18/9/2 Graph search

Goal of a graph search

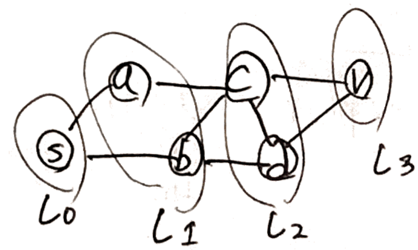
⇒ Find everything findable from a given start vertex.
⇒ Don't explore any thing twice.

BFS vs DFS's difference

⇒ How to choose the frontier vertex is different.
B ⇒ Using queue (FIFO) and get shortest path.
D ⇒ Using stack (LIFO) and explore aggresively.
B ⇒ Compute ~~Comp.~~ connected components of an undirected graph.
D ⇒ Compute connected components of an directed graph.
⇒ Either is $O(m+n)$ complexity.
↑ # of edges ↑ # of nodes

BFS

- Explore nodes in "layers".
- ~~Run~~ Can compute shortest path.

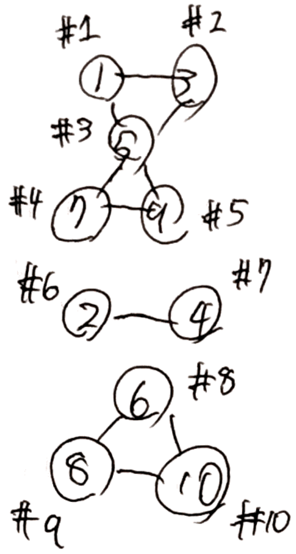


Application
⇒ Check undirected connectibity.

Application : Connected components via BFS

To compute all components

- all nodes unexplored. [all nodes labled from 1 ~ n]
- For i = 1 to n
  - If i not yet explored
    - BFS(G, i)



1st Loop : i = 1 , explore ~~from 1 ~~~ 1 3 5 7 9.

2nd loop : i = 2, explore 2, 4.

3rd loop : i = 3, ⇒ already explored skip!

4th loop : i = 4, ⇒ skip

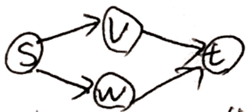
5th loop : i = 5, ⇒ skip

6th loop : i = 6, explore 6 8 10

⇒ Very useful to find connection between two nodes.

DFS

- explore aggresively deeper (Immidiate neighber). Only back track when necessary.
- Can get a topological ordering of a directed acyclic graph.
  Label of nodes counted from starting node in right order.



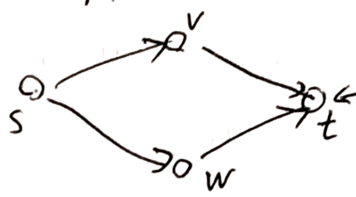
$\begin{cases} \overset{1}{s} \to \overset{2}{v} \to \overset{3}{w} \to \overset{4}{t} \\ s \to w \to v \to t \end{cases}$ ⇒ Both right topol. order.

Straight forward Solution for topological order.

※ Note : Every directed acyclic graph has a sink vertex.

(Otherwise it will have directed circle)

- Let v be a sink vertex of G.
- Set $f(v) = n$
- Recurse of $G - \{v\}$ (minus)



Sink vertex (No outbound edge)

1. Set t as sink vertex of G
2. Set t label = 4
3. Delete t from G.
4. Now v and w become the sink vertecies.
5. Either v or w can be next sink ~~vertice~~ vertex and choose w.
6. Set w label = 3
7. Delete w from G.
8. Now v is only sink vertex.
9. Set v label = 2
10. Delete v from G
11. Now s is only sink vertex
12. Set s label = 1

⇒ Can run in linear time.

But DFS is more vercitle.

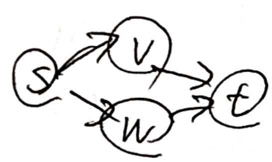
DFS for topological order

- Outer for loop.

DFS loop (graph G)
- Mark all nodes unexplored
- Global ver. "Current_label" = n (Count down for each labeling) (To keep track of ordering)
  - for each vertex v ∈ G:
    - if v not yet explored by previous DFS calls.
      - DFS (G, v)

DFS (graph G, start node v)
- mark v explored.
- for every edge (s, v):
  - if v not yet explored
    - DFS (G, v)
- Set f(s) = current_label
- ~~current_label -= 1~~

e.g.



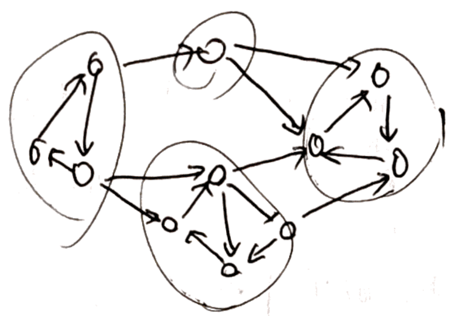
1. Currenlabel = 4
2. Select first vertex as V. Do DFS.
3. Only distanation from V is t.
4. Set f(t) = 4
5. Decrement the current_label to 3.
6. Set f(v) = 3
7. Decrement current_label to 2.
8. 2nd loop. skip t.
9. 3rd loop. Select s for DFS.
10. S can go to v and w. but v is explored.
11. Set f(w) = 2

:

Strongly connected Components (SCCs)

Meaning: SCCs of a directed graph G has following property.

$\Rightarrow$ path $\underbrace{u \rightsquigarrow v}_{\text{Arbertrary source and dist.}}$ and path $v \rightsquigarrow u$ exists in G

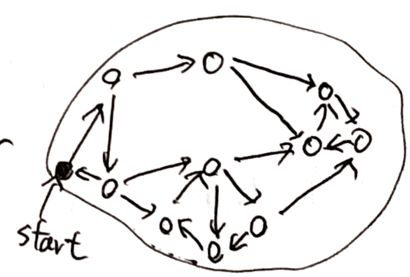


- Each component in circle is strongly connected because it is a directed circle.
- But, as a total graph, it is not a SSCs because it's not possible to start from right most region to left most region

DFS is useful method to defect the SCCs

→ Defects the SCCs

However



It might discovers entire graph if it start from a wrong node.

⟸

Need some trick to defect the right size of SCCs.

Kasaraju's Two-pass Algorithm

Theorem: Can compute the SCCs in $O(m+n)$ time.

Steps:
1. Let G^{rev} = G with the all edges direction reversed.
2. Run DFS-loop on G^{rev} ← Goal: Compute the "magical order" of nodes.
3. Run DFS-loop on G ← Goal: Discover the SCCs one-by-one.

DFS-Loop

global variable $t = 0$ [For "finishing times" in 1st pass]
[# of nodes processed so far]

global variable $s = Null$ [For label "Leader node" in 2nd pass]
[current source vertex]

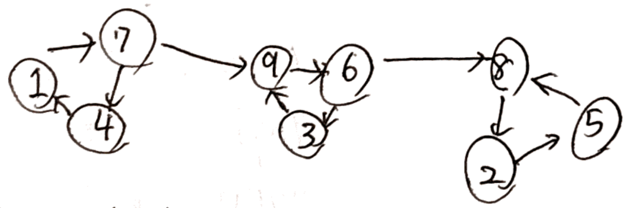
Nodes labelled from 1 to n.

```
For i=n down to 1:
    if i not yet explored
        s:=i
        DFS(G,i)
```

```
DFS (graph G, node i)
  - mark i as explored (For rest of DFS-loop)
  - Set leader(i):= node s
  - for each arc (i,j) ∈ G:
      - if j not yet explored.
          - DFS(G,j)
  - t++
  - set f(i):= t
```

$f(i)$: Finishing time

e.g. [1st DFS-loop on G^{rev}]

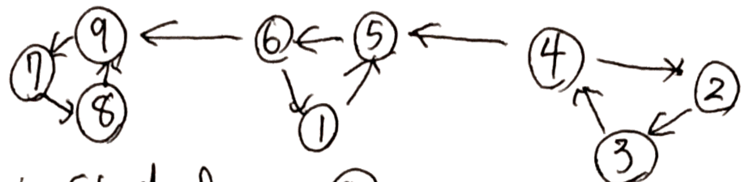


Nodes	1	2	3	4	5	6	7	8	9
Finishing time	7	3	1	8	2	5	9	4	6

1. Start from 9
2. Go to 6
3. Go to 3 (or 8)
4. Dead end $f(3) = 1$
5. Backtrack to 6
6. Go to 8
7. Go to 2
8. Go to 5
9. Dead end $f(5) = 2$
10. Backtrack to 2, $f(2) = 3$
11. Back track to 8, $f(8) = 4$
12. Back track to 6, $f(6) = 5$
13. Back track to 9, $f(9) = 6$
14. Skip 5 and resume DFS from 7 ......

e.g. [2nd DFS-loop]

- Changed all node names to the finish time from 1st-DFS.
- Bring back the directions of graph back.

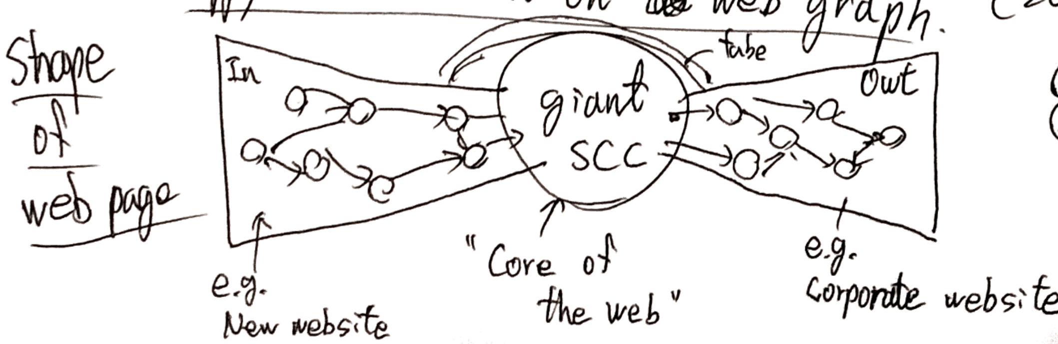


1. Start from 9, S:=9 start DFS
2. Go to 7
3. Go to 8, deadend
4. Mark 7 8 9 as SCCs. with 9 as leader node.
5. Skip 7, 8 due to the fact those already discovered.
6. Start DFS from 6, S:=6
7. Go to 1
8. Go to 5 deadend.
9. Mark 1 5 6 as SCCs with 6 as leader node.
10. Skip 5
11. Start 4 DFS, S:=4
12. Go to 2
13. Go to 3 deadend.
14. Mark 2 3 4 as SCCs, with 4 as leader node.

⟹ Works to find SCC generally.

Application

Apply SCC search on ~~a~~ web graph. (2000's research)



① all 4 parts (Giant, In, Out, tube)
② Within CORE, very well connected.

Proper iterative DFS

DFS 12

loop1: stack[12] test=12 ex[12] ed[11]
loop2: Stack[11] test=11 ex[12,11] ed[10]
loop3: st[10] test=10 ex[12,11,10] ed[7, ~~12~~]
loop4: st[7] test=7 ex[12,11,10,7] ed[4,5,8,9]
loop5: st[4 5 8 9] test=9 ex[12,11,10,7,9] ed[~~10~~]

9.FT = 1

BT 9 7 = 9.parents.

explored ~~deadend~~ 7 = False

loop6: St[4 5 8] test=8 ex[12,11,10,7,9,8] ed [6,~~7~~]

loop7: st[4 5 8 6] test=6 ex[12,11,10,7,9,8,6] ed [3, 5]

loop8: St[4 5 3 5] test=5 ex[12,11,10,7,9,8,6,5] ed (2,~~4~~)

loop9: St [4 5 3 2 4] test=4 ex[12,11,10,7,9,8,6,5,4] ed [2]

loop10: St [4 5 3 2 2] test 2 ex[12,11,10,7,9,8,6,5,4,2] ed [1]

loop11: St [4 5 3 2 1] test 1 ex[12,11,10,7,9,8,6,5,4,2,1] ed [ ]

1.FT= 2

BT(1) 2 = 1 parents

explored 2 = True 2 FT=3

BT(2) 4 = 2 parents

explored 4 = True 4.FT= 4

BT 4 5 = 4 parents

explored 5 = True 5.FT=5

BT 5 6 = 5 parents

explored 6 = False

loop 12: 1 Skip

loop 13: 2 Skip

loop 14: [4 5 3] test 3 ex[all] ed ~~[2]~~

3.FT=6

BT 3 6 = 3.parents

explored 6 = True 6 FT=7

BT 6 8 = 6 parents 8 FT= 8 11.FT=11

7 FT=9 12 FT=12

10 FT = 10