Graph2

June 7, 2020

1 Directed Graph

A directed graph is a graph where each edge has a start vertex and an end vertex

1.1 Examples

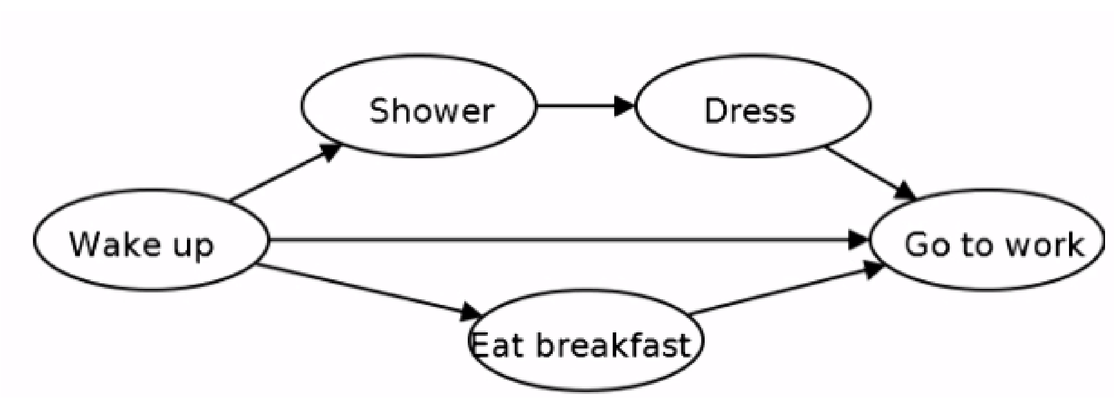
Directed graph might be used to represent - Streets with one-way roads - Links between webpages - Followers on social network - dependencies between tasks

2 Directed DFS

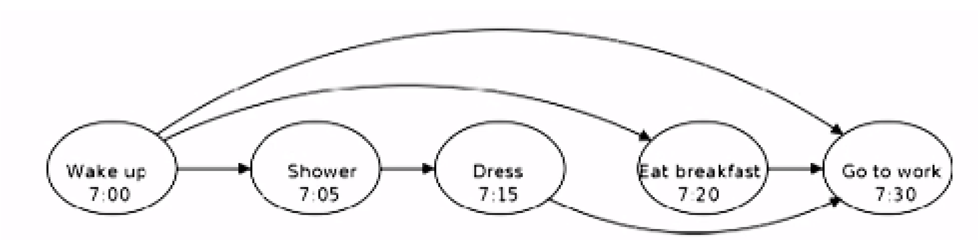
Can run **DFS** in directed graphs - Only follow directed edges - $ explore(v) $ finds all vertices reachable from $ v $ - Can still compute pre- and post- orderings

3 Linear Orderings

Imagine you had the following morning routine as shown below



we would like to order tasks to respect dependencies as below, is it possible



It may not be possible to order a graph if there are cycles

3.1 Cycle

A **cycle** in a graph $ G $ is a sequence of vertices $ v_1, v_2, ..., v_n $ so that $ (v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n), (v_n, v_1) $ are all edges

Theorem: > If $ G $ contains a cycle, it cannot be linearly ordered

Proof: - Has cycle $ v_1, ..., v_n $ - Suppose linearly ordered - Suppose v_k comes first - Then $v_k $ comes before $ v_{k-1}$, contradiction

4 DAGs

A directed graph $ G $ is a **Directed Acyclic Graph** (DAG) if it has no cycles

You need to be a DAG to linearly order, but is it sufficient?

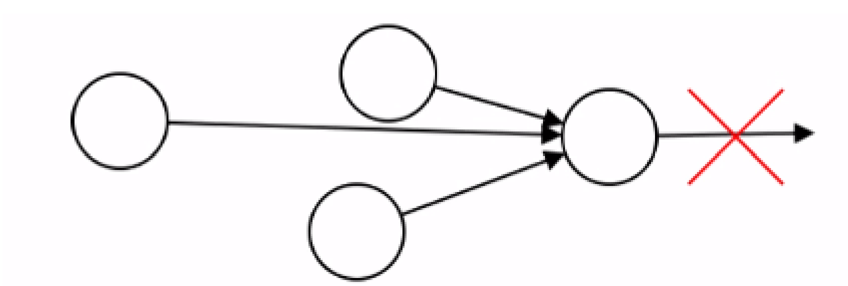
5 Topological Sort

Theorem: > Any DAG can be linearly ordered

5.1 Last Vertex

Consider the last vertex in the ordering. It cannot have any edges pointing out of it.

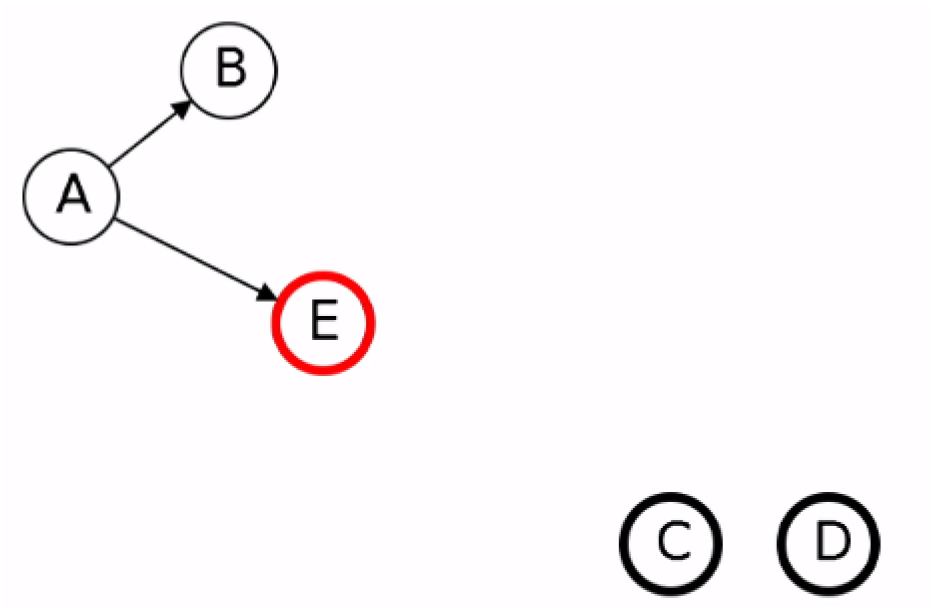
coming last means you can have edges pointing to it, but not from it



5.2 Source and Sinks

A source is a vertex with no incoming edges. A sink is a verted with no outgoing edges

Basic Idea: of implementing a topological sort - Find sink - Put at end of order - Remove from graph - Repeat



How do we even find a sink?

5.2.1 Follow Path

Follow path as far as possible $v_1 \to v_2 \to \ldots \to v_n$.

Eventually either: - Cannot extend (found sink) - Repeat a vertex (have a cycle)

5.3 Algorithm

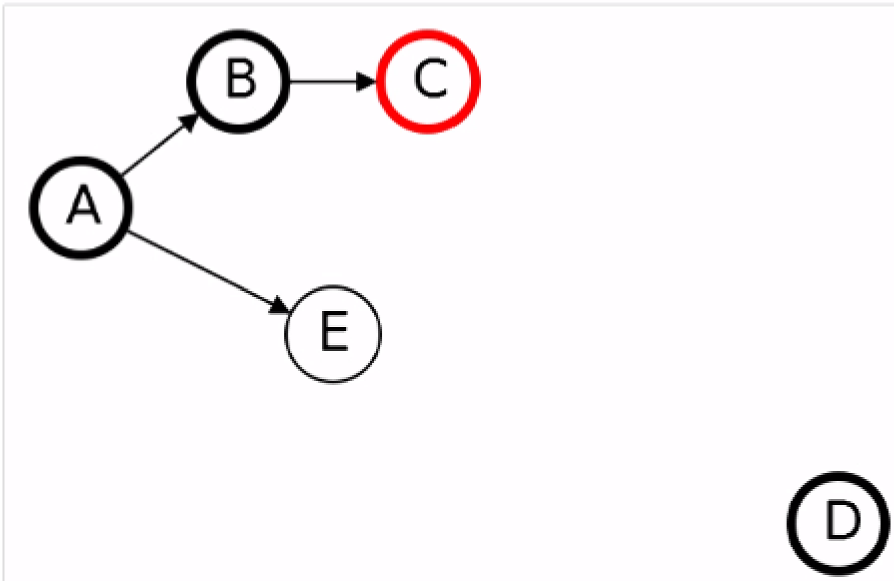
```
LinearOrder(G)

while G non-empty:
  Follow a path until cannot extend
  Find sink v
  Put v at end of order
  Remove v from G
```

5.3.1 Big O

- $O(|V|)$ paths
- Each takes $O(|V|)$ time
- Runtime $O(|V|^2)$

To speed up algorithm, instead of going all the way back to the beginning, we simply back up 1 step



In-other words, we simply do a **DFS**. More specifically, were doing a post-order!

5.4 Better Algorithm

```
TopologicalSort(G)

DFS(G)
sort vertices by reverse post-order
```

Theorem: > if $ G $ is a DAG, with an edge $ u $ to $ v $, $ post(u) > post(v) $

Proof:

Consider the following cases 1. Explore $ v $ before exploring $ u $ 2. Explore $ v $ while exploring $ u $ 3. Explore $ v $ after exploring $ u $ (cannot happen since there is an edgee)

Case I: Explore $ v $ before exploring $ u $ - Cannot reach $ u $ from $ v $ (DAG) - Must finish exploring $ v $ before finding $ u $ - $ post(u) > post(v) $

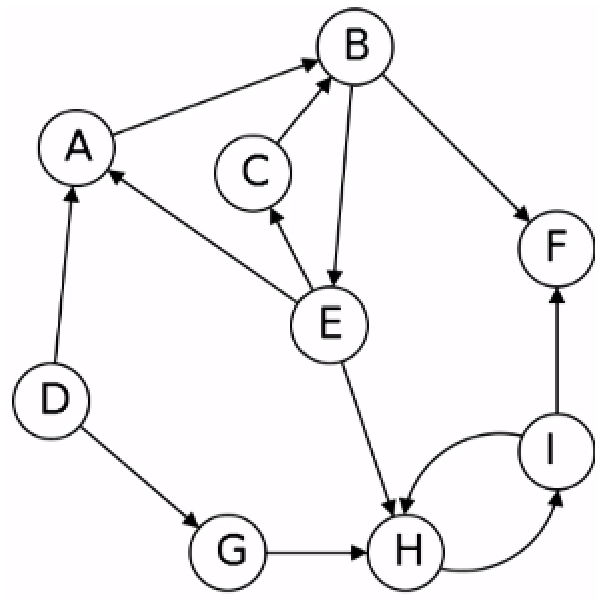
Case II: Explore $ v $ while exploring $ u $ - Must finish exploring $ v $ before can finish exploring $ u $ - Therefore $ post(u) > post(v) $

6 Strongly Connected Components

6.1 Connectivity in Digraphs

In undirected graphs, have connected components or you dont

Directed graphs are more complicated



In the graph above, from vertex $ D $ you can reach every other graph. But D is a source vertex, once you leave it, you cannot come back

$ F $ is the oppsite, once you reach $ F $, you cannot get out of it

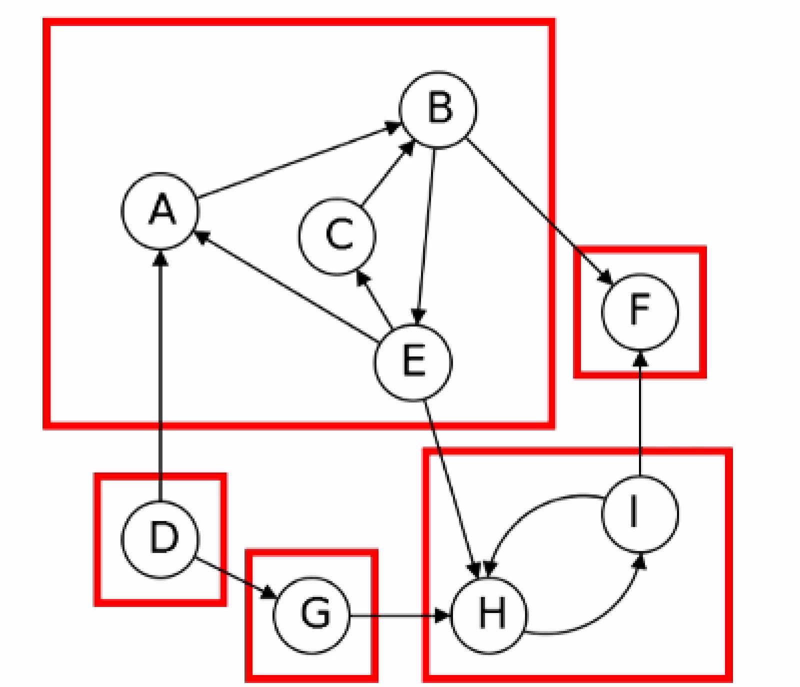
6.2 Possible Notions

- Connected by edges in any directions
- One vertex reachable from another
- Two vertices both reachable from the other

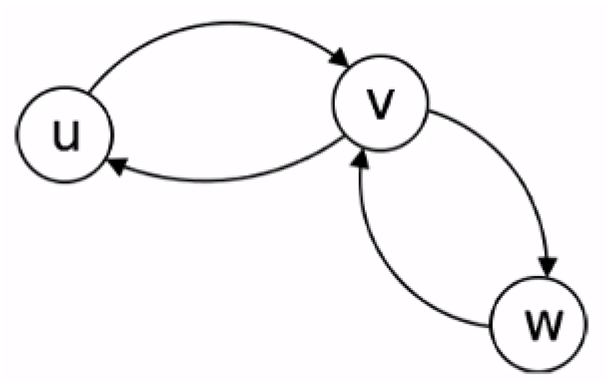
6.3 Strongle Conencted Components

> Two vertices $ v, w $ in directed graph are **connected** if you can reach $ v $ from $ w $ and can reach $ w $ from $ v $

Theorem: > A directed graph can be partitioned into **strongly connected components** where two vertices are connected if and only if there are in he same component

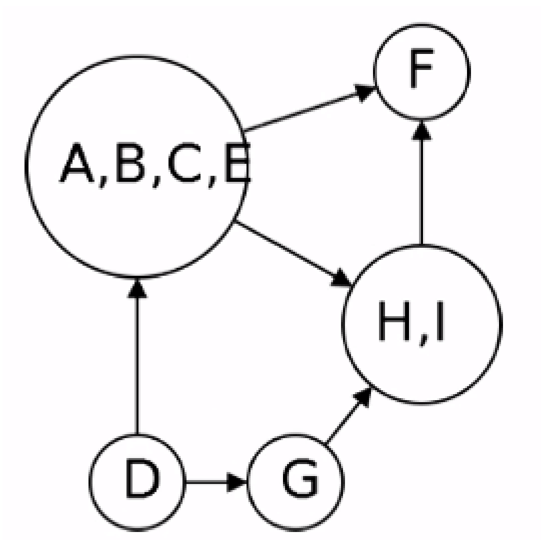


Proof: Need to show an equvalence relation



6.4 Metagraph

Even strongly connected components have edges to other components. We can draw something called the metagraph to show these



Theorem: > The metagraph of a graph $ G $ is always a DAG

Proof: > Suppose its not. Must be a cycle $ C $. Any nodes in cycle can reach any others. Should all be in same $ SCC $. Contradiction.

7 Computing Strongly Connected Components

Problem: - Input: A directed graph $ G $ - Output: The strongly connected components of $ G $

7.1 Easy Algorithm

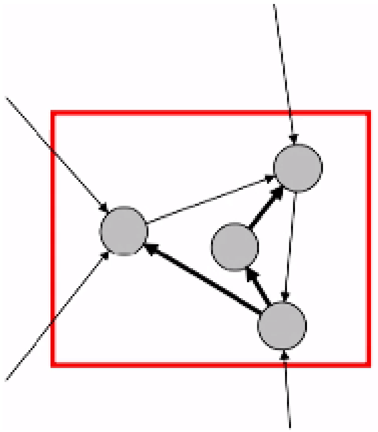
```
EasySCC(G)

for each vertex v:
  run explore(v) to determine
    vertices reachable from v
for each vertex v:
  find the u reachable from v that
    can also reach v
these are the SCCs
```

Problem with it is the runtime, which is $ O (|V|^2 + |V||E|) $

7.2 Sink Components

Idea: If \$ v \$ is in a sink SCC, \$ explore(v) \$ finds vertices reachable from \$ v \$. This is exactly the SCC of \$ v \$



7.3 Finding Sink Components

Need a way to find a sink SCC

Theorem: > if \$ C \$ and \$ C' \$ are two strongly connected components with an edge from some vertex of \$ C \$ to some vertex of \$ C' \$, then largest post in \$ C \$ bigger than latgest post in \$ C' \$

Proof: - Cases 1. Vist \$ C \$ before visit \$ C' \$ 2. Vist \$ C' \$ before vist \$ C \$

Case I: Visit \$ C \$ first - Can reach everything in \$ C' \$ from \$ C \$ - Explore all of \$ C' \$ while exploring \$ C \$ - \$ C \$ has largest post

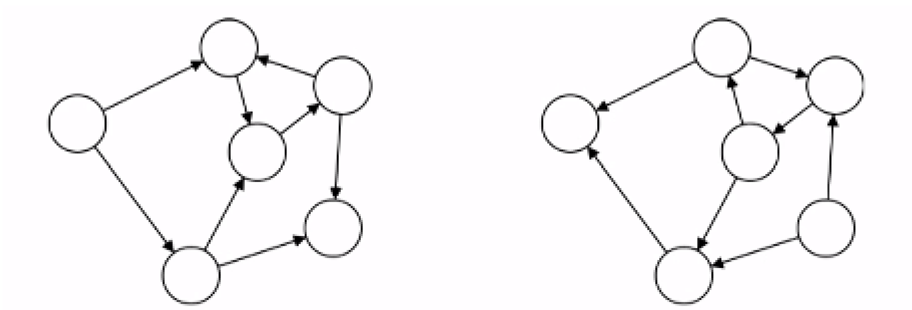
Case II: Visit \$ C' \$ first - Cannot reach \$ C \$ from \$ C' \$ - Must finish exploring \$ C' \$ before exploring \$ C \$ - \$ C \$ has larget post

Conclusion: The vertex with the largest post-order number is a source component!

the problem, however, is we want to find a sink component. We can solve this by taking the reverse graph

7.4 Reverse Graph

Let \$ G^R \$ be the graph obetained from \$ G \$ by reversing all of the edges



Reverse graph Components: - $ G^R $ and $ G $ have same SSCs - Source components of $ G^R $ arew sink components of $ G $

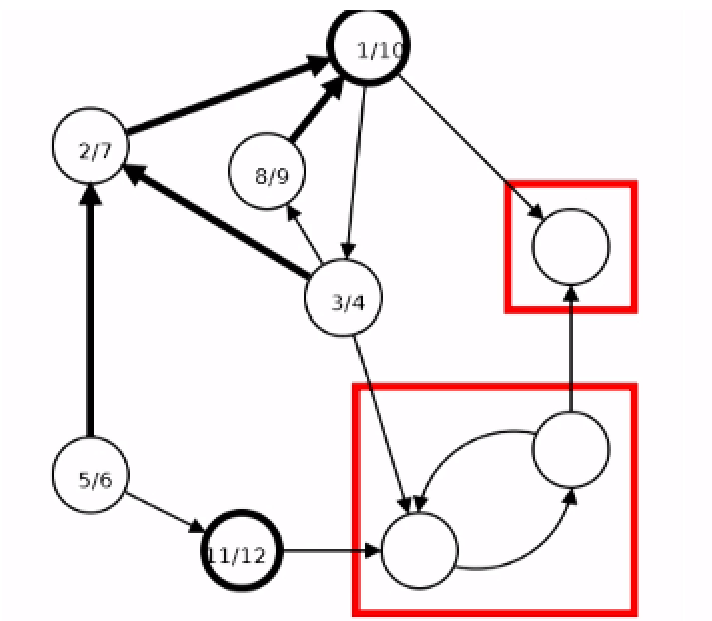
Find sink components of $ G $ by running DFS on $ G^R $

7.5 Basic Algorithm

```
SCCs(G)

run DFS(G^R)
let v have largest post number
run Explore(v)
vertices found are first SCC
Remove from G and repeat
```

You DFS the revers graph, the largest post number is the sink, we explore that and find the vertex and that is our first component. We remove it from the graph and do it again



Algorithm may be ineffecient due to having to re-run the DFS

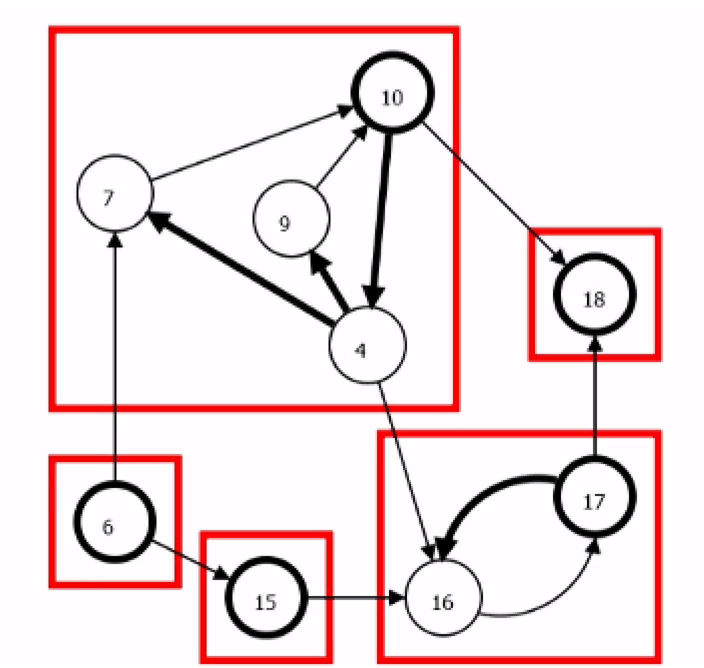
Improvement: - Don't need to rerun **DFS** on $ G^R $ - Largest remaining post number comes from sink component

7.6 New Algorithm

SCCs(G)

Run DFS(G^R)
for $v \in V$ in reverse postorder:
  if not visited(v):
    Explore(v)
    mark visited vertices
      as new SCC

Again, we simply store the post-order numbers and simply begin to visit them in reverse order from greatest to least



7.7 Runtime

Big O: - Esentially **DFS** on $ G^R $ and then on $ G $ - Runtime $ O (|V| + |E|) $