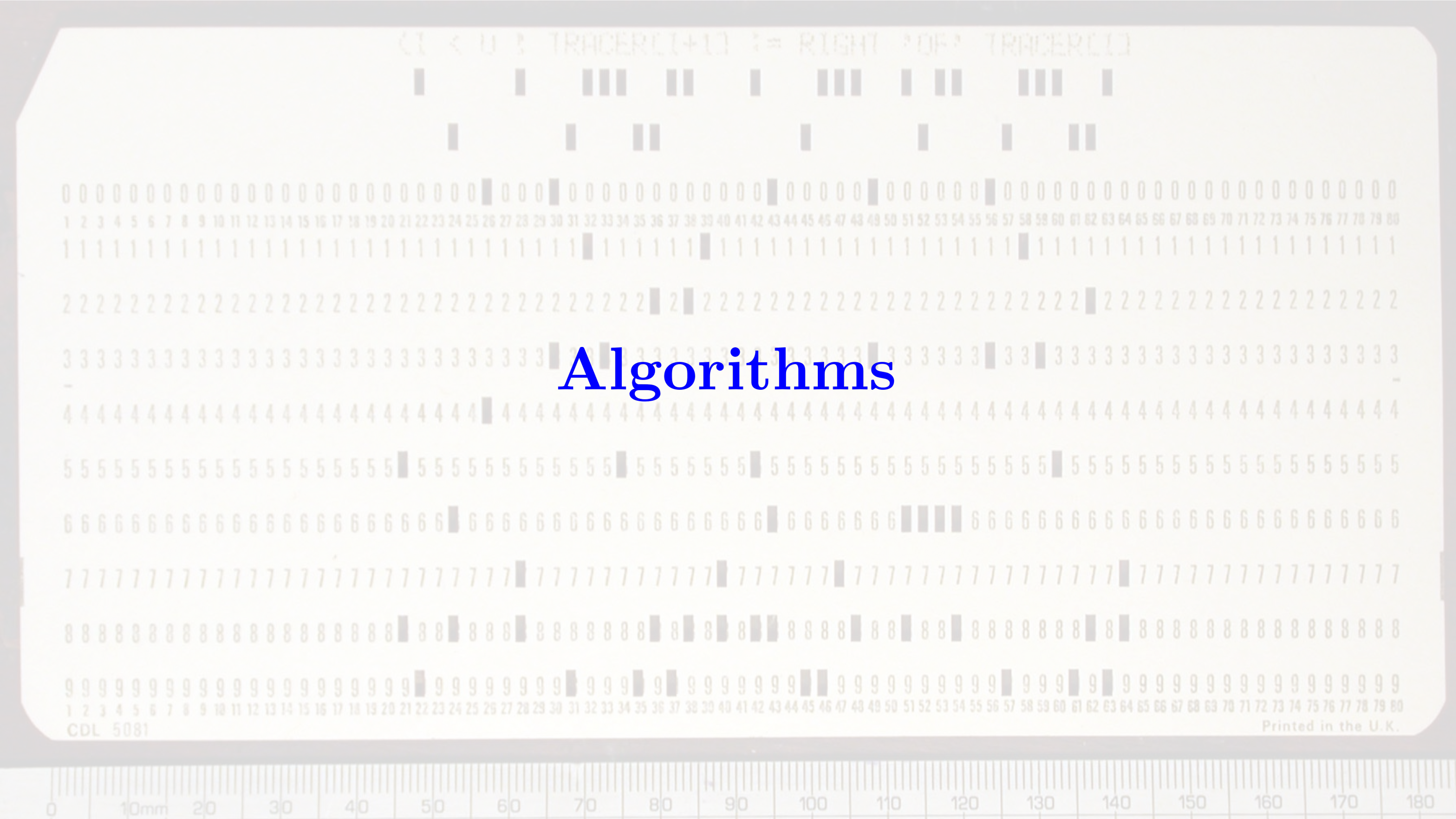
Algorithms



Reading

Required. Graphs and Graph Traversal: Sections 3.1-3.2.

not discussed $\rightarrow$ in class **Please read Sections 3.3 and 3.4 on your own.**

Graph Connectivity: Section 3.5

DAGs and Topological Ordering: Section 3.5

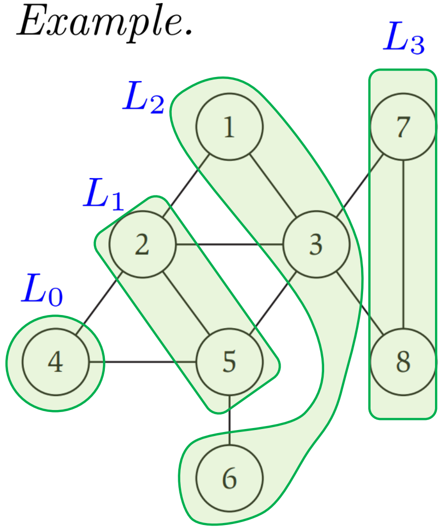
Suggested. Sections 7.1-8.5 of Roughgarden's video lectures.

http://algorithmsilluminated.org/

Breadth First Search (BFS)

```
Q = {}  # empty queue

BFS(s):
    Q.enqueue(s)
    mark s as "discovered"

    while not Q.empty():

        u = Q.dequeue()
        visit u and mark it "visited"

        for each edge (u,v):
            if v is not "discovered":
                Q.enqueue(v)
                mark v as "discovered"
```

Example.



Execution of $BFS(4)$:

Front *Rear*

$\cancel{4}$ $\cancel{2}$ $\cancel{5}$ $\cancel{1}$ $\cancel{3}$ $\cancel{6}$ $\cancel{7}$ $\cancel{8}$

vertices are visited in the order they are queued

Breadth First Search (BFS)

```
Q = {}  # empty queue

BFS(s):
    Q.enqueue(s)
    mark s as "discovered"

    while not Q.empty():

        u = Q.dequeue()
        visit u and mark it "visited"

        for each edge (u,v):
            if v is not "discovered":
                Q.enqueue(v)
                mark v as "discovered"
```

What is the running time if the graph has n vertices and m edges?

$O(m+n)$

In each iteration of the while loop, we process a distinct vertex u and we look at all its neighbors v.

The time for this is proportional to $1+d(u)$ where $d(u)$ is the degree of u in the graph.

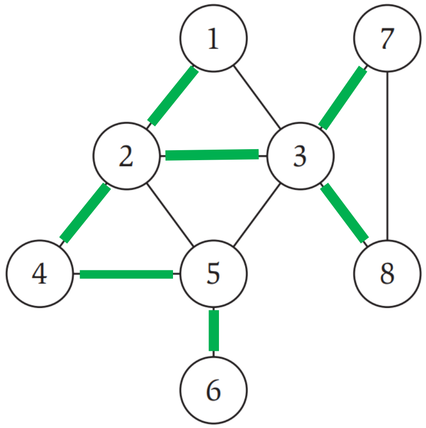
$$\begin{aligned}\text{Total time} &\propto \sum_u (1+d(u)) \\ &= n+2m \\ &= O(n+m)\end{aligned}$$

Breadth First Search (BFS)

```
Q = {}  # empty queue
T = {}  # empty tree

BFS(s):
    Q.enqueue(s)
    mark s as "discovered"

    while not Q.empty():
        u = Q.dequeue()
        visit u and mark it "visited"

        for each edge (u,v):
            if v is not "discovered":
                T = T ∪ {(u,v)}
                Q.enqueue(v)
                mark v as "discovered"
```

BFS Tree.

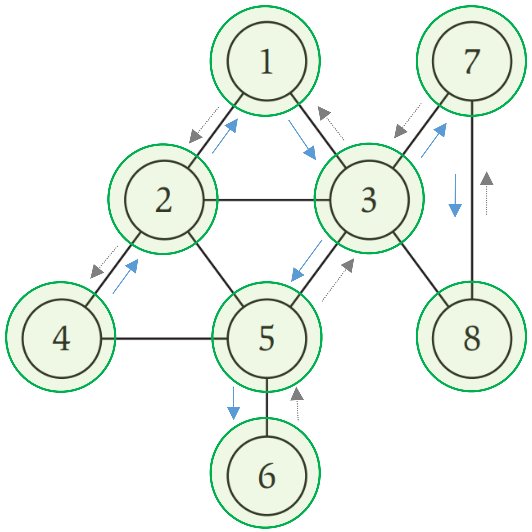
The tree defined by the edges through which we discover new vertices.



Tree for BFS started at 4.

Depth First Search (DFS)

```
DFS(u):
    visit u and mark it "visited"
    for each edge (u,v):
         if v is not "visited":
                DFS(v)
```



In what order are the vertices visited if we execute $DFS(4)$?

The precise order depends on the order in which we look at the neighbors of any vertex

One possible order: 4 2 1 3 5 6 7 8

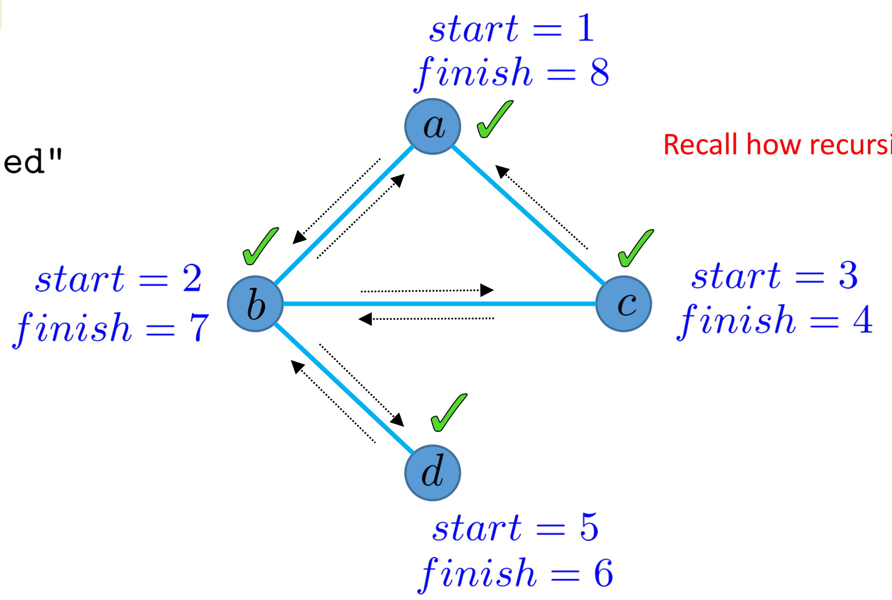
The edges marked with arrows form a tree known as the DFS tree.

Visualization: https://www.cs.usfca.edu/~galles/visualization/DFS.html

Depth First Search (DFS)

```
T = {}; // T is the DFS tree
time = 0

DFS(u):
    visit u and mark it "visited"

    time += 1
    u.start = time

    for each edge (u,v):
        if v is not "visited":
            T = T ∪ {(u,v)}
            DFS(v)

    time += 1
    u.finish = time
```

Execution of DFS(a):



Recall how recursion works

DFS Tree. Edges through which we discover new vertices.

The order of visiting the vertices is *<u>not unique</u>* *since the neighbors of a vertex can be process in any order.*

Depth First Search (DFS)

```
DFS(u):
    visit u and mark it "visited"
    for each edge (u,v):
         if v is not "visited":
                DFS(v)
```

Running time?

assuming that the graph is connected and has n vertices and m edges

DFS is called on each vertex exactly once. *Why?*

When we do DFS at a vertex u, we go over all its neighbors, and recurse on neighbors that are not yet visited.

Time spent for DFS at u, apart from recursing, is $\propto 1 + d(u)$.

The time for recursing is accounted for at the vertex we call DFS on.

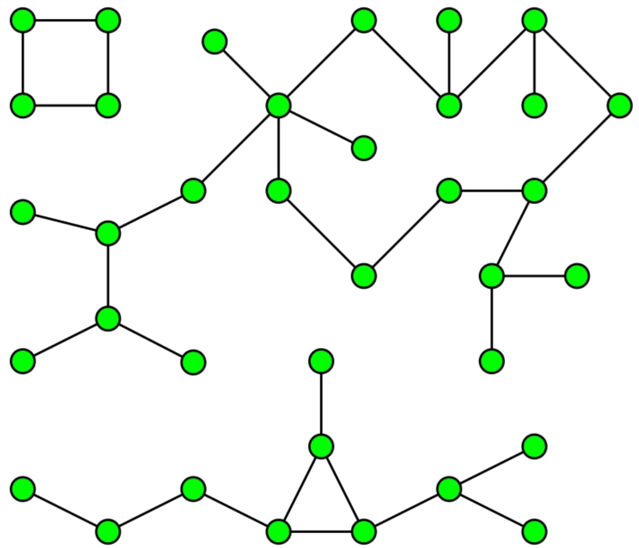
Total time $\propto \sum_u (1 + d(u)) \; = n + 2m = O(m + n)$.

Connected Components

Suppose that we have a graph G given in the adjacency list representation.

How do we find out the number of connected components in G?

Observation. If we start DFS or BFS on any vertex v, it visits all vertices in the connected component containing v.



```
num_components = 0

for each vertex v:
    if v is not "visited":
      DFS(v)
      num_components += 1
```

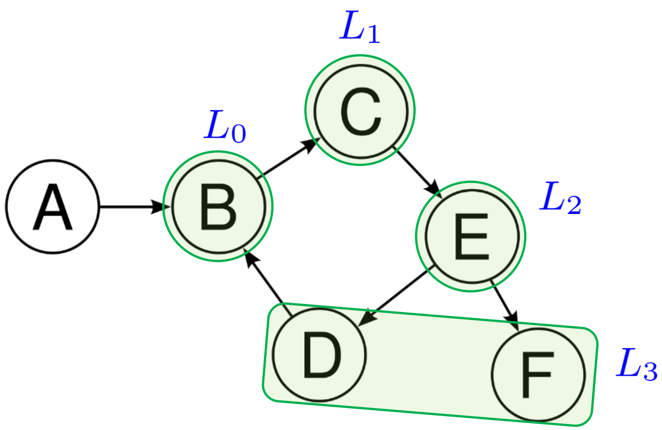
The for loop here executes DFS only if there is a non-visited node remaining.

Running time? $O(m+n)$ *where* $n = \#$ *vertices and* $m = \#$ *edges in* G

BFS in Directed Graphs

```
Q = {}  # empty queue

BFS(s):
    Q.enqueue(s)
    mark s as "discovered"

    while not Q.empty():

        u = Q.dequeue()
        visit u and mark it "visited"

        for each directed edge u->v:
            if v is not "discovered":
                Q.enqueue(v)
                mark v as "discovered"
```

← *we only take outgoing edges from u*



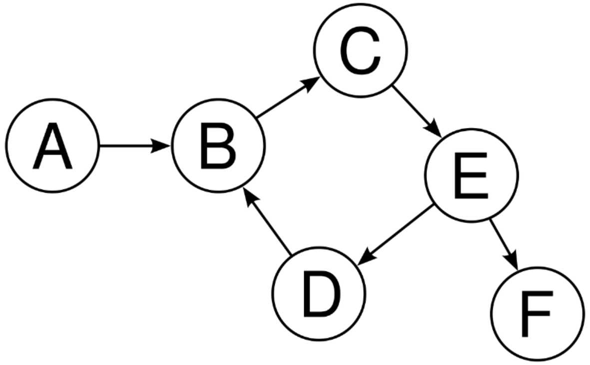
In what order are the vertices visited if we start BFS at B?

$B,\ C,\ E,\ D,\ F$

Note: vertex A is not visited since there is no directed path from B to A

DFS in Directed Graphs

```
DFS(u):
    visit u and mark it "visited"
    for each directed edge u->v:
        if v is not "visited":
            DFS(v)
```



In what order are the vertices visited if we start DFS at C?

C, E, D, B, F

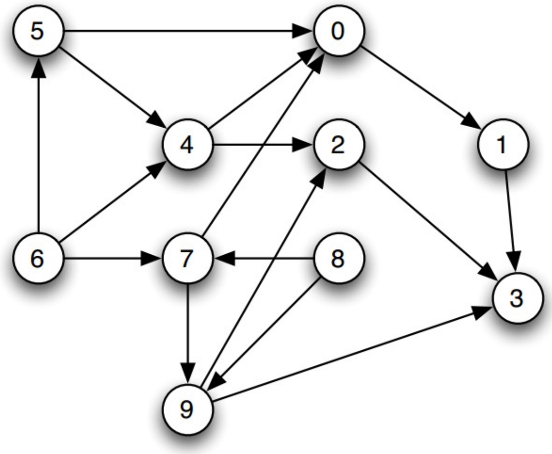
Note: vertex A is not visited since there is no directed path from C to A

Topological Sorting

- Directed acyclic graphs (DAGs) are directed graphs with no cycles.
- DAGs are a very common structure in computer science.
- DAGs can be used to encode precedence relations or dependencies in a natural way.
- Example: we have a set of tasks labeled {1, 2, ... , n} that need to be performed, and there are dependencies among them stipulating, for certain pairs i and j, that i must be performed before j.
    - *For example, the tasks may be courses, with prerequisite requirements stating that certain courses must be taken before others.*
    - *Or the tasks may correspond to a pipeline of computing jobs, with assertions that the output of job i is used in determining the input to job j, and hence job i must be done before job j.*

Topological Sorting

Each node is a task.



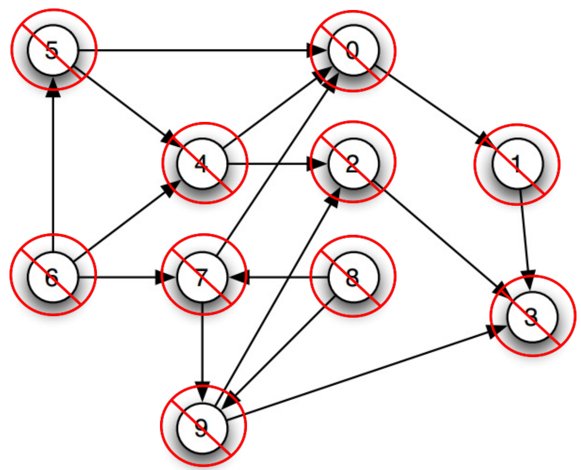
Directed edge from i to j means
"Task i must be done before task j"

Want: Find an order in which the tasks can be executed.

Topological Sorting: Algorithm

At least one

Observation. If there are no directed cycles, there must be a vertex with no incoming edges.



We can safely make such a vertex the first vertex in our ordering.

We can then remove this vertex and recurse!

One possible ordering obtained this way for the above graph:

6 5 8 4 7 0 9 2 1 3

Exercise. *How do we implement this algorithm so that it runs in $O(m+n)$ time?*