KOSARAJU'S ALGORITHM

How it Works

- 2 pass algo
- goes thru graph twice

to find strongly connected components

1st pass :

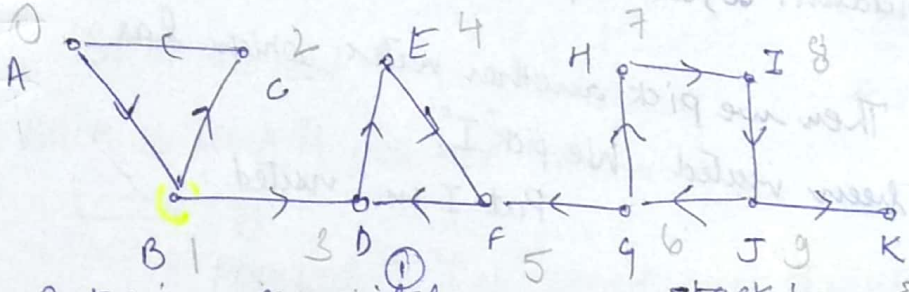
(i) DFS on graph

(ii) Order the vertices of this graph by finish times in decreasing order

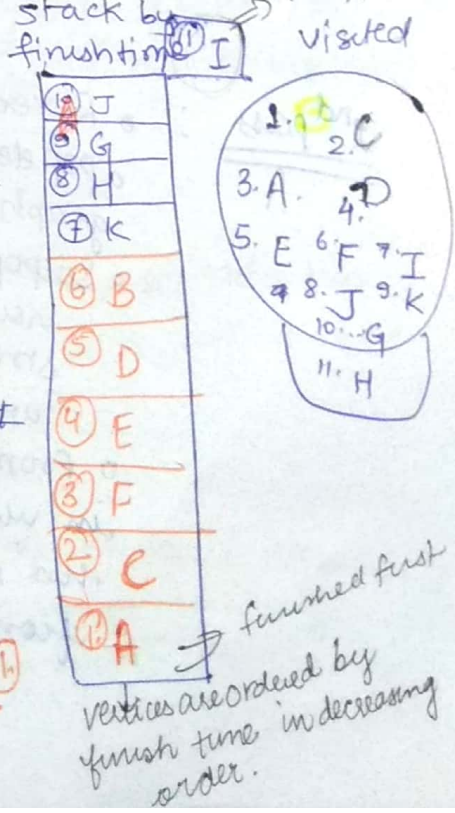
2 Data structures :

* Set : to keep track of all visited vertices
* Stack : to order vertices by finish time.

① First pass of algo:
We can start from any vertex. Let's start from B.



- Put B in ~~stack~~ visited.
- explore the children of B. Let's say go to C
- C is not visited put C in visited. ②
  Explore the children of C (only one child A). & A is not visited so put A in visited ③
- Explore children of A, it's B. ~~So put~~ B is already visited. So don't go to B. So at this point we're done visiting all children of A ⇒ A is done & finished. So A is first vertex to finish. So let's put A ④ in stack.



* Then go back to recursion to goto C to C has no other children so C is totally done visiting C. done, so we're So we put C in stack (2)

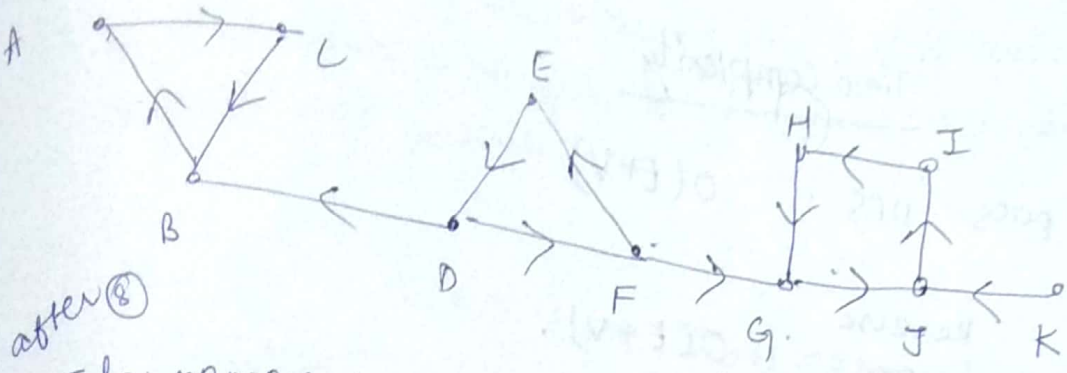
* Then we come to B, B has another child D so we go in direction of D. to D is not visited so we'll put D in visited. (4)

* G done. F has child D but D is already visited so we go back to F. F has no more children to be explored so we'll put F in stack. (3)

* After (5) from D we go back to B, B has explored all its children. So, we'll put B into stack.

" Then we pick another vertex which has not been visited. We pick "I".
Put I in visited.

2nd pass :
- Reverse the graph.
- pop elements out of stack & do a DFS on reversed graph (1)
- So pop I, since I is not visited, put I in visited. (2) & thus I is my first strongly connected component starting with I.
- from I go to H, since H not visited, put H in visited & also add H as a part of this strongly connected component. & from H we go to G.

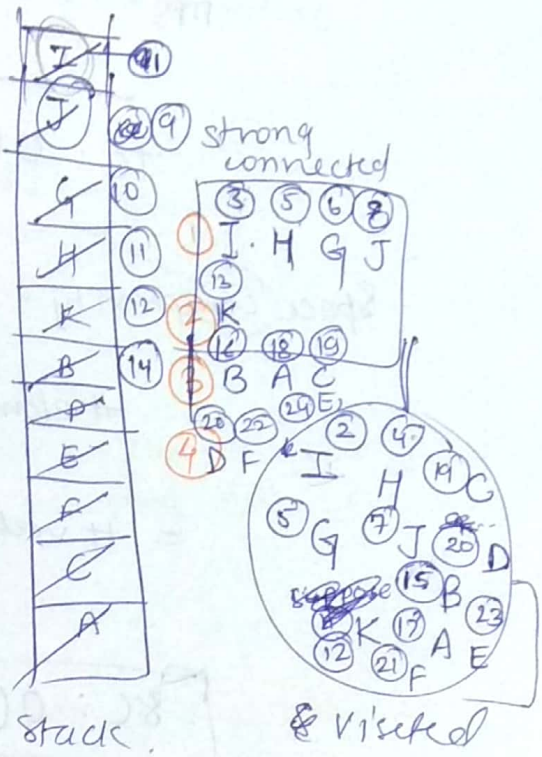


after (8)

J has no more children to be explored. G has no more children.

~~I has no~~ H has "

I has no "



So, we are done exploring all nodes from vertices from I.
so, this is one first component.

So, at this point,

Go Back to stack & pop next element

Pop out J, J is already part of visited, so donoting

IHGJ — POP G, G is "

BAC — POP H, H "

DFE — POP K, (K) is not visited, so add K in visited.

Start DFS from K

(13) K has no child to be explored other than J
so K is only ~~same~~ vertex of this strongly connected compo.

Time complexity

1st pass DFS : $O(E+V)$

Reverse DFS : $O(E+V)$.

TC : $O(E+V)$

Space complexity :

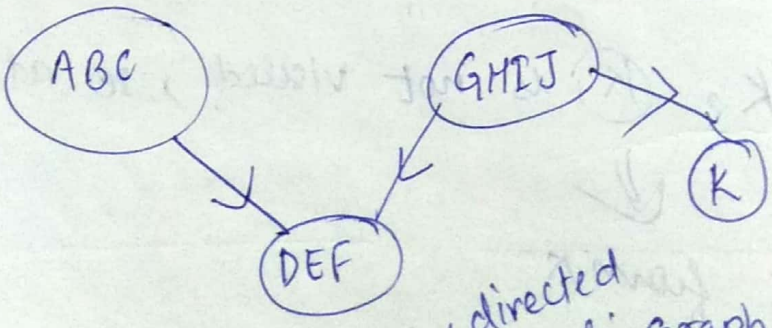
elements in stack

= # vertices = V = # vertices in visited set.

SC : $O(V)$

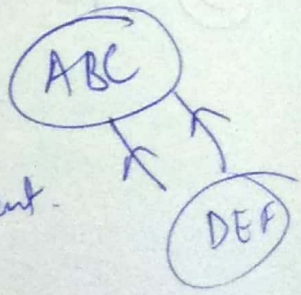
---

2. WHY IT WORKS — make connected component into one vertex & create a new graph.



This graph is guaranteed directed acyclic graph.

Because if there's an edge from ↑ then this combined together would be 1 strongly connected component.

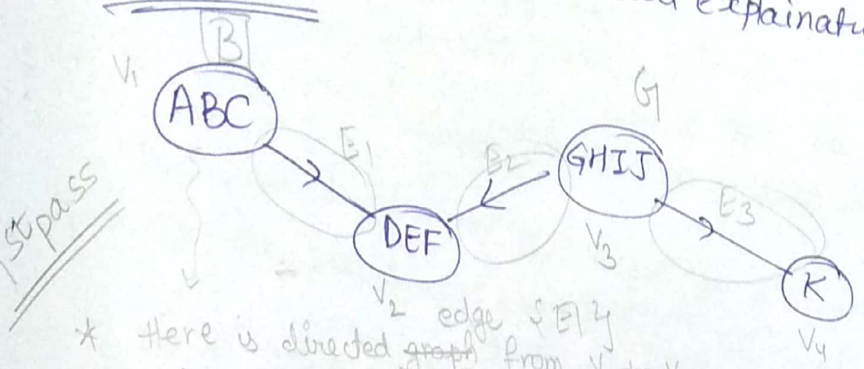


cout << endl;

The fact that we are having this 2 diff vertices is that there is no cycle & it means that this edge is not possible.

CLRS book - Full explaination

Intuition :



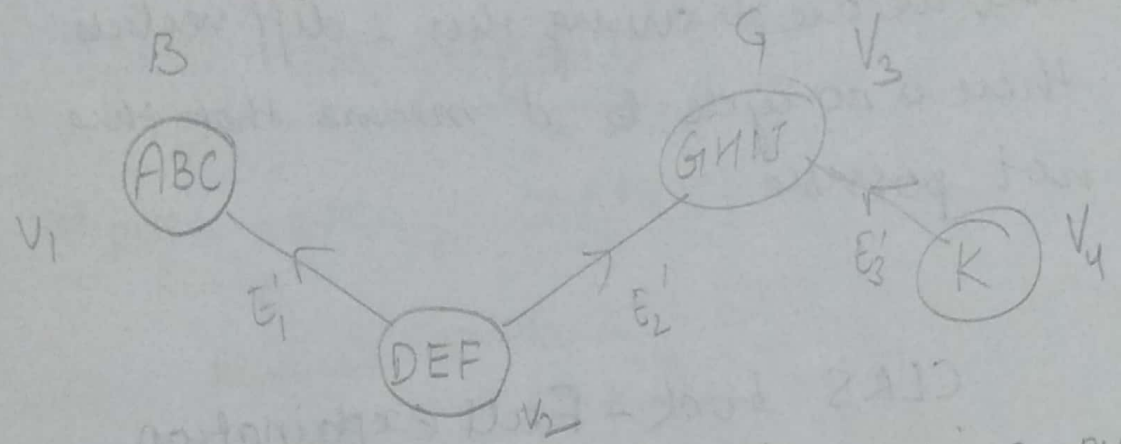
1st pass

* Here is directed ~~graph~~ from V_1 to V_2. $\{E1\}$
This shows that there is an atleast one vertex in ABC which will end after exploring DEF.
{Let's say it is B}

* Here is also directed edge from V_3 to V_2. $\{E_2\}$
This shows that there is atleast one vertex in GHIJ which will end after exploring DEF
{Let's say it is G1}

* edge from V_3 to V_4 $\{E_3\}$
so, there is atleast one vertex in GHIJ which will end after exploring K.
{Let's say it is G1}.

2nd pass Reverse the edges *



* So here we see that E_1' & E_2' pointing out of V_2. So atleast one among DEF must be there that will end after exploring ABC & GHIJ

* & since K is dependent on V_3 to end, it will finish up after GHIJ completes its execution.

* Let's say ABC finished exploring & there is no dependency among ABC given the fact that they are strongly connected, even if graph edges are reversed among ABC, they all are reachable.

& GHIJ finished exploring, then DEF can explore or K can explore in any order

* So, we got 4 strongly connected components

This proof is intuitive. Full proof can be seen in CLRS book.

CODE
17:20

```
#include <bits/stdc++.h>
using namespace std;
class Graph
{
  public:
      int V;
      list<int> *l;
      list<int> *m;

      Graph(int v)
      {
         V = v;
         l = new list<int>[v];
         m = new list<int>[v];
       }

      void addEdge(int u, int v)
      {
          l[u].push_back(v);
          m[v].push_back(u);
       }
       void DFSutilReverse(int v, bool visited[])
       {
           visited[v] = true;
           cout << v << ", ";
           for(auto itr = m[v].begin();
                 itr != m[v].end(); ++itr)
           {
              if (visited [*itr] == false)
                 DFSutilReverse(*itr, visited);
           }
       }
  void showGraph()
  { for (int i = 0; i < V ; i++)
      { cout << i << "->";
        for(auto itr = l[i].begin();
              itr != l[i].end(); ++itr)
                 cout << *itr << ", ";
  } } cout << endl;
```

```
void fillOrder(int v, bool visited[], stack<int> &Stack)
{
    visited[v] = true;
    list<int>:: iterator i;
    for(i = l[v].begin(); i != l[v].end(); ++i)
        if (visited[*i] == false)
            fillOrder(*i, visited, Stack);

    Stack.push(v);

}

void printSCC()
{
    stack<int Stack>
    bool *visited = new bool[V];
    for (int i=0; i<V; i++)
        visited[i] = false;
    for(int i=0; i<V;i++)   // Fill instack
        if (visited[i]==false)
            fillOrder(i, visited, Stack);

    for(int i=0; i<V; i++)
        visited[i] = false;

    while(Stack.empty() == false)   // process vertices order defined by Stack
    {
        int v = Stack.top();
        Stack.pop();
        if (visited[v] == false)
        {
            DFSUtilReverse(v, visited);
            cout<<endl;
        }
    }
}
};
```

```
while ( Stack.empty() == false)
{ cout << stack.top() << " ";
  stack.pop();
}
```

(20)

20

```
int main()
{
  int V = 11;
  Graph g(V);
  g.addEdge(0,1);
  g.addEdge(1,2)
```

2,0
1,3
3,4
5,3
4,5
6,5
6,7
9,6
7,8
8,9
9,10

```
  g.showGraph();
  cout<<" "<<"\n";
  g.printSCC();
  return 0;
}
```

OUTPUT:

0 → 1
1 → 2,3
2 → 0
3 → 4
4 → 5
5 → 3
6 → 5,7
7 → 8
8 → 9
9 → 6,10
10 →

6,9,8,7
10
0,2,1
3,5,4