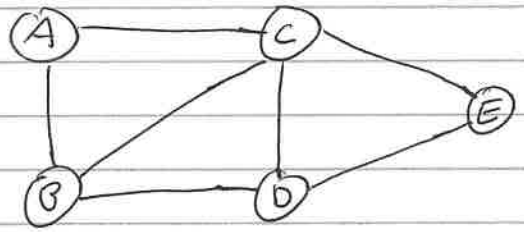
DFS, Topsort, and SCC. L5

encodings

Welcome to the first ~~unit~~ lecture of unit 2 on graph algorithms. Graphs are useful topological and combinatorial devices to study many problems. A graph is a collection of n nodes denoted by a set V, and a collection of edges $E \subseteq V \times V$. Edges may be weighted and/or directed. We need a way to first encode graphs to be used as inputs in algorithms.

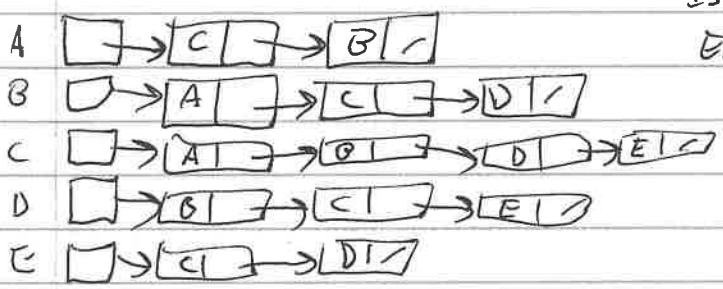
matrix



	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
D	0	1	1	0	1
E	0	0	1	1	0

One obvious way is as an adjacency matrix. If the graph was directed then you could even use rows as incoming and columns as outgoing. Checking specific edges takes constant time, but one bit must be stored for each edge and nonedge. If the graph is sparse, say $E = O(V)$, then this still takes $O(V^2)$ $(= O(n^2))$ space. Another representation is by using adjacency lists.

list



Each vertex has its own linked list of its neighbors. Each edge appears atleast once if the graph is directed. Checking if a specific edge exists is now no longer constant time, but we can still loop over a vertexes neighbors. It is now optimal in size, with $|V|$ lists totalling $|E|$ elements. The size of this is then $2|E| + |V|$ for undirected graphs, or $O(|V| + |E|)$. Both the list and the matrix encoding of graphs can be easily modified if the graph is weighted. Most algorithms, we will use the adjacency list representation the matrix representation is used rarely, but it does have some linear algebra tricks.

DFS.

Depth-First-Search is a graph traversal algorithm. It is a primitive to be used in the construction of other, actual, algorithms. It visits each node in a way which prioritizes "depth". It searches deeper before searching wider.

```
def explore(G, v):
    visited[v] = true
    previsit(v)
    for each e ∈ E  e=(v,u)
        if not visited[u]
            explore(G,u)
    postvisit(v)
```

```
def dfs(G):
    for each v∈V:
        visited[v] = false
    for each v∈V:
        if not visited(v)
            explore(v).
```

proof of correctness

Given a graph and a vertex v, explore will touch everything reachable from v. If there are vertices not reachable from v, they will be hit in the main for loop of dfs(G). Lets quickly prove correctness of explore. Suppose there exists a vertex u in G such that explore(G,v) does not touch u but there is a path to u. Let z be the last vertex in this path reached by explore(G,v). v ⋯ z → z' ⋯ u. Let z' be the node immediately after it. So z is visited, z' is not. But the explore call on z would have visited z' because of the edge z→z', contradiction.

runtime

The runtime is $O(|V|+|E|)$, as each vertex is visited once, and edge twice. The edges are visited at different times in different loops, but each is hit twice.

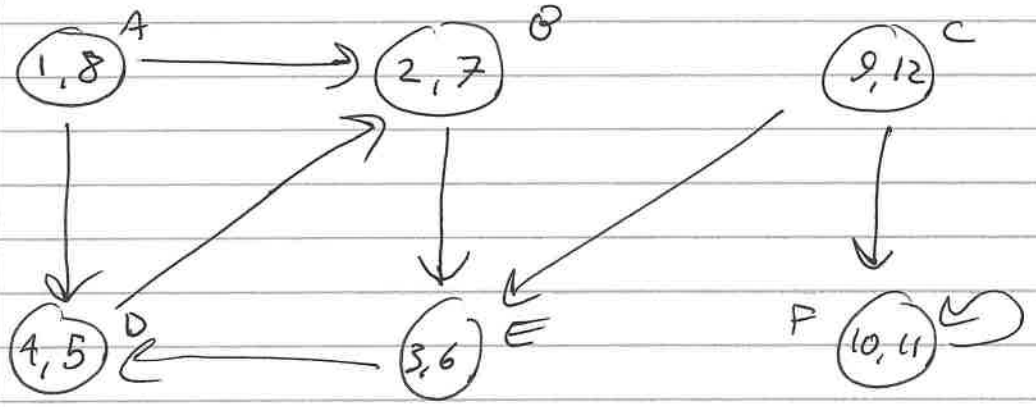
pre and post visit can be modified so that you can use dfs as a primitive lets view the execution of DFS on a graph with pre and post visit implemented like clocks. Also notice that DFS's implicit data structure is a stack.

counters

```
counter = 1
def previsit(v)
    pre[v] = counter
    counter++
```

```
def postvisit
    post[v] = counter
    counter++
```

example

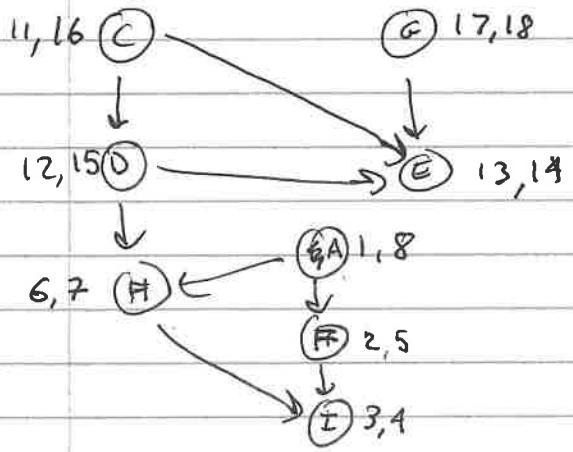


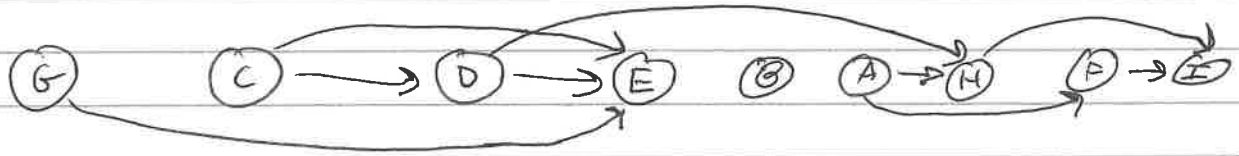
Topsort.

DAG

A Directed Acyclic Graph (DAG) is a graph which is directed and has no directed loops. Many concepts can be represented like a DAG as a set of objects with some strict precedence or priorities.

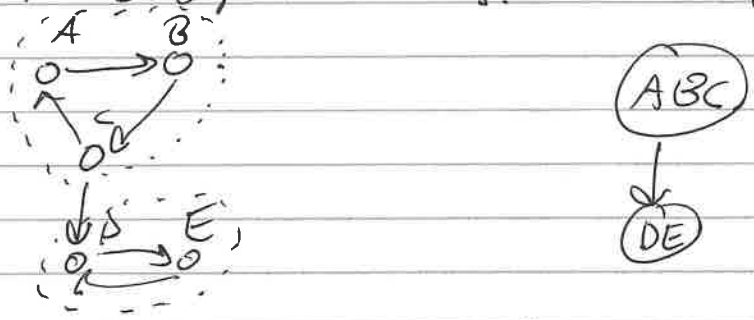
A topological sort of a DAG is an ordering of the vertices ~~such that~~ as $v_1, v_2, \ldots$ such that there is no edge $v_i \to v_j$ if $j \leq i$. Think of it like a drawing of a DAG with all edges facing $\rightarrow$. A graph is a DAG if and only if it can be topologically sorted. Let's give an easy algorithm to finding ~~pos~~ a topsort of a DAG. Note in a DAG if $post(u) < post(v)$. Then v must be at a higher precedence than u. Actually, running DFS on a DAG, the post numbers are exactly the topsort! No need to call sort after, simply modify post to print out things as it called, then reverse the list. This takes one DFS call and one reversal, so $((|V|+|E|)+|V| = O(|V|+|E|)$ linear time.





SCCs

A strongly connected component is a subset of vertices in a graph which are all reachable from one another. In an undirected graph, partitioning a graph into its components is easy. Each top level DFS call of explore must be on a new component. In a directed graph this is more complicated. In directed graphs, we defined two nodes to be connected if there is both a $u \to v$ path and a $v \to u$ path. The metagraph is a graph whose nodes represent one component in the original graph. It represents the connections between components but not with the components themselves. Note the Metagraph is always a DAG.



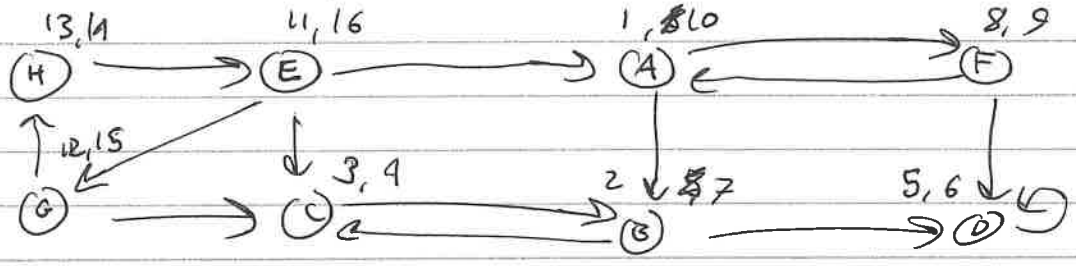
We argue some facts about SCCs to help justify an efficient algorithm. If C, C' are distinct SCCs and there is an edge in the metagraph $C \to C'$ then the highest post number of C is bigger than the highest post number of C'.

Proof: Suppose DFS sees C before C', then all of C' is explored, so the first node touched of C has a higher post than any of C'. If C' is touched first by DFS, all of C' gets explored, but none of C, so then it gets explored later and has a higher post number. We get the immediate as a lemma The node with the highest post lies in a source SCC.

Second, note if we call explore on any node in a sink SCC, we can find the entire component, but the node may not necessarily be the sink. Consider G^R, the reversal of G. The reversal of any SCC is the same SCC. a path $u \to v$ and $v \to u$ is also a path $v \to u$ and $u \to v$. So G, G^R have the same components. This gives us our algorithm. Find an element of the source of G, pretend it is in the sink of G^R, remove this sink component from G^R, repeat.

```
def SCC(G)
    dfs(G)  get posts in decreasing order
    compute G^R
    while posts not empty
        v = max(posts)
        component = explore(G^R, v)
        print/mark component
        remove component from G^R and posts
```

G



G^R