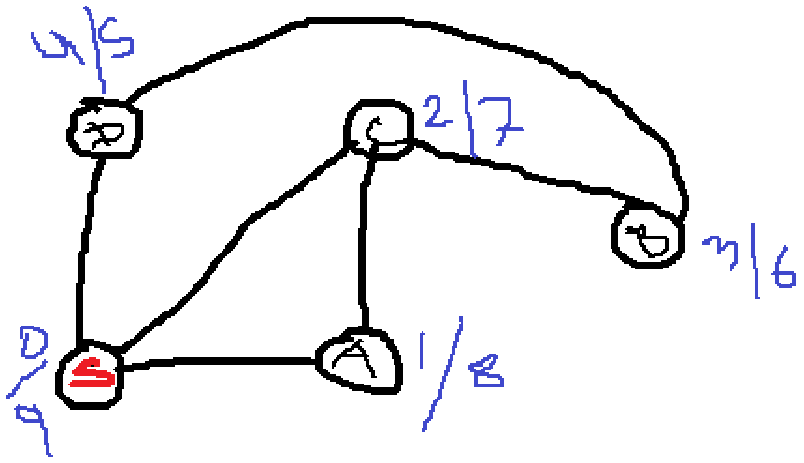
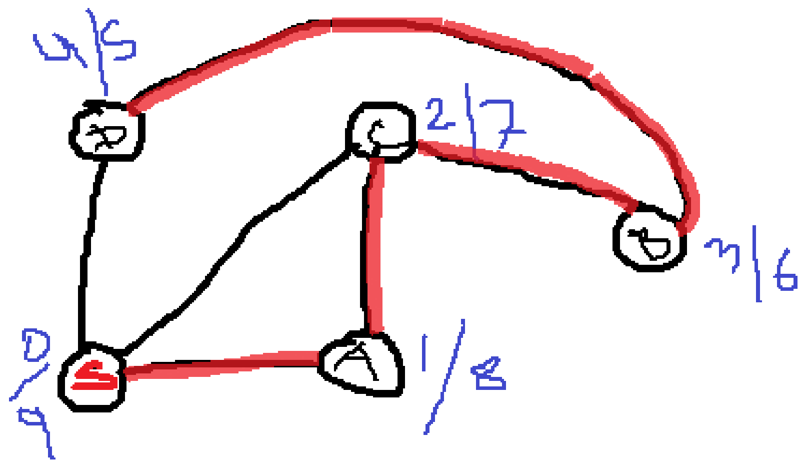
Graph Search Algorithms - II (DFS)

I will be discussing another technique of graph search called the depth first search (DFS). Well most of us have implemented DFS in real world unknowingly. When we are searching for a shop for example, we start from a point and keep going deep searching for that shop. If we come to a dead end we back track and choose another path and start searching. DFS is implemented when we solve a maze. Now let us implement DFS on the following graph.



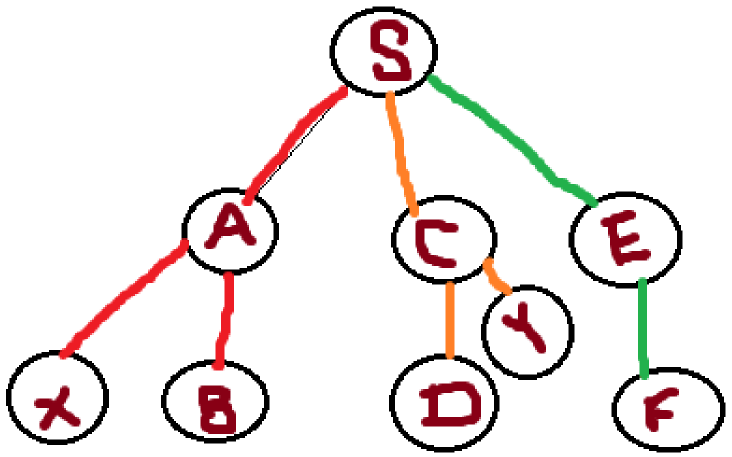
We start from the source vertex s. There are 3 adjacent **unvisited** vertices from s. We visit any one of them. DFS is usually implemented using a time stamp, that is, we will keep record (starting time) when a vertex is visited and an ending time when all the adjacent vertices of that vertex are visited. Therefore when we start from s, it starting time is 0. Let us go to A and the starting time will be 1. From A there are 2 ways - to C and back to A. As A is visited will go to C and record its starting time as 2. From C B is the only unvisited one so visit B and from B to D. Record their starting times accordingly.
Now from D there is no more unvisited vertex therefore we back track to that vertex from where we reached to D. We can do that by looking at the starting times. D has the starting time of 4 and we came to D from a vertex with starting time 3. As we are back tracking from D it means dealing with D is complete and therefore we will give it an ending time which is 1 + the last starting/ending time.
We can use color notion just like we did in BFS. White for unvisited, grey for visited but not finished yet and black for the ones completely dealt.

While running DFS all the paths were not used. If the ones used are picked from the graph we will see a tree has been formed. For the above graph the paths used are shown below marked in red.



This tree is called the DFS tree and the edges used are called **tree edges**. The edges not touched are called **back edges**.

Implementation of DFS: After reading all the explanation about the working of DFS, it is quite obvious to think the implementation would be quite complex. But fortunately it is not. If you look closely DFS on a vertex is just calling DFS on all its neighbors. If not convinced then follow the diagram below.



S is the source and we run DFS on S. Let us forget starting time and ending time for now. DFS on S means for each neighbor of S we will go downward selecting one vertex at a time. If we select A first, we will go downward for each neighbor of A selecting one vertex at a time, which is basically DFS on A. When all the neighbors of A are visited we visit each neighbor of C, which is DFS on C. Therefore we can conclude that DFS on a vertex is calling DFS on all its neighbors. Hence it is recursion.

The algorithm of DFS looks like this:

```
DFS(S)
    colour[S]=grey
    for each adjacent vertex v of S
        if (colour[v]==white)
            DFS(v)
```

Now the question that might drive you crazy is where is **backtracking!?!** I have mentioned it so much and its significance while explaining but I have not put it anywhere in the algorithm. WHY? Well the answer is the algorithm is recursive and backtracking is a 'part' of recursion that does not need to be explicitly handled. This is the beauty of recursion that it eases the job for us.
Now if we bring the starting and ending time, we need to make a minor modification of the above algorithm.

```
time = 0;
DFS(S)
    colour[S]=grey
    starting_time[S]= time++
    for each adjacent vertex v of S
        if (colour[v]==white)
            DFS(v)
    colour[S]=black
    ending_time[S]=time++
```

We have heard or learned that DFS uses a stack. Well yes, it does. Recall what happens during recursion. Each function call is saved in a stack. As DFS is a recursive algorithm we do not need to use a stack separately.

If you are still not convinced with the recursive procedure, I have broken it down using loops and stack for understanding.

```
for every vertex v in graph G
    colour[v] = white
// let source vertex be denoted as u
time = 0
push u in stack S
colour[u] = grey
starting_time[u] = time++
while (S not empty)
    d = pop[S]
    for each neighbor vertex v of d
        if (color[v]==white)
            push[S]=v
            colour[v] = grey
            starting_time[v] = time++
    colour[d]=black
    ending_time[d]=time++
```