Vietnam National University - Ho Chi Minh

University of Science

Faculty of Information Technology

.......o0o.......

LAB REPORT: SEARCH STRATEGIES

Course: Artificial Intelligence

Theory teacher:	**Nguyen Ngoc Thao**	
Instructors practice:	**Nguyen Thanh An**	
	Ho Thi Thanh Tuyen	
	Le Ngoc Thanh	
Student:	**Hoang Huu Minh An**	**ID: 20127102**
Class:	**20CLC01**	

Lab 01: Search Strategies

Contents:

I. Information: 3

II. Introduction: 3

III. Search Algorithm: 4

1. Breadth-first search: 5
   a. Idear: 5
   b. Description algorithm: 5
   c. Complexity: 5
   d. Pros and cons: 5
2. Tree-search depth-first search: 6
   a. Idear: 6
   b. Description algorithm: 6
   c. Complexity: 6
   d. Pros and Cons: 6
3. Uniform-cost search: 7
   a. Idear: 7
   b. Description algorithm: 7
   c. Complexity: 7
   d. Pros and Cons: 7
4. Iterative deepening search: 8
   a. Idear: 8
   b. Description algorithm: 8
   c. Complexity: 8
   d. Pros and Cons: 8
5. Greedy best first search: 9

a. Idear: 9
b. Description algorithm: 9
c. Complexity: 9
d. Pros and Cons: 9
6. Graph-search A*: 10
a. Idear: 10
b. Description algorithm: 10
c. Complexity: 10
d. Pros and Cons: 11
7. Steepest-ascent Hill-climbing: 11
a. Idear: 11
b. Description algorithm: 12
e. Complexity: 12
f. Pros and Cons: 12
IV. Experimental Results and Comments: 13
1. Table: 13
2. Bar Chart: 16
3. Overall comment of algorithm: 17
V. Lab Organization and Programing Notes: 17
VI. Applications of search algorithms: 18
VII. Conclusion: 18
VIII. List of References: 19

Lab 01: Search Strategies

I. Information:

This Lab01: Search Strategies is presented by Hoang Huu Minh An-20127102.

II. Introduction:

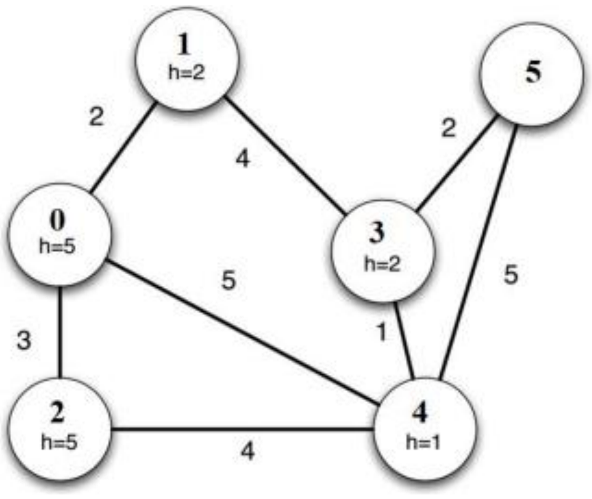
Search algorithms are algorithms that help in solving search problems. A search problem consists of a search space, start state, and goal state. Search algorithms help the AI agents to attain the goal state through the assessment of scenarios and alternatives.

The algorithms provide search solutions through a sequence of actions that transform the initial state to the goal state. Without these algorithms, AI machines and applications cannot implement search functions and find viable solutions.

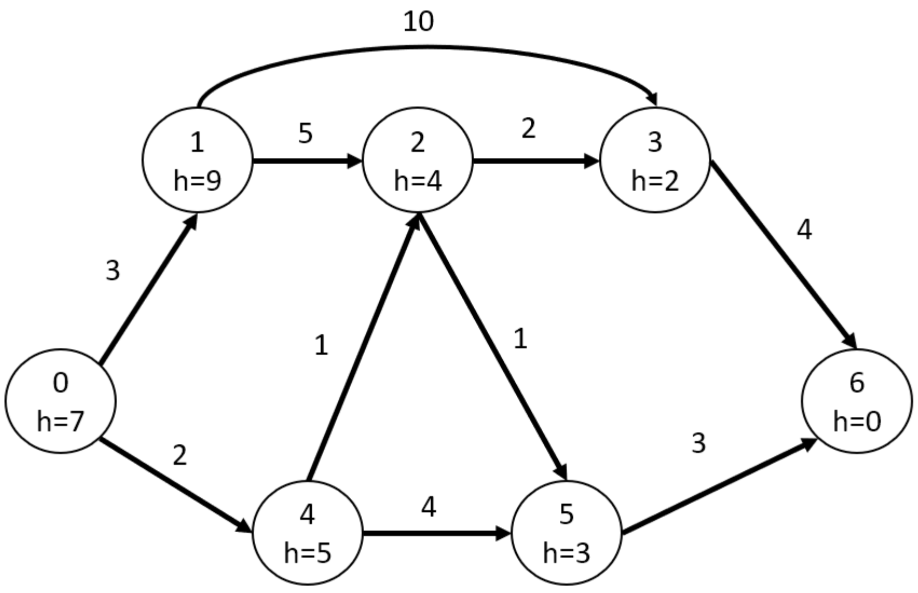
Write a program to find an (optimal) path from the source node to the destination node on a graph, using either of the following strategies:

- 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* (AStar)
- Steepest-ascent Hill-climbing (HC)

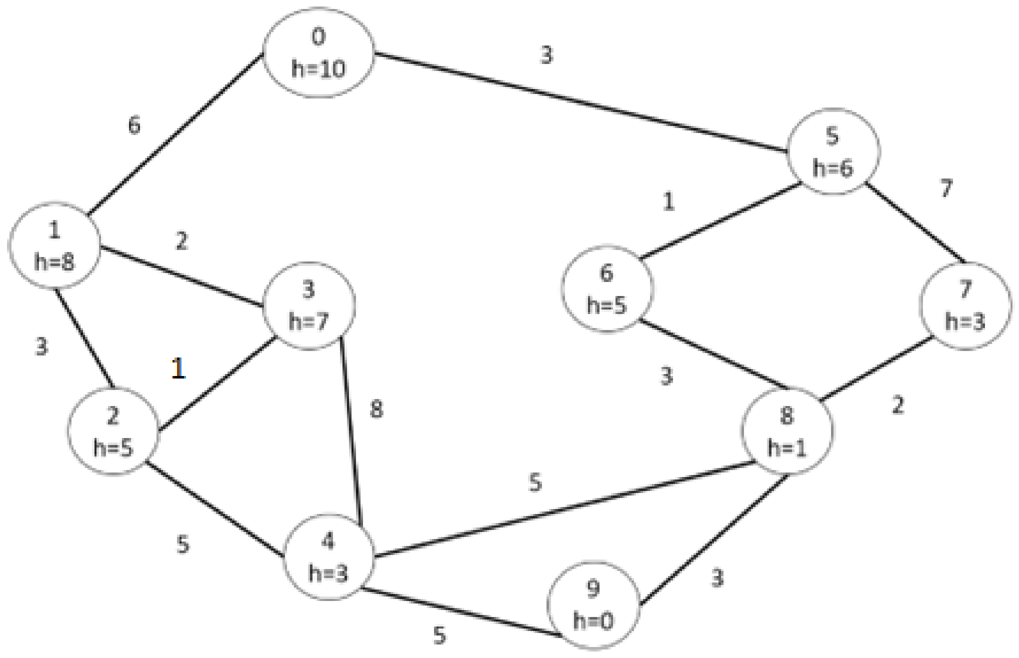
Run the above strategies on a set of 3 different graphs:



Graph 1



Graph 2



Graph3

III. Search Algorithm:

1. Breadth-first search:

a. Idear:

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

b. Description algorithm:

Implementation: FIFO queue is frontier

Step 1: Initial expandedList, path and frontier is empty list. An dict visited, Which contains visited nodes, each key is a visited node, each value is adjacent node visited before it. An isGoal, Which check if goal return expandedList, path

Step 2: If src == des then return expandedList, path else let node is Initial-state(src)

Step 3: push node to frontier and expandedList

Step 4: Loop if frontier not is empty, do pop frontier and add node to expandedList. If isGoal then break

Step 5: for each adjacent node, if adjacent node not in expanded list and frontier then visited[adjacent node] = node if adjacent node is src then let isGoal = True break else push to frontier

Step 6: Go step4

Step 7: find path by function found_Path(visited, src, des)

c. Complexity:

- Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state. Search take time $O(b^d)$
- Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier, Roughly the last tier take $O(b^d)$
- Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
- Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

d. Pros and cons:

Pros:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps

Cons:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

2. Tree-search depth-first search:

a. Idear:

Depth-first search is recursive algorithm for traversing a tree or graph data structure. It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path. DFS uses frontier LIFO.

Expand deepest unexpanded node.

b. Description algorithm:

Initial visited, expandedList is empty, check if(src == des) then return expandedList, path. Else recursion dfs (visited, expandedlist, let node is src, des)
If(node == des) then visited append(node) return True

Else push node to visited and expanded, for each adjacent node if(node in visied) then continue, Avoid repeated states by checking new states against those on the path from the root to the current node. else goal = recursion with node , check if(goal) return True

If no solution visited.pop() and return false

Finally, Return expandedList, visited

c. Complexity:

- Time Complexity of DFS algorithm Some left prefix of the tree, and it could process the whole tree!, If the maximum depth m is finite, it takes time O(b^m)
- Space Complexity: Only has siblings on path to root, so $O(bm)$ → linear space
- Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- Optimality: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

d. Pros and Cons:

Pros:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node
- It takes less time to reach to the goal node than BFS algorithm

Cons:

- There is the possibility that many states keep recurring, and there is no guarantee of finding the solution.

- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop

3. Uniform-cost search:

a. Idear:

UCS is different from BFS and DFS because here the costs come into play. Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The goal is to find a path where the cumulative sum of costs is the least. A uniform-cost search algorithm is implemented by the priority queue. A uniform-cost search algorithm is implemented by the priority queue.

b. Description algorithm:

Similar BFS, Initial expandedList, path and frontier priority queue is empty list. An dict visited, Which contains visited nodes, each key is a visited node, each value is adjacent node visited before it, pathCost = 0

Check if(src == des) then push src to path and return expandedList, path Else push [0, src] frontier with 0 is pathCost, src is node

Loop do if(frontier not is empty), let node = pop(Frontier), let pathCost(new) = node[0] (pathCost prev), if node[1] is des, then push node[1] to expandedlist return expandedList, path else push node[1] to expandedlist. For each adjacent node if(childNode in frontier and expandedList) visited[ChildNode] = node (store path) push [pathCost + g(), childNode] to frontier elseif in frontier and pathCost of childNode < pathCost of node Frontier the update pathCost

Find path by found_path(visited, src, des) return expandedList, path

c. Complexity:

- Time Complexity: Process all nodes with cost less than cheapest solution, Let **C*** is Cost of the optimal solution, and **ε** is each step to get closer to the goal node. Then the number of steps is = $C^*/\varepsilon+1$, the worst-case time complexity of Uniform-cost search is $O(b^{1 + [C^*/\varepsilon]})$ (exponential in effective depth)
- Space Complexity: Roughly the last tier, so $O(b\ 1+ C * /\epsilon\ )$
- Completeness: Uniform-cost search is complete, such as if there is a solution, UCS will find it
- Optimality: Uniform-cost search is always optimal as it only selects a path with the lowest path cost

d. Pros and Cons:

Pros:

- Uniform-cost search is always optimal as it only selects a path with the lowest path cost
- UCS is complete only if states are finite and there should be no loop with zero weight

Cons:

- Explores options in every "direction"
- No information on goal location
- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop

4. Iterative deepening search:

a. Idear:

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown

b. Description algorithm:

Initial visited, expanded list is empty, d is deep limit = 0

Loop true if(__dls(expandedList, visited, let node is src, des, d) is "cutoff") then let expandedList, path = [] else break, d increase

__dls() function, if(src == des) then push node to visited return True elseif d == 0 then push node to visited and expanded return "cuttoff" else cutoff_occurred = False, push node to visited, expanded for each adjacent node if node in visited continue, call recusion goal = __dls(expanded, visit, childNode, des, d - 1). If goal is True return True else if goal = "cuttoff" cut_occurred = True pop(visited) the last element. End loop if cut_occrred then return "cutoff" else return False

c. Complexity:

- Time Complexity: Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **O(b^d)**
- Space Complexity: similar to DFS
- Completeness: This algorithm is complete is if the branching factor is finite.
- Optimality: if step cost = 1

d. Pros and Cons:

Pros:

- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency

Cons:

- The main drawback of IDS is that it repeats all the work of the previous phase

5. Greedy best first search:

a. Idear:

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n)

b. Description algorithm:

Initial expandedList, path and frontier is empty list. An dict visited, Which contains visited nodes, each key is a visited node, each value is adjacent node visited before it. An isGoal, Which check if goal return expandedList, path.

If src is des, then push src to path return expandedList, path,

Push ((0, src, des)) with fomula (h(n), node, parent node), loop do if frontier is not empty. If isGoal break, user library heapq to implement priority queue, heapq.heapify(frontier), let (hNode, node, parent) = pop(frontier), push node to expandedList, let visited[node] = parent.

Loop true, create openList is [] for each adjacent node if node is des then visited[childNode] = node, isGoal = true break. Check if childNode not in expandedList and not in frontier then push to openList. End loop for adjacent node, if isGoal break, if(open_list is empty, this mean no solution) then break, use heapq to find node has heuristic value smallest , let visited[node] = parent push node to expandedList, let frontier += openList to get the node have heuristic smaller than.

c. Complexity:

- Time Complexity: $O(b^m)$ → reduced substantially with a good heuristic
- Space Complexity: keeps all nodes in memory-> $O(b^m)$
- Completeness: if it is a graph-search instance in finite state spaces
- Optimality: Greedy best first search algorithm is not optimal

d. Pros and Cons:

Pros:

- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms
- This algorithm is more efficient than BFS and DFS algorithms

Cons:

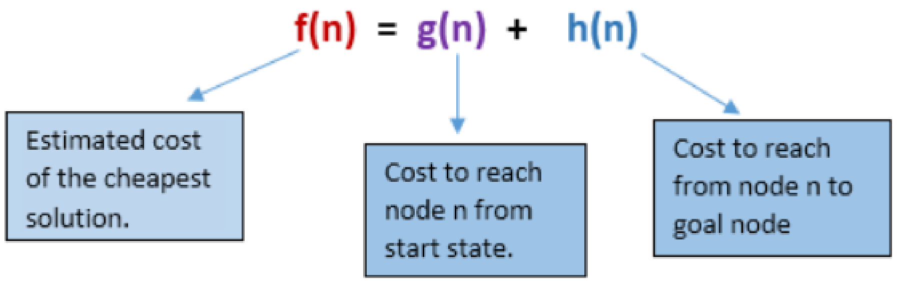
- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS This algorithm is not optimal.
- This algorithm is not optimal

6. Graph-search A*:

a. Idear:

A* tree search works well, except that it takes time re-exploring the branches it has already explored. In other words, if the same node has expanded twice in different branches of the search tree, A* search might explore both of those branches, thus wasting time. Ensure to compute a path with minimum cost, Ensure to compute a path with minimum cost

A* Graph Search, or simply Graph Search, removes this limitation by adding this rule: do not expand the same node more than once


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

b. Description algorithm:

Initial expandedList, path and frontier is empty list. An dict visited, Which contains visited nodes, each key is a visited node, each value is adjacent node visited. If src == des then push src to path return expandedList, path.

Push (0, src, 0, src) with fomula (f(n), node, g(n), parent) to frontier loop do if frontier is not empty. Use heapq.heapify(forntier) to be priority queue,
let (f, node, g_path_cost, parent) = pop(frontier). If node in expandedList then continue

Push node to expandedList let visieted[node] = parent. Check goal if node is des then brek

For each adjacent node if node not in expandedList then push (f_child, child_node, g_childNode, node) with g_child += g_path_cost, f_child = childNode.heuristic + g_child.

c. Complexity:

- Time Complexity: The number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d)
- Space Complexity: The space complexity of A* search algorithm is O(b^d), keep all nodes in memory
- Completeness: A* algorithm is complete as long as: Branching factor is finite, Cost at every action is fixed or if all step costs exceed some finite ϵ and if b is finite
- Optimality:
  - ➢ Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search.
  - ➢ Consistency: Second required condition is consistency for only A* graph-search

d. Pros and Cons:

Pros:

- A* search algorithm is the best algorithm than other search algorithms
- A* search algorithm is optimal and complete
- This algorithm can solve very complex problems

Cons:

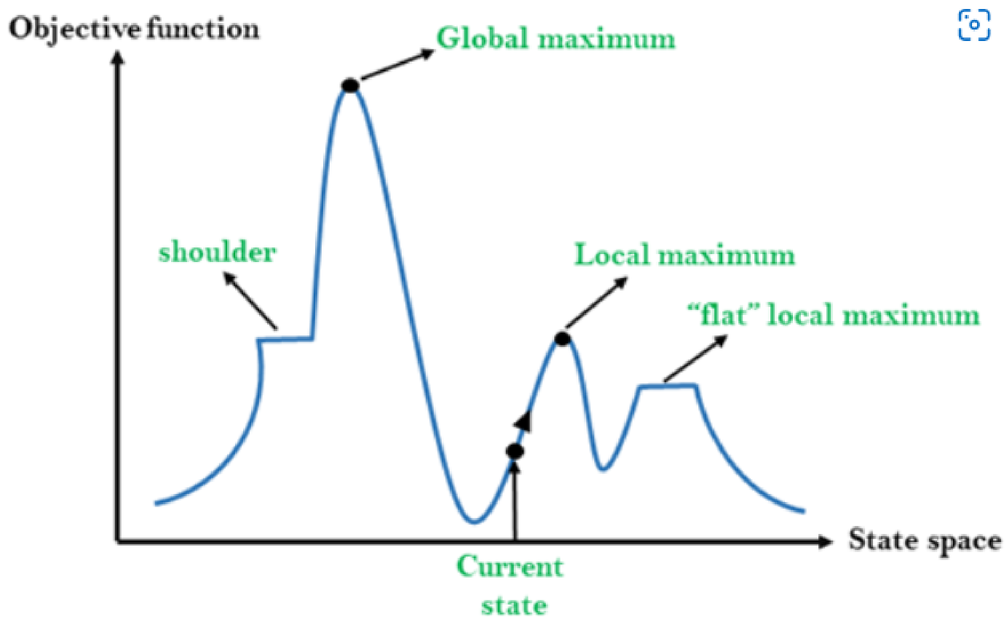
- It does not always produce the shortest path as it mostly based on heuristics and approximation
- A* search algorithm has some complexity issues
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems

7. Steepest-ascent Hill-climbing:

a. Idear:

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing



algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

b. Description algorithm:

Initial expandedList, path and frontier is empty list. If src == des then return expandedList, path.

Initial current_node is src old_state = None this store prev state of the current node

Loop do if old_state != current_node, push current_node to path, expandedList. Let Success be a state such that nay the current state will be better than it. For each adjacent node. Generate new_state if is goal, push new_state to path then return expanded list else if new_state.heuristic < success.heuristic then success = new_state if success.heuristic <= current_node then current_node = success

e. Complexity:

- Time Complexity: Has a time complexity of O(∞)
- Space Complexity: a space complexity of O(b).
- Hill climbing is neither complete *nor* optimal

f. Pros and Cons:

Pros:

- Hill Climbing can be used in continuous as well as domains
- Hill climbing technique is very useful in job shop scheduling, automatic programming, circuit designing, and vehicle routing
- Hill climbing is also helpful to solve pure optimization problems where the objective is to find the best state according to the objective function

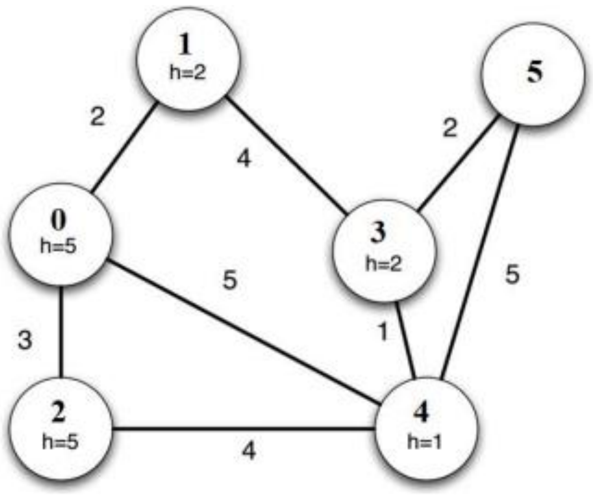
Cons:

- Local Maxima: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go
- Ridges: a ridge is shown in Figure 4.4. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate
- Plateaux: a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. A hill-climbing search might get lost on the plateau

IV. Experimental Results and Comments:

1. Table:

➢ Graph 1:



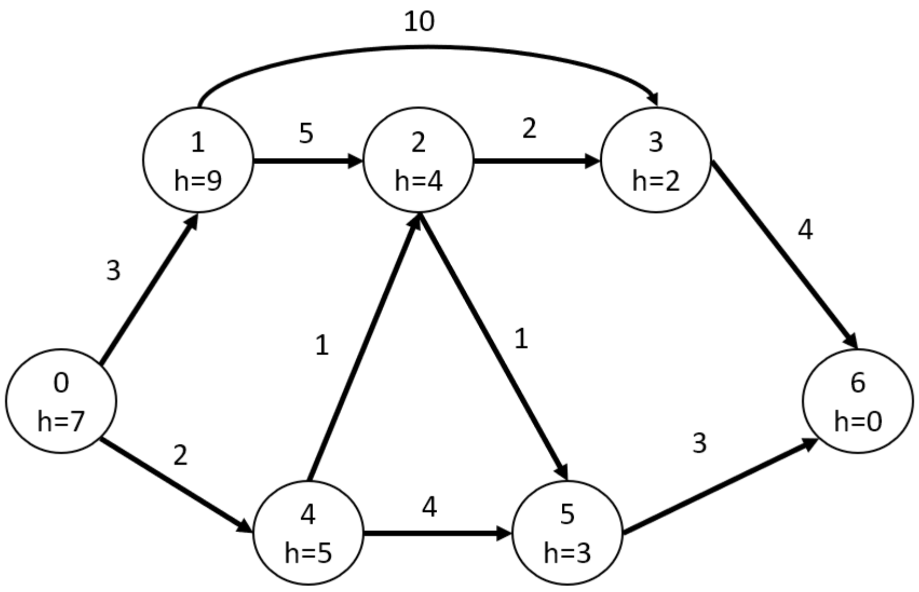
Start Node : 0

End Node: 5

Search Strategies	List of		Total Nodes
BFS	Expanded Nodes	0 1 2 4	4

	Path	0 4 5	
DFS	Expanded Nodes	0 1 3 4 2	5
	Path	0 1 3 4 5	
UCS	Expanded Nodes	0 1 2 4 3 5	6
	Path	0 1 3 5	
IDS	Expanded Nodes	0 1 3 2 4 4 2 3	8
	Path	0 4 5	
GBFS	Expanded Nodes	0 4	2
	Path	0 4 5	
A*	Expanded Nodes	0 1 4 2 3 5	6
	Path	0 1 3 5	
HC	Expanded Nodes	0 4	2
	Path	0 4 5	

- Graph 2:



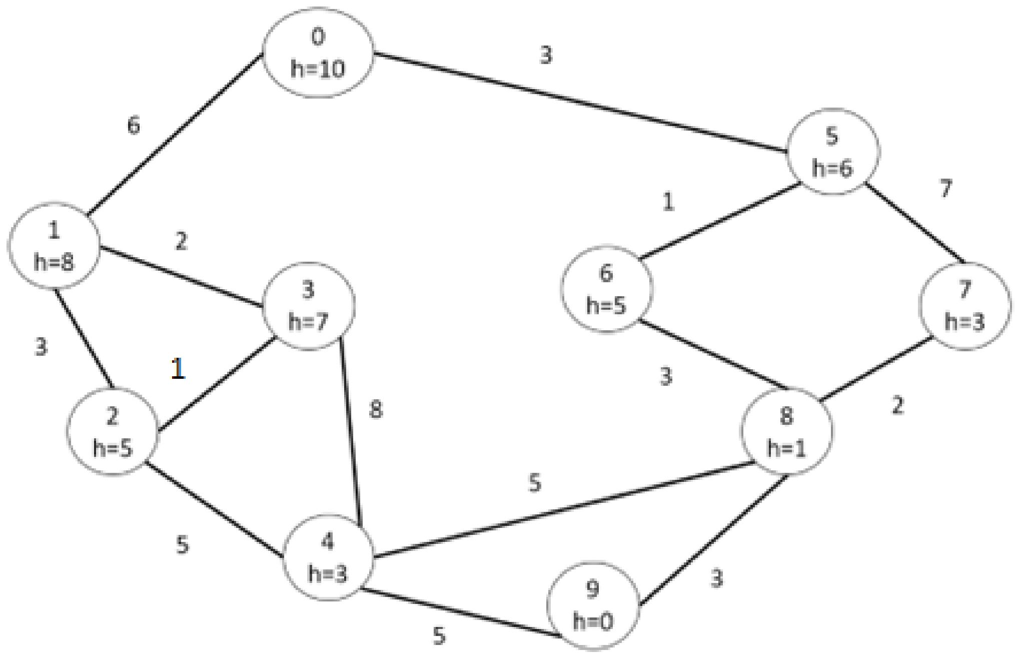
Start Node : 0

End Node: 6

Search Strategies	List of		Total Nodes
BFS	Expanded Nodes	0 1 4 2 3	5
	Path	0 1 3 6	

DFS	Expanded Nodes	0 1 2 3	4
	Path	0 1 2 3 6	
UCS	Expanded Nodes	0 4 1 2 5 3 6	7
	Path	0 4 2 5 6	
IDS	Expanded Nodes	0 1 2 3 5 3	6
	Path	0 1 3 6	
GBFS	Expanded Nodes	0 4 5	3
	Path	0 4 5 6	
A*	Expanded Nodes	0 4 2 3 5 6	6
	Path	0 4 2 5 6	
HC	Expanded Nodes	0 4 5	3
	Path	0 4 5 6	

- Graph 3:



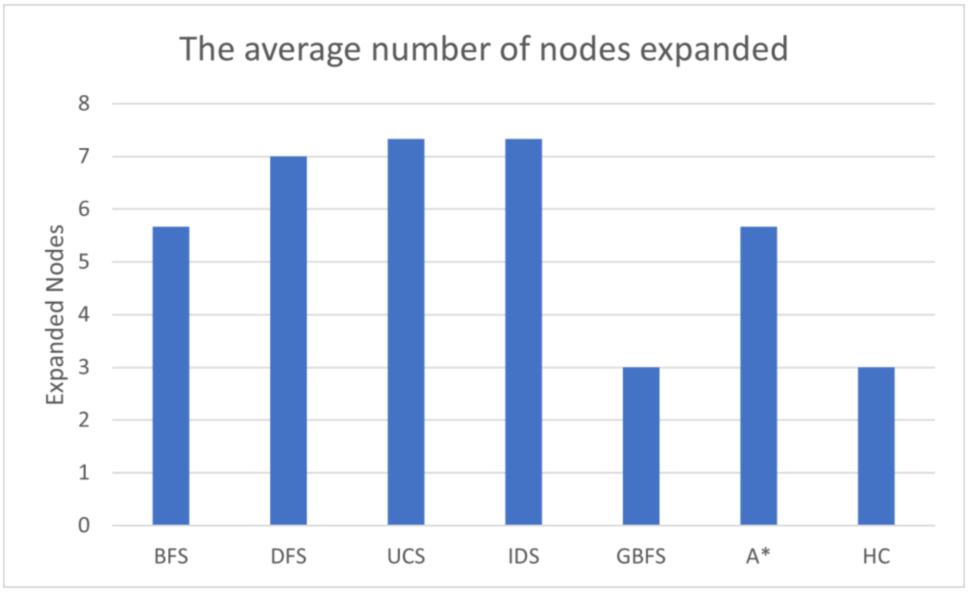
Start Node : 0

End Node: 9

Search Strategies	List of		Total Nodes
BFS	Expanded Nodes	0 1 5 2 3 6 7 4	8
	Path	0 1 2 4 9	
DFS	Expanded Nodes	0 1 2 3 4 8 6 5 7 7 5 6	12
	Path	0 1 2 3 4 8 9	
UCS	Expanded Nodes	0 5 6 1 8 3 2 7 9	9
	Path	0 5 6 8 9	
IDS	Expanded Nodes	0 1 2 3 4 4 3 8	8
	Path	0 1 2 4 9	
GBFS	Expanded Nodes	0 5 7 8	4
	Path	0 5 7 8 9	
A*	Expanded Nodes	0 5 6 8 9	5
	Path	0 5 6 8 9	
HC	Expanded Nodes	0 5 7 8	4
	Path	0 5 7 8 9	

2. **Bar Chart:**

Search Strategies	Graph 1	Graph 2	Graph 3	AVG
BFS	4	5	8	5.6667
DFS	5	4	12	7
UCS	6	7	9	7.3333
IDS	8	6	8	7.3333
GBFS	2	3	4	3
A*	6	6	5	5.6667
HC	2	3	4	3



This section is divided into an analysis of the searching algorithms, implementation of the searching algorithms , results of searching

- HC and GBFS have the lowest number of extended nodes.
- UCS and IDS have the highest number of extended nodes
- The Breadth-First Search Algorithm was won more against the Depth-First Search and loss more against the A* Search algorithm and the Best First Search algorithm
- The Depth-First Search Algorithm was not won more against all the remaining algorithms

3. **Overall comment of algorithm:**
   - The Breadth-First Search traverses the optimal path to reach, but The time complexity of the algorithm is high because to find the optimal path.
   - By computing the heuristic value between the adjacent node to the node and the adjacent node heuristic value, the A* Search method additionally employs cost estimation. When compared to other algorithms, this one performs well and follows the best path to the node.
   - HC, GBFS the number of nodes processed is also less. The expanded list of the algorithm is low compared to others.

V. Lab Organization and Programing Notes:

The code of Lab01 include 4 Class:

- graph.py: include data data structure G, vertex
- file.py: include 2 method to read from file input.txt, write to file output.txt

- requirement_input.py: include globl variable SOURCE_NODE, DESTINATION_NODE, STRATEGY_SEARCH to store value when read input from input.txt. STRATEGY_SEARCH is list store string strategy. Corresponding to each stratege it is index in the list
- main.py: the main function run comand_line, argument from command line: `python main.py <input_file_path>

Data Structure:

- G: This is a data structure for creating a graph, with an adjacency matrix and a list of vertices as properties. The method of the object consists of a constructor whose initial values are an adjacency matrix and a list of heuristic lists of vertices, the add vertex function, which is intended to be used to create a list of vertices with a corresponding list of heuristics, and the sort sort strategy functions.
- Vertex: this is a data structure for creating a vertex, with atribute _name and heuristic. The method __lt__ operator to compare two vertex following their ascending order of index values because _name is index.

Library: heapq: priority queue, copy: deep copy matrix

Progaming: Pyton 3.10

File input: input1.txt is graph 1, input2.txt is graph 2, input3.txt is graph 3

VI. Applications of search algorithms:

➢ Vehicle routing

Search algorithms are used in solving real-life vehicle routing problems, which are optimization problems. A vehicle routing problem seeks to establish the optimal series of routes that should be taken by vehicles to deliver customers. This is a combinatorial optimization problem that seeks to minimize the cost and minimize the time taken to reach a given destination. The search algorithm is used to search for the best solution

➢ Retrieving records from databases

Search algorithms are also used to retrieve information or records from databases. The desired record is configured as the desired state. The algorithm searches in the search space and goal tests are made until the desired record is found

VII. Conclusion:

This article looked at the essentials of search algorithms in artificial intelligence. To summarize, we have learned the following:

- Search algorithms are algorithms that help in solving search problems. A search problem consists of a search space, start state, and goal state
- These algorithms are important because they help in solving AI problems and support other systems such as neural networks and production systems
- The main properties of search algorithms include optimality, completeness, time complexity, and space complexity.
- Search algorithms work by defining the problem (initial state, goal state, state space, space cost, etc) and conducting search operations to establish the best solution to the given problem.
- There are two main types of search algorithms: informed algorithms and uninformed algorithms

- The main applications of search algorithms include vehicle routing, nurse scheduling, record retrieval, and industrial processes.

VIII. List of References:

https://www.javatpoint.com/hill-climbing-algorithm-in-ai
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
https://www.section.io/engineering-education/understanding-search-algorithms-in-ai/
https://www.youtube.com/watch?v=j1H3jAAGlEA&t=1640s
https://www.youtube.com/watch?v=6TsL96NAZCo
2021-Lecture03-P2-UninformedSearch.pdf
2021-Lecture03-P3-InformedSearch.pdf
2021-Lecture04-LocalSearch.pdf

Thank you for waching !!