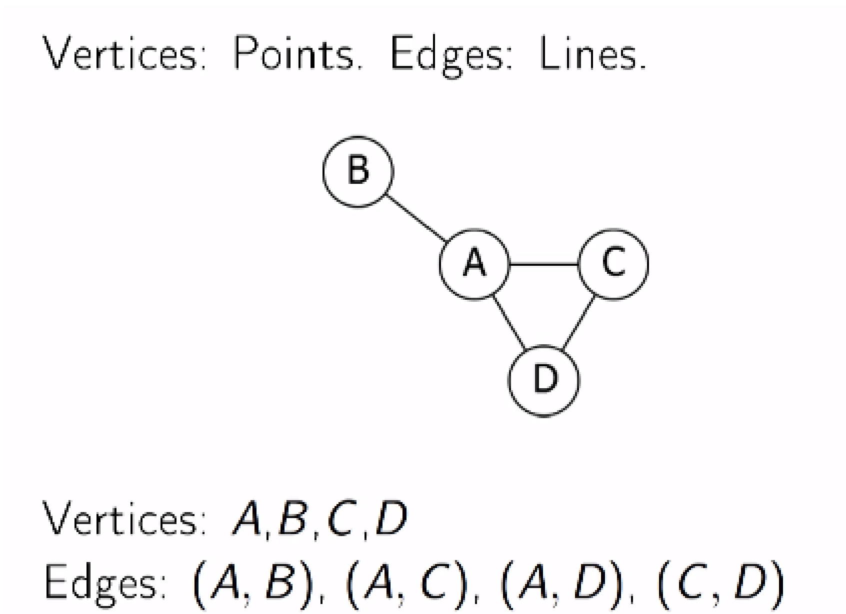
Graphs 1

June 6, 2020

1 Basics

Graphs: - Represent connections between objects - Describe many important phenomena

Definition > An (unidirected) Graph is a collection of $ V $ of vertices and a collection of $ E $ of edges each of which connects a pair of vertices

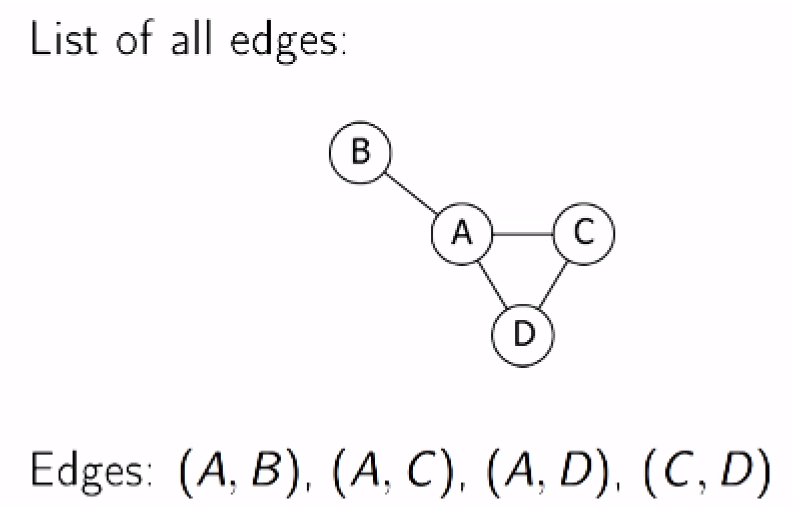


1.1 Loops and Multiple Edges

Loops connect a vertex to itself and sometimes multiple edges exist between the same vertices

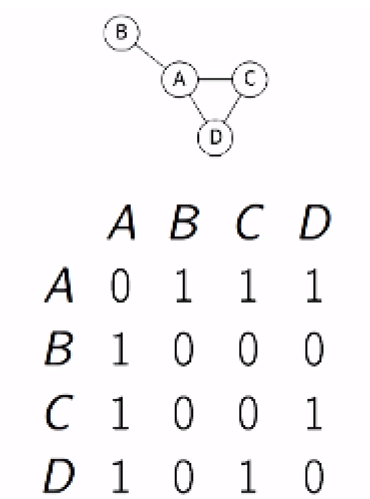
1.2 Representing Graphs

1.2.1 Edge List



1.2.2 Adjacency Matrix

Entries 1 if there is an edge, 0 if there is not. Were esentailly making a lookup table

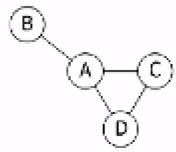


	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	1	0	0	1
D	1	0	1	0

1.2.3 Adjacency List

For each vertex, a list of adjacent vertices. For each vertex, we store its neighbors

For each vertex, a list of adjacent vertices.



A adjacent to B, C, D
B adjacent to A
C adjacent to A, D
D adjacent to A, C

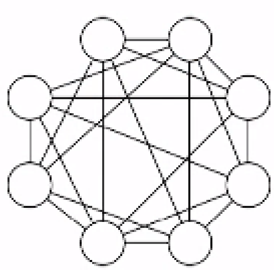
Big O Different Operations are faster in different representations

Op.	Is Edge?	List Edge	List Nbrs.
Adj. Matrix	$\Theta(1)$	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert)$
Edge List	$\Theta(\lvert E\rvert)$	$\Theta(\lvert E\rvert)$	$\Theta(\lvert E\rvert)$
Adj. List	$\Theta(\deg)$	$\Theta(\lvert E\rvert)$	$\Theta(\deg)$

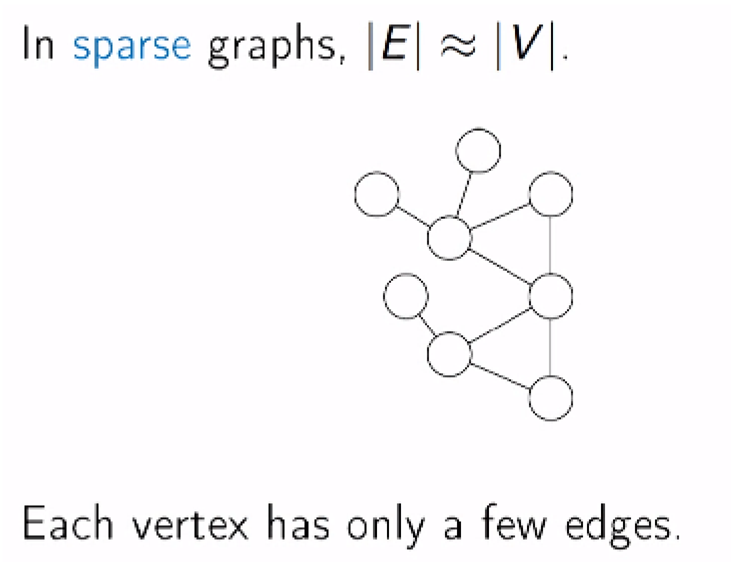
Graph algorithms runtimes depend on $ | V | $ and $ |E |$

the runtime depends largly on the density of the graph

In dense graphs, $|E| \approx |V|^2$.



A large fraction of pairs of vertices are connected by edges.



2 Exploring Graphs

2.1 Path

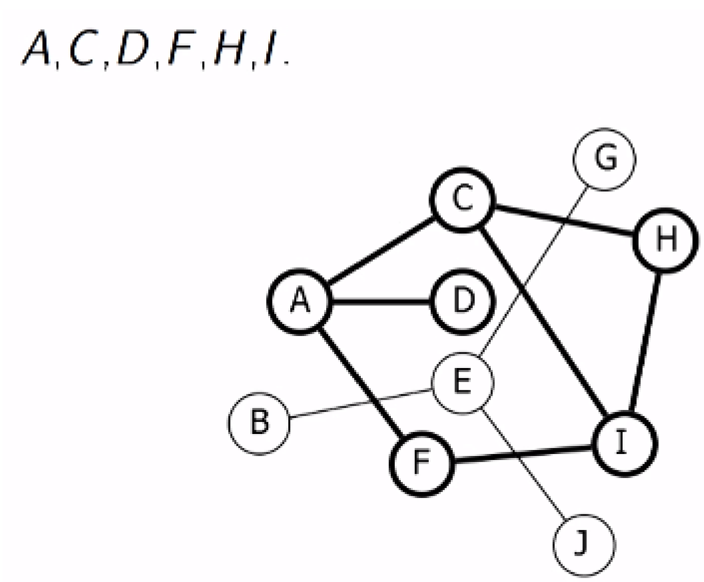
A path in a graph G is a sequence of vertices $v_0, v_1, ... v_n$ so that all $i_1 (v_i, v_{i+1})$ is an edge of G

2.2 Reachability

Input: Graph G and veertex s

output: The collection of vertices v of G so that there is a path from s to v

in the graph below, the vertices A, C, D, F, H, I are reachable from A



Basic Idea: We want to make sure that we have explored every edge leaving every vertex we have found

Pseudocode

```
Component(s)

DiscoveredNodes ← {s}
while there is an edge e leaving
DiscoveredNodes that has not been
explored:
  add vertex at other end of e to
  DiscoveredNodes
return DiscoveredNodes
```

What we need to do: - keep track of visited vertices - keep track of unprocessed vertices - which order we want to see the nodes

2.3 Depth First Ordering

We will explore new edges in Depth First order. We will follow a long path forward only backtracking when we hit a dead end

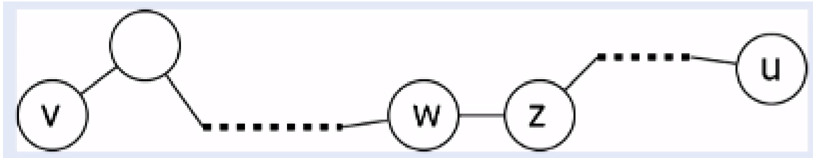
```
Explore(v)

visited(v) ← true
for (v, w) ∈ E:
  if not visited(w):
    Explore(w)
```

we need adjacency list representation!

So in DFS, where we use recursion, we basically backtrack when we pop the stack. When we return from a recursive policy!

Therom: Iff all vertice start unvisited, $ Explore(v) $ marks as visited exactly the vertices reachable from $ v $

Proof: - Only explore things reachable from $ v $ - $ w $ not marked as visited unless explored - if $ w $ explored, neighbors explored - $ u $ reachable from $ v $ by path - Assume $ w $ furthest along path explored 

- Must explore next item

Reach all Vertices: Sometimes you want to find all vertices of $ G $, not just thoese reachable from $ v $. The algorithm we would use in this case would be the $ DFS $ algorithm

```
DFS(G)

for all v ∈ V:    mark v unvisited
for v ∈ V:
  if not visited(v):
    Explore(v)
```

2.3.1 Runtime

Number of calls to explore: - Each explored vertex is marked visited - No vertex is explored after visited once - Each vertex is explored exactly once

Checking for neighbors:

- Each vertex checks each neighbor
- Total number of neighbors over all vertices is $ O(|E|) $

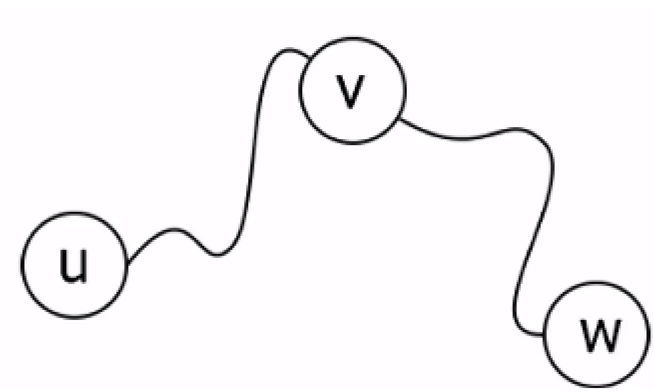
Big O: - $ O(1) $ work per vertex - $ O (1) $ work per edge - Total $ O (|V| + |E|) $

3 Connectivity

Theorm: > The vertices of a graph $ G $ can be partitioned into **Connected Components** so that $ v $ is reachable from $ w $ if and only if there are in the same connected component

Its like island, so bridges connect smaller islands together but some islands dont have bridges

Proof: Need to show reachability is an **equivalence relation**. Namely: - $ v $ is reachable from $ v $ - if $ v $ reachable from $ w $, $ w $ reachable from $ v $ - if $ v $ reachable from $ u $, and $ w $ reachable from $ v $, $ w $ reachable from $ u $



3.1 Problem

Connected Components: - Input: Graph $ G $ - Output: The connected components of $ G $

Idea: $ Explore(v) $ finds the connected component of $ v $. Just need to repeat to find other components. Modify DFS to do this. Modify goal to label connected components

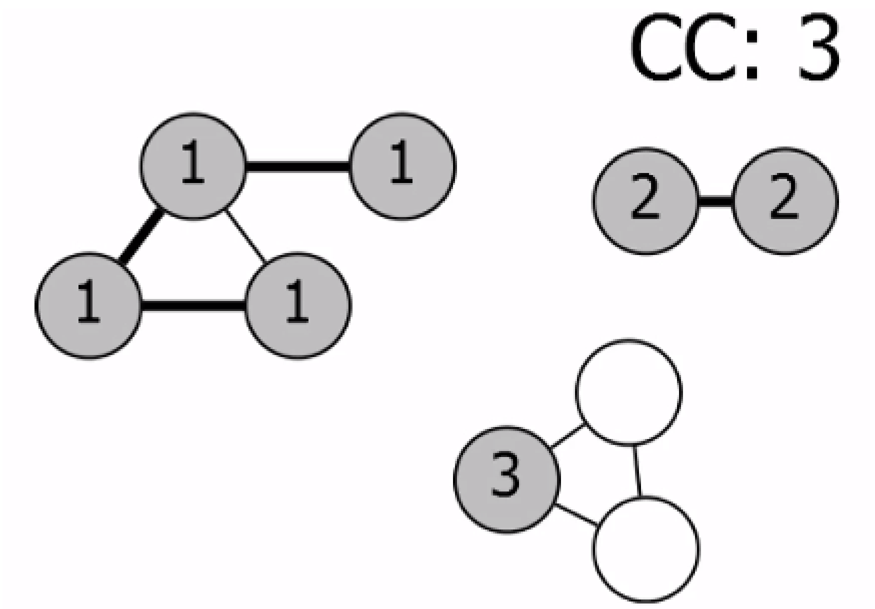
Explore(v)

```
visited(v) ← true
CCnum(v) ← cc
for (v, w) ∈ E:
  if not visited(w):
    Explore(w)
```

Modifications of DFS:

DFS(G)

```
for all v ∈ V mark v unvisited
cc ← 1
for v ∈ V:
  if not visited(v):
    Explore(v)
    cc ← cc + 1
```



Correctness: - Each new explore finds new connected component - Eventually find every vertex - Runtime still $ O (|V| + |E|) $

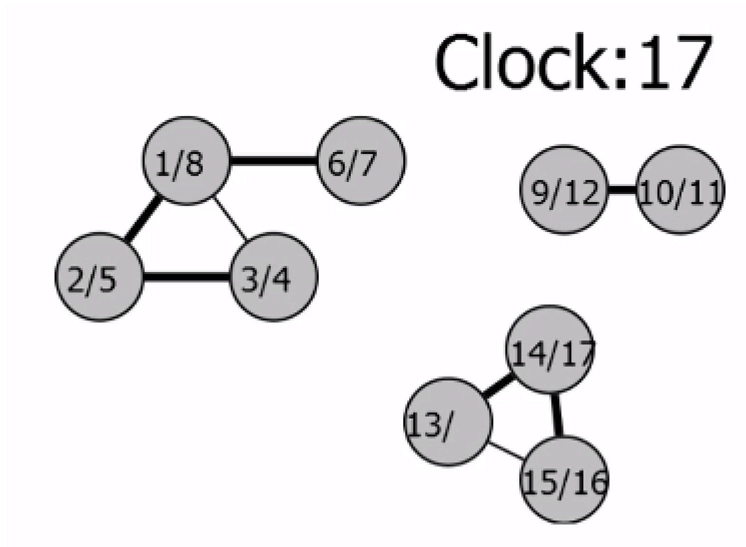
3.2 Pre-visit and Post-visit Orderings

Sometimes we dont just want to label a node as visited and want to do extra work. If we had functions $ previsit(v) $ and $ postvisit(v) $, this is where we would add them

```
Explore(v)

visited(v) ← true
previsit(v)
for (v, w) ∈ E:
  if not visited(w):
    explore(w)
postvisit(v)
```

Clock: - we might want to keep a clock with previsit/postvisit - clock ticks at each pre/post visit - records previsit and postvist times for each $ v $



3.2.1 Computing Pre- and Post- Numbers

Initialize clock to 1

previsit(v)

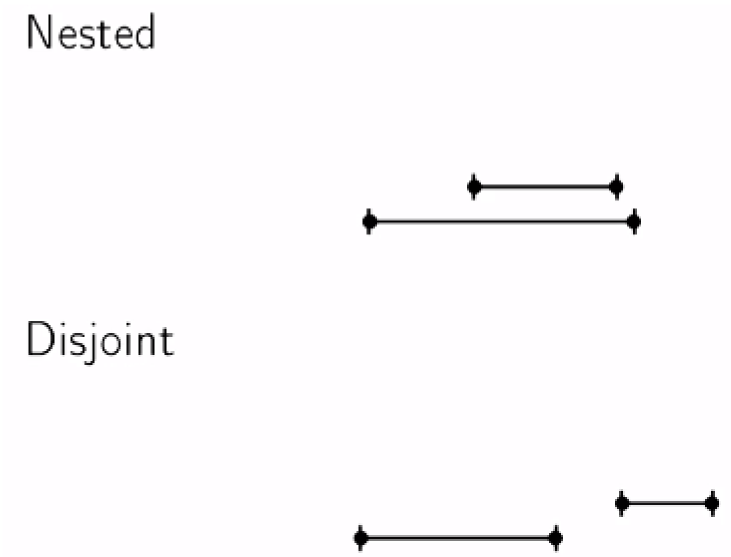
pre(v) $\leftarrow$ clock
clock $\leftarrow$ clock $+ 1$

postvisit(v)

post(v) $\leftarrow$ clock
clock $\leftarrow$ clock $+ 1$

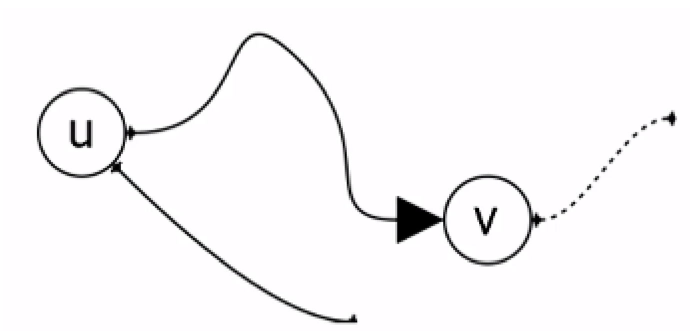
Result: Previsit and Postvisit numbers tell us about the execution of DFS

Lemma: > For any vertices $ u $, $ v $ the intervals $ [pre(u), post(u)]$ and $ [pre(v), post(v)] $ are either nested or disjoint

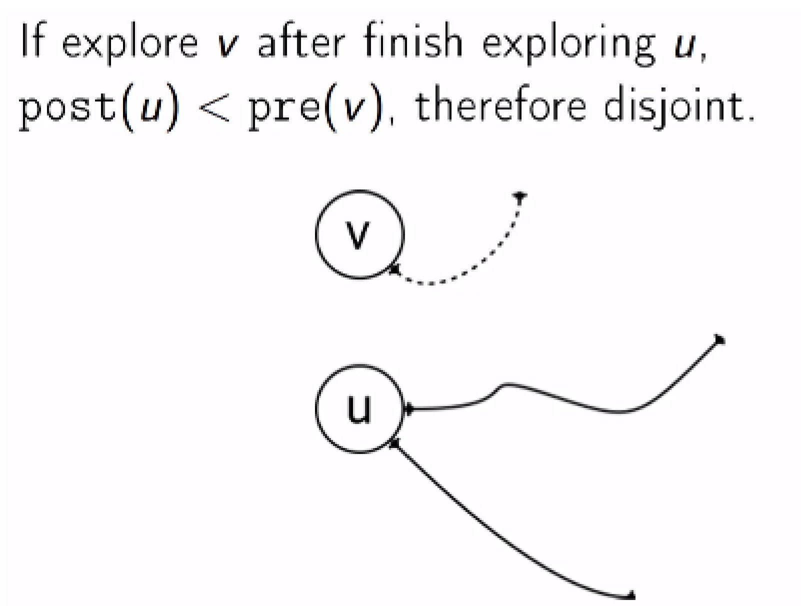


Proof: Assume that u visited before v.

Two cases: 1. Find v while exploring u (u an ancestor of v) 2. Find v after exploring u (u is a cousin og v)



Case 1:



Case 2: