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

LAB PRACTICE #1: SEARCHING

INTRODUCTION TO ARTIFICIAL INTELLIGENCE – CSC14003

—o0o—

INSTRUCTOR(S)

MS. NGUYỄN NGỌC THẢO
MR. NGUYỄN THANH TÌNH
MS. HỒ THỊ THANH TUYỀN

—o0o—

STUDENT'S INFORMATION

FULL NAME: LÊ PHƯỚC PHÁT
CLASS: 22CLC10
STUDENT'S ID: 22127322
STUDENT'S EMAIL: lpphat22@clc.fitus.edu.vn

HO CHI MINH CITY, JULY 2024

INTRODUCTION

In today's rapidly evolving technological landscape, the ability to efficiently search and navigate through vast amounts of data is a critical skill for computer scientists and artificial intelligence practitioners. This lab report documents the implementation and comparative analysis of various graph search algorithms, providing an in-depth exploration of their operational intricacies and performance metrics.

Graph search algorithms are foundational techniques in the field of Artificial Intelligence (AI) and are widely used in diverse applications such as route planning, puzzle solving, and network analysis. This lab focuses on seven essential algorithms: Breadth-first search (BFS), Tree-search Depth-first search (DFS), Uniform-cost search (UCS), Iterative deepening search (IDS), Greedy best-first search (GBFS), Graph-search A* (A*), and a variant of Hill-climbing (HC). Each of these algorithms offers unique approaches to traversing and exploring graph structures, highlighting different strengths and trade-offs in terms of efficiency, memory usage, and practical applicability.

The primary objective of this lab is to implement these algorithms, perform path searches on given graphs, and systematically compare their performance. By reading input data from a file and writing the results to an output file, we ensure a structured approach to evaluating each algorithm's efficacy. The lab not only emphasizes the correct implementation of these algorithms but also encourages critical analysis through the comparison of their runtime and memory consumption.

This report has been meticulously divided into complete parts, including the following:

[1] Teachers' comments & Pledge.

[2] Self-evaluation of the completion rate of the lab and other requirements.

[3] Describing and implementing all the various graph search algorithms.

- **Bread-first search (BFS)**
- **Tree-search depth-first search (DFS)**
- **Uniform-cost search (UCS)**
- **Iterative deepening search (IDS)**
- **Greedy best-first search (GBFS)** using the given heuristics
- **Graph-search A* (A*)** using the given heuristics
- **Hill-climbing (HC) variant**

[4] Complete and detailed description of the program system overview.

- Detailed description of the program system overview.
- Detailed description of all the supporting classes and functions in the program.
- Detailed description of all the main algorithms and functions in the programs.

[5] Experimental evaluation and comments

- **Experimental evaluation**: Generating and describing all test cases with different attributes and performance metrics of all algorithms. Showing all the results of all given test cases.
- **Experimental comments**: General and detailed comments for each algorithm and draw general conclusions.

[6] References

Through this lab, we gain valuable insights into the operational dynamics of graph search algorithms and their implications in the broader context of AI research and development.

s

I appreciate feedback and constructive criticism from my teachers who will be grading my project. This will help me identify areas for improvement and refine my system. I am very open to suggestions for enhancing my program.

Tuesday, July 9th, 2024

Lê Phước Phát

TABLE OF CONTENTS

TEACHERS' COMMENTS

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

Date: July …, 2024
Graded Teacher (s)

PLEDGE

I declare that this research was conducted by me, under the supervision and guidance of the teachers of the **Introduction to Artificial Intelligence – CSC14003** subject: Ms. **Nguyen Ngoc Thao**, Mr. **Nguyen Thanh Tinh**, and Ms. **Ho Thi Thanh Tuyen**. The results of this study are legal and have not been published in any form before. All documents used in this study were collected by myself and from various sources, and are fully listed in the references section. In addition, we also use the results of several other authors and organizations. All are properly cited. In case of copyright infringement, we are responsible for such action. Therefore, **Ho Chi Minh City University of Science (HCMUS)** is not responsible for any copyright violations committed in my research.

RESEARCH PROJECT

I. Self–evaluating the completion rate of the lab and other requirements

No.	General works	Detailed works	Completion rate (%)
1	BFS Algorithm	BFS detailed description (report)	100 %
		BFS implementation (code)	100 %
2	DFS Algorithm	DFS detailed description (report)	100 %
		DFS implementation (code)	100 %
3	UCS Algorithm	UCS detailed description (report)	100 %
		UCS implementation (code)	100 %
4	IDS Algorithm	IDS detailed description (report)	
		IDS implementation (code)	100 %
5	GBFS Algorithm	GBFS detailed description (report)	100 %
		GBFS implementation (code)	100 %
6	A* Algorithm	A* detailed description (report)	
		A* implementation (code)	100 %
7	Hill-climbing Algorithm	HC detailed description (report)	
		HC implementation (code)	100 %
8	Program System	Detailed description of the program system overview.	100 %
		Detailed description of all the supporting classes and functions in the program.	100 %
		Detailed description of all the main algorithms and functions in the programs.	100 %
9	Experiments (Report & Implementation)	Generating and describing all test cases with different attributes and performance metrics of all algorithms. Showing and analyzing all the results of all given test cases.	100 %
		General and detailed comments for each algorithm and draw general conclusions.	100 %

II. Detailed program system description

1. Program System Overview

In this **lab 1- searching algorithms**, this structure and layout as depicted in ***"Figure 1. The structure of source lab 1"*** is organized into a hierarchical directory format.

a. Root directory: **22127322**

This is the main project directory name after ***my student ID (22127322)***. It contains several subdirectories and files necessary for the project.

b. Subdirectory: **docs**
This directory will contain all documents for the whole project. The report named ***"22127322_Report.pdf"*** will include documentation, an explanation of all algorithms and program systems, experimental results, comments, and conclusions related to the lab on searching algorithms.

c. Subdirectory: **src**

- Subdirectory: **algorithms**
  This directory consists of all the source code files about all searching algorithms (*7 **searching algorithms***). All the files below about all searching algorithms will be described in detail in the part ***"III. Detailed algorithm description and implementation"***:
  - **"bfs.py"**: contains the implementation of the Breadth-First Search algorithm (BFS).
  - **"dfs.py"**: contains the implementation of the Depth-First Search algorithm (DFS).
  - **"ucs.py"**: contains the implementation of the Uniform-Cost Search algorithm (UCS).
  - **"ids.py"**: contains the implementation of the Iterative Deepening Search algorithm (IDS).
  - **"gbfs.py"**: contains the implementation of the Greedy Best-First Search algorithm (GBFS).
  - **"a_star.py"**: contains the implementation of the A* Search algorithm (A*).
  - **"hill_climbing.py"**: contains the implementation of the Hill Climbing algorithm (HC).
- Subdirectory: **common**
  This directory contains the supporting classes and functions that help to run all searching algorithms successfully. All the supporting classes and functions below will be described in detail in part ***II.2 - Supporting classes and functions***:
  - **"node.py"**: defines the Node class used by used by the search algorithms.
  - **"problem.py"**: defines the Problems class which represents the problem domain.
  - **"frontier.py"**: defines the frontier class (StackFrontier and QueueFrontier) used for BFS and DFS algorithms.
  - **"utils.py"**: contains utility functions such as: expand(), read_input(), write_output(), and reconstruct_path().
- Subdirectory: **test**

This subdirectory consists of all test case information which is dedicated to testing the implementation of the search algorithms. It is divided into two further subdirectories: **"input"** and **"output"**

- Subdirectory: **input**

  This subdirectory contains **6 test cases**, which **means 6 input files**:

  - **"input1.txt"**
  - **"input2.txt"**
  - **…**
  - **"input6.txt"**

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

  - The first line represents the number of nodes in the graph.
  - The second line consists of two integers, which means the initial state and the goal state, respectively.
  - The subsequent lines contain an adjacency matrix representing the graph where the value at row *i* and column *j* represents the cost of an edge between nodes *i* and *j*.
  - The final line represents a list of heuristic values for each node, used by heuristic-based algorithms such as A* and GBFS algorithms.

  The structure of these files ensures that the algorithms can be tested against a variety of scenarios, evaluating their performance and correctness.

  <u>Example:</u>

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

  <u>Note:</u> For each test case, it will be described in detail about the structure, layout of the given graph, and the way to find out the results in part ***"IV. Experimental evaluation and comments"***.

- Subdirectory: **output**

  This subdirectory contains **6 output files**, which means **6 results for 6 test cases** above in the subdirectory **"input":**
  - **"output1.txt"**
  - **"output2.txt"**
  - **…**
  - **"output6.txt"**

  These files store the results generated by running the search algorithms on the corresponding input files. Each output file typically contains:
  - The name of the algorithm.
  - The path found by the algorithm from the initial state to the goal state. If no path is found, it will be '-1'.
  - The time taken by the algorithm to find the path.
  - The memory usage during the execution of the algorithm.

  Example:

```
BFS:
Path: 0 -> 1 -> 3
Time: 0.0000003 seconds
Memory: 8 KB
...
Hill-climbing:
Path: 0 -> 2 -> 3
Time: 0.0000003 seconds
Memory: 8 KB
```

  Note: For each result of the test case, it will be used for analyzing and comparing the performance and efficiency of different search algorithms based on runtime and memory usage. All results and comments about them are described in detail in part ***"IV. Experimental evaluation and comments"***.

d. Files in the **Root Directory**
- **main.py:** This is the main script to execute the search algorithm. It might include main functions such as ***measure_time ()***, ***run_algorithm ()***, and ***main ()***.
- **requirements.txt:** This file lists the dependencies required to run the project. It might include libraries and packages that need to be installed for the code to work.

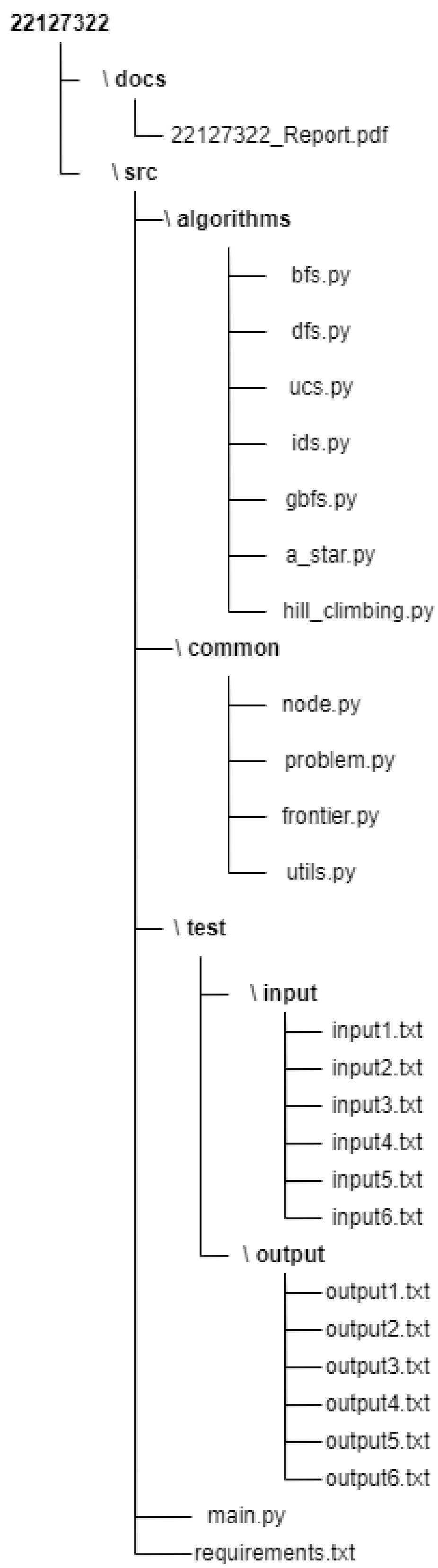

Figure 1. The structure of source lab 1. Searching

2. Supporting classes and functions

a. Class Node (...)

Implementation file: node.py

```
class Node
    method __init__ (self, state, parent = None, action = None,
path_cost = 0, heuristic = 0):
        self.state ← state
        self.parent ← parent
        self.action ← action
        self.path_cost ← path_cost
        self.heuristic ← heuristic
    method __lt__ (self, other):
        return (self.path_cost + self.heuristic) < (other.path_cost +
other.heuristic)
```

Pseudo-code #1. Class Node (…)

The class **"Node"** is designed to represent a single node in graph search algorithms. This class encapsulates the properties and behaviors of nodes within the state space of any search strategy.

The **constructor method __init__** initializes a new instance of the "Node" class with its **state**, **parent**, **action**, **path_cost**, and **heuristic** attributes of the node based on the provided arguments, which means that:

- **"state"**: represents the specific state of the node within the problem space. In this situation, the state of this node will be default in the initial of the problem in class Problems.
- **"parent"**: points to the parent node from which the current node was generated.
- **"action"**: stores the action that was applied to the parent node to generate the current node. It helps in tracking the sequence of actions taken to reach the current state.
- **"path_cost"**: holds the cumulative cost of the path from the initial node to the goal node. In the initial step, it will default to 0.
- **"heuristic"**: represents the heuristic value of the current node. In the initial step, it will default to 0 for search algorithms not using heuristic values or in ***problem.heuristics[problems.initial]*** for one using heuristic values.

The **__lt__ method** is implemented to enable the less-than comparison for nodes. It is used primarily when nodes are stored in priority queues. The comparison is based on the sum of the path cost and the heuristic value of the nodes.

b. Class Problems (…)

Implementation file: problem.py

```
class Problems
    method __init__ (initial, goal, adjacency_matrix, heuristics)
        self.initial ← initial
        self.goal ← goal
        self.adjacency_matrix ← adjacency_matrix
        self.heuristics ← heuristics
    method is_goal(state) returns boolean
        return state == goal
    method actions(state) returns list of actions
        return list of indices i where adjacency_matrix[state][i] > 0
    method result (state, action) returns new_state
        return action
    method action_cost(s, action, s_prime) returns cost
        return adjacency_matrix[s][action]
```

Pseudo-code #2. Class Problems (…)

The **class "Problem"** encapsulates a problem space in which search algorithms operate. It provides the essential components and methods needed to define, navigate, and evaluate the problem.

The **constructor method __init__** initializes the **"Problems"** instance with attributes: **initial state**, **goal state**, **adjacency matrix**, and **heuristic values**. This method sets up the problem space for the search algorithms.

- **"initial"**: is the initial state of the problem, representing the starting node for the searching graph.
- **"goal"**: is the goal node of the searching graph, which the search aims to reach.
- **"adjacency_matrix"**: is a matrix representing the graph, each element at "adjacency_matrix[i][j]" indicates the cost of a directed path from node "i" to node "j". For example, if **"adjacency_matrix[1][2] = 5",** it means there's a path from node 1 to node 2 with a cost of 5.
- **"heuristics"**: is a list where each element that represents the heuristic value (estimated cost) from a current node to the goal node. This "heuristics" is used by search algorithms like A* or GBFS.

The **"is_goal" method** will check if the given node is the goal node. This method returns **"True"** if "state" equals "goal", otherwise returns **"False"**.

The **"actions" method** will return a list of possible actions (successors of the node (state)) from the current node (state). It finds all indices i in "adjacency_matrix[state]" where the value is greater than 0.

The **“result” method** will return the new state resulting from applying the given action to the current state. In this implementation, the action directly corresponds to the new state, simplifying the result computation.

The **“action_cost” method** retuns the cost associated with moving from state “s” to state “s_prime” by taking the given action. This cost is retrieved from the adjacency matrix, which provides the cost of the edge between the states.

c. *Class StackFrontier(…) and class QueueFrontier(…)*
Implementation file: frontier.py

```
class StackFrontier
    method __init__ ()
        frontier ← empty list
    method add(node)
        append node to frontier
    method contains_state(state) returns boolean
        return any node.state == state for each node in frontier
    method empty () returns boolean
        return length of frontier == 0
    method remove () returns node
        if empty ()
            raise Exception ("empty frontier")
        else
            node ← last element in frontier
            remove last element from frontier
            return node
```

Pseudo-code #3. Class StackFrontier (…)

The **“StackFrontier” class** ís a data structure that implements a stack-based frontier for DFS algorithm. A stack frontier follows the **Last – In – First – Out (LIFO)** principle, where the most recently added element is the first to be removed.
The **constructor method __init__** initializes an empty list (frontier) to represent the stack. This list (frontier) will store the nodes that are added to the frontier.
The **“add” method** will add a current node to the frontier. This operation appends the node to the end of the list, following the Last-In-First-Out (LIFO) principle of a stack.
The **“contains_state” method** checks if any nodes in the frontier contain the specified state. This method returns “True” if any node in the frontier has the same state as the given state, otherwise, it returns “False”.

The **"empty" method** checks if the frontier is empty. This method returns "True" if the list (frontier) is empty, which means that the length of the list (frontier) equals 0, otherwise, it returns "False".
The **"remove" method** removes and returns the last node added to the frontier. Before attempting to remove a node, it checks if the frontier is empty. If it is, an exception is raised with the message "empty frontier". If the frontier is not empty, it uses the "pop" method to remove and return the last node from the list.

Note: The class **"StackFrontier"** is used only for the **DFS algorithm**.

```
class QueueFrontier inherits StackFrontier
    method remove () returns node
      if empty ()
        raise Exception ("empty frontier")
      else
        node ← first element in frontier
        remove first element from frontier
   return node
```

Pseudo-code #4. Class QueueFrontier (…)

The **"QueueFrontier" class** is a specialized data structure that inherits the **"StackFrontier" class**, which operates as a queue instead of a stack. This means it has all the methods and attributes of the **"StackFrontier" class**, but it overrides the **"remove" method** to implement queue behavior. This class is designed for implementing BFS, where nodes are explored in the order they are added based on the **First-In-First-Out (FIFO)** principle.

The **"remove" method** first checks if the **"frontier"** list is empty by calling the **"empty" method** inherited from **"StackFrontier" class**. If the **"frontier"** is empty, it raises an exception with the message **"empty frontier"**. This indicates that there are no nodes left to remove, and the search cannot proceed. If the **"frontier"** is not empty, the method proceeds to remove and return the first node added to the frontier, following the **FIFO principle**.

Note: The class **"QueueFrontier"** is used only for the **BFS algorithm**.

d. *Function: expand (…)*
   Implementation file: utils.py

```
function expand (problem: Problems, node : Node) returns a
generator of child nodes
    s ← node.state
    for each action in problem.actions(s) do
        s_prime ← problem.result (s, action)
        path_cost ← node.path_cost + problem.action_cost (s, action,
s_prime)
        heuristic ← problem.heuristics[s_prime]
       child ← Node (state = s_prime, parent = node, action =
action, path_cost = path_cost, heuristic = heuristic)
       yield child
```

Pseudo-code #5. expand (…) function

The **"expand"** function generates the successors (child nodes) from a given node in the context of a search problem, which means that it is exploring the search space by applying possible actions to the current state and generating the subsequent states.

This function takes two arguments:

- **"problem"**, an instance of the "Problems" class, which encapsulates the problem-specific details as above description about the "Problems" class.
- **"node"**, an instance of the "Node" class, representing the current state in the graph search, as above description about the "Node" class.

The **"expand"** function starts by extracting the current state from the node. For each possible action from this state, it calculates the resulting state. The new path cost is computed by adding the cost of the action to the current node's path cost. The heuristic value for the new state is also retrieved. A new child node is then created with:

- **"state"** is the resulting state after applying the action.
- **"parent"** is the current node for this child node
- **"action"** is the action that leads to this state of the chide node.
- **"path_cost"** is the new path cost above.
- **"heuristic"** is the heuristic value for the new state.

Finally, the child node is yielded to be used in further exploration.

This function facilitates the exploration of the search space in a structured and efficient manner.

e. *Function: reconstruct_path (…)*
Implementation file: utils.py

```
function reconstruct_path (node: Node) returns path
   path ← an empty list
   while node is not null do
      append node.state to path
      node ← node.parent
   end while
   reverse path
   return path
```

Pseudo-code #6. reconstruct_path (…) function

The **"reconstruct_path"** function will reconstruct the path (the sequence of states) from the initial state to the goal state by following the parent links from the goal node back to the initial node.
This function takes a node (assumed to be the goal node) as its argument. It starts by initializing an empty list named "path" to store the sequence of states that form the path from the initial node (state) to the goal node (state). The function then enters a loop that continues as long as the current node is not **None**. In each iteration of the loop, the current node's state is added to the **"path"** list, and the function moves to the parent of the current node. This process effectively traces the path backward from the goal state to the initial state by following the chain of parent pointers.
Once the loop has completed, the **"path"** list contains the states from the goal state to the initial state, in reverse order. Therefore, the function reverses the **"path"** list to obtain the correct order, from the initial state to the goal state.
Finally, the function returns the reconstructed path as a list of states.

f. *Function: read_input (…)*
Implementation file: utils.py

The **"read_input" function** reads an input file in folder **"test/input"** and constructs an instance of the **"Problems" class**. This function takes the path of the input file (the name of the input file) as its argument. It begins by opening the given input file for reading and loading all lines into the "lines" list. After reading the input file, it is closed to free up resources.

The first line (line[0]) of the file indicates the number of nodes in the problem graph, which is converted to an integer and stored in **"num_nodes"**. The second line (line[1]) contains the start and goal nodes, which are split into two integers and assigned to **"start"** and **"goal"**, respectively.

Subsequently, the function reads the adjacency matrix which represents the graph's connectivity, from lines **2** to **2 + num_nodes - 1**. Each line within this range is split into a list of integers and appended to the **adjacency_matrix** list. Following the adjacency matrix, the heuristic values, utilized in heuristic search algorithms, are read from the next line (lines[2 + num_nodes]) and converted into a list of integers. Finally, the function returns an instance of the **"Problems" class**, initialized with the start state, goal state, adjacency matrix, and heuristic values.

```
function read_input (file_path: str) returns Problems
    file ← open file_path for reading
    lines ← read all lines from file
    close file

    num_nodes ← convert lines [0] to integer
    start, goal ← convert lines [1] to two integers
    adjacency_matrix ← empty list

    for i ← 2 to 2 + num_nodes - 1 do
        row ← convert lines[i] to list of integers
        append row to adjacency_matrix
        heuristics ← convert lines [2 + num_nodes] to list of integers

    return Problems (initial=start, goal=goal,
adjacency_matrix=adjacency_matrix, heuristics=heuristics)
```

Pseudo-code #7. read_input (…) function

g. *Function: write_output (…)*
Implementation file: utils.py
The **"write_output" function** writes the results of search algorithms to an output file. This function takes two arguments: **output_file**, which specifies the path to the output file, and **results**, a dictionary containing the results of various search algorithms.
The function starts by opening the output file in write mode. For each algorithm in the **results** dictionary, it retrieves the algorithm's name, path, duration, and memory usage. It then writes the algorithm's name to the file, followed by the path. If the path is **None**, it writes "Path: -1" to indicate that no path was found. Otherwise, it writes the path, formatting it as a sequence of states separated by " -> ".
Next, the function writes the time taken by the algorithm to complete, formatted to eight decimal places, and the memory usage in kilobytes, formatted to two decimal places. Finally, it writes a separator line for readability before moving on to the next algorithm's results.

If any errors occur during the file writing process, the function raises a **ValueError** with a message indicating the issue. This error handling ensures that problems with writing the output file are properly reported.

```
function write_output (output_file: str, results: list)
    try
        file ← open output_file for writing
        for each (name, result) in results.items() do
            path, duration, memory_usage ← result
            write name to file
            if path is null
                write "Path: -1" to file
            else
                write "Path: " + join path with " -> " to file
            write "Time: " + format duration with 8 decimal places
        + " seconds" to file
            write "Memory: " + format memory_usage with 2
        decimal places + " KB" to file
             write "\n===============\n\n" to file
         end for
         close file
    catch Exception e
         raise ValueError ("Error writing output file " + output_file +
": " + e)
```

Pseudo-code #8. write_output (…) function

3. Main functions and algorithms

a. Function: run_algorithm (…)

Implementation file: main.py

```
function run_algorithm (algorithm, problem) returns path
   path ← algorithm(problem)
   return path
```

Pseudo-code #9. run_algorithm (…) function

The **"run_algorithm" function** serves as a wrapper to execute a specified search algorithm on a given problem instance. This function takes two arguments: **algorithm**, which is the search algorithm to be executed, and **problem**, an instance of the Problems class representing the specific problem to be solved.

Inside the function, the algorithm is executed on the problem by calling **algorithm(problem)**. The result of this execution is stored in the variable

path. This path represents the sequence of states from the initial state to the goal state as determined by the algorithm.
Finally, the function returns the path found by the algorithm.

b. *Function: measure_time (…)*
Implementation file: main.py

```
function measure_time (algorithm: dictionary, problem:
Problems) returns (path, duration, memory_usage)
   tracemalloc.start()
   start_time ← time.perf_counter()
   path ← run_algorithm (algorithm, problem)
   end_time ← time.perf_counter()
   duration ← end_time - start_time
   current, peak ← tracemalloc.get_traced_memory()
   tracemalloc.stop()
   return path, duration, peak / 1024
```

Pseudo-code #10. measure_time (…) function

The **“measure_time” function** is designed to measure both the execution time and memory usage of a specified search algorithm applied to a given problem instance.
This function takes two arguments: **“algorithm”**, the search algorithms to be executed, and **“problem”**, an instance of the “Problem” class representing the problem to be solved.
This function will use two specific libraries in Python: **tracemalloc** and **time**.

- **“tracemalloc.start()”**: initiates the tracing of memory allocations, which allows the function to monitor memory usage during the algorithm's execution.
- **“start_time ← time.perf_counter()”**: captures the current time in seconds before the algorithm starts. This high-resolution timer is used to measure the precise duration of the algorithm's run time.
- **“path”**: the **“run_algorithm” function** is called with the specified “algorithm” and “problem” executing the search algorithm on the problem instance. The resulting path from the initial state to the goal state is stored in the **path** variable.
- **“end_time = time.perf_counter()”**: captures the current time in seconds immediately after the algorithm finishes. This timestamp, combined with the start time, will be used to calculate the total duration of the algorithm's execution.
- **“duration = end_time - start_time”**: computes the total time taken by the algorithm by subtracting the start time from the end time.

- **"current, peak = tracemalloc.get_traced_memory()":** retrieves the current and peak memory usage during the algorithm's execution. The peak memory usage is of particular interest as it indicates the maximum memory consumed at any point during the run.
- **"tracemalloc.stop()"**: ends the tracing of memory allocations.

Finally, the function returns a tuple containing the **path** found by the algorithm, the **duration** of the execution in seconds, and the **peak** memory usage converted to kilobytes **(peak / 1024)**.

c. *Function: main (…)*
Implementation file: main.py

```
function main (input_file: str, output_file: str)
    try
        problem ← read_input (input_file)
    catch Exception e
        print ("Error reading input file " + input_file + ": " + e)
        return
    algorithms ← {
        "BFS": bfs,
        "DFS": dfs,
        "UCS": ucs,
        "IDS": ids,
        "GBFS": gbfs,
        "A*": a_star_search,
        "Hill-climbing": hill_climbing,
    }
    results ← empty dictionary
    for each (name, algorithm) in algorithms do
        try
            path, duration, memory_usage ← measure_time
        (algorithm=algorithm, problem=problem)
            results[name] ← (path, duration, memory_usage)
        catch Exception e
            print ("Error running " + name + " on " + input_file + ": "
        + e)
            results[name] ← (None, 0, 0)
    try
        write_output (output_file, results)
    catch Exception e
        print ("Error writing output file " + output_file + ": " + e)
```

Pseudo-code #11. main (…) function

The **"main" function** orchestrates the entire process of reading the problem, running various search algorithms, and writing the results to an output file.

III. Detailed algorithm description and implementation

1. Breadth-first search (BFS)

Implementation file: bfs.py

```
function BFS (problem: Problems) returns a solution path or failure
  node ← Node (state = problem.initial)
  if problem.is_goal(node.state) then
    return reconstruct_path(node)
  frontier ← QueueFrontier()
  add node to frontier
  reached ← set containing problem.initial
  while not frontier.empty() do
    node ← frontier.remove() // chooses the shallowest node in frontier
    for each child in expand (problem, node) do
      if problem.is_goal (child.state) then
          return reconstruct_path(child)
      if child.state not in reached and not frontier.contains_state(child.state)
      then
          add child.state to reached
          add child to frontier
  return failure
```

Pseudo-code #12. Breadth-first search algorithm (BFS)

The algorithm begins by initializing a node representing the initial state of the problem and checks if the initial state is the goal state. If it is, the algorithm reconstructs and returns the path to this goal (the supporting functions '**reconstruct_path**' as described in the above section). If not, the algorithm proceeds by initializing a frontier queue, which will store nodes to be explored, and adding the initial node to this queue. Additionally, a reached set is initialized to keep track of all states that have been explored to avoid redundant work.

The algorithm enters a loop that continues until there are no more nodes to explore, indicated by an empty frontier (as class **QueueFrontier()** will inherit class **StackFrontier(),** which has the described '**empty**' method, as in **Pseudo-code #2**). Within this loop, the algorithm removes a node from the frontier (following the First-In-First-Out, or FIFO, principle) because the frontier is **QueueFrontier()**, which has a '**remove**' method override. It then expands this node to generate its children nodes, which represent possible states the system can transition into from the current state. For each child node generated, the algorithm checks if its state is the goal state. If a goal state is found, the path to this state is reconstructed from the current node and returned as the solution.

If the child node's state is not the goal state, the algorithm checks if this state has already been reached or is present in the frontier. This ensures that each state is processed only once, preventing unnecessary precomputation and infinite loops. If the state of the child node is neither in the reached set nor in the frontier, it is added to the reached set, and the child node itself is added to the frontier for future exploration.

2. **Depth-first search (DFS)**
   Implementation file: dfs.py

```
function DFS (problem: Problems) returns a solution path or failure
    node ← Node (state = problem.initial)
            if problem.is_goal(node.state) then
               return reconstruct_path(node)
            frontier ← StackFrontier()
            add node to frontier
            while not frontier.empty() do
               node ← frontier.remove() // choose the deepest node in frontier
               for each child in expand (problem, node) do
                  if problem.is_goal(child.state) then
                     return reconstruct_path(child)
                  add child to frontier
   return failure
```

Pseudo-code #13. Tree-search depth-first search algorithm (DFS)

The **DFS algorithm** starts by creating a **node** with the initial state of the problem and checking if this node is the goal state. If the initial state is the goal, the algorithm reconstructs and returns the path to this goal state immediately.

If the initial state is not the goal, the algorithm proceeds by initializing a stack-based frontier, starting with the initial node. The stack-based frontier ensures that the algorithm explores the deepest nodes first, adhering to the **Last-In-First-Out** (LIFO) principle.

The algorithm enters a loop that continues until the frontier is empty, meaning there are no more nodes to explore. Within this loop, the algorithm removes the most recently added node from the frontier (the deepest node) for exploration.

For the node removed from the frontier, the algorithm expands it to generate its child nodes. For each child node generated, the algorithm checks if it represents the goal state. If a child node is the goal, the algorithm reconstructs and returns the path from the initial state to this goal state.

If a child node is not the goal, the algorithm adds this child node to the frontier for further exploration. This process continues, with the algorithm repeatedly exploring the deepest nodes, expanding them, and adding their children to the frontier until the goal is found or the frontier is empty.

If the frontier becomes empty and no goal state has been found, the algorithm returns a failure, indicating that no solution path exists for the given problem.

3. Uniform-cost search (UCS)

Implementation file: ucs.py

```
function UCS (problem: Problems) returns a solution path or failure
  start_node ← Node (state = problem.initial, parent = None, action = None,
path_cost = 0)
  frontier ← priority queue ordered by path_cost, initially containing (0,
start_node)
  reached ← dictionary with key problem.initial and value start_node
  while frontier is not empty do
     // chooses the lowest-cost node in the frontier, which means that it removes
the node with the lowest PATH_COST
     _, current_node ← POP (frontier)
     if problem.is_goal(current_node.state) then
       return reconstruct_path(current_node)
     for each child in expand (problem, current_node) do
       if child.state is not in reached or (child.path_cost <
reached[child.state].path_cost) then
          reached[child.state] ← child
          PUSH (frontier, (child.path_cost, child))
  return failure
```

Pseudo-code #14. Uniform-cost search algorithm (UCS)

The algorithm starts by creating a **start_node** with the initial state of the problem, no parent, no action, and a path cost of 0. This node is then added to a priority queue, called **frontier**, which orders nodes by their path costs. Additionally, a **reached** dictionary is initialized to keep track of the best-known path cost to each state, starting with the initial state mapped to the **start_node**.

The main loop of the **UCS algorithm** continues as long as the **frontier** is not empty. Within the loop, the node with ***the lowest path cost*** is removed from the **frontier** using **heappop**, and this node is referred to as **current_node**. If the state of **current_node** matches the goal state, the algorithm reconstructs and returns the path from the initial state to the goal state using the **reconstruct_path** function.

Next, the **current_node** is expanded to generate its children using the **expand** function. For each child node generated, the algorithm checks if the child's state has not been reached yet or if the path cost to this state is lower than the previously known path cost. If either condition is met, the **reached** dictionary is updated with the new lower path cost for this state, and the child is added to the **frontier** with its path cost as the priority using **heappush**.

If the goal state is not **reached** and the **frontier** becomes empty, the algorithm returns **failure**, indicating that no solution path was found. This ensures that the UCS algorithm always finds the optimal path to the goal, if one exists, by exploring the least-cost paths first.

4. **Iterative deepening search (IDS)**
   Implementation file: ids.py

```
function DLS (problem: Problems, limit: int) returns a solution path or
cutoff or failure
  return RECURSIVE-DLS (node = Node (problem.initial), problem,
limit)
function RECURSIVE_DLS (node: Node, problem: Problems, limit: int)
returns a solution path or cutoff or failure
  if problem.is_goal (node.state) then
    return reconstruct_path (node)
  else if limit = 0 then
    return cutoff // the signal represents no solution within the depth limit
  else
    cutoff_occurred ← false
    for each child in expand (problem, node) do
      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 // no solution for the whole graph search problem
```

Pseudo-code #15. Depth limited search algorithm (DLS)

The **DLS function** initializes the search by creating a node from the initial state of the problem and calling the recursive helper function **RECURSIVE_DLS**.

The **RECURSIVE_DLS function** takes a node (= **Node(state=problem.initial)**), the problem, and the current depth limit as arguments. It first checks if the current node's state is the goal state, returning the reconstructed path if it is. If the depth limit is zero, it returns a **"cutoff"** signal, indicating that the limit has been reached without finding a solution. If the depth limit is not zero, it initializes a flag **"cutoff_occurred"** to track whether any recursive calls result in a cutoff.

The function then expands the current node to generate its children (using the **“expand”** function) and recursively calls itself for each child, decrementing the depth limit by one. If any recursive call returns a **"cutoff"** signal, the **“cutoff_occurred”** flag is set to true. If a valid path is found (i.e., the result is not **"cutoff"** or **"failure"**), it is immediately returned.
After all children have been explored, if any recursive call resulted in a cutoff, the function returns **"cutoff"**; otherwise, it returns **"failure"**, indicating no solution was found within the depth limit

```
function IDS (problem ← Problems) returns a solution path or failure
    depth ← 0
    while true do
        result ← DLS (problem, depth)
        if result ≠ cutoff then
            return result
        depth ← depth + 1
```

Pseudo-code #16. Iterative deepening search algorithm (IDS)

The **IDS function** initializes a variable depth to **0**, which represents the current depth limit for the search. It enters an infinite loop, which ensures that the search will continue until a solution is found or all possible depths are exhausted.
Within the loop, the function calls the **DLS function**, passing the problem and the current depth limit as arguments. The **DLS function** performs a depth-first search up to the specified depth and returns one of three possible results:

- **a solution path** if the goal is found
- **"cutoff"** if the depth limit is reached without finding the goal
- **"failure"** if no solution exists within the depth limit.

The result of the **DLS function** is stored in the result variable. If the result is not **"cutoff"**, it means that either a solution path has been found or it has been determined that no solution exists within the current depth limit. In either case, the result is returned, ending the search.
If the result is **"cutoff"**, the depth limit is incremented by 1, and the loop continues with the new depth limit. This process of incrementing the depth limit and performing a depth-limited search is repeated until a solution is found or it is determined that no solution exists.

5. **Greedy best-first search (GBFS)**
Implementation file: gbfs.py

function **GBFS (problem: Problems)** **returns** a solution path **or** failure
node ← Node (state = problem.initial, parent = None, action = None, path_cost=0, heuristic = problem.heuristics[problem.initial])
frontier ← **a priority queue ordered by** heuristic**, with** node **as an element**
reached ← **a dictionary with key** problem.initial **and** value node
while frontier **is not** empty **do**
_, current_node ← **POP (frontier)** *// chooses the node with the* ***lowest*** *heuristic value from the frontier*
if problem.is_goal(current_node.state) **then**
return **reconstruct_path(current_node)**
for each child **in** **expand (problem, current_node)** **do**
if child.state **not in** reached **or** child.heuristic < reached[child.state].heuristic then
reached[child.state] ← child
add (child.heuristic, child) **to** frontier
return failure

Pseudo-code #17. Greedy best-first search algorithm (GBFS)

The algorithm starts by creating a **"start_node"** with the initial state, no parent, no action, a path cost of zero, and a heuristic value from the problem's heuristics in the **"Problems" class**.

This node **("start_node")** is placed in a priority queue **"frontier"** (implemented using **"heapq"**) that orders nodes by their heuristic values. A dictionary called **"reached"** is also initialized to keep track of the best heuristic value encountered for each state, starting with the initial state.

The main loop of the function continues until the **frontier** is empty. In each iteration, the node with the lowest heuristic value is popped from the frontier.

- If this node's state is the **goal** state (checks by using the **"is_goal" method** of the "Problems" class), the function reconstructs and returns the path from the initial state to the goal state using the **"reconstruct_path" function**.
- If the node's state is not the **goal** state, the function generates successors by expanding the current node, using the **"expand"** function. Each successor's state is evaluated, and if it has not been reached yet before or has a better heuristic value than previously recorded, which means that if the new heuristic is lower than the previously known heuristic to this state, the reached dictionary will be updated by this successor node's state with the new lower heuristic, and this node is added to the frontier with its heuristic as the priority.

If the loop exits without finding the goal state, the function returns **None**, indicating that no path was found.

This algorithm prioritizes nodes that appear closer to the goal based on heuristic values, aiming for efficient navigation through the state space.

6. **Graph-search A* (A*)**
Implementation file: a_star.py

```
function a_star (problem: Problems) returns a solution path or failure
    node ← Node (state = problem.initial, path_cost = 0, heuristic =
problem.heuristics[problem.initial])
    frontier ← a priority queue ordered by node.path_cost + node.heuristic,
with node as an element
    reached ← a dictionary with key problem.initial and value node
    while frontier is not empty do
        _, current_node ← POP (frontier) // chooses the node with the lowest
        sum of the path cost and heuristic value of this node.
        if problem.is_goal(current_node.state) then
            return reconstruct_path (current_node)
        for each child in expand (problem, current_node) do
            if child.state not in reached or child.path_cost <
        reached[child.state].path_cost then
                reached[child.state] ← child
                PUSH (frontier, (child.path_cost + child.heuristic, child))
    return failure
```

Pseudo-code #18. Graph-search A* (A*)

The algorithm starts by creating a **"start_node"** with the initial state of this node's problem, no parent, no action, a path cost of zero, and a heuristic value derived from the problem's heuristic function. This node is then placed in a priority queue ("frontier") ordered by the sum of its path cost and heuristic value. Additionally, a dictionary called **"reached"** is initialized to track the best path cost encountered for each state, starting with the initial state.

The main loop continues until the **"frontier"** is empty. In each iteration, the node with the lowest estimated total cost **(path cost + heuristic)** is popped from the frontier.

- If this node's state is the **goal** state (checks by using **"is_goal" method** of the "Problems" class, it returns ***"True"***), the function reconstructs and returns the path from the initial state to the goal state using the **"reconstruct_path" function**.
- If the current node's state is not the **goal** state (checks by using **"is_goal" method** of the "Problems" class, it returns ***"False"***), the function expands the current node to generate its children, using the "expand" function with the **"problem"** search state space and the **"current_node"**, which has the lowest sum of path cost and heuristic value. For each child node, it checks if the child state has not been reached before or if the new path cost is lower than the previously recorded path cost for that state. If either condition is

true, the child node is added to the **“reached”** dictionary and the **“frontier”** with its estimated total cost.

Finally, if the loop exits without finding the goal state, the function returns **None**, indicating that no path was found.

7. **Hill-climbing (HC) variant**

Implementation file: ***hill_climbing.py***

Function hill-climbing (problem: Problems) returns a solution path **or** failure
start_node ← **Node (state = problem.initial, parent = None, action = None, path_cost = 0, heuristic = problem.heuristics[problem.initial])**
current_node ← start_node
while true **do**
if problem.is_goal(current_node.state) **then**
return reconstruct_path (current_node)
neighbors ← **expand (problem, current_node)**
if neighbors **is** empty **then**
return failure
next_node ← **MIN (neighbors, heuristic)**
if next_node.heuristic ≥ current_node.heuristic **then**
return failure
current_node ← next_node

Pseudo-code #19. Hill-climbing variant (HC)

The **“hill-climbing” algorithm** attempts to find a path from an initial state to a goal state by iteratively moving to the neighbor with the lowest heuristic value. This algorithm begins by creating a **“start_node”** representing the initial state with no parent, no action, a path cost of zero, and a heuristic value derived from the problem's heuristic function. This node is set as the **“current_node”**.

The algorithm then enters a loop that continues indefinitely. In each iteration, it first checks if the **“current_node”** is the goal state.

- If it is the **goal** state (using the **“is_goal” method** in the “Problems” class), the function reconstructs and returns the path from the initial state to the goal state using the **“reconstruct_path”** function.
- If the **“current_node”** is not the **goal** state (using the **“is_goal” method** in the “Problems” class), the function expands the **“current_node”** to generate its neighbors using the **“expand” function**. If no neighbors are found, the algorithm returns **None**, indicating that it is stuck and cannot proceed further.

Next, the algorithm identifies the neighbor with the **lowest** heuristic value using the **“min”** function with a lambda function that extracts the heuristic value from each neighbor. If the neighbor with the **lowest** heuristic value has a heuristic that is greater than or equal to the current node's heuristic, the function returns **None**, indicating that it cannot find a better path and is therefore stuck.

If a neighbor with a lower heuristic value is found, the **"current_node"** is updated to this neighbor, and the loop continues. This process repeats until the goal state is found or no better neighbors are available.

IV. Experimental evaluation and comments

1. Experimental evaluation

With each specific case, I ran my experiment on about 6 test cases calculated running time, and consumed memory for each searching algorithm by using two Python libraries: **time** and **tracemalloc**.
I run my experiments on my laptop, with the following configuration:

- **Processor**: 12th Gen Intel(R) Core (TM) i5-12450H, 2000 MHz, 8 Core(s), 12 Logical Processor(s)
- **CPU clock**: 1.60 GHz

For each section below, I will consider all searching algorithms that the final path between the start node and the goal node is being found, and the running time and consumed memory will be calculated for each test case.
Note: In test case 1, I will describe in detail the way how to find out the path solution for each search algorithm. In the remaining test case, I just show you the result of the path, runtime, and memory usage for analyzing and comparing their performance and efficiency.

a. *Experiment 1 (test case 1)*

- *Detailed test case description*

In this experiment, we have the graph search with 8 nodes which this graph has multiple paths between the start node and the goal node. Now, I will describe the test case 1 input with some different attributes below:
Number of nodes: 8
Started node: 0
Goal node: 7
Adjacency matrix:

$$\begin{bmatrix} 0 & 3 & 5 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 0 & 14 \\ 0 & 1 & 0 & 0 & 4 & 8 & 0 & 0 \\ 0 & 0 & 2 & 0 & 4 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 12 & 16 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Heuristics:

$$[9 \quad 1 \quad 8 \quad 6 \quad 2 \quad 7 \quad 3 \quad 0]$$

- **File input1.txt**

```
8
0 7
0 3 5 7 0 0 0 0
0 0 0 0 5 0 0 14
0 1 0 0 4 8 0 0
0 0 2 0 4 3 0 0
0 0 0 0 0 7 9 0
0 0 0 0 0 0 12 16
0 0 0 0 0 0 0 10
0 0 0 0 0 0 0 0
9 1 8 6 2 7 3 0
```

Next, we will analyze this graph with some features:

- The provided graph is a **directed weighted** graph with **8 nodes** labeled from **0** to **7**. It depicts various connections between these nodes, where each directed edge has a specific weight, and each node has a heuristic value denoted in **blue**.
- The start node is **green**, and the goal node is **red**.

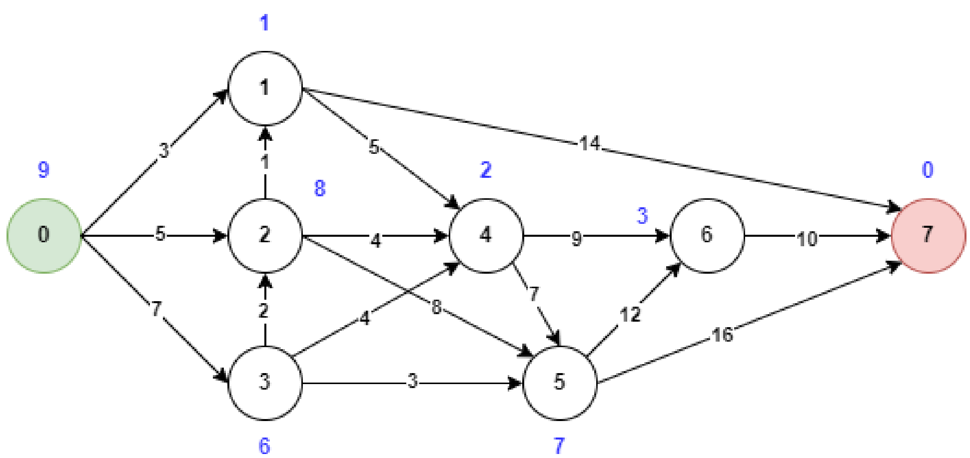

Figure 2. Graph for experiment 1 (test case 1)

- <u>*Detailed analysis of the algorithm*</u>
  - **Execution time**
    - **DFS** is the fastest with 0.00006840 seconds.
    - **A*** is the slowest with 0.00021600 seconds.
    - The other algorithms (BFS, UCS, IDS, GBFS, Hill-Climbing) have intermediate execution times, with not too significant differences.
  - **Memory usage**

- **Hill-Climbing** uses the least memory with 1.31 KB.
- **BFS** uses the most memory with 3.61 KB.
- The other algorithms have memory usage in between, with GBFS and IDS consuming less memory compared to BFS and DFS.

- **File output1.txt**

```
BFS:
Path: 0 -> 1 -> 7
Time: 0.00013000 seconds
Memory: 3.61 KB

DFS:
Path: 0 -> 3 -> 5 -> 7
Time: 0.00006840 seconds
Memory: 3.19 KB
UCS:
Path: 0 -> 1 -> 7
Time: 0.00011860 seconds
Memory: 2.36 KB

IDS:
Path: 0 -> 1 -> 7
Time: 0.00015280 seconds
Memory: 2.08 KB

GBFS:
Path: 0 -> 1 -> 7
Time: 0.00011130 seconds
Memory: 1.49 KB

A*:
Path: 0 -> 1 -> 7
Time: 0.00021600 seconds
Memory: 2.02 KB

Hill-climbing:
Path: 0 -> 1 -> 7
Time: 0.00010480 seconds
Memory: 1.31 KB
```

b. *Experiment 2 (test case 2)*

- *Detailed test case description*

In this experiment, we have the graph search with 6 nodes which this graph has multiple paths between the start node and the goal node. Now, I will describe the test case 2 input with some different attributes below:

Number of nodes: 6

Started node: 0
Goal node: 5
Adjacency matrix:

$$\begin{bmatrix} 0 & 2 & 3 & 0 & 0 & 0 \\ 2 & 0 & 0 & 5 & 2 & 0 \\ 3 & 0 & 0 & 0 & 5 & 0 \\ 0 & 5 & 0 & 0 & 1 & 2 \\ 0 & 2 & 5 & 1 & 0 & 4 \\ 0 & 0 & 0 & 2 & 4 & 0 \end{bmatrix}$$

Heuristics:

$$[\mathbf{4} \quad \mathbf{3} \quad \mathbf{9} \quad \mathbf{2} \quad \mathbf{5} \quad \mathbf{0}]$$

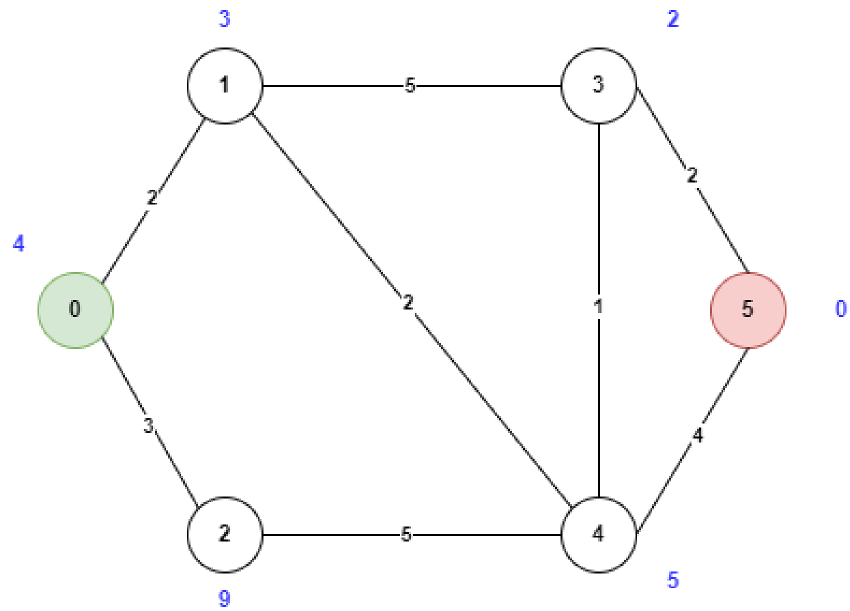

Figure 3. Graph for experiment 2 (test case 2)

- **File input2.txt**

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

Next, we will analyze this graph with some features:

- The provided graph is an **undirected weighted** graph with **6 nodes** labeled from **0** to **5**. It depicts various connections between these nodes, where each directed edge has a

specific weight, and each node has a heuristic value denoted in **blue**.
  - The start node is **green**, and the goal node is **red**
- *Detailed analysis of the algorithm*
  - **Execution time**
    - **GBFS** is the fastest with 0.00004830 seconds.
    - **BFS** is the slowest with 0.00011510 seconds.
    - The other algorithms (IDS, DFS, UCS, A*, Hill-Climbing) have intermediate execution times, with DFS and Hill-Climbing being notably faster.
  - **Memory usage**
    - **GBFS** uses the least memory with 1.49 KB.
    - **IDS** uses the most memory with 2.68 KB.
    - The other algorithms have memory usage in between, with UCS, A*, and Hill-Climbing consuming less memory compared to BFS, DFS, and IDS.
  - **File ouput2.txt**

```
BFS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00011510 seconds
Memory: 2.28 KB

DFS:
Path: 0 -> 2 -> 4 -> 5
Time: 0.00005540 seconds
Memory: 2.06 KB

UCS:
Path: 0 -> 1 -> 4 -> 3 -> 5
Time: 0.00007460 seconds
Memory: 1.72 KB

IDS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00011000 seconds
Memory: 2.68 KB

GBFS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00004830 seconds
Memory: 1.49 KB

A*:
Path: 0 -> 1 -> 4 -> 3 -> 5
Time: 0.00008090 seconds
Memory: 1.88 KB
```

```
Hill-climbing:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00005810 seconds
Memory: 1.59 KB
```

c. *Experiment 3 (test case 3)*

- *Detailed test case description*

In this experiment, we have the graph search with **6** nodes which this graph has multiple paths between the start node and the goal node. Now, I will describe the test case 3 input with some different attributes below:

Number of nodes: 6

Started node: 0

Goal node: 5

Adjacency matrix:

$$\begin{bmatrix} 0 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 5 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Heuristics:

$$\begin{bmatrix} 6 & 5 & 3 & 1 & 2 & 0 \end{bmatrix}$$

Next, we will analyze this graph with some features:

- The provided graph is an **undirected weighted** graph with **6 nodes** labeled from **0** to **5**. It depicts various connections between these nodes, where each directed edge has a specific weight, and each node has a heuristic value denoted in **blue**.
- The start node is **green**, and the goal node is **red**

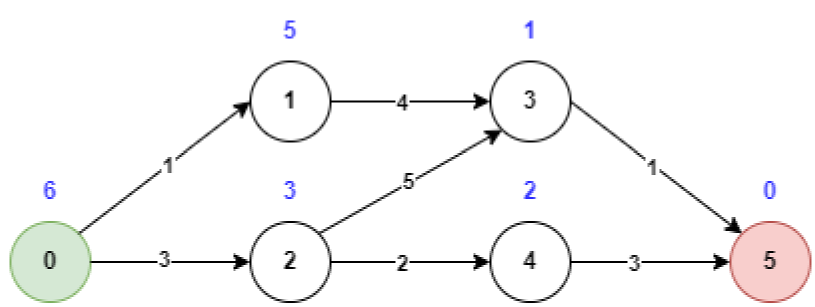

Figure 4. Graph for experiment 3 (test case 3)

- **File input3.txt**

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

- *Detailed analysis of the algorithm*
  - **Execution time**
    - Fastest: GBFS (0.00003660 seconds)
    - Slowest: BFS (0.00019460 seconds)
    - Moderate: A* (0.00004260 seconds), DFS (0.00004290 seconds), Hill-Climbing (0.00005070 seconds), UCS (0.00005740 seconds), IDS (0.00009190 seconds)
  - **Memory usage**
    - Lowest: Hill-Climbing (1.31 KB)
    - Highest: IDS (2.63 KB)
    - Moderate: GBFS (1.49 KB), A* (1.49 KB), UCS (1.55 KB), DFS (1.70 KB), BFS (2.38 KB)
  - **File output3.txt**

```
BFS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00019460 seconds
Memory: 2.38 KB

DFS:
Path: 0 -> 2 -> 4 -> 5
Time: 0.00004290 seconds
Memory: 1.70 KB

UCS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00005740 seconds
Memory: 1.55 KB

IDS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00009190 seconds
Memory: 2.63 KB

GBFS:
Path: 0 -> 2 -> 3 -> 5
```

```
Time: 0.00003660 seconds
Memory: 1.49 KB

A*:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00004260 seconds
Memory: 1.49 KB

Hill-climbing:
Path: 0 -> 2 -> 3 -> 5
Time: 0.00005070 seconds
Memory: 1.31 KB
```

d. *Experiment 4 (test case 4)*

- *Detailed test case description*

In this experiment, we have the graph search with **6** nodes which this graph has multiple paths between the start node and the goal node. Now, I will describe the test case 3 input with some different attributes below:

Number of nodes: 6

Started node: 0

Goal node: 5

Adjacency matrix:

$$\begin{bmatrix} 0 & 3 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 7 \\ 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 7 & 4 & 0 \end{bmatrix}$$

Heuristics:

$$[13 \quad 11 \quad 12 \quad 5 \quad 1 \quad 0]$$

Next, we will analyze this graph with some features:

- The provided graph is an **undirected weighted** graph with **6 nodes** labeled from **0** to **5**. It depicts various connections between these nodes, where each directed edge has a specific weight, and each node has a heuristic value denoted in **blue**.
- The start node is **green**, and the goal node is **red**

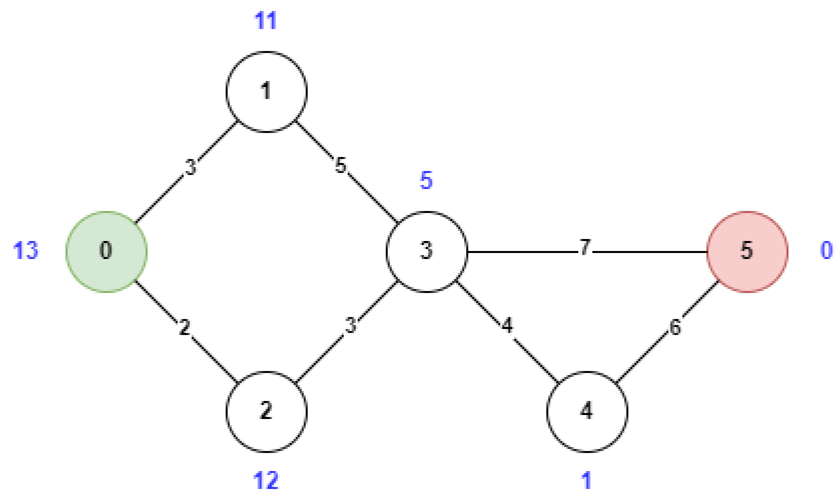

Figure 5. Graph for experiment 4 (test case 4)

- **File input4.txt**

```
6
0 5
0 3 2 0 0 0
0 0 0 5 0 0
0 0 0 3 0 0
0 0 0 0 4 7
0 0 0 0 0 6
0 0 0 7 4 0
13 11 12 5 1 0
```

- *Detailed analysis of the algorithm*
    - **Execution time**
        - Fastest: GBFS (0.00003790 seconds)
        - Slowest: BFS (0.00009590 seconds)
        - Moderate: DFS (0.00004080 seconds), Hill-Climbing (0.00004750 seconds), UCS (0.00005290 seconds), A* (0.00007290 seconds), IDS (0.00008440 seconds)
    - **Memory usage**
        - Lowest: Hill-Climbing (1.23 KB)
        - Highest: IDS (2.63 KB)
        - Moderate: GBFS (1.49 KB), UCS (1.55 KB), DFS (1.69 KB), A* (1.77 KB), BFS (2.29 KB)

- **File output4.txt**

```
BFS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00009590 seconds
Memory: 2.29 KB

DFS:
Path: 0 -> 2 -> 3 -> 5
Time: 0.00004080 seconds
Memory: 1.69 KB

UCS:
Path: 0 -> 2 -> 3 -> 5
Time: 0.00005290 seconds
Memory: 1.55 KB

IDS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00008440 seconds
Memory: 2.63 KB

GBFS:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00003790 seconds
Memory: 1.49 KB

A*:
Path: 0 -> 2 -> 3 -> 5
Time: 0.00007290 seconds
Memory: 1.77 KB

Hill-climbing:
Path: 0 -> 1 -> 3 -> 5
Time: 0.00004750 seconds
Memory: 1.23 KB
```

e. *Experiment 5 (test case 5)*

- *Detailed test case description*

In this experiment, we have the graph search with **6** nodes which this graph has multiple paths between the start node and the goal node. Now, I will describe the test case 3 input with some different attributes below:

Number of nodes: 6
Started node: 0
Goal node: 5
Adjacency matrix:

$$\begin{bmatrix} 0 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 5 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Heuristics:

$$[10 \quad 8 \quad 5 \quad 5 \quad 4 \quad 0]$$

Next, we will analyze this graph with some features:

- The provided graph is an **undirected weighted** graph with **6 nodes** labeled from **0** to **5**. It depicts various connections between these nodes, where each directed edge has a specific weight, and each node has a heuristic value denoted in **blue**.
- The start node is **green**, and the goal node is **red**

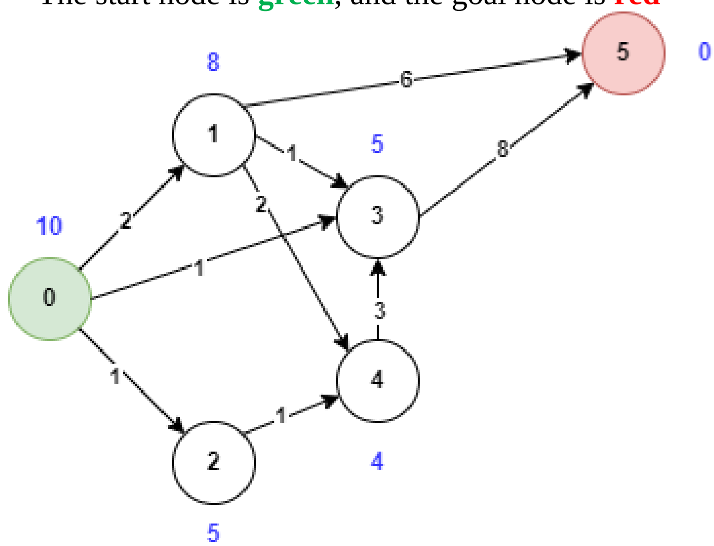

Figure 6. Graph for experiment 5 (test case 5)

- **File input5.txt**

```
6
0 5
0 2 1 1 0 0
0 0 0 1 2 6
0 0 0 0 1 0
0 0 0 0 0 8
0 0 0 3 0 0
0 0 0 0 0 0
10 8 5 5 4 0
```

- <u>*Detail analysis of the algorithm*</u>

  - **Execution time**
    - Fastest: DFS (0.00008930 seconds)
    - Slowest: BFS (0.00020120 seconds)
    - Moderate: GBFS (0.00012010 seconds), Hill-Climbing (0.00012970 seconds), UCS (0.00017790 seconds), A* (0.00011660 seconds), IDS (0.00014480 seconds)
  - **Memory usage**
    - Lowest: Hill-Climbing (1.26 KB)
    - Highest: BFS (2.42 KB)
    - Moderate: GBFS (1.49 KB), UCS (1.66 KB), DFS (1.57 KB), A* (1.55 KB), IDS(1.97 KB)
- **File output5.txt**

```
BFS:
Path: 0 -> 1 -> 5
Time: 0.00020120 seconds
Memory: 2.42 KB

DFS:
Path: 0 -> 3 -> 5
Time: 0.00008930 seconds
Memory: 1.57 KB

UCS:
Path: 0 -> 1 -> 5
Time: 0.00017790 seconds
Memory: 1.66 KB

IDS:
Path: 0 -> 1 -> 5
Time: 0.00014480 seconds
Memory: 1.97 KB

GBFS:
Path: 0 -> 3 -> 5
Time: 0.00012010 seconds
Memory: 1.49 KB

A*:
Path: 0 -> 3 -> 5
Time: 0.00011660 seconds
Memory: 1.55 KB

Hill-climbing:
Path: -1
Time: 0.00012970 seconds
Memory: 1.26 KB
```

2. Experimental comments

a. Runtime

- The BFS algorithm's performance is running with the highest runtime for almost all search algorithms.
- The DFS and GBFS algorithm are the fastest, making it suitable for scenarios where time is critical.

b. Memory usage

- Hill-Climbing is the most memory-efficient, making it ideal for environments with memory constraints.

c. Overall balance

- A* and UCS provide a good balance between speed and memory efficiency, making them versatile for various scenarios.

*

REFERENCES

During the process of researching and implementing the project **Lab#1: Searching – CSC14003 – Introduction to Artificial Intelligence**, I used and referenced some of the following open and electronic documents:

Programming

[1] Python Program for Breadth First Search or BFS for a Graph - Geeksforgeeks (Last updated: 12/07/2024)

[2] Greedy Best first search algorithm -Geeksforgeeks (Last updated: 13/07/2024)

[3] Breadth First Search in Python (with Code) | BFS Algorithm (Last updated: 14/07/2024)

[4] Introduction to Hill Climbing | Artificial Intelligence - Geeksforgeeks (Last updated: 10/07/2024)

Report

[5] *Slides CSC14003 – Introduction to Artificial Intelligence 2024 – Nguyen Ngoc Thao – Nguyen Hai Minh (Access date: 14/07/2024)*

[5] Github: search-strategies | Author: Kiều Công Hậu (Last updated: 13/07/2024)