DEBARK UNIVERSITY

**College of Natural and Computational Science**

**Department of Computer Science**

**Individual Assignment**

**Course Title: Design and Analysis of Algorithms**

**Course Code: cosc3094**

Fikadu Getaye 1300358

**Submitted To: Mr. Shimels T.**

**Submition Date: 27-06-2016**

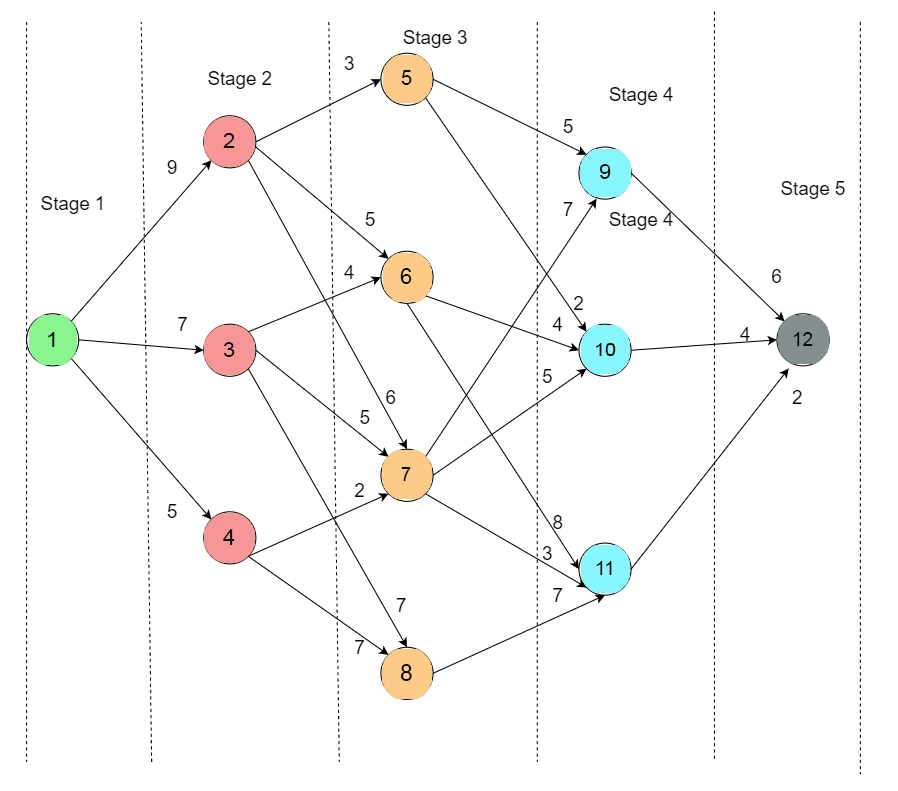
**Debark, Ethiopia**

**Multistage Graph**

A multistage graph falls under the class of a directed and weighted graph. Here the vertices are divided into stages. The first and last stages will have a single vertex representing the starting or ending vertex. Also, between the starting and ending vertex, there will be vertices in different stages. Which will connect the starting and ending vertex.

The main aim of this graph is to find the minimum cost path between starting and ending vertex. This article will discuss the multistage graph and its implementation.

Sample Example



In the above graph, the cost of an edge is represented as c(i, j). We have to find the minimum cost path from vertex 1 to vertex 12. We will be using this below formula to find the shortest cost path from the source to the destination:

cost(i,j)=minimum{c(j,l)+cost(i+1,l)}

Step 1

In Step 1, we use the forward approach (cost(5,12) = 0).

Here, 5 is the stage number and 12 is a node. Since there are no edges outgoing from vertex 12, the cost is 0.

Step 2

Now, cost(4,9) = cost(9,12)= 6

cost(4,10) = cost(10,12)= 4

cost(4,11) = cost(11,12)= 2

Step 3

cost(3,5) = minimum{cost(5,9)+cost(4,9),c(5,10)+cost(4,10)}

minimum{5+6,2+4}

minimum{11,6} = 6

cost(3,5) = 6

cost(3,6)=minimum{cost(6,10)+cost(4,10),cost(6,11)+cost(4,11)}

minimum{4+4,8+2}

minimum{8,10}=8

cost(3,6)=8

cost(3,7)=minimum{cost(7,9)+cost(4,9),cost(7,10)+cost(4,10),cost(7,11)+cost(4,11)} minimum{7+6,5+4,3+2}

minimum{13,9,5}=5

cost(3,7)=5

cost(3,8)=cost(8,11)+cost(4,11)=7+2=9 cost(3,8)=9

Step 4

cost(2,2)=minimum{cost(2,5)+cost(3,5),cost(2,6)+cost(3,6),cost(2,7)+cost(3,7)} minimum{3+6,5+8,6+5}

minimum{9,13,11}=9

cost(2,2) = 9

cost(2,3)=minimum{c(3,6)+cost(3,6),cost(3,7)+cost(3,7),cost(3,8)+cost(3,8)} minimum{4+8,5+5,7+9}

minimum{12,10,16}=10

cost(2,3)=10

cost(2,4)=minimum{cost(4,7)+cost(3,7),cost(4,8)+cost(3,8)}

minimum{2+5,7+9}

minimum{7,16}=7

cost(2,4)=7

Step 5

cost(1,1)=minimum{cost(1,2)+cost(2,2),cost(1,3)+cost(2,3),cost(1,4)+cost(2,4)} minimum{9+9,7+10,5+7}

minimum {18,17,12}=12

cost(1,1)=12

**Optimal Binary Search Trees**

A binary search tree is a special kind of binary tree in which the nodes are arranged in such a way that the smaller values fall in the left subnode, and the larger values fall in the right subnode.

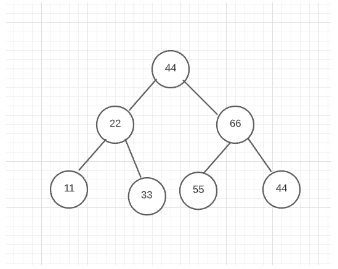
So now, what is an optimal binary search tree, and how are they different than normal binary search trees. The cost of searching a node in a tree plays a very big role, and it is decided by the frequency and a key value of the node. Often, our goal is to reduce the cost of searching, and that is done through an optimal binary tree.

For our better understanding, we can take the following example:

let us assume the binary tree has the following keys:

11, 22, 33, 44, 55, 66, 77.

So the binary tree will be arranged in the following manner:



As we can see the root node is 44, and the smaller values with respect to the root value, are placed in the left subtree, and the larger values with respect to the root value are placed in the right subtree.

Now moving on, we will check how many binary search trees can be made by using the keys we have taken in our example:

The formula for it is: 2n Cn / n+1

Therefore, the total number of binary search trees that can be created with the keys mentioned is 5.

Now to calculate the cost for the operation will depend on how many comparisons it needs to make, so we will find that by the following method:

As we can see in the tree mentioned above, we will need to make 3 comparisons, so the average number of comparisons will be: 1+2+3/3 = 2. Similarly, We will find the comparisons for various other forms of BST using the same keys.

After having done the comparisons we see that the tree mentioned below has the least number of comparisons, and hence the least cost, and qualifies as a balanced binary search tree, or the optimal binary search tree.

**Depth First Search (DFS)**

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

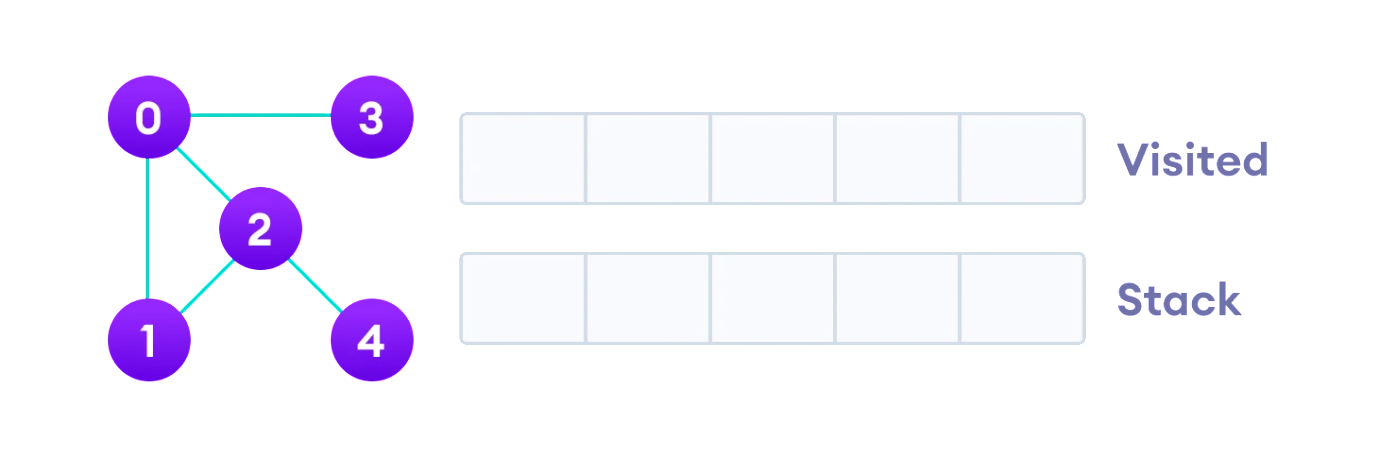
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

**The DFS algorithm works as follows:**

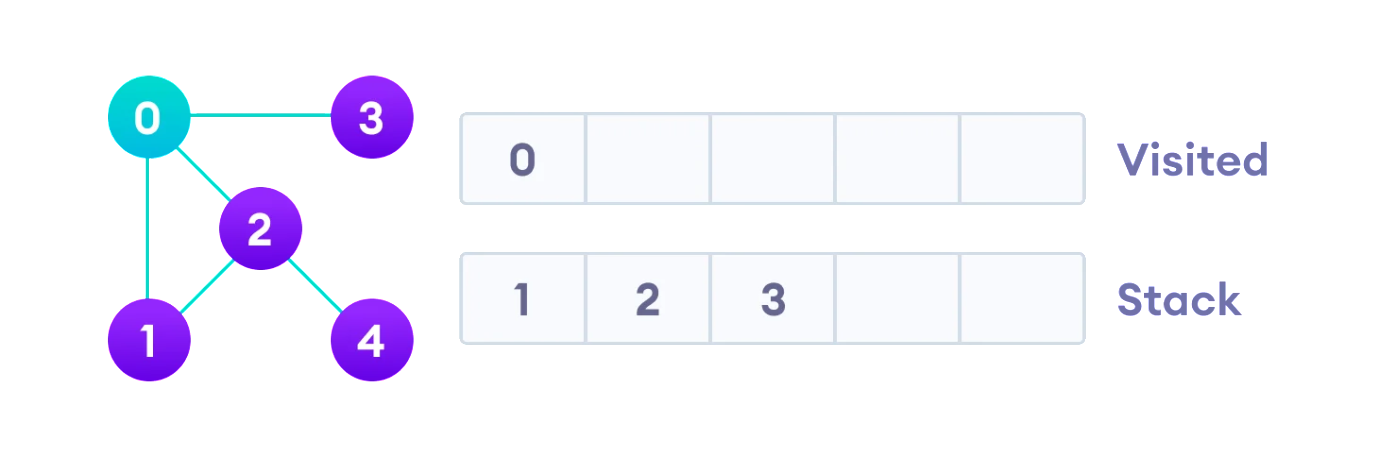
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

**Depth First Search Example**

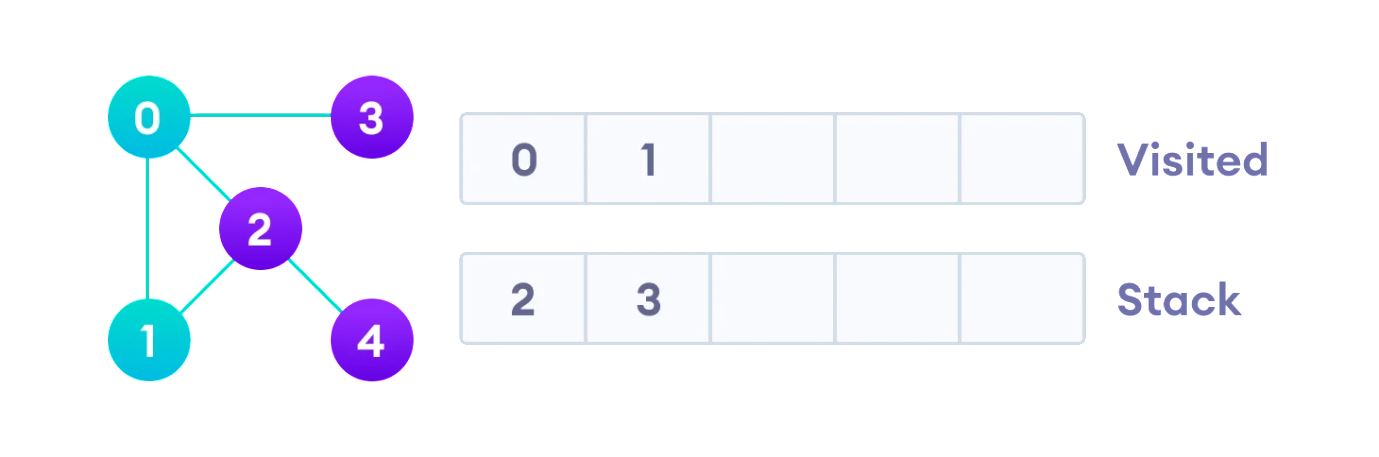
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



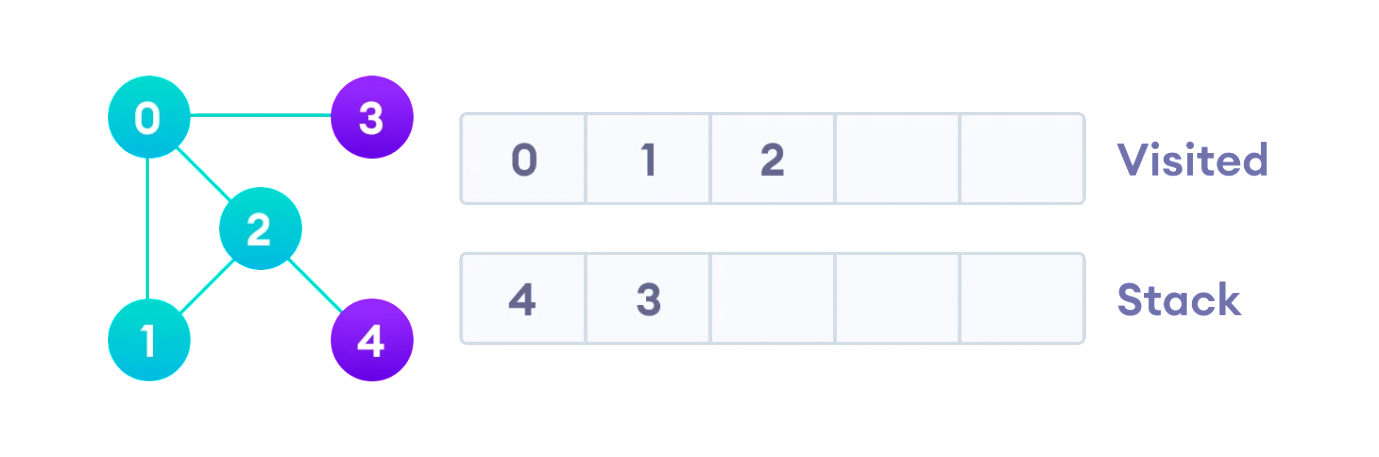
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

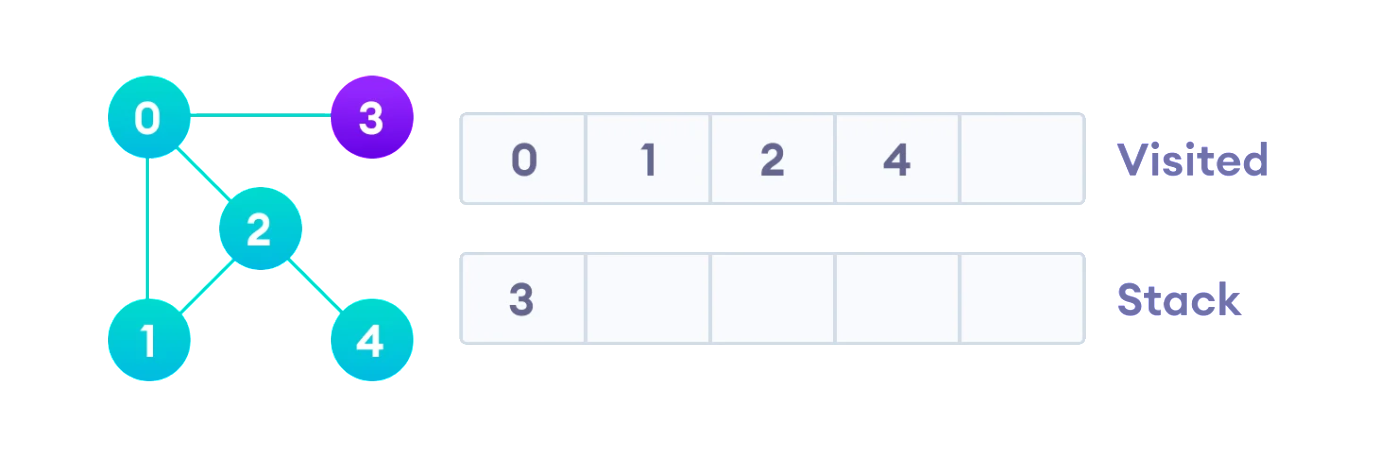


Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.





After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

