University of Science - VNUHCM
Falcuty of Information Technology
---o0o---

Lab 1 Report

SEARCHING ALGORITHMS

Course : **Introduction to Artificial Intelligence**

Instructors : **Nguyen Ngoc Thao**
Nguyen Tran Duy Minh
Nguyen Thanh Tinh

Class : **22CLC06**

Student : **Vo Nguyen Phuong Quynh**
22127360

Ho Chi Minh City, July 2024

TABLES OF CONTENTS

1 Information

1.1. About student

- Full name: Vo Nguyen Phuong Quynh
- Student ID: 22127360
- Class ID: 22CLC06

1.2. About project

a) Completion assessment table

No	Details	Đánh giá
1	Implement BFS correctly.	10%
2	Implement DFS correctly.	10%
3	Implement UCS correctly.	10%
4	Implement IDS correctly.	10%
5	Implement GBFS correctly.	10%
6	Implement A* correctly.	10%
7	Implement Hill=climbing correctly.	10%
8	Generate at least 5 test cases for all algorithm with different attributes. Describe them in the experiment section.	10%
9	Report algorithms, experiment with some reflection or comments.	20%
TOTAL		100%

b) Tools

- Language: Python 3.11
- IDE: Visual Studio Code
- Support libraries:
  - `time`: Used to measure the execution time of algorithms in milisecond.
  - `tracemalloc`: Utilized to track memory allocations by KB.

- `queue.PriorityQueue`: Provides a priority queue data structure, essential for certain graph search algorithms like UCS, A*.

```
import time as tm
import tracemalloc as tcml
from queue import PriorityQueue
```

2 Algorithm Implementations

2.1. Class Graph

The Graph class serves as a fundamental structure for representing graphs and provides essential methods for graph manipulation and traversal.

a) Attributes

- vertices: Represents the number of vertices (nodes) in the graph.
- graph: An adjacency matrix initialized with default weights (e.g., all edges set to 0).
- heuristic: A list storing heuristic values associated with each vertex, initialized to zero for each vertex.

b) Methods

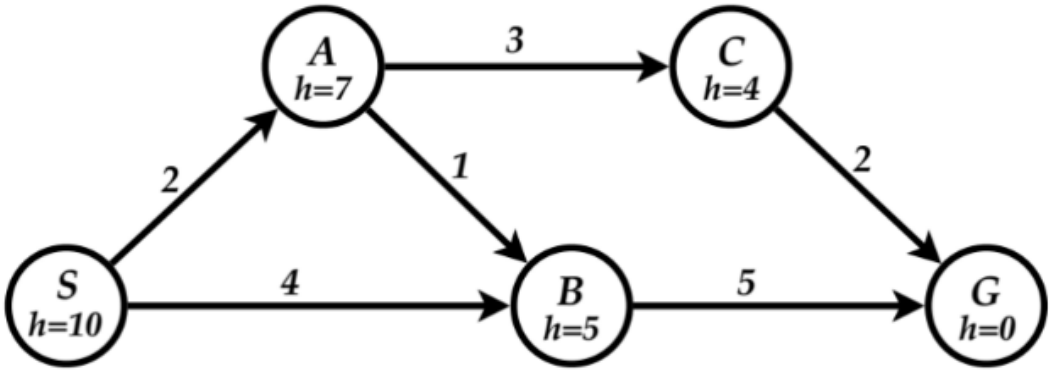
Phương thức	Mục đích
`def __init__(self, vertices):`	Initializes the Graph object with a specified number of vertices.
`def printGraph(self):`	Prints the adjacency matrix (`self.graph`) and the heuristic values (`self.heuristic`) of the graph.
`def BFS(self, start, goal) → tuple:`	Performs Breadth-First Search (BFS) traversal from a specified start node to a goal node in the graph.
`def DFS(self, start, goal) → tuple:`	Performs Depth-First Search (DFS) traversal from a specified start node to goal node in the graph.

`def DLS(self, start, goal, limit, visited) → tuple:`	Performs Depth-Limited Search (DLS) starting from start node with a depth limit limit, considering visited nodes.
`def IDS(self, start, goal) → tuple:`	Performs Iterative Deepening Search (IDS) from start node to goal node in the graph.
`def UCS(self, start, goal) → tuple:`	Performs Uniform-Cost Search (UCS) from start node to goal node in the graph.
`def GBFS(self, start, goal) → tuple:`	Performs Greedy Best-First Search (GBFS) from start node to goal node in the graph, using heuristic values (`self.heuristic`).
`def Astar(self, start, goal) → tuple:`	Performs A* Search from start node to goal node in the graph, using heuristic values (`self.heuristic`).
`def hillClimbing(self, start, goal) → tuple:`	Performs Hill-Climbing search variant from start node to goal node in the graph, focusing on local optimal solutions, using heuristic values (`self.heuristic`).

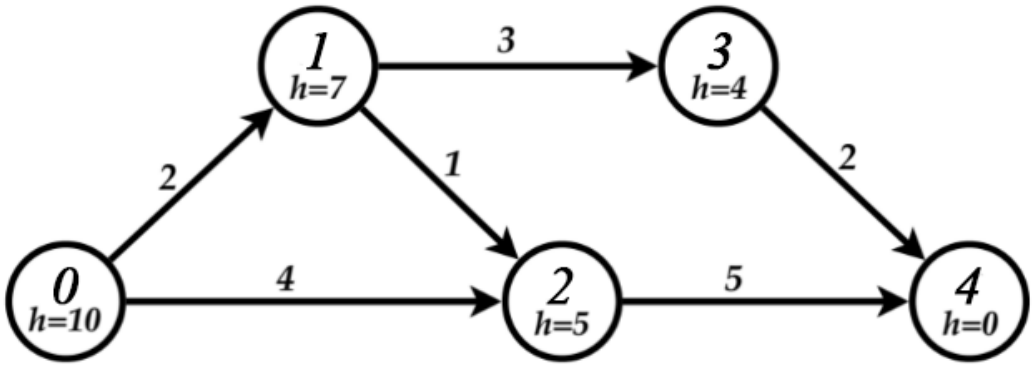
For all search algorithms (except DLS), the methods return a `tuple (path, runtime, memory)`, with:

- `path_found` is the path from `start` to `goal` or `-1` if not found. DLS returns only the path list and is used to support IDS.
- `runtime`: running time of algorithm in milisecond.
- `memory`: memory usage in kilobyte (KB).

In this report, to prove that the algorithms are implemented correctly, let's use the Graph from Exercise 2 (with solution) from Review Exercise 02 in Cours study document of Ms. Nguyen Ngoc Thao.



To fit with the defined `Class Graph`, we transform alphabet nodes to number nodes, let's call it "Default graph":



File `input.txt`:

```
5
0 4
0 2 4 0 0
0 0 1 3 0
0 0 0 0 5
0 0 0 0 2
0 0 0 0 0
10 7 5 4 0
```

2.2. Breadth-first search (BFS)

```
def BFS(self, start, goal) → tuple:
```

a) Idea:

Breadth-First Search (BFS) algorithm explores nodes in the order of their depth from the starting node: "Expand the root node first, then all the successors of the root node are next, then their successors, and so on." This makes BFS particularly suitable for finding the shortest path in unweighted graphs.

- It uses a **FIFO queue** frontier to manage the nodes to be explored, ensuring that all nodes at the present depth level are explored before moving on to nodes at the next depth level.
- **Early goal test**: The path is considered found when the goal node was added into the frontier *(not the expanded list)*.

b) Steps

Initialization:

- Start Time and Memory Tracking: Start measuring runtime and memory usage.
- `inFrontier`: A list to keep track of whether a node is in the frontier (initialized to `False`).
- `frontier`: A FIFO queue for BFS (initialized with the start node).
- `backpointer`: A list to keep track of the path (initialized to -1 for all nodes).

BFS loop

- Dequeue Node: The first node in the frontier queue is dequeued for exploration.
- Explore successors: For each successor of the current node:
  - If the successor is not already in the frontier:
    - Enqueue successor: Add the successor to the frontier.
    - Mark the successor as in the frontier (`inFrontier`).
    - Update `Backpointer`: Set the backpointer for the succesor to the current node.
    - Early goal test: If goal is generated, break out of the loop.

Check goal

- Path Found: If the goal is in the frontier, reconstruct the path from the goal to the start using the backpointer.
- Path Not Found: If not, set path to -1.

c) Check answer

Output for the Default graph:

```
BFS
Path: 0 -> 2 -> 4
Time:  0.0000000000000000 seconds
Memory: 0.19 KB
```

The path in alphabet: S → B → G, with also same with the solution in Review Exercise 02.

2.3. Tree-search Depth-first search (DFS)

```
def DFS(self, start, goal) → tuple:
```

a) Idea

Depth-First Search (DFS) algorithm that explores as far as possible along each branch before backtracking: "Always expand the deepest node in the frontier first." DFS is particularly useful for scenarios where the solution is located deep in the search space.

- Using a **LIFO stack** to manage the nodes to be explored, ensuring that the deepest nodes are explored first.

- **Early goal test**: The path is considered found when the goal node was added into the frontier *(not the expanded list)*.

b) Steps

Initialization:

- Start Time and Memory Tracking: Start measuring runtime and memory usage.
- `frontier`: A **LIFO** stack for DFS (initialized with the start node).
- `visited`: List to keep track of visited nodes (initialized to `False`)
- `backpointer`: A list to keep track of the path (initialized to -1 for all nodes).
- `havePath`: Boolean flag to indicate if the goal has been generated.

DFS loop

- Pop Node: The last node in the frontier queue is popped for exploration.
- Check Visited: If the node has already been visited, continue to the next iteration.
- Mark as Visited: Mark the current node as visited.
- Explore successors: For each successor of the current node:
  - If the successor has not been visited:
    - Push Successor: Add the successor to the frontier stack.
    - Update Backpointer: Set the backpointer for the successor to the current node.
    - Early goal test: If goal is generated, set `havePath` to `True` and break out of the loop.

Check goal

- Path Found: If the goal is in the frontier, reconstruct the path from the goal to the start using the backpointer.
- Path Not Found: If not, set `path` to -1.

c) Check answer

Output for the Default Graph

```
DFS
Path: 0 -> 1 -> 2 -> 4
Time:  0.0000000000000000 seconds
Memory: 0.19 KB
```

The path in alphabet: S → A → B → G, with also same with the solution in Review Exercise 02.

2.4. Uniform-cost search (UCS)

```
def UCS(self, start, goal) → tuple:
```

a) Idea

Uniform-Cost Search (UCS) is a graph traversal algorithm that extends Breadth-First Search by considering edge costs: "UCS resembles the mechanism of **Dijkstra's algorithm**." UCS is particularly useful for finding the least costly path in weighted graphs.

- It uses a **priority queue** to explore the node with the lowest path cost from the start node, ensuring that the shortest path is found in terms of cost.
- **Late goal test**: The path is considered found when the goal node was reached (added to expanded list).

b) Steps

Initialization:

- Start Time and Memory Tracking: Start measuring runtime and memory usage.
- `frontier`: Priority queue initialized with the start node and cost 0.
- `cost`: List to keep track of the minimum cost to reach each node (initialized to a large value).
- `visited`: List to keep track of visited nodes (initialized to `False`)
- `backpointer`: A list to keep track of the path (initialized to -1 for all nodes).
- `havePath`: Boolean flag to indicate if the goal has been generated.

UCS loop

- Get Node: The node with the lowest path cost is dequeued from the priority queue.
- Check Goal: If the current node is the goal, set `havePath` to True and break

out of the loop.

- Check Visited: If the node has already been visited, continue to the next iteration.
- Mark as Visited: Mark the current node as visited.
- Explore successors: For each successor of the current node:
  - If the successor has not been visited:
    - Calculate Cost: Update the cost to reach the successor by the formula: *cost = cost(current) + edge(current, successor)*
    - Update Frontier and Backpointer: If the new cost is **lower**, update the cost, add the successor to the frontier, and update the backpointer.

Check goal

- Path Found: If the goal is reached, reconstruct the path from the goal to the start using the backpointer.
- Path Not Found: If not, set `path` to -1.

c) Check answer

Output for the Default Graph:

```
UCS
Path: 0 -> 1 -> 3 -> 4
Time:  0.0000000000000000 seconds
Memory: 4.14 KB
```

The path in alphabet: S → A → C → G, with also same with the solution in Review Exercise 02.

2.5. Iterative deepening search (IDS)

```
def IDS(self, start, goal) -> tuple:
```

a) Idea

Iterative Deepening Search (IDS) repeatedly applies Depth-Limited Search (DLS) with increasing depth limits until a goal is found: "We supply a **depth limit *l*** to keep DFS from wandering down an infinite path… Then IDS will gradually increase the limit until a goal is found or a failure value is returned from DLS". This approach ensures that the shallowest goal is found with minimal memory usage.

- IDS combines the space efficiency of DFS with the completeness of BFS.

- DLS: use **LIFO stack** for frontier and **early goal test**.

  DLS recuresive

  - Base case:
    - If the `start` node equals the `goal`, return a list containing just the `start` node, indicating the path is found.
    - If the `limit` of exploration depth is reached (`limit == 0`), return an empty list, indicating the search has exhausted its depth limit.
  - Mark the start node as visited in the visited list to prevent revisiting during the search.
  - Explore successors: For each successor of the current node:
    - If the successor has not been visited: Recursively call `DLS` on `start` replace by each successor with the depth limit reduced by one (limit - 1).
  - Path construction:
    - If a path (`path`) is found in the recursive call, insert `start` at the beginning of `path` to maintain the correct path order from `start` to `goal`.
    - Return the constructed path to the caller.
    - If no path found after exploring all successors within the depth limit, return an empty list.

  Initialization for IDS

  - Start Time and Memory Tracking: Start measuring runtime and memory usage.
  - Initialize the depth limit (`limit`) to 0.

  IDS loop

  - Continuously increase the `limit` from `0` to `self.vertices` (number of nodes).
  - Initialize new `visited` list for each iteration of DLS to track visited nodes.
  - Call `path = DLS(current limit, start, goal)`:
    - If DLS returns a non-empty path (path found), break out of the loop.

Check path

- Path Found: If the path is not empty alter exiting the loop, store it.
- Path Not Found: If not, set `path` to -1.

b) Check answer

Output for the Default Graph:

```
IDS
Path: 0 -> 2 -> 4
Time:  0.0000000000000000 seconds
Memory: 0.12 KB
```

The path in alphabet: S → B → G, with also same with the solution in Review Exercise 02.

2.6. Greedy best-first search (GBFS)

```
def GBFS(self, start, goal) -> tuple:
```

a) Idea

Greedy Best-First Search (GBFS) algorithm that selects the node that appears to be closest to the goal based on heuristic value: "Expand first the node with the ***lowest h(n) value*** – the node that appears to be closest to the goal." GBFS is particularly useful when a heuristic is available that estimates the cost to reach the goal.

- It uses a **priority queue** to explore nodes with the lowest heuristic value first, aiming to reach the goal as quickly as possible.
- **Early goal test**: The path is considered found when the goal node was added into the frontier *(not the expanded list)*.

b) Steps

Initialization:

- Start Time and Memory Tracking: Start measuring runtime and memory usage.
- `frontier`: A **FIFO** queue for BFS (initialized with the start node).
- `visited`: List to keep track of visited nodes (initialized to `False`).
- `backpointer`: A list to keep track of the path (initialized to -1 for all nodes).
- `havePath`: Boolean flag to indicate if the goal has been generated.

GBFS loop

- Get Node: The node with the lowest heuristic value is dequeued from the priority queue.
- Check Visited: If the node has already been visited, continue to the next iteration.
- Mark as Visited: Mark the current node as visited.
- Explore successors: For each successor of the current node:
  - If the successor is not already in the frontier:
    - Push Successor: Add the successor to the frontier with its heuristic value.
    - Update Backpointer: Set the backpointer for the successor to the current node.
    - Chek goal: If goal is generated, set `havePath` to `True` and break out of the loop.

Check goal

- Path Found: If the goal is in the frontier, reconstruct the path from the goal to the start using the backpointer.
- Path Not Found: If not, set `path` to -1.

c) Check answer

Output for the Default Graph:

```
GBFS
Path: 0 -> 2 -> 4
Time:  0.0000000000000000 seconds
Memory: 4.16 KB
```

The path in alphabet: S → B → G, with also same with the solution in Review Exercise 02.

2.7. Graph-search A* (A*)

```
def Astar(self, start, goal) -> tuple:
```

a) Idea

A* (A-star) search algorithm extends UCS algorithm by using heuristics to guide the search. It combines the cost to reach the node and an estimate of the cost from the node to the goal (heuristic) to prioritize nodes in the frontier. A* ensures finding the

least-cost path to the goal if the heuristic is admissible (never overestimates the true cost).

- It uses a **priority queue** to store estimated total cost with node in frontier.
- **Late goal test**: The path is considered found when the goal node was reached (added to expanded list).

b) Steps

Initialization:

- Start Time and Memory Tracking: Start measuring runtime and memory usage.
- `frontier`: Priority queue initialized with the start node and cost 0.
- `cost`: List to keep track of the minimum cost to reach each node (initialized to a large value).
- `visited`: List to keep track of visited nodes (initialized to `False`)
- `backpointer`: A list to keep track of the path (initialized to -1 for all nodes).
- `havePath`: Boolean flag to indicate if the goal has been generated.

A loop*

- Get Node: The node with the lowest estimated total cost is dequeued from the priority queue.
- Check Goal: If the current node is the goal, set `havePath` to True and break out of the loop.
- Check Visited: If the node has already been visited, continue to the next iteration.
- Mark as Visited: Mark the current node as visited.
- Explore successors: For each successor of the current node:
  - If the successor has not been visited:
    - Calculate Cost: Update the cost to reach the successor by the formula: *cost = cost(current) – heuristic(current) + edge(current, successor) + heuristic(successor)*
    - Update Frontier and Backpointer: If the new cost is **lower**, update the cost, add the successor to the frontier, and update the backpointer.

Check goal

- Path Found: If the goal is reached, reconstruct the path from the goal to the start using the backpointer.
- Path Not Found: If not, set `path` to -1.

c) Check answer

Output for the Default Graph:

```
A*
Path: 0 -> 1 -> 2 -> 4
Time:  0.0000000000000000 seconds
Memory: 4.04 KB
```

The path in alphabet: S → A → B → G, with also same with the solution in Review Exercise 02.

2.8. Hill-climbing (HC) variant

```
def hillClimbing(self, start, goal) -> tuple:
```

a) Idea

Hill climbing (also called greedy local search) starts with an arbitrary solution to a problem and makes incremental changes to the goal: "The algorithm heads in the direction that gives the steepest ascent and terminates when it reaches a "peak"".

- The algorithm terminates when no further improvements can be made, i.e., when it reaches a peak.
- Hill climbing can get stuck in **local minima** of heuristic value (or maxima of objective function).

b) Steps

Initialization:

- Start Time and Memory Tracking: Start measuring runtime and memory usage.
- `current`: Initialize the current node to the start node.
- `path`: Initialize the path with the start node.

Hill climbing loop

- Find Next Node: For each successor of the current node:
  - Compare Heuristics: If the successor has a lower heuristic cost than the current node, update `next` to the successor.

- Check for Improvement: If no improvement is found (i.e., `next` is the same as `current`) => reach the peak => break out of the loop.
- Update Path and Current Node: Append the next node to the path and set it as the current node.

Check goal

- Path Found: If the current node is the goal after exiting the loop, return the path.
- Path Not Found: If not, set `path` to -1.

c) Check answer

Output for the Default Graph:

```
Hill Climbing
Path: 0 -> 2 -> 4
Time:  0.0000000000000000 seconds
Memory: 0.06 KB
```

The path in alphabet: S → B → G. In Review Exercise 02 does not have the solution for Hill Climbing algorithm but for self-test by hand, the result is totally the same.

3
Experiment

3.1. Self-generated test cases:

For best experiment and comparing, I use 3 data sets: small, medium and large, put in the folder `\self-test`. Each dataset is structured to assess algorithm efficiency across 5 types of graph: admissible heuristic, inadmissible heuristic, random graph, cycle graph, sparse graph.

a) Small Dataset (n = 20):

```
n = 20; start = 0; goal = 19
```

- Admissible Heuristic Graph: A graph where heuristic values are consistent with the actual shortest path distances from each node to the goal:
  - Heuristic values designed to smaller than the true cost.
- Inadmissible Heuristic Graph: A graph where heuristic values may

overestimate or underestimate actual distances to the goal.
    - Heuristic values may mislead search algorithms but still provide some guidance.
- Random Graph: A graph where edges are randomly distributed among nodes.
    - Varying density of edges to simulate different connectivity scenarios.
- Cycle Graph: A graph that forms a cycle, where each node is connected to exactly two other nodes.
    - Forms a single cycle structure without any additional edges.
- Sparse Graph: A graph with fewer edges relative to the number of nodes.
    - Limited number of connections between nodes to simulate sparsely connected networks.

b) Medium Dataset (n = 50)

```
n = 50; start = 0; goal = 49
```

Repeat the structure above for the medium dataset, adjusting the characteristics to fit the larger size (50 nodes) of the graphs.

c) Large Dataset (n = 500)

```
n = 500; start = 0; goal = 499
```

Repeat the structure for the large dataset, emphasizing the increased size (500 nodes) and potentially more complex characteristics of the graphs.

3.2. Experiment setup:

a) Function set up:

- `def printResult(path, runtime, memory, file):` Writes the `path`, `runtime`, and `memory` usage results to the output file.
- `def run(inputFile, outputFile):` Executes all graph search algorithms on a single input file and writes the results to the output file:
    - BFS
    - DFS
    - UCS
    - IDS
    - GBFS
    - A*

- Hill climbing

- `def runSetofData(size):` Executes the `run` function for all datasets of a specific size category.

b) Main execution block

- Runs the `run` function on for default graph in `input.txt` file.
- Executes `runSetofData` for each size category: small, medium, and large.

3.3. Path returned comparison

In this section, we only compare path results in Medium dataset

a) BFS, DFS, IDS and GBFS

Graph	BFS	DFS	IDS	GBFS
Admissible	0 → 49	0 → 49	0 → 49	0 → 49
Inadmissible	0 → 49	0 → 49	0 → 49	0 → 49
Random	0 → 49	0 → 49	0 → 49	0 → 49
Cycle	0 → 25 → 48 → 31 → 28 → 42 → 49	0 → 12 → 18 → 43 → 37 → 3 → 27 → 40 → 45 → 34 → 9 → 16 → 10 → 47 → 17 → 46 → 21 → 7 → 24 → 11 → 5 → 29 → 41 → 22 → 38 → 14 → 32 → 20 → 4 → 26 → 30 → 2 → 36 → 19 → 13 → 1 → 6 → 8 → 39 → 35 → 15 → 44 → 33 → 23 → 49	0 → 25 → 48 → 31 → 28 → 42 → 49	0 → 25 → 48 → 31 → 28 → 42 → 49
Sparse	0 → 2 → 43 → 49	0 → 2 → 9 → 5 → 4 → 49	0 → 2 → 43 → 49	0 → 2 → 22 → 46 → 49

b) UCS vs A*

Graph	UCS	A*
Admissible	0 → 26 → 49	0 → 26 → 49
Inadmissible	0 → 47 → 21 → 22 → 7 → 49	0 → 15 → 7 → 49
Random	0 → 29 → 37 → 22 → 46 → 31 → 49	0 → 5 → 49
Cycle	0 → 25 → 48 → 31 → 28 → 42 → 49	0 → 25 → 48 → 31 → 28 → 42 → 49
Sparse	0 → 8 → 25 → 43 → 49	0 → 8 → 25 → 43 → 49

c) Hill climbing

Paths returned from Hill climbing algorithm usually not found for all graph types. Except the Admissible graph: 0 → 49.

3.4. Runtime & memory usage analys

I have analys the runtime and memory usage with 5 types of graph of 3 datasets, when running all algorithms.

- Runtime: miliseconds
- Memory usage: kilobyte (KB)

a) Small Dataset

Number of nodes: 20.

	admissible_h		inadmissible_h		random		cycle		sparse	
	runtime	*memory*	*runtime*	*memory*	*runtime*	*memory*	*runtime*	*memory*	*runtime*	*memory*
BFS	0	0.59	0	0.59	0	0.59	0	0.44	0	0.38
DFS	0	0.44	0	0.44	0	0.44	0	0.59	0	0.38
UCS	0.99826	5.51	1.2579	6.05	1.4379	7.2	0	4.06	1.00064	4.34
IDS	0	0.25	0	0.25	0	0.25	0	0.25	1.00207	0.19
GBFS	0	3.95	0	3.89	0	4.22	0	3.71	0	3.62
A*	0.99993	4.84	2.01035	5.23	0.93508	5.16	0	4.11	1.0035	4.3
Hill climbing	0	0.6	0	0	0	0	0	0	0	0

b) Medium Dataset

Number of nodes: 50

	admissible_h		inadmissible_h		random		cycle		sparse	
	runtime	*memory*	*runtime*	*memory*	*runtime*	*memory*	*runtime*	*memory*	*runtime*	*memory*
BFS	0	1.25	0	1.25	0	1.25	0	0.91	0	1.09
DFS	0	0.88	0	0.88	0	0.88	0	1.25	0	1.06
UCS	0.99921	7.64	2.84934	7.64	2.08354	12.84	1.00112	4.98	0.99659	6.62
IDS	0	0.47	0	0.47	0	0.47	0	0.47	0	0.47
GBFS	0	4.45	0	4.45	0	4.47	0	4.11	0	6.16
A*	0.99802	6.48	5.51081	7.55	1.58501	8.05	0	5.08	1.10102	6.34
Hill climbing	0	0.6	0	0	0	0	0	0	0	0

c) Large Dataset

Number of nodes: 500

	admissible_h		inadmissible_h		random		cycle		sparse	
	runtime	*memory*	*runtime*	*memory*	*runtime*	*memory*	*runtime*	*memory*	*runtime*	*memory*
BFS	0.08106	19.81	0	19.81	0	19.81	156.003	17.66	6.00541	12.47
DFS	0	8.22	0	8.22	0	8.22	66.052	14.19	13.9923	25.03
UCS	223.043	235.56	249.579	121.64	19.0659	146.38	249.926	39.92	171.694	149.32
IDS	0	4.16	0.70197	4.16	0.91672	4.16	24519.2	10.09	5.08569	4.16
GBFS	2.00152	23.47	4.53634	23.05	2.00033	23.05	83.8122	18.27	36.9096	57.84
A*	51.6346	84.28	498.97	100.45	91.3768	146.13	304.874	39.83	153.881	67.12
Hill climbing	1.00827	0.03	0.53072	0.06	1.00112	0.06	1.00075	0.03	0	0.06

4
Reflection & Comments

4.1. Reflection

a) *Peth returned's reflection*

- BFS, IDS, and GBFS: These algorithms tend to produce similar results in finding paths. However, DFS does not guarantee the shortest path and may miss it altogether, especially noticeable in cycle graphs and sparse graphs. DFS is more prone to wandering into deep branches without ensuring optimality.
- Admissible Heuristic Graphs: UCS (Uniform-Cost Search) and A* (A-Star Search) consistently yield the same path due to the accurate heuristic guiding the search process. However, in graphs with inadmissible heuristics, A* can return different, often suboptimal paths, indicating the sensitivity of A* to the heuristic's accuracy.
- Hill Climbing: This algorithm performs well in admissible heuristic graphs, reliably finding the path from start to goal. In other types of graphs, it often gets stuck in local maxima, failing to find the optimal path. This is particularly evident in random and sparse graphs where the heuristic might not provide sufficient guidance.

b) *Runtinme & memory usage's reflection*

Small dataset n = 20

- *BFS and DFS* consistently showed zero runtime across all graph types, indicating near-instantaneous traversal in small graphs. Memory usage is minimal and uniform, aligning with their straightforward traversal nature.
- *UCS and A** have similar runtimes for admissible heuristics, but A* takes significantly more time with inadmissible heuristics, reflecting its sensitivity to heuristic accuracy. Both algorithms show higher memory usage in random and sparse graphs due to the cost of maintaining multiple paths and heuristic evaluations.
- *IDS* shows zero runtime for most graph types except the sparse graph, where minimal runtime is observed, possibly due to its iterative deepening

nature needing more iterations. Memory usage is consistently low, indicating efficiency in smaller datasets.

- *GBFS* consistently shows zero runtime across all graph types, reflecting its greedy nature of quickly converging to the goal. Memory usage varies *(because of the priority queue)*, but generally remains low, highlighting its space efficiency.
- *Hill Climbing* has zero runtime across all graphs, showing quick convergence, likely due to its greedy nature which can quickly find local maxima. Zero memory usage indicates minimal state storage, highlighting its space efficiency.

Medium dataset n = 50

- *BFS and DFS*: Still zero runtime, with increasing memory usage variations due to larger graph sizes. Memory starts to diverge, especially in cycle and sparse graphs, due to increased traversal space.
- *UCS and A** exhibit increased runtime with larger datasets and more complex graphs, notably with inadmissible heuristics. Memory usage spikes in random and sparse graphs underline the overhead of managing extensive state spaces.
- *IDS* shows zero runtime for most graph types except sparse graphs, reinforcing its efficiency in mid-sized datasets. Memory usage remains low, highlighting its space efficiency.
- *GBFS & Hill Climbing*: same as Small dataset.

Large Dataset n = 500

- *BFS and DFS* start showing significant runtime increases, especially in cycle and sparse graphs with extensive traversal space. Memory usage significantly increases, reflecting the extensive state spaces traversed.
- *UCS and A** show substantial runtime, particularly A* in inadmissible heuristics, highlighting performance degradation with poor heuristics. Both algorithms exhibit substantial memory usage spikes, especially in inadmissible heuristics and random graphs, due to the complexity and depth of state spaces.
- *IDS* runtime spikes in complex graphs like cycle graphs, showcasing the impact of iterative deepening on larger datasets. Memory usage remains

relatively low, but runtime spikes indicate the trade-off in iterative deepening.

- *GBFS* runtime increases in larger datasets but remains relatively low compared to other algorithms. Memory usage increases significantly, particularly in cycle and sparse graphs, reflecting the larger state space it has to manage.
- *Hill Climbing*: Minimal runtime across most graph types, indicating quick but possibly suboptimal solutions due to local maxima. Maintains minimal memory usage same as other dataset.

4.2. Comments and Insight thoughts

Throughout the experiment and reflection, some key insights have emerged regarding the performance of graph search algorithms across 3 dataset sizes and 5 graph types:

- For small datasets, BFS, DFS, and IDS are efficient and sufficient due to their low runtime and memory usage. While for larger datasets or when optimal paths are crucial, UCS and A* are preferable, particularly when admissible heuristics are available. However, the heuristic's accuracy must be ensured to prevent performance degradation.
- UCS and A* face scalability issues with larger datasets, highlighting the need for efficient heuristic design and potential hybrid approaches to manage state space complexity. IDS shows that iterative deepening can efficiently manage memory usage but may face runtime challenges in complex graphs.
- There is a trade-off between efficiency (runtime and memory usage) and optimality (accuracy of the path found). Algorithms like BFS, IDS, and GBFS offer efficiency, while UCS and A* provide optimal solutions at higher resource costs.
- Hill Climbing's efficiency makes it suitable for problems where quick solutions are needed, but mechanisms to escape local maxima should be considered for better performance.
- The quality of heuristics is paramount for algorithms like A*, GBFS and Hill Climbing. Admissible heuristics ensure optimal paths and efficient performance, while inadmissible heuristics can lead to suboptimal paths and increased computational costs.

References

1) Course study document of Ms. Nguyen Ngoc Thao
2) *"Cơ sở AI"* lecture book from VNH-HCM University of Science
3) *"Artificial Intelligence: A modern Approach (Fourth Edition)"* book

___END.___