CS 163 & CS 265: Graph Algorithms

Week 1: Basics

Lecture 3: Reachability, DFS, and BFS

David Eppstein
University of California, Irvine

Fall Quarter, 2022

Reachability and implicit graphs

Spiders

We've discussed how web search engines build indexes of web pages and sort the results of queries, but how did they find those pages in order to build an index of them?

Web crawler or spider

Algorithm that explores the web graph or any unknown graph finding all of the vertices that it can reach by following links from a given starting vertex

Example: find all web pages on the www.ics.uci.edu web site that can be reached from the starting page https://www.ics.uci.edu/

Pages that are not linked from other pages will not be found

Implicit graph representation

Web graph: We do not already have its vertices and edges in our computer (that's the problem we're trying to solve) but they still exist out on the internet

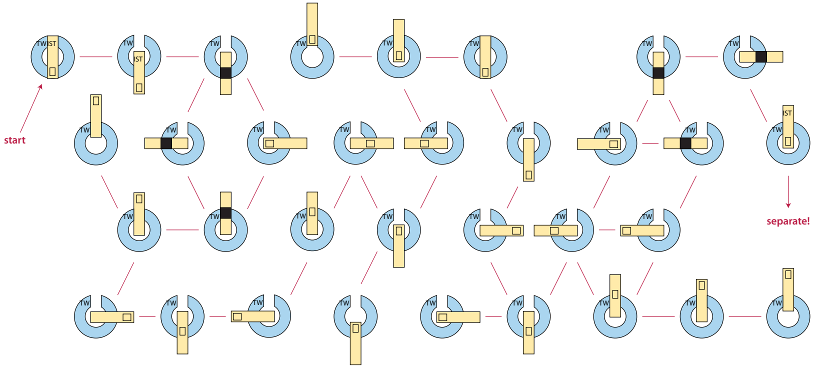
We can represent any vertex by its url, a string

Given any vertex we can find its outgoing edges by reading and parsing the html content at that url

More generally, an implicit graph is any graph that is not already represented in our computer, where we have some way of describing vertices and finding their edges

Another application of implicit graphs: Puzzles

Vertices = states of the puzzle, edges = moves



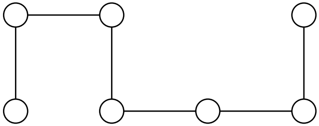
Some puzzles like Rubik's cube have many more states than we can store in computer memory at once

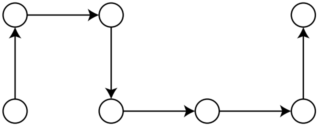
Real-world application in planning sequences of actions

Paths and walks

Path in an undirected graph: alternating sequence of vertices and edges, without repetitions, starting and ending at vertices, with each edge sandwiched by its two endpoints

In a directed graph: same, but each edge should have starting endpoint before it and destination endpoint after it in the sequence, so that the edges are all oriented consistently along the path





Walk: allow repeated vertices and edges

Unfortunately some sources use different terminology: “simple path” for no repetitions, or “path” allowing repetitions

A single vertex by itself is a path (very short alternating sequence)

Reachability

Vertex v can reach vertex w in graph G
$\Leftrightarrow$ G has a path starting at v and ending at w
$\Leftrightarrow$ G has a walk starting at v and ending at w

Repetitions in any walk can be shortcut to make a path

Every vertex can reach itself by a one-vertex path

Reachability is transitive: if G has paths $u \rightsquigarrow v$ and $v \rightsquigarrow w$, we can connect them together to form a walk $u \rightsquigarrow w$

In directed graphs, it can be asymmetric: there might exist a path $u \rightsquigarrow v$ but not one in the reverse direction

In undirected graphs, it is symmetric, any path $u \rightsquigarrow v$ can be reversed to give a path $v \rightsquigarrow u$

Formulation of the web spider problem

Given an implicit graph G and a starting vertex s

Find and return the set of all vertices that can be reached from s

Algorithm doesn't have to know that the graph is implicit, but in the web spider application some graph operations are limited: for instance we cannot easily find incoming edges to vertices

Web spiders don't need to return the paths from s to the reachable vertices but other applications might need them
(e.g. for puzzles, paths are sequences of moves that solve them)

Depth-first search

Depth-first search

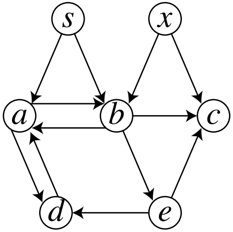
Easy recursive exploration method

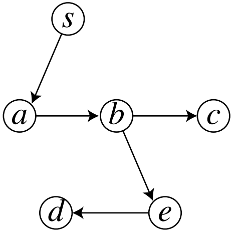
- Main idea: Whenever you find a new reachable vertex v, recursively explore its neighbors, the vertices that can be reached in one more step from v
- Start by exploring the starting vertex s and its neighbors
- Very important optimization: keep track of the vertices you've already explored and don't 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

Breadth-first search

Breadth-first search

Probably a better choice for web spiders:

- Also linear time, $O(m)$
- Only slightly more complicated to implement
- Explores vertices in order by # steps from start, spreading load among multiple servers
- For nested web sites, will continue exploring rest of the web so everything else is eventually reached

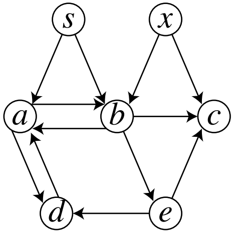
Intuitive idea:

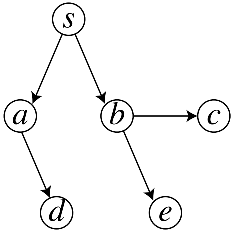
- Queue (first-in first-out) data structure of vertices to explore
- Explore vertices in queue order
- When we find an edge to an unreached vertex add it to the queue

Breadth-first reachability pseudocode

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

    reached = new set containing s
    queue = new queue containing s

    while queue is not empty:
        remove first item from queue; call it v
        for each edge in G from v to w:
            if w is not already in reached:
                add w to both reached and queue

    return reached
```

Breadth-first reachability example





queue: s; reached: s

remove s
add unreached neighbors a,b
queue: a,b; reached: s,a,b

remove a
add unreached neighbor d
queue: b,d
reached: s, a, b, d

remove b
add unreached neighbors c, e
queue: d, c, e
reached: s, a, b, d, c, e

remove d, c, e
(no more unreached neighbors)

Tree in bottom shows which *v* was the first to find each vertex *w*, but it is also a tree in the graph, with shortest paths from *s* to other vertices

It is called the BFS tree

The algorithm can be modified to construct and return it

Analysis of breadth-first search

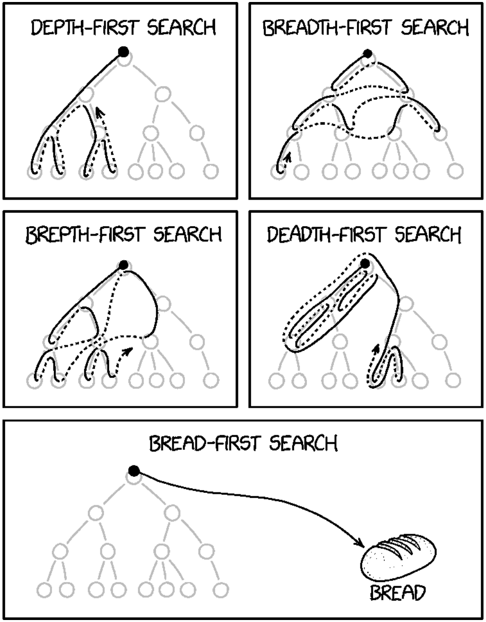
Even though it's a loop rather than a recursion,
it's mostly the same as DFS

Each vertex is only added to the queue when not in reached, and
at the same time is added to reached, so added once per vertex $\Rightarrow$
also removed once per vertex

Each edge is looped over only when we remove one of its
endpoints from the queue $\Rightarrow$ once or twice per edge

Constant work for each vertex or edge that we loop over
$\Rightarrow O(m)$ time

Other exploration orders are also possible



https://xkcd.com/2407/

The morals of the story

The web graph is already in a graph representation,
an implicit graph, even before we start exploring it

So are the graphs (state spaces) of puzzles and planning problems

Both depth-first and breadth-first search can be used to find all of
reachable vertices of a (finite) graph in linear time

Breadth-first may work better for the web graph (less bursty
network traffic, better behavior when faced with infinite graphs)