CS 163 & CS 265: Graph Algorithms

Week 1: Basics

Lecture 1c: DFS and Connectivity

David Eppstein
University of California, Irvine

Winter Quarter, 2024

Depth-first search

Depth-first search

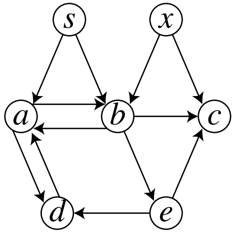
Easy recursive exploration method

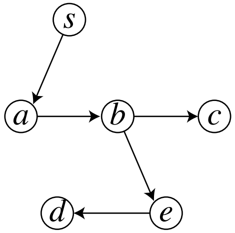
- Main idea: Whenever you find a new reachable vertex v, recursively explore its neighbors, the vertices that can be reached in one more step from v
- Start by exploring the starting vertex s and its neighbors
- Very important optimization: keep track of the vertices you've already explored and don't re-explore if you reach them again
  Otherwise infinite loops are possible
- The set of already-explored vertices can be re-used as the return value from the search

Depth-first reachability pseudocode

```
def reachable(G,s):
    // arguments are an implicit graph G
    // and the starting vertex s

    reached = set()

    def recurse(v):
        add v to reached
        for each edge in G from v to w:
            if w is not already in reached:
                recurse(w)

    recurse(s)
    return reached
```

Depth-first reachability example





```
recurse(s)
  reached: $\{s\}$
  recurse(a)
    reached: $\{s, a\}$
    recurse(b)
      reached: $\{s, a, b\}$
      recurse(c)
        reached: $\{s, a, b, c\}$
      recurse(e)
        reached: $\{s, a, b, c, e\}$
        recurse(d)
          reached:
          $\{s, a, b, c, d, e\}$
return $\{s, a, b, c, d, e\}$
```

Tree in bottom shows recursive calls, but it is also a tree in the graph, with paths from s to all other vertices

It is called the DFS tree

The algorithm can be modified to construct and return it

Analysis of depth-first search

- We have at most one call to recurse per vertex, because after that it is in the reached set and won't be called again
- Inside each recursive call there is a loop over its neighbors
- Naive analysis: n calls $\times$ n iterations of the loop $= O(n^2)$
  but we can do much better!

- In a directed graph, each edge is looped over at most once (in the recursive visit to its start vertex). In undirected, at most twice (once for each endpoint). So total number of times the loop runs, over all calls to recurse, is $O(m)$
- Each loop run takes $O(1)$ time to find the next edge, check membership in reached, and set up a recursive call
- Total over the whole algorithm: n calls to recurse$\times O(1)$ work outside the loop $+$ $O(m)$ runs of the loop $\times$ $O(1)$ work inside the loop $= O(n+m) = O(m)$

Issues with DFS in web spider application

- When exploring parts of the web graph spread across multiple web servers, it will hit single servers in hard bursts rather than spreading the load across the servers

- Doesn't deal well with implicit graphs that are not finite

  Example: In the root folder of your web site, set up a symbolic link named "nest" that points back to the root folder itself. Now you have a site with infinitely many pages of the form "http://site/nest/nest/nest/nest/..."

  Spider will get stuck in this trap and never return to the rest of the web

Connected components

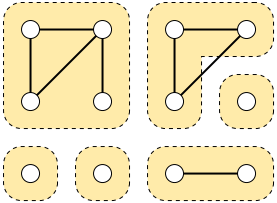
Warm-up: Connectivity in undirected graphs

Undirected reachability is an equivalence relation

- Every vertex can reach itself
- If u can reach v, then v can reach u (reverse the path)
- If u can reach v and v can reach w then u can reach w (join two paths to form a walk)

Equivalence classes: subsets of vertices such that vertices are in the same subset $\Leftrightarrow$ they can reach each other

They are called connected components



This graph has 6 connected components

Vertices that are not connected to anything else belong to one-vertex components

Every vertex is in exactly one component

Finding the connected components

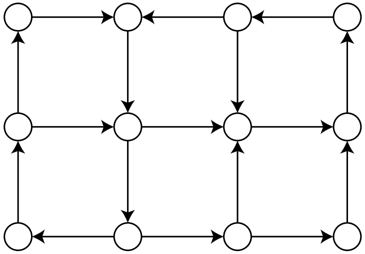
DFS or BFS can find one component; repeat until whole graph is explored

```
def components(G):
    reached = empty set
    components = empty list

    def recurse(v):
        add v to reached and to last component in list
        for each edge in G from v to w:
            if w is not already in reached:
                recurse(w)

    for each vertex v in G:
        if v is not in reached:
            add new empty set to components
            recurse(v)

    return components
```

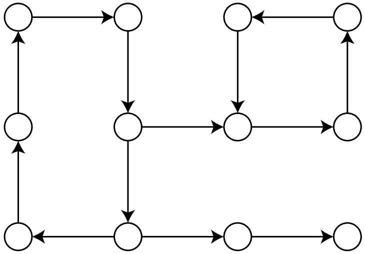
Strong connectivity definitions

Directed strong connectivity

A directed graph is strongly connected
if every vertex can reach every other vertex



Strongly connected



Not strongly connected

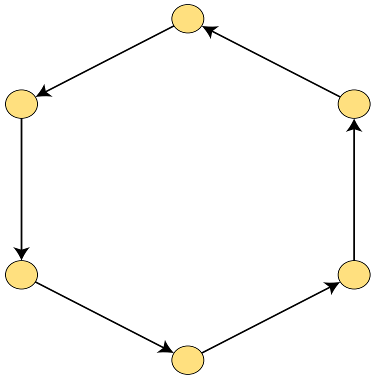
A useful property for a web site or one-way street map to have!
People browsing the site or driving the streets don't get stuck

Simplest strongly connected graphs: Cycles

Cycle = Walk that starts and ends at the same vertex and has no other repetitions

- In undirected graphs: Connected graph where all vertices touch two edges
- In directed graphs: Strongly connected graph where all vertices have one edge in and one edge out

(If repetitions are allowed: “circuit”, “tour” “closed trail”)



Testing whether a graph is strongly connected

Testing reachability from each vertex would be too slow

n runs of reachability, each with time $O(m)$ — total $O(mn)$

Instead:

- Choose an arbitrary starting vertex s
- Use reachability to check that s can reach all vertices
- Construct graph G^R with edge directions are reversed from G
- Use a second run of the reachability algorithm to check that s can reach all vertices in G^R $\Leftrightarrow$ every vertex can reach s in G
- If both checks succeed, every two vertices u and v can reach each other by a walk from u to s and then from s to v

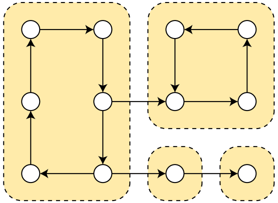
Time $= 2 \times O(m) = O(m)$

A finer-grained version of strong connectivity

Two vertices u and v are strongly connected (made-up notation: $u \rightleftharpoons v$) if there are paths or walks both ways, $u \rightsquigarrow v$ and $v \rightsquigarrow u$

This is an equivalence relation:

- Every vertex v has $v \rightleftharpoons v$ (it has one-vertex paths to itself)
- If $u \rightleftharpoons v$ then $v \rightleftharpoons u$ (swap the two paths)
- If $u \rightleftharpoons v$ and $v \rightleftharpoons w$ then $u \rightleftharpoons w$: join paths into longer walks $u \rightsquigarrow v \rightsquigarrow w$ and $w \rightsquigarrow v \rightsquigarrow u$

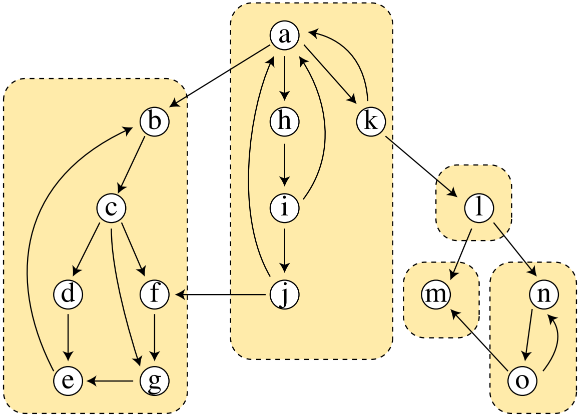


Its equivalence classes are called strongly connected components

Every vertex belongs to exactly one of them (even the vertices that are not part of any cycles)

Once we know the components, we can check whether $u \rightleftharpoons v$ just by testing if they are in the same component as each other

Another example

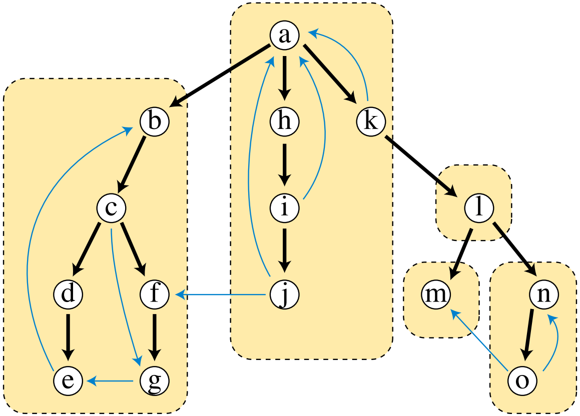


Two vertices that belong to a cycle ⇒ in the same component

Some components can include more than one cycle

Depth-first algorithm for strong connectivity

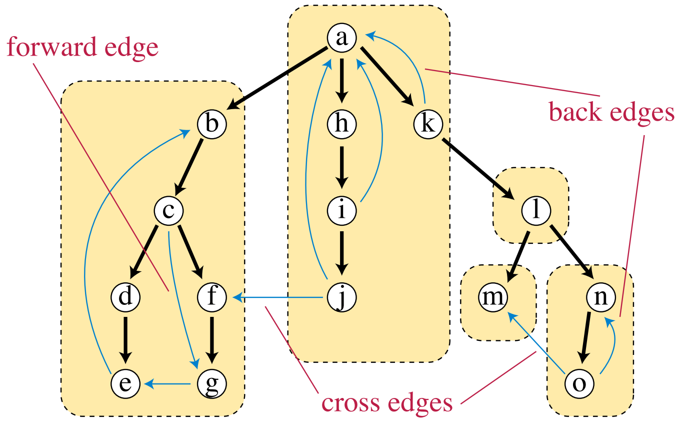
Same example



Heavy black edges show the depth-first search tree

Every component is a connected subtree! Once DFS enters a component, it can't leave until whole component has been explored

Classification of edges in DFS



If an edge is not in the DFS tree, it must be forward (from ancestor to descendant), back (from descendant to ancestor), or cross (right to left, from a subtree later in the search to an earlier one)

Main ideas of DFS SCC algorithm

- Use a stack of vertices
- When we start a recursive call at a vertex v, push v onto the stack
- When we're about to return from the call for v, if v is the top vertex of a component subtree:
    - Start making a new component
    - Repeatedly pop vertices from the stack into the new component until we pop v

But how to recognize the top vertices of components?

Top vertices of components

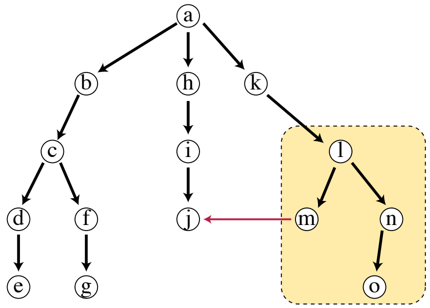
A vertex v is the top vertex of its component

if and only if

There is no "escape edge" from v or from the subtree below v

leading to a vertex earlier than v

that has not already been popped into a different component

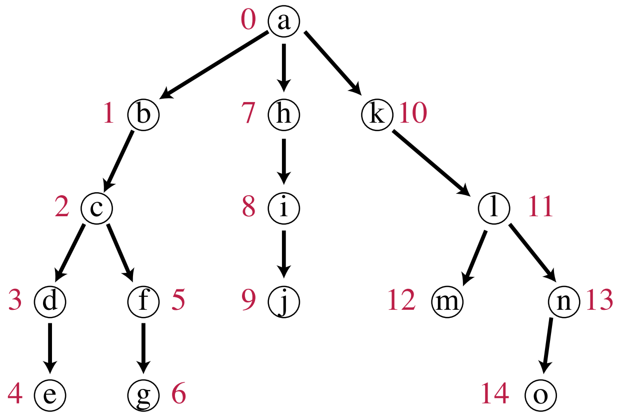


In the example, l is the top vertex of its (one-vertex) component

Its subtree does have an edge to the earlier vertex j

But j will already be popped by the time we reach l

DFS numbers



To be able to tell which of two vertices was reached earlier by DFS

Decorate each vertex by its position in order of when it was reached (keep a counter, and increment each time you assign it to a vertex)

Escape numbers

(Called “link” in the Wikipedia article on this algorithm)

The escape number of v is the smallest DFS number of an earlier vertex w such that:

- w has not yet been popped into a component when we explore v and its subtree
- There is an edge from v or its subtree to w

If no such w exists, we just use the DFS number of v itself

Then v is the top vertex of its component if and only if its escape number is not smaller than its DFS number

Summary of algorithm ideas

Use depth-first search, modified in the following ways:

- Use a stack to collect vertices that are not yet in components; push each vertex when we start its recursive visit
- Decorate vertices with their DFS numbers
- Decorate vertices with escape numbers, calculated as minimum of (DFS number of self, escape numbers of children, DFS numbers of edges from self to unpopped vertices)
- When we are about to return from a vertex v whose escape number equals its DFS number, make a new component and pop the stack into it until reaching v

Whole algorithm

```
def strongcomps(G):
    dfsn = new dictionary
    // reached <=> in dfsn
    // #reached = len(dfsn)

    esc = new dictionary
    popped = new set
    comps = new list
    theStack = new stack

    for v in G:
        if v not in dfsn:
            recurse(v)

    return comps
```

```
def recurse(v):
    esc[v] = dfsn[v] = len(dfsn)
    theStack.push(v)
    for each edge from v to w:
        if w not in dfsn:
            recurse(w)
            esc[v] =
              min(esc[v],esc[w])
        else if w not in popped:
            esc[v] =
              min(esc[v],dfsn[w])
    if dfsn[v] == esc[v]:
        add new list C to comps
        x = None
        while x != v:
            x = theStack.pop()
            add x to C
            add x to popped
```

Morals of the story

Both undirected reachability and directed strong connectivity are equivalence relations

We can partition the vertices into “components” so that two vertices are reachable or strongly connected exactly when they both belong to the same component

Finding components takes $O(m)$ time using modifications to DFS

Same ideas can be used for some other problems (e.g. “blocks” of undirected graphs: do two vertices belong to a cycle?)