**Practical File**

**of**

**Artificial Intelligence LAB**

**[BTIT 515-18]**

**B.Tech – 5th Semester**

**[Batch 2022-26]**

****

**Submitted To: Submitted By:**

**Er.Chandandeep Kaur Harpreet**

**AP, IT Dept. 2206421**

**Department of Information Technology**

**Rayat Bahra Institute of Engineering**

**and Nano-Technology, Hoshiarpur**

**Rayat Bahra Institute of Engineering & Nano-Technology**

**Hoshiarpur**

**Affiliated to IKG PTU Jalandhar, Approved by AICTE**

**New Delhi**

**List of Practicals**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Name of the Practical** | **Page No.** | **Date** | **Remarks** |
| **1.** | **Write programme to conduct uninformed and informed search.** |  |  |  |
| **2.** | **Write a programme to conduct game search.** |  |  |  |
| **3.** | **Write programme to construct a Bayesian network from give data.** |  |  |  |
| **4.** | **Write programme to run value and policy iteration in a grid world.** |  |  |  |
| **5.** | **Write a programme to do reinforcement learning in a grid world.** |  |  |  |

**PRACTICAL NO :- 1**

**AIM :- Write a program to conduct uninformed and informed search**

**UNINFORMED SEARCH ALGORITHMS**

**Introduction:**

Uninformed search is one in which the search systems do not use any clues about the suitable area but it depends on the random nature of search. Nevertheless, they begin the exploration of search space (all possible solutions) synchronously. The search operation begins from the initial state and providing all possible next steps arrangement until goal is reached. These are mostly the simplest search strategies, but they may not be suitable for complex paths which involve in irrelevant or even irrelevant components. These algorithms are necessary for solving basic tasks or providing simple processing before passing on the data to more advanced search algorithms that incorporate prioritized information.

**Following are the various types of uninformed search algorithms:**

1. Breadth-first Search (BFS)
2. Depth-first Search (DFS)

**1. Breadth-first Search (BFS):**

* Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
* BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
* The breadth-first search algorithm is an example of a general-graph search algorithm.
* Breadth-first search implemented using FIFO queue data structure.

**Advantages of Breadth-first Search:-**

* BFS will provide a solution if any solution exists.
* If there are more than one solution for a given problem, then BFS will provide the minimal solution which requires the least number of steps.
* It also helps in finding the shortest path in goal state, since it needs all nodes at the same hierarchical level before making a move to nodes at lower levels.
* It is also very easy to comprehend with the help of this we can assign the higher rank among path types.

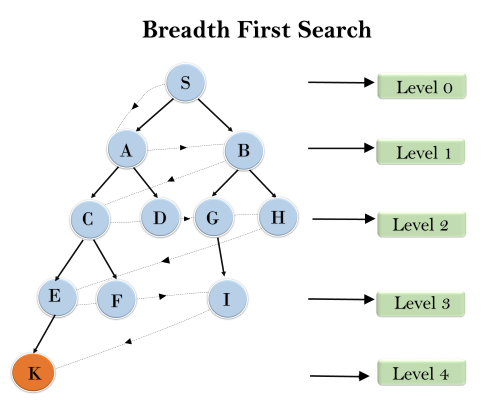
**Disadvantages of Depth-first**:-

* 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.
* It can be very inefficient approach for searching through deeply layered spaces, as it needs to thoroughly explore all nodes at each level before moving on to the next

**Example:**

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K

****

**Source code:**

public class BFS {  
 class Graph{  
 private int V;  
 private LinkedList<Integer>adj[];  
 Graph(int V)  
 {  
 V=v;  
 adj = new LinkedList[v];  
 for(int i=0;i<v;++i)  
 adj[i] = new LinkedList();  
 }  
 void addEdge(int v ,int w)  
 {  
 adj[v].add(w);  
 }  
 void BFS(int s)  
 {  
 boolean visited[]= new boolean[v];  
 LinkedList<Integer>queue=new LinkedList<Integer>();  
 visited[s]=true;  
 queue.add(s);  
 while(queue.size()!=0)  
 {  
 s=queue.poll();  
 System.out.println(s + " ");  
 Iterator<Integer> i=adj[s].listIterator();  
 while(i.hasNext())  
 {  
 int n = i.next();  
 if(!visited[n])  
 {  
 visited[n]=true;  
 queue.add(n);  
 }  
 }  
 }  
 }  
 }  
 public static void main(String[] args)  
 {  
 Graph g=new Graph(4);  
 g.addEdge(0,1);  
 g.addEdge(0,2);  
 g.addEdge(1,2);  
 g.addEdge(2,0);  
 g.addEdge(2,3);  
 g.addEdge(3,3);

System.out.println("following is Breadth First Traversal"+"(starting from vertex 2");  
 g.BFS(2);  
 }  
}

**OUTPUT:**

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

**2. Depth-first Search**

* Depth-first search isa 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 a stack data structure for its implementation.
* The process of the DFS algorithm is similar to the BFS algorithm.

**Note:** Backtracking is an algorithm technique for finding all possible solutions using recursion.

**Advantage:**

* 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 (if it traverses in the right path).
* With the help of this we can stores the route which is being tracked in memory to save time as it only needs to keep one at a particular time.

**Disadvantage:**

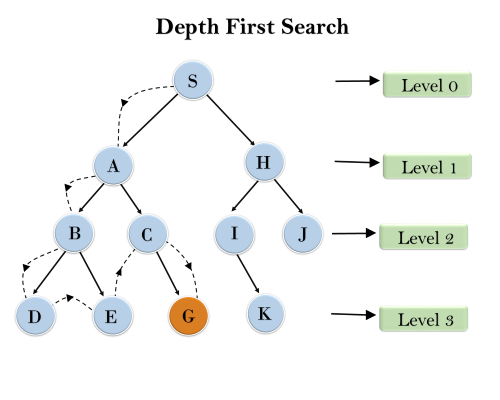
* There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
* DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.
* The depth-first search (DFS) algorithm does not always find the shortest path to a solution.

**Example:**

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



**Source Code:**

public class DFS {  
 class GfG{

static void addEdge(List<List<Integer> > adj,int s, int t)

{  
 adj.get(s).add(t);

adj.get(t).add(s);

}

Static void DFSRec(List< List<Integer> > adj,Boolean[] visited, int s)

{

Visited[s] = true;

System.out.print(s + " ");

For (int I: adj.get(s))

{

If( !visited[i]) {

DFSRec(adj,visited,i);

}

}

}

Static void DFS(List< List<Integer> > adj, int s)

{

Boolean[] visited = new boolean [adj.size()];

DFSRec(adj, visited, s);

}

public static void main(String[] args)  
 {  
 int V = 5;

List< List<Integer> > adj = new ArrayList<>(V);

for (int I = 0; I < V; i++)

{

Adj.add(new ArrayList<>());

}

Int[] [] edges = {

{1,2}{1,0}{2,0}{2,3}{2,4}

};

For (int[] e : edges) {

addEdge(adj,e[0],e[1]);

}

int source =1;

system.out.println(“DFS from source: “ +source);

DFS(adj,source);

}

**Output :**

DFS from source: 1

1 2 0 3 4

**INFORMED SEARCH**

**INTRODUCTION**

*Informed search algorithms*, also known as *heuristic search algorithms*, are an essential component of Artificial Intelligence (AI). These algorithms use domain-specific knowledge to improve the efficiency of the search process, leading to faster and more optimal solutions compared to uninformed search methods. By incorporating heuristics, informed search algorithms can make educated guesses about which paths to explore in a search space, ultimately reducing the time and computational resources required to find a solution**.**

**Informed Search Algorithms:**

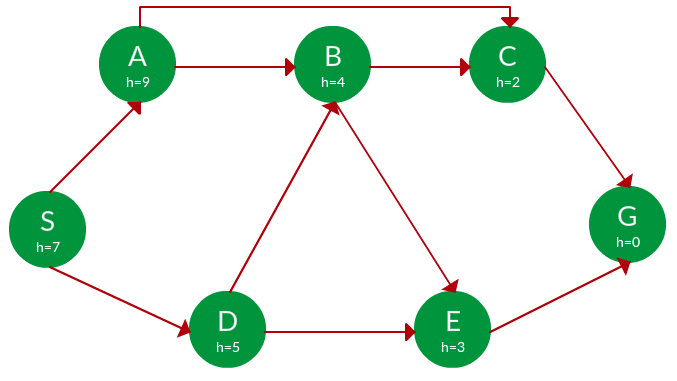
Here, the algorithms have information on the goal state, which helps in more efficient searching. This information is obtained by something called a *heuristic.*   
In this section, we will discuss the following search algorithms.

1. Greedy Search
2. A\* Tree Search
3. A\* Graph Search

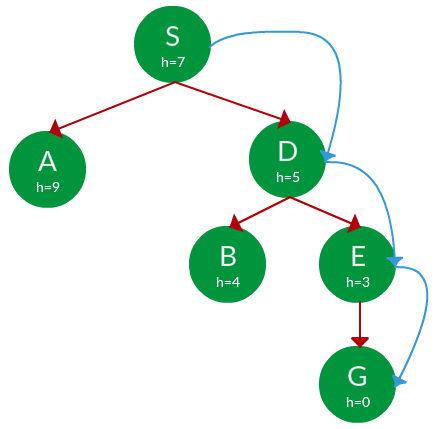
**Search Heuristics:**In an informed search, a heuristic is a *function* that estimates how close a state is to the goal state. For example – Manhattan distance, Euclidean distance, etc. (Lesser the distance, closer the goal.) Different heuristics are used in different informed algorithms discussed below.

**1.Greedy Search:**

In greedy search, we expand the node closest to the goal node. The “closeness” is estimated by a heuristic h(x).   
**Heuristic:**A heuristic h is defined as-   
h(x) = Estimate of distance of node x from the goal node.   
Lower the value of h(x), closer is the node from the goal.   
**Strategy:**Expand the node closest to the goal state, *i.e.* expand the node with a lower h value.   
**Example:**Find the path from S to G using greedy search. The heuristic values h of each node below the name of the node. 



**Solution.** Starting from S, we can traverse to A(h=9) or D(h=5). We choose D, as it has the lower heuristic cost. Now from D, we can move to B(h=4) or E(h=3). We choose E with a lower heuristic cost. Finally, from E, we go to G(h=0). This entire traversal is shown in the search tree below, in blue.



**Path:**   S -> D -> E -> G

**Advantage:**Works well with informed search problems, with fewer steps to reach a goal.   
**Disadvantage:**Can turn into unguided DFS in the worst case.

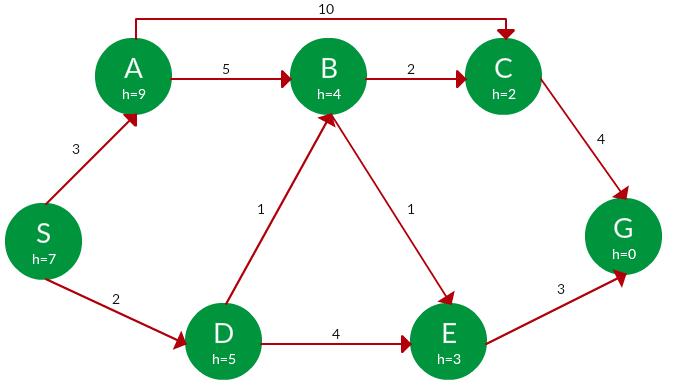
**2.A\* Tree Search:**

A\* Tree Search, or simply known as A\* Search, combines the strengths of uniform-cost search and greedy search. In this search, the heuristic is the summation of the cost in UCS, denoted by g(x), and the cost in the greedy search, denoted by h(x). The summed cost is denoted by f(x).   
**Heuristic:**The following points should be noted wrt heuristics in A\* search.

* Here, h(x) is called the **forward cost** and is an estimate of the distance of the current node from the goal node.
* And, g(x) is called the **backward cost** and is the cumulative cost of a node from the root node.
* A\* search is optimal only when for all nodes, the forward cost for a node h(x) underestimates the actual cost h\*(x) to reach the goal. This property of *A\** heuristic is called **admissibility**.

Admissibility:     
**Strategy:**Choose the node with the lowest f(x) value.   
**Example:**

 Find the path to reach from S to G using A\* search.



**Solution.** Starting from S, the algorithm computes g(x) + h(x) for all nodes in the fringe at each step, choosing the node with the lowest sum. The entire work is shown in the table below.   
  
Note that in the fourth set of iterations, we get two paths with equal summed cost f(x), so we expand them both in the next set. The path with a lower cost on further expansion is the chosen path. 

|  |  |  |  |
| --- | --- | --- | --- |
| **Path** | **h(x)** | **g(x)** | **f(x)** |
| **S** | **7** | **0** | **7** |
|  |  |  |  |
| **S -> A** | **9** | **3** | **12** |
| **S -> D** | **5** | **2** | **7** |
|  |  |  |  |
| **S -> D -> B** | **4** | **2 + 1 = 3** | **7** |
| **S -> D -> E** | **3** | **2 + 4 = 6** | **9** |
|  |  |  |  |
| **S -> D -> B -> C** | **2** | **3 + 2 = 5** | **7** |
| **S -> D -> B -> E** | **3** | **3 + 1 = 4** | **7** |
|  |  |  |  |
| **S -> D -> B -> C -> G** | **0** | **5 + 4 = 9** | **9** |
| **S -> D -> B -> E -> G** | **0** | **4 + 3 = 7** | **7** |

**Path:**   S -> D -> B -> E -> G   
**Cost:**   7 

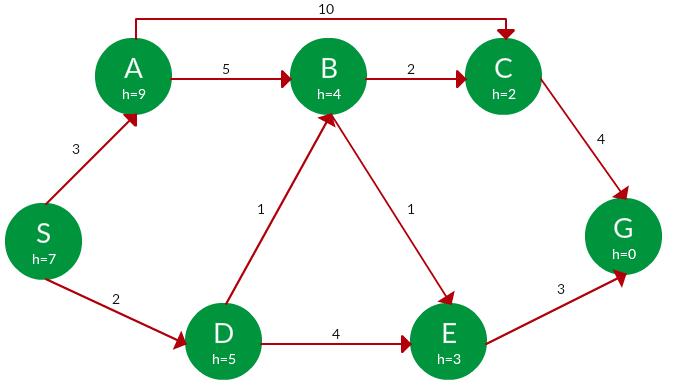
**3.**[**A\* Graph Search**](https://www.geeksforgeeks.org/a-search-algorithm/)**:**

* 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
* A\* Graph Search, or simply Graph Search, removes this limitation by adding this rule: **do not expand the same node more than once.**
* **Heuristic.**Graph search is optimal only when the forward cost between two successive nodes A and B, given by h(A) – h (B), is less than or equal to the backward cost between those two nodes g(A -> B). This property of the graph search heuristic is called **consistency**.

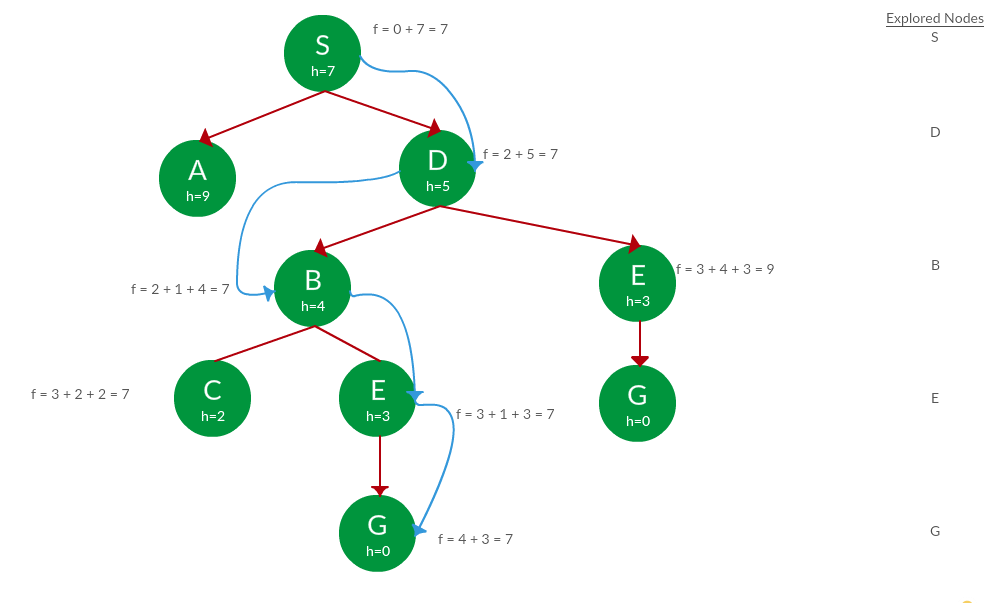
Consistency:

**Example:**

 Use graph searches to find paths from S to G in the following graph.



the **Solution.** We solve this question pretty much the same way we solved last question, but in this case, we keep a track of nodes explored so that we don’t re-explore them.



**Path:**   S -> D -> B -> E -> G   
**Cost:**   7