graph)

Input	Heurestic	Start	Goal
5 2 4 0 7 0 2 0 7 0 3 0 8 0 3 0 5 0 2 0 5 0 6 0 8 0 6 0 7 8 10 6 0	Node 0: 7 Node 1: 8 Node 2: 10 Node 3: 6 Node 4: 0	2	4

Table 3 : Test case 02

Result:

```
BFS :
Path: 2 -> 1 -> 4
Time: 0.1021999996737577 ms
Memory: 4.671875 KB
DFS :
Path: 2 -> 3 -> 4
Time: 0.060400000165827805 ms
Memory: 1.3623046875 KB
UCS :
Path: 2 -> 1 -> 4
Time: 0.11319999975967221 ms
Memory: 2.8466796875 KB
IDS :
Path: 2 -> 1 -> 4
Time: 0.15050000001792796 ms
Memory: 3.703125 KB
GBFS :
Path: 2 -> 3 -> 4
Time: 0.07849999974496313 ms
Memory: 1.4951171875 KB
Astar :
Path: 2 -> 1 -> 4
Time: 0.19720000000233995 ms
Memory: 1.7841796875 KB
HC :
Path: 2 -> 3 -> 4
Time: 0.09690000024420442 ms
Memory: 1.3466796875 KB
```

Figure 28 : Result of test case 02

Comment:

- Fastest runtime: DFS
- Slowest runtime: A*
- Largest memory usage: BFS
- Lowest memory usage: Hill-climbing
- Explain: DFS runs fastest because it runs to the deepest node, and luckily the goal node is the deepest node when the algorithm performs. A* runs slowest because of the heurestic quality, and it explores many unnecessary nodes. BFS has the largest memory usage since it traverses all nodes at each depth levels. Hill-climbing has the lowest memory usage since it only needs to keep track of the current node and the neighboring nodes being considered and once it moves to a new state, it discards the previous state, so the algorithm only stores the path from start node to goal node.

3. Test case 3:

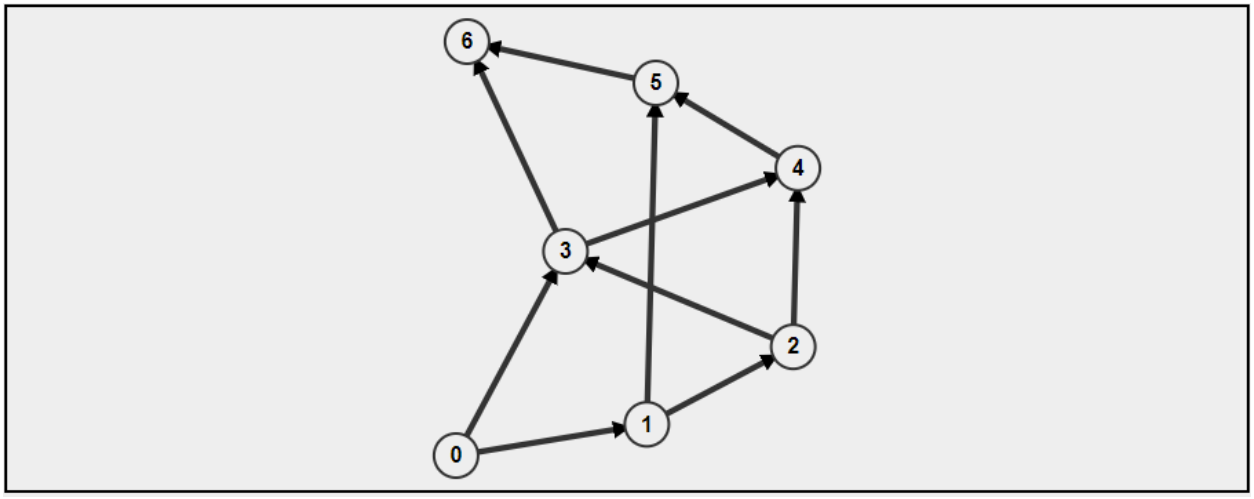


Figure 29 : Test case 03 (Non-weighted directed graph)

Input	Heurestic	Start	Goal
7 0 6 0 1 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 9 7 6 5 4 2 0	Node 0: 9 Node 1: 7 Node 2: 6 Node 3: 5 Node 4: 4 Node 5: 2 Node 6: 0	0	6

Table 4 : Test case 03

Result:

```
BFS :
Path: 0 -> 3 -> 6
Time: 0.136099999963335 ms
Memory: 4.96875 KB
DFS :
Path: 0 -> 3 -> 6
Time: 0.07810000033714459 ms
Memory: 1.3623046875 KB
UCS :
Path: 0 -> 3 -> 6
Time: 0.24149999990186188 ms
Memory: 2.8935546875 KB
IDS :
Path: 0 -> 3 -> 6
Time: 0.12570000035339035 ms
Memory: 4.0107421875 KB
GBFS :
Path: 0 -> 3 -> 6
Time: 0.07479999976567342 ms
Memory: 1.4951171875 KB
Astar :
Path: 0 -> 3 -> 6
Time: 0.08470000011584489 ms
Memory: 1.5419921875 KB
HC :
Path: 0 -> 3 -> 6
Time: 0.0542000002496934 ms
Memory: 1.3466796875 KB
```

Figure 30 : Result of test case 03

Comment:

- Fastest runtime: Hill-climbing
- Slowest runtime: UCS
- Largest memory usage: BFS
- Lowest memory usage: Hill-climbing
- Explain: Hill climbing has the fastest runtime and lowest memory usage since it only needs to keep track of the current node and the neighboring nodes being considered and once it moves to a new state, it discards the previous state, so the algorithm only stores the path from start node to goal node. UCS runs slowest because it may need to explore many different path to make sure it finds the optimal one, and this graph has no weight while UCS calculates the path cost. BFS has the largest memory usage since it traverses all nodes at each depth levels.

4. Test case 4:

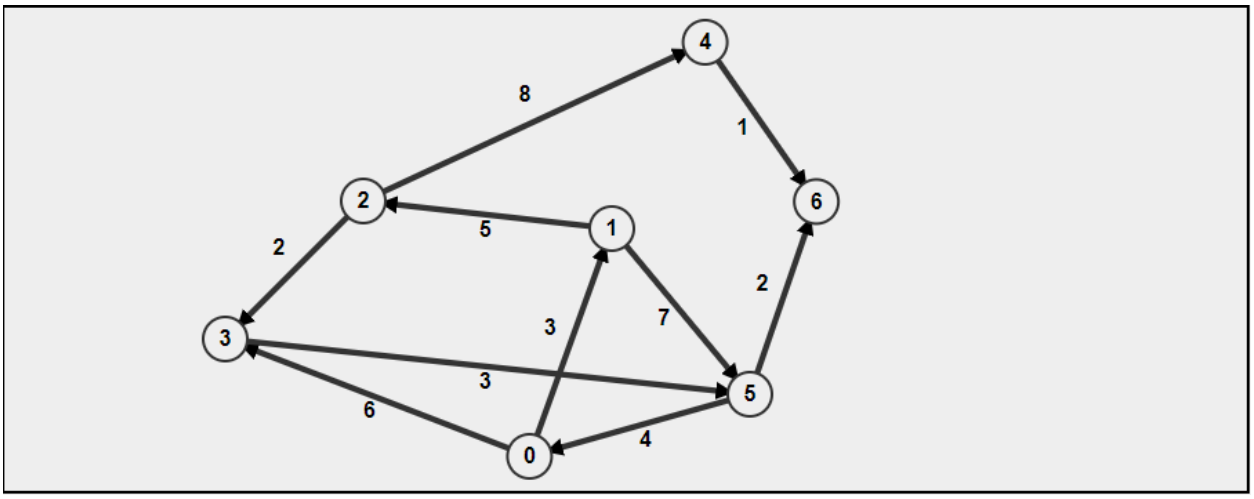


Figure 31 : Test case 04 (Weighted directed graph)

Input	Heurestic	Start	Goal
7 0 6 0 3 0 6 0 0 0 0 0 5 0 0 7 0 0 0 0 2 8 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 1 4 0 0 0 0 0 2 0 0 0 0 0 0 0 10 8 6 5 3 2 0	Node 0: 10 Node 1: 8 Node 2: 6 Node 3: 5 Node 4: 3 Node 5: 2 Node 6: 0	0	6

Table 5 : Test case 04

Result:

```
BFS :
Path: 0 -> 1 -> 5 -> 6
Time: 0.14580000015484984 ms
Memory: 4.8203125 KB
DFS :
Path: 0 -> 3 -> 5 -> 6
Time: 0.07270000014614197 ms
Memory: 1.4482421875 KB
UCS :
Path: 0 -> 3 -> 5 -> 6
Time: 0.2108999997290084 ms
Memory: 2.7841796875 KB
IDS :
Path: 0 -> 1 -> 5 -> 6
Time: 0.21090000018375576 ms
Memory: 5.3544921875 KB
GBFS :
Path: 0 -> 3 -> 5 -> 6
Time: 0.08389999993596575 ms
Memory: 1.6044921875 KB
Astar :
Path: 0 -> 3 -> 5 -> 6
Time: 0.11610000001383014 ms
Memory: 1.9560546875 KB
HC :
Path: 0 -> 3 -> 5 -> 6
Time: 0.06660000008196221 ms
Memory: 1.4560546875 KB
```

Figure 32 : Result of test case 04

Comment:

- Fastest runtime: Hill-climbing
- Slowest runtime: IDS
- Largest memory usage: IDS
- Lowest memory usage: DFS
- Explain: Hill climbing has the fastest runtime since it only needs to keep track of the current node and the neighboring nodes being considered and once it moves to a new state, it discards the previous state, so the algorithm only stores the path from start node to goal node. IDS has the slowest runtime and largest memory usage due to traversing all nodes at each depth levels using recursive method. DFS has lowest memory usage because it runs to the deepest node, and luckily the goal node is the deepest node when the algorithm performs.

5. Test case 5:

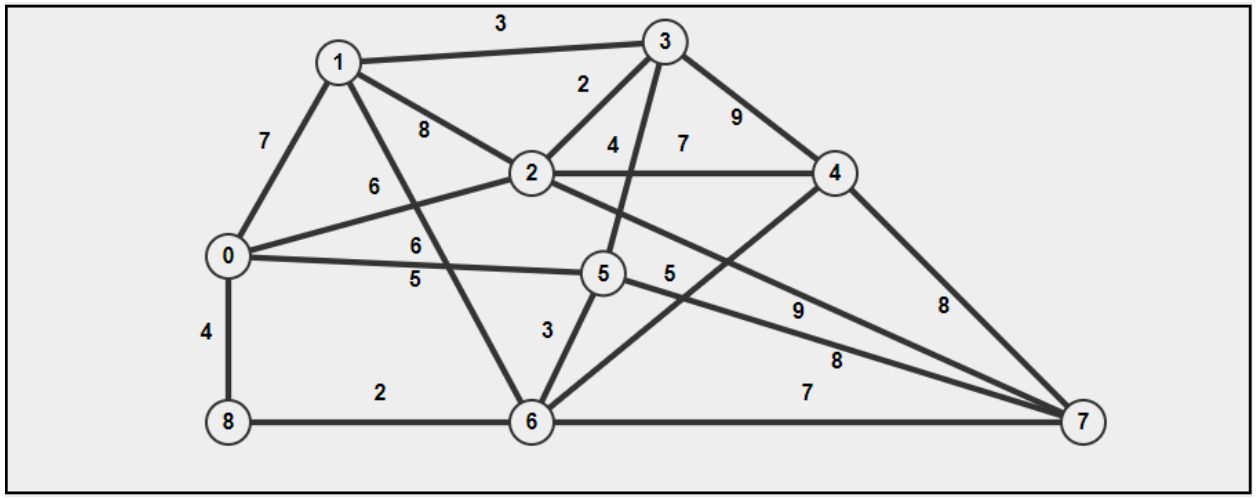


Figure 33 : Test case 05 (Big weighted undirected graph)

Input	Heurestic	Start	Goal
9 0 7 0 7 6 0 0 5 0 0 4 7 0 8 3 0 0 6 0 0 6 8 0 2 7 0 0 9 0 0 3 2 0 9 4 0 0 0 0 0 7 9 0 0 5 8 0 5 0 0 4 0 0 3 8 0 0 6 0 0 0 3 0 7 2 0 0 9 0 8 0 7 0 0 4 0 0 0 0 0 2 0 0 11 9 7 6 5 8 10 0 12	Node 0: 11 Node 1: 9 Node 2: 7 Node 3: 6 Node 4: 5 Node 5: 8 Node 6: 10 Node 7: 0 Node 8: 12	0	7

Table 6 : Test case 05

Result:

```
BFS :
Path: 0 -> 2 -> 7
Time: 0.2051999999821419 ms
Memory: 5.265625 KB
DFS :
Path: 0 -> 8 -> 6 -> 7
Time: 0.14850000025035115 ms
Memory: 1.6513671875 KB
UCS :
Path: 0 -> 8 -> 6 -> 7
Time: 0.2262000002701825 ms
Memory: 4.5263671875 KB
IDS :
Path: 0 -> 5 -> 7
Time: 0.2137000001312117 ms
Memory: 4.2998046875 KB
GBFS :
Path: 0 -> 2 -> 7
Time: 0.1729999999042775 ms
Memory: 1.765625 KB
Astar :
Path: 0 -> 5 -> 7
Time: 0.25860000005195616 ms
Memory: 2.3203125 KB
HC :
Path: 0 -> 2 -> 7
Time: 0.1428999999006919 ms
Memory: 1.4873046875 KB
```

Figure 34 : Result of test case 05

Comment:

- Fastest runtime: Hill-climbing
- Slowest runtime: A*
- Largest memory usage: BFS
- Lowest memory usage: Hill-climbing
- Explain: Hill climbing has the fastest runtime and lowest memory usage since it only needs to keep track of the current node and the neighboring nodes being considered and once it moves to a new state, it discards the previous state, so the algorithm only stores the path from start node to goal node. A* runs slowest because of the heurestic quality, and it explores many unnecessary nodes. BFS has the largest memory usage since it traverses all nodes at each depth levels.

V. System descriptions and Programming notes:

1. Specifications of the testing system:

- Hardware specifications: Intel® Core™ i7-4600U CPU @ 2.10Hz (4 CPUs) ~ 2.7GHz
- Operating system : Windows 10 Pro 64-bit
- Coding language used : python 3.10.6
- IDE used : Visual Studio Code.

2. Measurement method:

- For measuring memory usage, used tracemalloc to trace memory allocations and identify where memory is being used in the code.
- For measuring runtime, used time to measure the execution time of the code.

3. Libraries used:

- deque: provides an implementation of double-ended queue data structure.
- numpy : handle the adjacency matrix.
- time : measuring time.
- tracemalloc : measuring memory used.
- heapq : provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

4. Functions and classes:

- class Node: Handle traversing by nodes in graph.
- bfs, dfs, ucs, dls, ids, gbfs, astar, hc : Implement searching algorithms.
- read_file: Read the adjacency matrix, heurestic values, start and goal node from the input file.
- execute_algorithm: Execute the searching algorithms.
- output_file: Write the algorithms' result to file.
- main: input the directory of the input and output file and calling the functions defined above.

5. Notes:

- There are 5 test cases I provided in the submitting folder (graph01.txt, graph02.txt,…) and an output file result.txt.
- To run these test cases, please change the value of the input_test variable in main function. (At the beginning of main function, for example, input_test = 'graph01'.)
- After executing, the result.txt will contain the result of the algorithms.

- If you want to input other test cases than the provided test case, input your test case's directory in the input_test variable. This works the same with the output file.

```
def main():
    input_test = 'graph05.txt'
    output_filename = 'result.txt'
```

Figure 35 : Input and output directory

VI. References:

[1] Bui Tien Len, lect-03-uninformed search, p. 5.
[2] https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/ , Accessed date : July 7th 2024.
[3] https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/ , Accessed date : July 7th 2024.
[4] https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search , Accessed date : July 8th 2024.
[5] https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/ , Accessed date : July 8th 2024.
[6] https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search , Accessed date : July 9th 2024.
[7] https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/ , Accessed date : July 9th 2024.