

DAAPY 11

Wednesday, November 15, 2023 3:00 PM

----- class 9 11/1/2023 -----

1

----- class 10 11/8/2023 -----

2

----- class 11 11/15/2023 -----

3

----- class 12 11/22/2023 -----

NO CLASS. THANKS GIVING ~~FALL~~ BREAK

Graph Alg

----- class 13 11/29/2023 -----

4

----- 12/6/2023 -----

NO CLASS

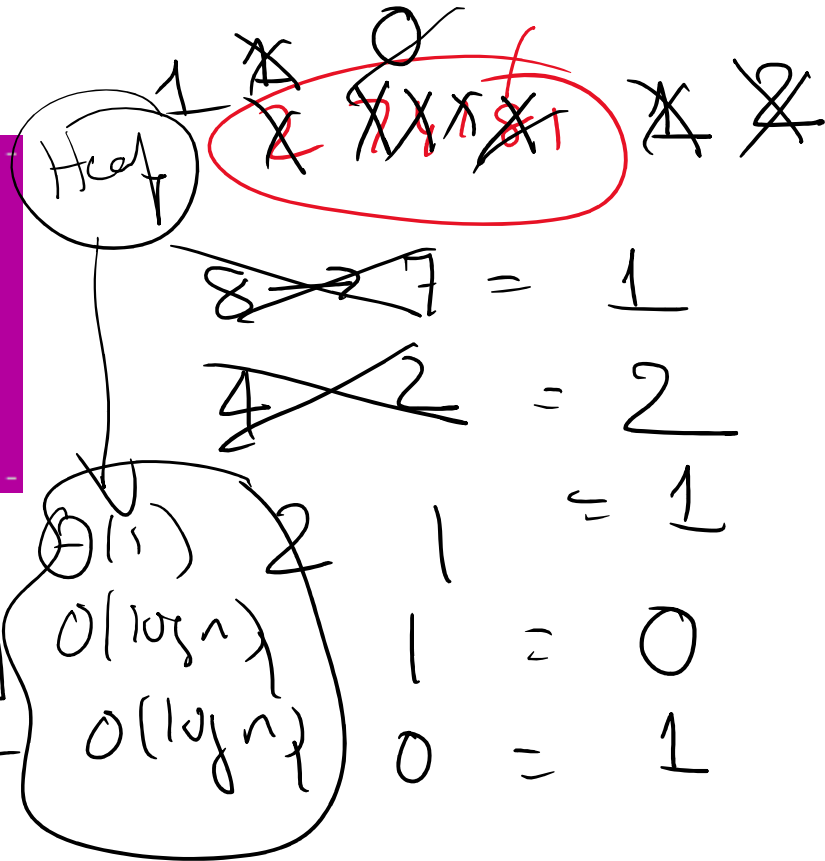
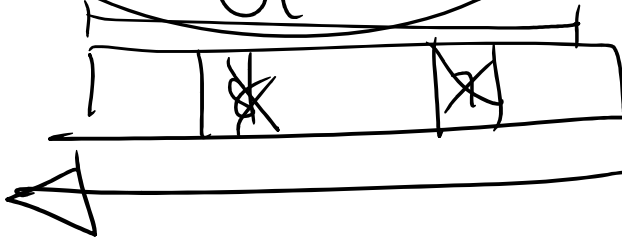
----- class 14 12/10/2023 ---
FINAL

9 to 12 Noon
PST

```

-----PROBLEM 1-----
0 1 2 3 4 5
2 7 4 1 8 1
Shrink problem. Expected ans = 1
Step 1 Combining: 3 7 weight= 1
Step 2 Combining: 4 2 weight= 2
Step 3 Combining: 2 1 weight= 1
Step 4 Combining: 1 1 weight= 0
Step 5 Combining: 1 0 weight= 1
-----PROBLEM 2-----

```

 $\Theta(n)$


True min heap

```

class Heap:
    def __init__(self, minn: "bool"):
        self._q = []
        Node.minheap = minn

    def insert(self, a: "list of int"):
        for e in a:
            n = Node(e)
            heapq.heappush(self._q, n)

    def add(self, a: "one int"):
        n = Node(a)
        heapq.heappush(self._q, n)

    def get_top(self) -> "int":
        return self._q[0].getdata()

    def get_top_and_remove(self) -> "int":
        n = heapq.heappop(self._q)
        d = n.getdata()
        return d

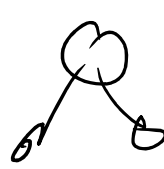
    def __len__(self):
        return len(self._q)

```

$n \log n$
 $O(\log n)$
 $\log n$
 $O(1)$
 $O(\log n)$

$h = \text{Heap}(\text{True})$

True or False



```
#####
```

```
class Node:
```

```
# static variable
```

```
minheap = True
```

```
def __init__(self, a: "int"):
```

```
    self._a = a
```

```
##Override __lt__
```

```
def __lt__(self, rhs: "Node") -> "bool":
```

```
    # print("HERE", self._a, rhs._a)
```

```
    if Node.minheap:
```

```
        if self._a < rhs._a:
```

```
            return True
```

```
        return False
```

```
    else:
```

```
        if self._a > rhs._a:
```

```
            return True
```

```
        return False
```

```
def getdata(self):
```

```
    return self._a
```

False

Self -

a

b

True

minh

MAX

```
def _alg(self):
    # build minheap for grow - grow is True
    # build maxheap for shrink - grow is False
    self._heap = Heap(self._grow)
    self._heap.insert(self._a)
```

```
step = 0
while True:
```

```
    n = len(self._heap)
```

```
    assert n >= 1
```

```
    if n == 1:
```

```
        self._increment_work()
```

```
        v = self._heap.get_top_and_remove()
```

```
        self._update_answer(v)
```

```
        return
```

```
    v1 = self._heap.get_top_and_remove()
```

```
    v2 = self._heap.get_top_and_remove()
```

```
    if self._grow:
```

```
        v = v1
```

```
        if v2 > v:
```

```
            v = v2
```

```
        v = v + 1
```

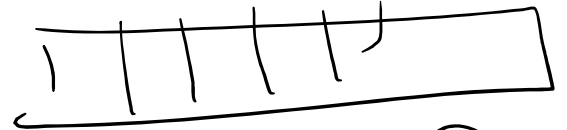
```
    else:
```

```
        v = v1 - v2
```

```
        if v < 0:
```

```
            v = -v
```

$n \log \sim \text{Heap}$ (24) $n > 1$ -2



v_1, v_2

$\log \sim$
 $\log \sim$

$n = 1$

$v_1 - v_2$

5 $4 = 1$ $\max(v_1, v_2) + 1$

```
file = outputFileBase + name + ".dot"  
g.write_dot(file)
```

test

g

network



```
def write_dot(self, f):  
    b = GraphDot(self, f)
```

graph by

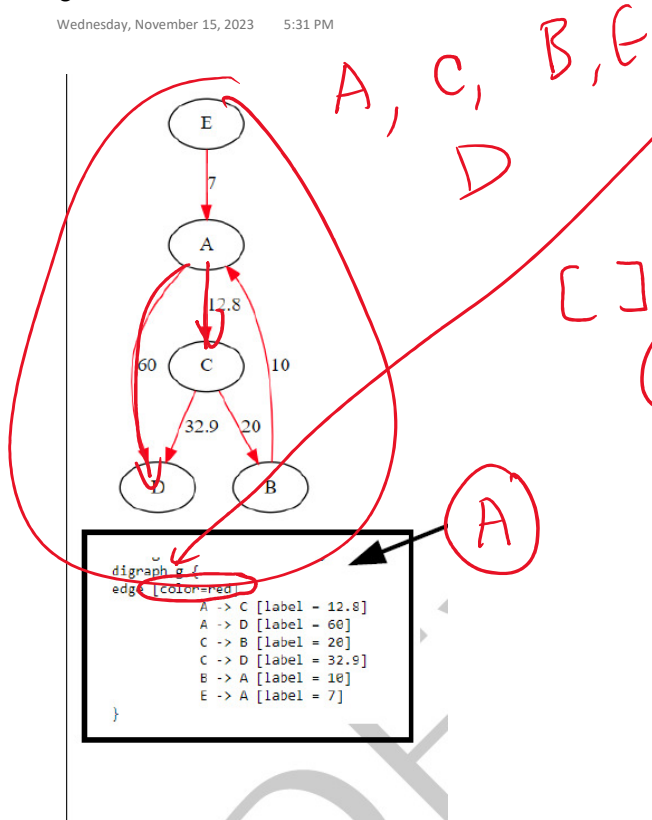
graph by

g



```
class GraphDot:  
    def __init__(self, g, f):  
        self._g = g # Handle to graph  
        self._f = f # File where you write graph in dot format  
        self._of = open(self._f, "w")  
        self._write_dot()  
        self._of.close()
```

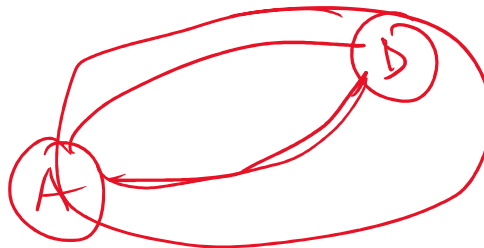
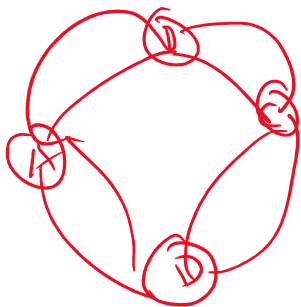
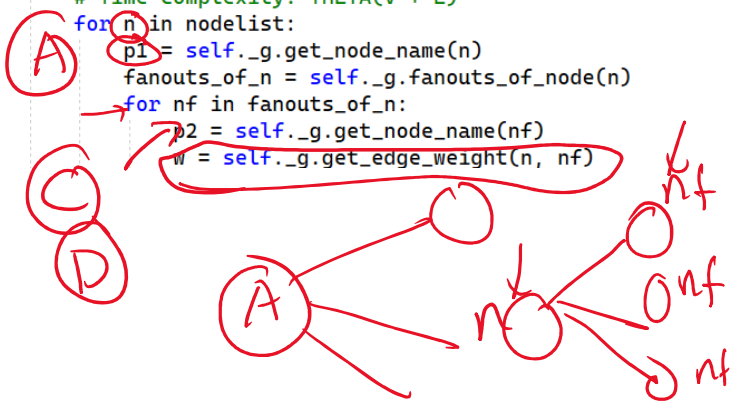
file name



```

def write_dot(self):
    self._of.write("## Jagadeesh Vasudevamurthy ###\n")
    self._of.write("digraph g {\n")
    t = self._g.get_graph_type()
    if t == GraphType.UNDIRECTED or t == GraphType.WEIGHTED_UNDIRECTED:
        self._of.write("\t edge [dir=None, color=red]\n")
    else:
        self._of.write("edge [color=red]\n")
    self._of.write("\n")
    nodelist = self._g.list_of_nodes()
    # Time complexity: THETA(V + E)
    for n in nodelist:
        p1 = self._g.get_node_name(n)
        fanouts_of_n = self._g.fanouts_of_node(n)
        for nf in fanouts_of_n:
            p2 = self._g.get_node_name(nf)
            w = self._g.get_edge_weight(n, nf)

```

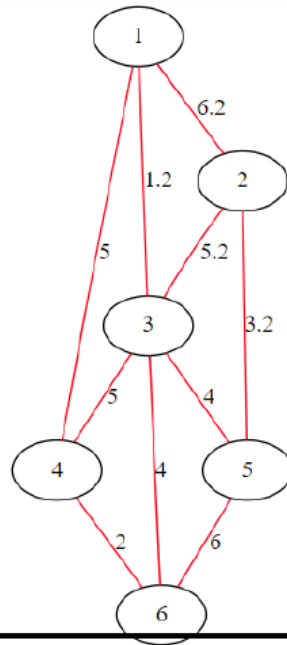


```

digraph g {
  edge [dir=none, color=red]
  1 -> 2 [label = 6.2]
  1 -> 3 [label = 1.2]
  1 -> 4 [label = 5]
  2 -> 3 [label = 5.2]
  2 -> 5 [label = 3.2]
  3 -> 4 [label = 5]
  3 -> 5 [label = 4]
  3 -> 6 [label = 4]
  4 -> 6 [label = 2]
  5 -> 6 [label = 6]
}

```

$2 \rightarrow 1$
 $2 \rightarrow$

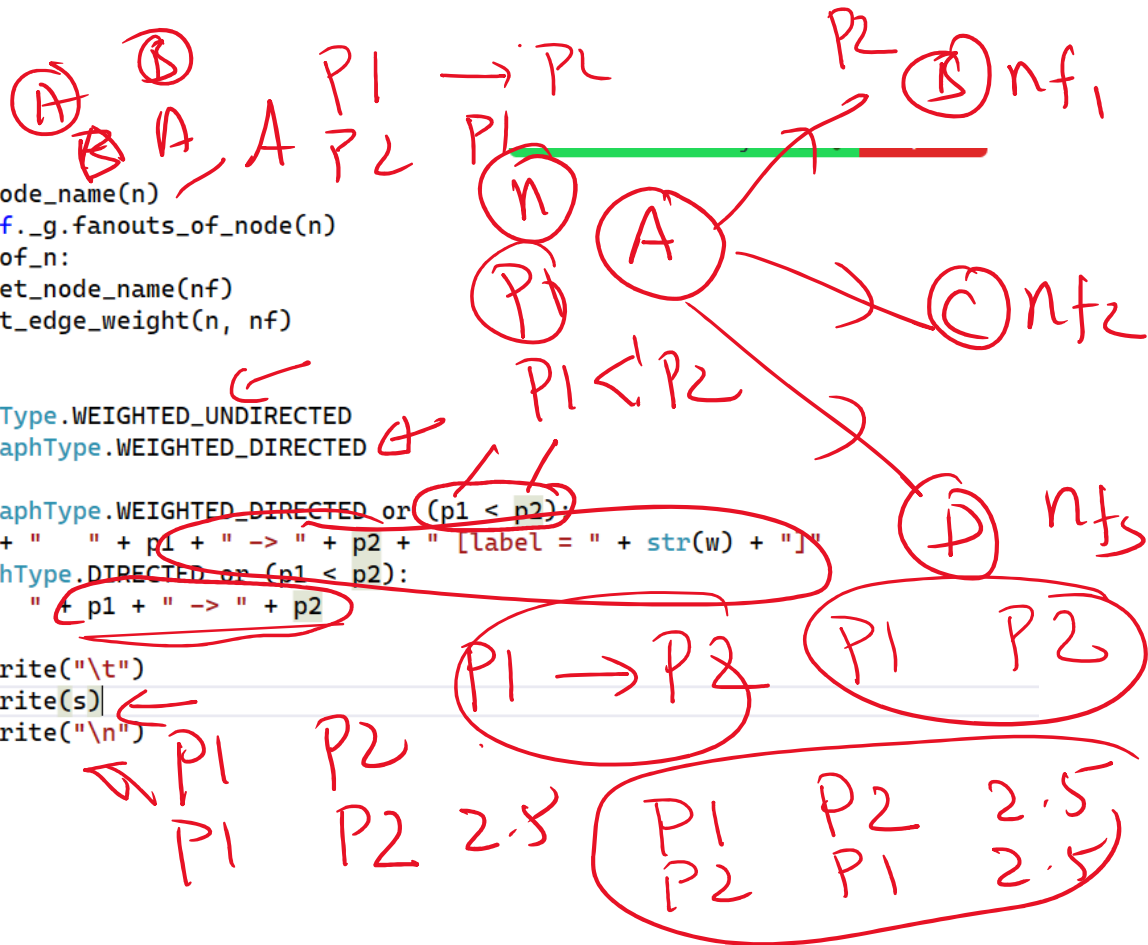


$P1$ $P2$
 $1 \rightarrow 2$ 6.2
 2 1


```

for n in nodelist:
    p1 = self._g.get_node_name(n)
    fanouts_of_n = self._g.fanouts_of_node(n)
    for nf in fanouts_of_n:
        p2 = self._g.get_node_name(nf)
        w = self._g.get_edge_weight(n, nf)
        s = ""
        if (
            t == GraphType.WEIGHTED_UNDIRECTED
            or t == GraphType.WEIGHTED_DIRECTED
        ):
            if t == GraphType.WEIGHTED_DIRECTED or (p1 < p2):
                s = s + " " + p1 + " -> " + p2 + " [label = " + str(w) + "]"
            elif t == GraphType.DIRECTED or (p1 < p2):
                s = s + " " + p1 + " -> " + p2
            if s != "":
                self._of.write("\t")
                self._of.write(s)
                self._of.write("\n")
    self._of.write("}")

```



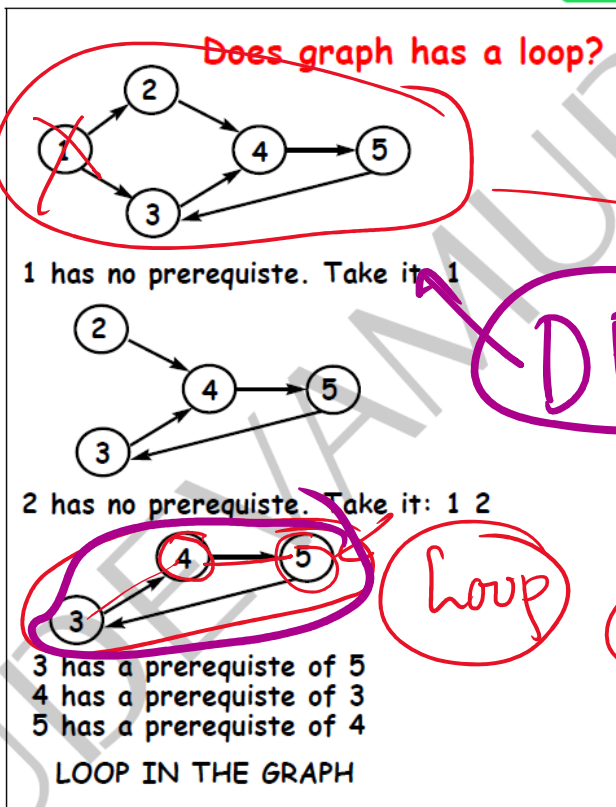


Figure 18.23: Completing courses in an university

g) Loop

Course work

DFS

1 : 1

2 : 2 ←

①

Graph has loop

Loop or NOT

Directed Graph

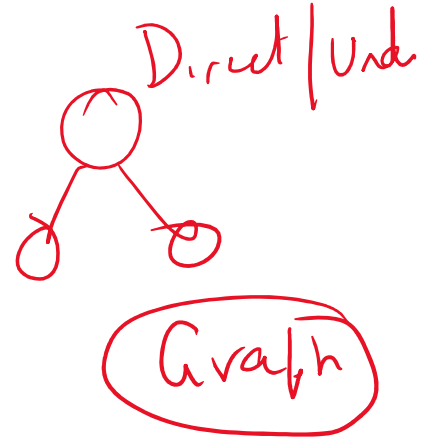
DAG

Traversal

PYTHONLIST []

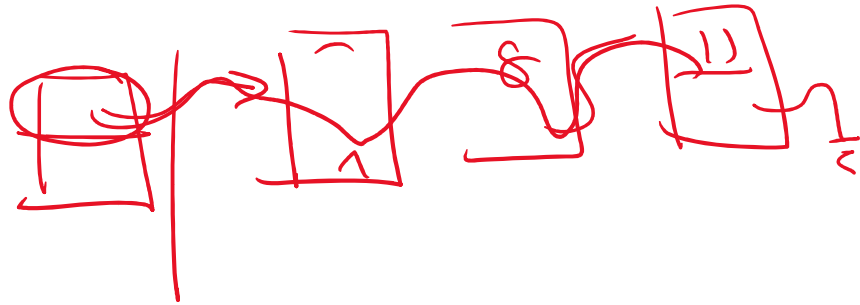
[5 8 11 2 15]

$n-1$



Graph

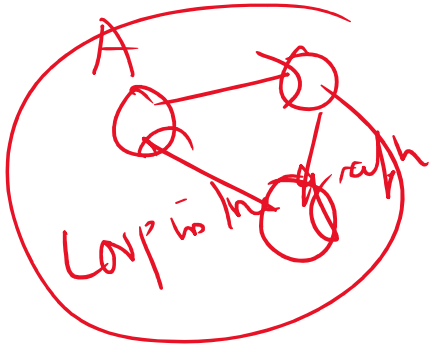
Singly linked



[2, 5, 6, 8]

Traversal Non Linear

(1) We need to visit each node
Exactly once



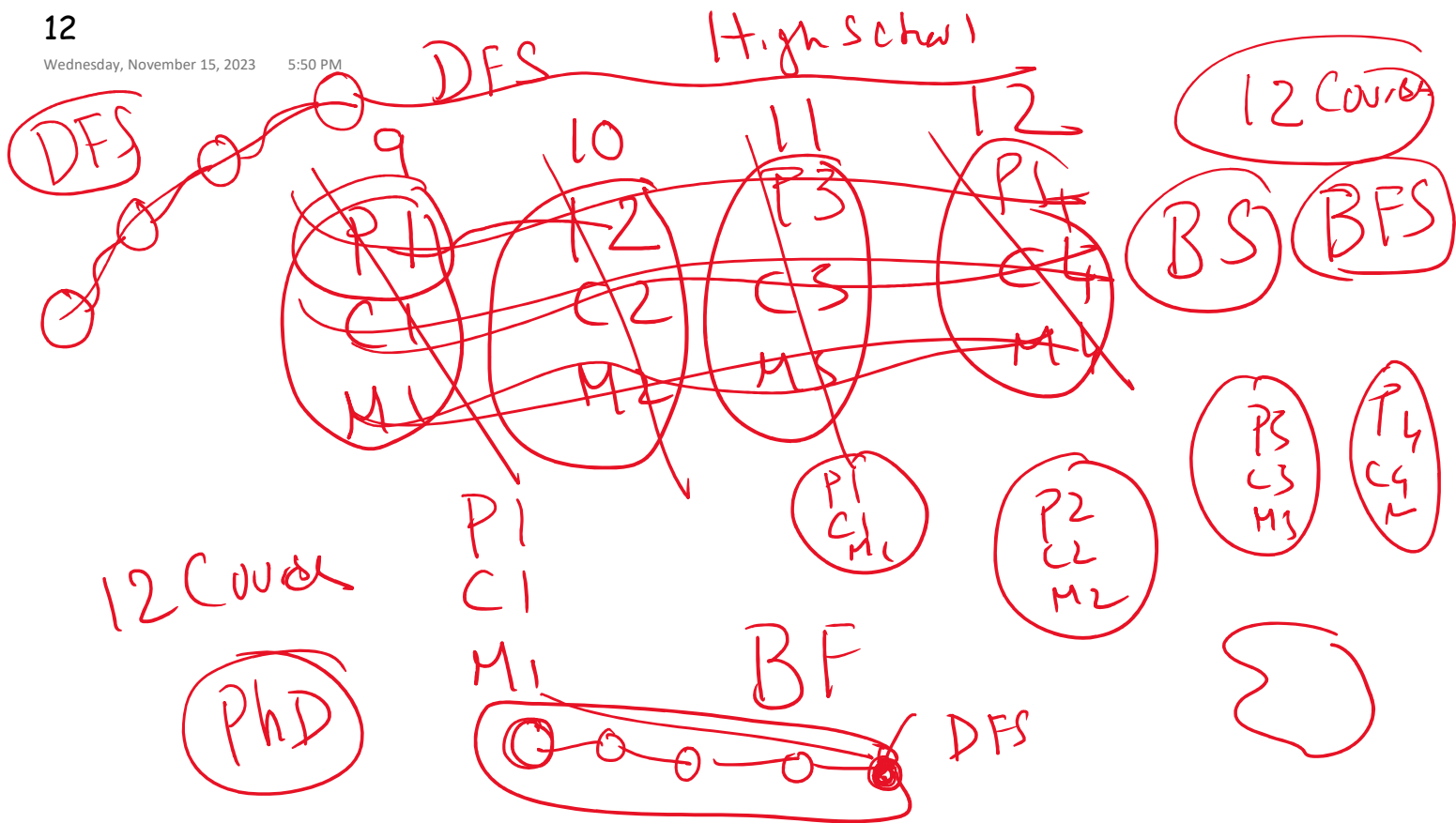
DFS
BFS

DFS

BFS

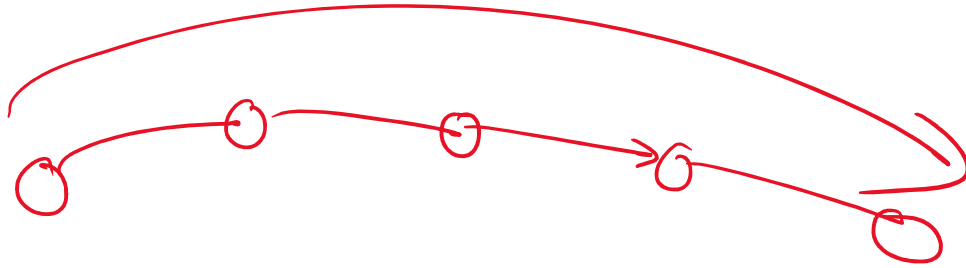
TREE

Problem





Directed und MIT TimeStamp

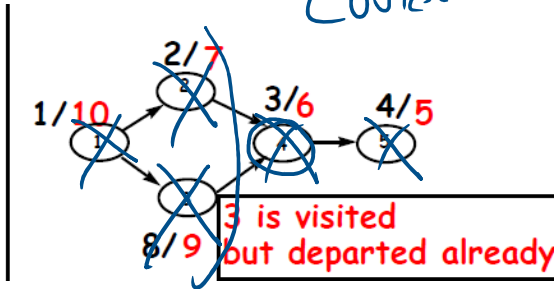


DFS

DFS

root

Course wr

IN OUT
1/10

2/7

3/6 4/5

1
3

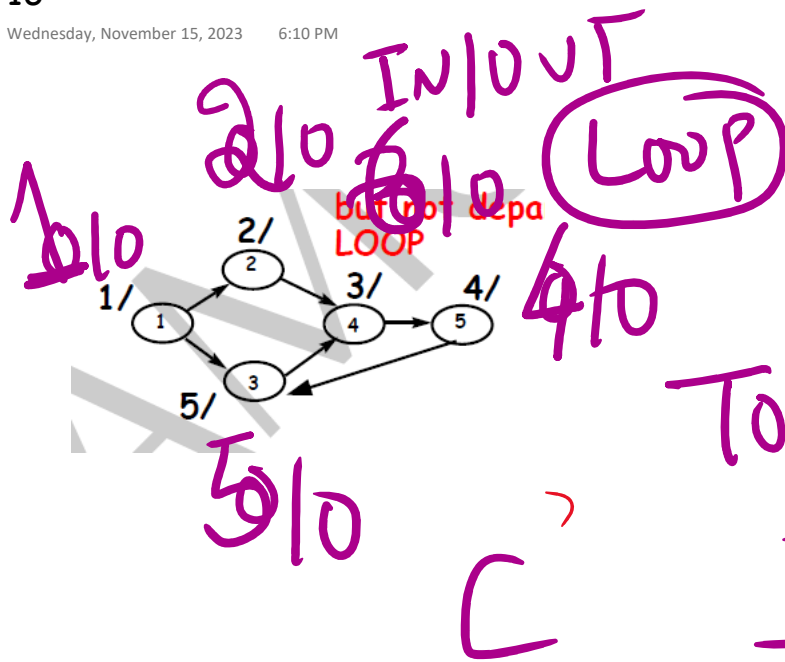
Loop

8/9

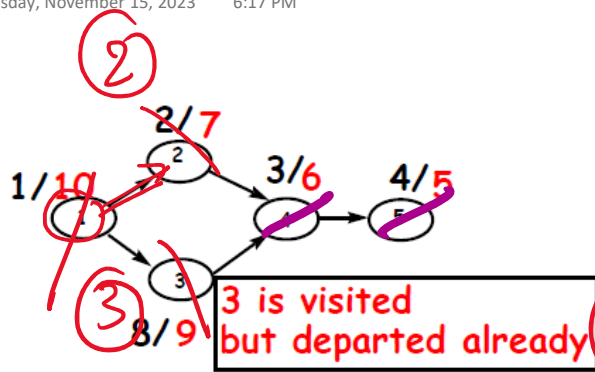
DFS order [5, 4, 2, 3, 1]

Reverse [1, 3, 2, 4, 5]

Topological Sort



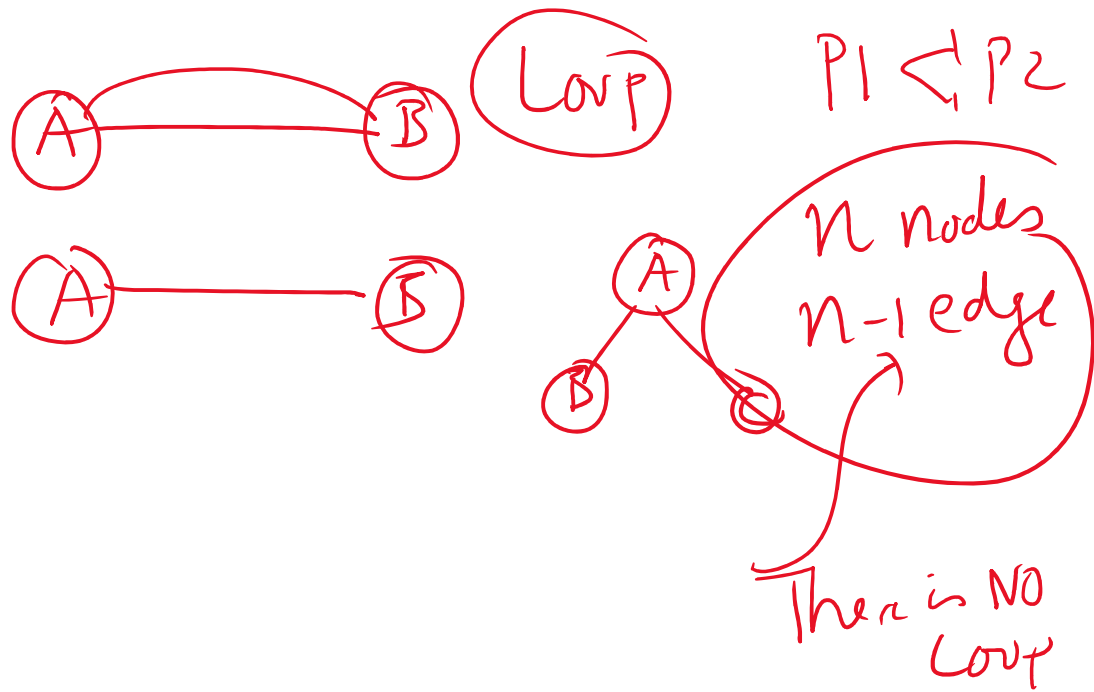
Topological Sort-Will NOT exist

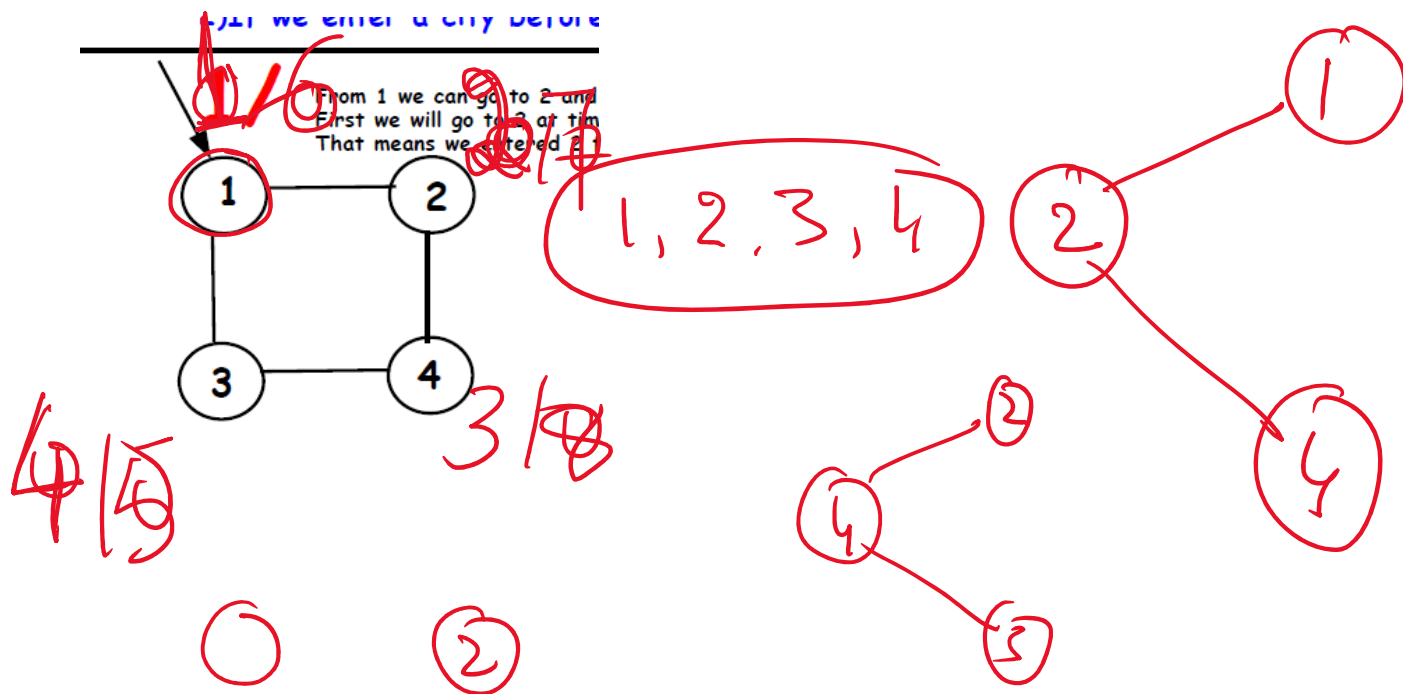


5,4,2,3,1
NO LOOP

1, 3, 2, 4, 5

1, 2, 3, 4, 5



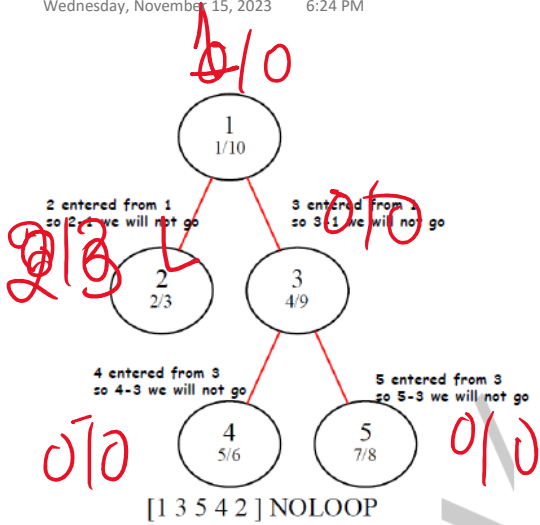


DFS ARRAY

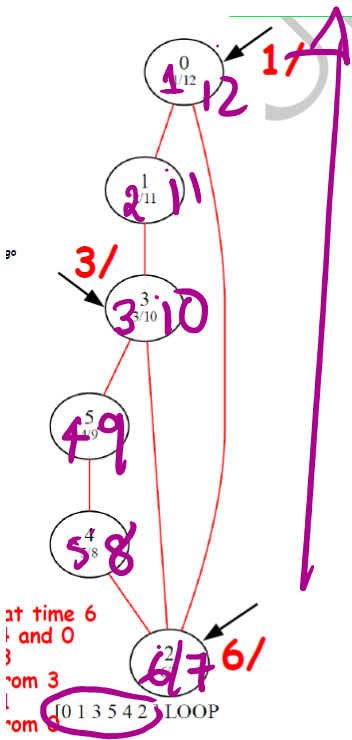
$$[2, \cancel{4}, 5, 3, 1]$$

$$[1, \cancel{2}, \cancel{3}, \cancel{4}, \cancel{5}]$$

10 (plg)

$$10 \log \text{ cat}$$


n nodes n-1 edges
Every node entered exactly once



HAS A Loop

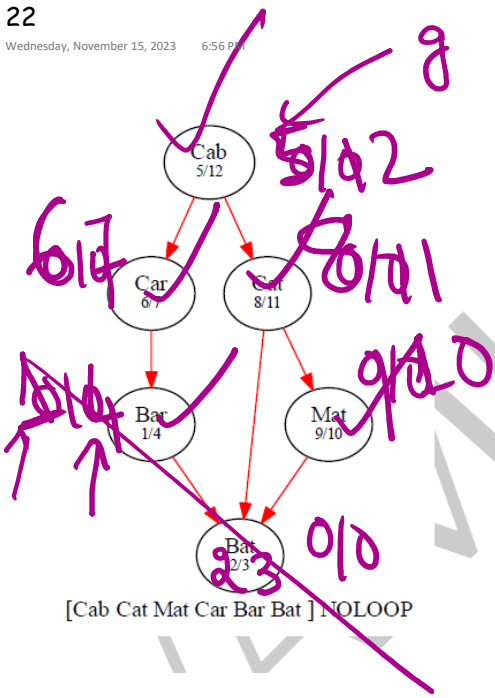
1/0

3/0

Loop

Record
Loop
6: 5 5

[2, 4, 3, 5, 1, 0]
[0, 1, 5, 3, 2, 4]



Exhibit

Bar

CAS

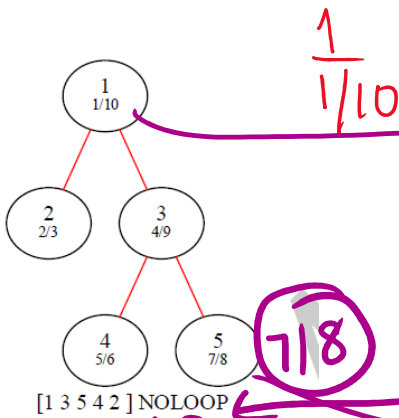
[BAT, Bar, Car, MAT
CAT, CAB]

[C

→

Σ

IN/OUT

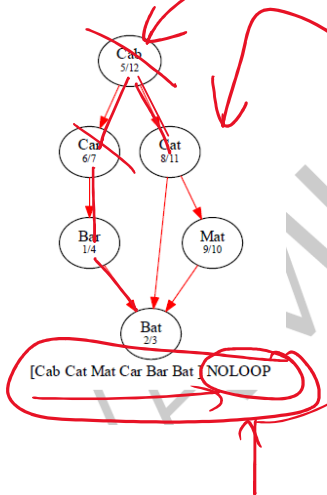


↑
AFTER RUN

```
## dot -Tpdf C:\scratch\outputs\dot\2dfs.dot -o C:\scratch\output:
digraph g {
  label = "[1 3 2 4 5 ] NOLOOP"
  1[label = <1<BR /><FONT POINT-SIZE="10">1/10</FONT>>]
  2[label = <2<BR /><FONT POINT-SIZE="10">2/7</FONT>>]
  3[label = <3<BR /><FONT POINT-SIZE="10">8/9</FONT>>]
  4[label = <4<BR /><FONT POINT-SIZE="10">3/6</FONT>>]
  5[label = <5<BR /><FONT POINT-SIZE="10">4/5</FONT>>]
  edge [color=red]
  1 -> 2
  1 -> 3
  2 -> 4
  3 -> 4
  4 -> 5
}
```

5 7/8

Topology can be



```
## Jagadeesh Vasudevamurthy ###
```

```
## dot -Tpdf C:\scratch\outputs\dot\catdfs.dot -o C:\scratch\outputs\dot\catdfs
```

```
digraph g {
```

```
    label = "[Cab Cat Mat Car Bar Bat ] NOLOOP"
```

```
    Bar[label = <Bar<BR /><FONT POINT-SIZE="10">1/4</FONT>>]
```

```
    Bat[label = <Bat<BR /><FONT POINT-SIZE="10">2/3</FONT>>]
```

```
    Cab[label = <Cab<BR /><FONT POINT-SIZE="10">5/12</FONT>>]
```

```
    Car[label = <Car<BR /><FONT POINT-SIZE="10">6/7</FONT>>]
```

```
    Mat[label = <Mat<BR /><FONT POINT-SIZE="10">9/10</FONT>>]
```

```
    Cat[label = <Cat<BR /><FONT POINT-SIZE="10">8/11</FONT>>]
```

```
    edge [color=red]
```

```
    Bar -> Bat
```

```
    Cab -> Car
```

```
    Cab -> Cat
```

```
    Car -> Bar
```

```
    Mat -> Bat
```

```
    Cat -> Bat
```

```
    Cat -> Mat
```

```
}
```

```

dfs_order = [] # Caller will Fill. List of Nodes
has_loop = [False] # List of size 1
work = [0] # List of size 1
dfs_dot_output_file = outputFileBase + gname + "dfs_dot"
g.dfs_using_time_stamp(gname, dfs_order, has_loop, work, dfs_dot_output_file)

print("DFS traversal is in")
print(dfs_dot_output_file)

print("DFS ORDER: ", end=" ")
for node in dfs_order:
    p1 = g.get_node_name(node)
    print(p1, end=" ")
print()

```

dfs_order = [] []

[1, 2, 3]

1
2/4

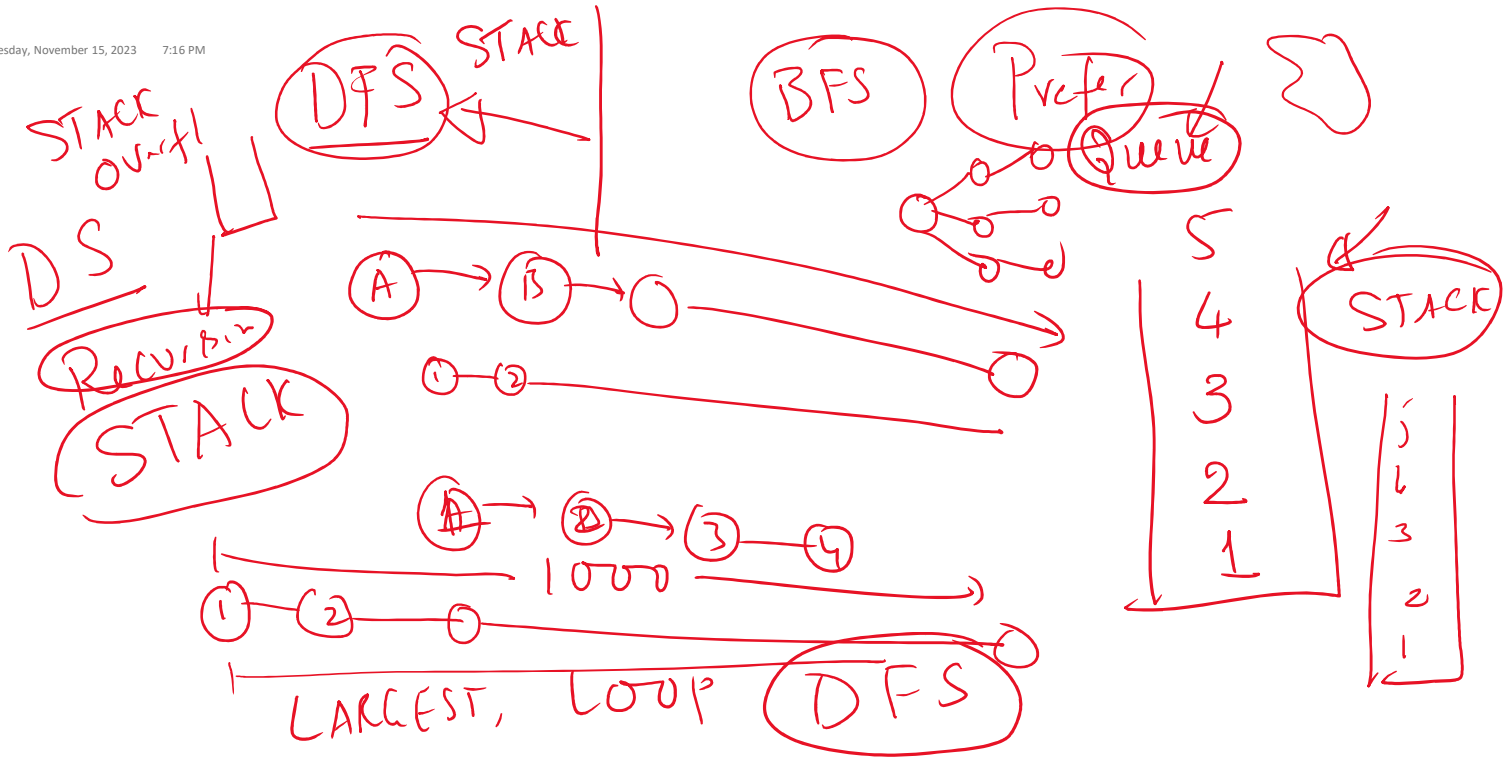
Topological
(Not a)

12.dot

12dfs.dot

1

True
1 FALSE

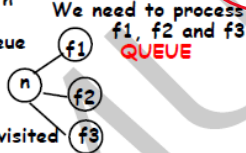


Shortest PATH When All the Weights are SAME

Breadth First Search

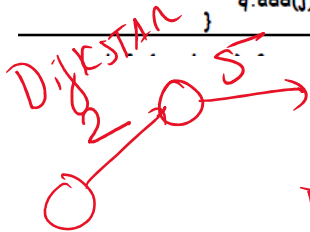
1. Explore the graph level by level
2. All nodes are visited exactly once
3. For every node n , we need to remember two things:
 - a) is n has visited already?
We can track this: `visited[n]`
 - b) Have we visited all the adjacent nodes of n ?
We can track this with a queue
Insert all the fanout of a node to a queue

For each fanout f of n
 if (`visited[f] == FALSE`) {
 `visited[f] = true` ;
 `q.add(j)` ; // j fanout has to be visited

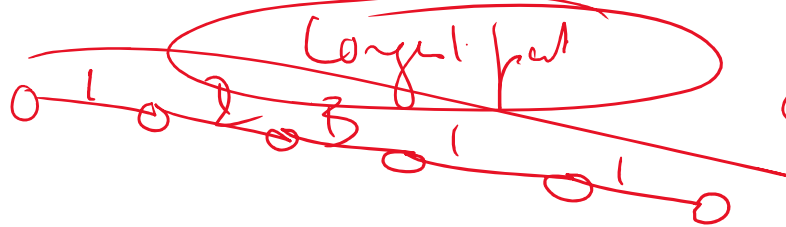


Loop

Each node exactly 1



DFS



Breadth First Search

1. Explore the graph level by level
2. All nodes are visited exactly once
3. For every node n , we need to remember two things:
 - a) is n has visited already?

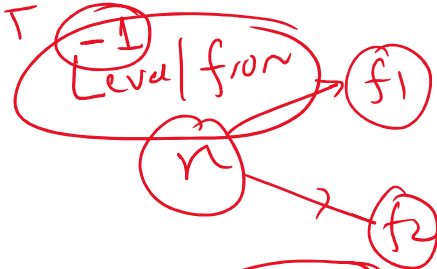
We can track this: `visited[n]`
 b) Have we visited all the adjacent nodes of n
 We can track this with a queue
 Insert all the fanout of a node to a queue

For each fanout f of n
 if (`visited[f] == FALSE`) {
 `visited[f] = true` ;
 `q.add(f)` ; // f fanout has to be visited
}

We need to process
 f_1, f_2 and f_3
QUEUE



2 IN/OUT 5 IN/OUT 0 IN/OUT



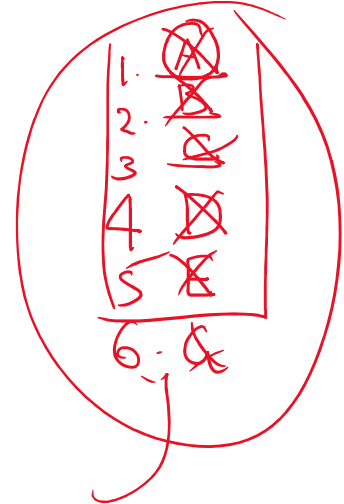
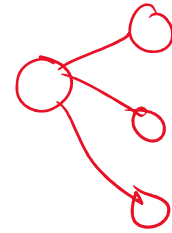
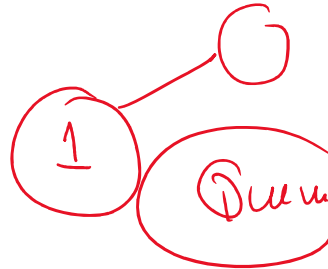
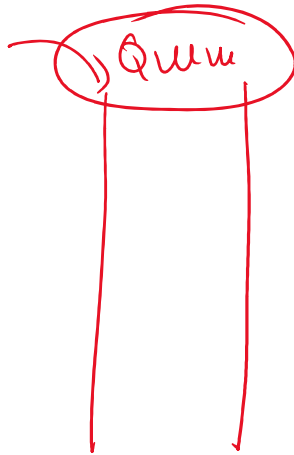
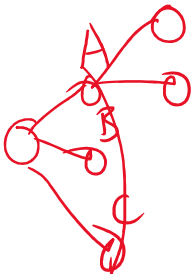
bowl Visited

True
false

Level From

-1

Queue



D
E
G

~~ST~~

$-1; O(V)$

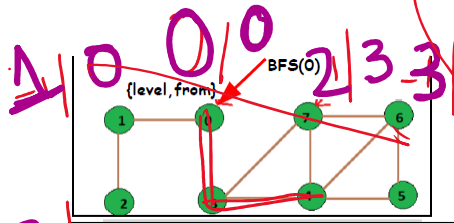
 $O(V+2E)$

- All

Handwritten notes and diagram:

- Level -1
- (From)
- n
- Var. bl
- (19)
- Starting
- Zawmle
- SJ
- Ny
- f
- 1
- SJO
- 1
- S-
- SJ
- Oabk -1
- M-1 -1

```
graph TD; Zawmle((Zawmle)) --> SJ1((SJ)); Zawmle --> Ny((Ny)); Zawmle --> f((f)); Zawmle --> SJO((SJO)); Zawmle --> Oabk((Oabk)); Zawmle --> M1((M-1)); SJ1 --> SJ2((SJ)); SJ1 --> SJO; SJ1 --> Oabk; SJ1 --> M1;
```



BFS order: 0 1 3 2 7 4 6 5
 BFS PATH: 0 0 0 1 3 3 7 4

Fru

1 → 0

2 1 4 0 2 3 3 4

9

4 → 3 → 0
 0 → 3 → 4

5 → 4 → 3 → 0

0 → 3 → 4 → 5

6 → 7 → 3 → 0

0 → 3 → 7 → 6

PATH

```

bfs(node s) {
  for each node of G assign {level, from} = -1;
  s.level = 0;
  s.from = 0;
  q.enqueue(s);
  while (q is not empty) {
    node n = q.dequeue();
    for each fanout_node(f of n) {
      if (f.level == -1) {
        f.level = n.level + 1;
        f.from = n;
        q.enqueue(f);
      }
    }
  }
}

```

$O(V)$ $O(E)$ $O(V+2E)$

$\frac{-1}{From}$

n

$\frac{0}{SJ}$

$\frac{-1}{A}$

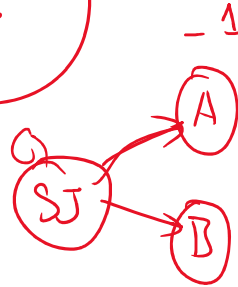
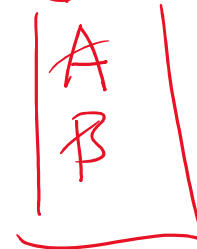
$\frac{-1}{B}$

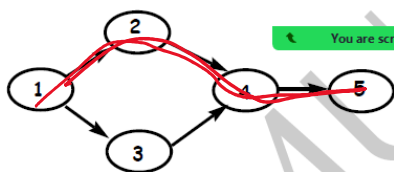
1. Finds shortest path if graph has same weight
2. We can find the path from s to t from 'from'
3. We know the distance from s to any node from 'level'
4. We can find all connected components
5. Each node is visited exactly once

Level

Visit

SJ





You are screen sharing

BFS order
BFS PA

1 2 3 4 5
1 1 1 2 4

Num Vertices = 5

Num Edges = 5

Work done = 10

BFS order = 1 2 3 4 5

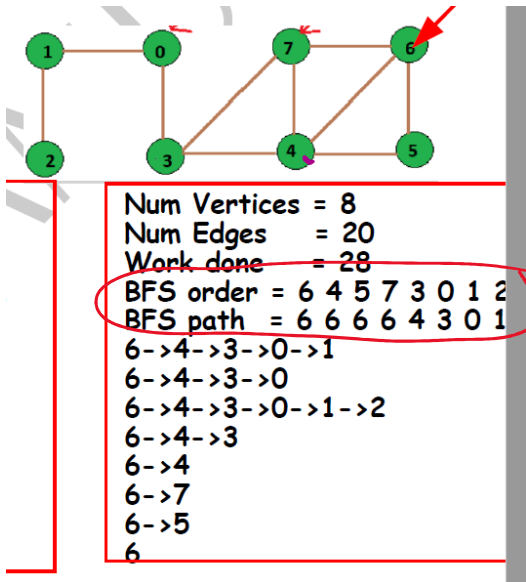
BFS path = 1 1 1 2 4

1
1 → 2
1 → 3
1 → 2 → 4
1 → 2 → 4 → 5

**SHORTEST
PATH**

5 → 4 → 2 → 1

4 → 2 → 1



$1 \rightarrow 0 \rightarrow 3 \rightarrow 4 \rightarrow 6$

$6 \rightarrow 4 \rightarrow 3 \rightarrow 0 \rightarrow 1$

$2 \rightarrow 1 \rightarrow 0 \rightarrow 3 \rightarrow 4 \rightarrow 6$

UNIFORM WEIGHT
NO WEIGHT

Longest path
DFS

NON UNIFORM
WEIGHT

NO ALG
EXISTS

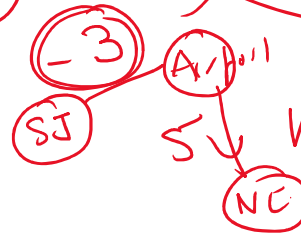
Polynomial

Shortest path

BFS

Dijkstra

≥ 0



Negative

Floyd D ALG