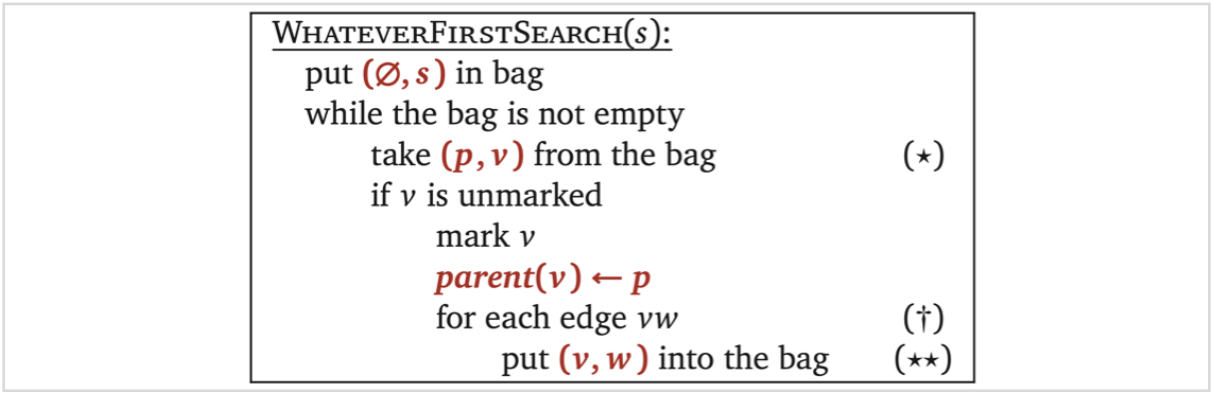
CS 4349.003.19F Lecture 13–October 9, 2019

Main topics for #lecture include #graph_traversal and #graph_reductions.

Whatever-First Search

- Last time, we looked at the following fundamental problem: given graph G and vertex s, what vertices are *reachable* from s? In other words, for what vertices v is there a path from s to v? In other other words, which vertices are in s's component? We'll stick with undirected graphs for most of today.
- We discussed a procedure for this problem called whatever-first search.
- Whatever-first search stores a set of edges in some data structure I'll call a "bag". The only thing you need to know about the bag is that we can put stuff into it and pull stuff out. Specifically, we'll put edges we think may belong to a path from s to some other vertex into the bag. Since we don't need to reach s, we'll start with (emptyset, s) in the bag.
- And we'll mark vertices as we reach them so we don't keep searching again from those same vertices.



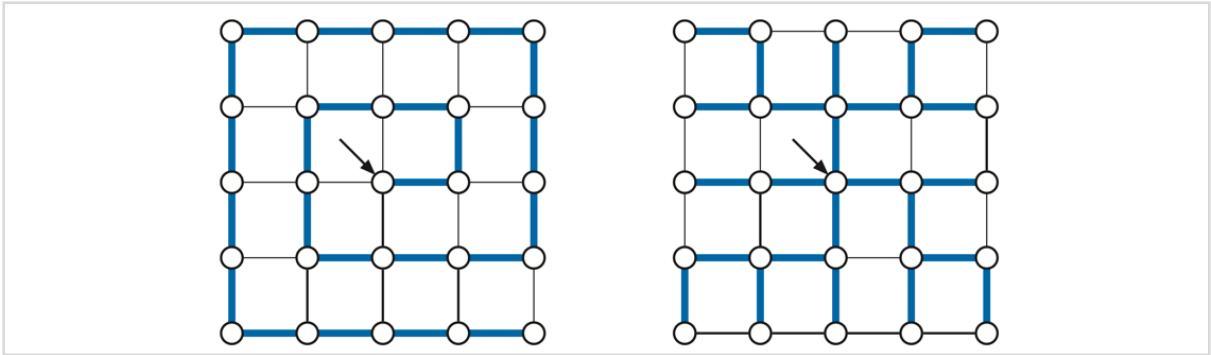
- Lemma: WhateverFirstSearch(s) marks every vertex reachable from s and only those vertices. Moreover, the set of pairs (v, parent(v)) with parent(v) ≠ emptyset defines a spanning tree of the component containing s.
- Proof:
  - First, everything is marked at most once thanks to that if statement.
  - Now, I'll prove we mark every reachable vertex v by induction on the shortest path distance from s to v. Doing induction over path length is pretty common when proving correctness of graph algorithms.
  - WhateverFirstSearch certainly marks s since we put it in the bag right away and immediately take it out. If v is reachable, there is a shortest path s ➔ … ➔ u ➔ v. Inductively, vertex u is marked, and then (u, v) is put in the bag. But later (u, v) is pulled out and v is marked if it isn't marked already.
  - Now to prove every marked vertex is reachable. Every pair (v, parent(v)) is an edge in the graph. I claim v ➔ parent(v) ➔ parent(parent(v)) ➔ … eventually leads to s by

induction on the order in which vertices were marked. s leads back to s. For any other vertex v, parent(v) was marked before v. Because parent(v) ➜ parent(parent(v))… leads back to s, v ➜ parent(v) ➜ parent(parent(v))… leads back to s.

- So, we mark exactly the reachable vertices. Further, the parent edges form a connected subgraph containing s and all other marked vertices. Every marked vertex except s has a unique parent, so the number of pairs is one less than the number of marked edges. A connected subgraph with one fewer edge than there are vertices is a tree (you can prove this statement using induction also!)

Analysis

- The time it takes to do the search depends upon what kind of bag we use. Say it takes T time to insert or remove something from the bag.
- The t for loop is executed once per marked vertex, so at most V times total.
- Each edge uv is put in the bag twice, once as (u, v) and once as (v, u) so the * * line runs at most 2E times.
- We can't take more out of the bag than we put in, so * runs at most 2E + 1 times.
- Assuming the graph is stored with an adjacency list, we can run that for loop for v in constant time per edge vw. The running time is O(V + ET).
- All that said, the bag we choose has some important consequences and can lead to algorithms you may have seen before.
- Suppose we implement the bag as a stack so we pull things out in the opposite order we add them. This leads to a depth-first search and *depth-first spanning tree*. The shape depends on the starting vertex and what order we add edges to the stack, but they tend to be long and skinny. We'll study their properties more next week. Pushing and popping from a stack takes O(1) time per operation, so the total running time is O(V + E).



- If we use a queue instead we get a breadth-first search and *breadth-first spanning tree*. Again, the exact look depends on the starting point and order we push edges, but these are short and bushy. In a couple weeks, we'll see that they contain shortest paths. Pushing and pulling takes O(1) time each, so the total running time is O(V + E).
- We could also use a priority queue for the bag to get what Erickson calls a *best-first search*. If we use a min-heap for our operations, then each insertion and removal takes O(log E)

time for O(V + E log E) time total. The spanning tree we get depends upon what keys we use for the priority queue. One choice leads to minimum spanning trees. Another leads to shortest paths trees. A third choice leads to a so-called *widest path tree*. Again, we'll cover at least the first two of these trees over the next couple weeks.

Finding Components

- Some of those variants have immediate applications. For example, breadth-first-search gives you shortest paths when the edges all have length 1. Minimum spanning trees model cheap ways to connect a network of computers.
- But WhateverFirstSearch is also useful as a subroutine for other algorithms.
- For example, suppose our graph is disconnected, so we cannot reach all vertices from a single vertex s. If we call WhateverFirstSearch(s), then we'll only visit and mark the vertices in s's component.
- We can visit and mark *every* vertex in the graph by iteratively performing fresh searches from still unmarked vertices.

```
WFSALL(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          WHATEVERFIRSTSEARCH(v)
```

- Now, this may seem odd. Why don't we just iterate over the vertices, marking each of them as we go?
- Well, that strategy assumes we *can* iterate over all vertices. If you're treating certain objects or phenomena as graphs, it may not be immediately how to list all the vertices.
- But maybe more importantly, WFSAll does an interesting thing with the graph. It marks all vertices in one component, then all in another, and so on. It ends up marking one complete component at a time until it is done visiting every component.
- Which means we can *count* components very easily.

```
COUNTCOMPONENTS(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          WHATEVERFIRSTSEARCH(v)
  return count
```

- A count like this could be useful for social networks like Facebook. If the count is 50, then

there are 50 isolated communities of friends. Maybe we should introduce each other? Or maybe Facebook wants to use different ads for each community.

- But to treat each community differently, we'd also need a way to know who is in each community. This is also a simple change to WFSAll and WhateverFirstSearch.

```
CountAndLabel(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          LabelOne(v, count)
  return count
```

```
⟨⟨Label one component⟩⟩
LabelOne(v, count):
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          comp(v) ← count
          for each edge vw
              put w into the bag
```

- Now we can tell if two vertices or people are in the same component by whether or not they have the same label. Even this procedure has further applications that we'll come back to later.
- Each of these variants marks each vertex once and puts each edge in or out of the bag twice, so they still run in O(V + ET) time. If we're only counting or labeling components, we may as well use breadth-first or depth-first search so the running time is only O(V + E).

Directed Graphs

- Whatever-first-search also works just fine in directed graphs.
- The only change we need to make is to put the *out*-neighbors of a vertex into the bag.

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge v→w
              put w into the bag
```

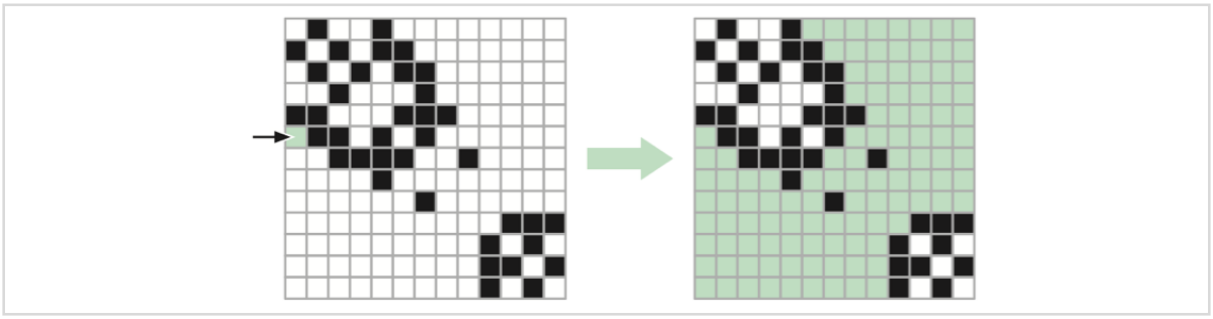
- We could also add the parent pairs or whatever, and you'll end up getting a spanning tree rooted at s containing every vertex *reachable* from s. Note that the vertices you reach may change depending on where you start since the graph is directed. We'll see this again on Monday.

Graph Reductions

- I want to finish today by giving an example of something I've said a couple times: you can often solve a problem by building a graph and then running your favorite graph algorithm

that you've already seen before with little to no modification.

- Say we're given a pixel map as a two-dimensional array P[1 .. n, 1 .. n]. Each entry is called a *pixel*, and its value is the color of that pixel. This is how simple drawing programs represent images.
- A *connected region* in a pixel map is a connected subset of pixels with the same color where two pixels are adjacent if they are immediate horizontal or vertical neighbors.
- You may have seen this little paint bucket icon in your favorite drawing program. It performs a *flood-fill* operation that changes every pixel in a connected region to a new color. The operation just needs to know the indices i and j of a single pixel in the region.



- We can perform a flood fill from (i, j) by using a simple *reduction* to WhateverFirstSearch:
  - Create a graph G = (V, E) where V is the set of pixels and E is the pairs of adjacent pixels sharing a color.
  - WhateverFirstSearch((i, j))
  - Color all marked vertices.
- Instead of writing a new algorithm from scratch, we just reduced to one we already knew! This required:
  - Specifying the *input* to the graph algorithm. We defined G = (V, E) based on the pixel map.
  - Saying how to use the *output*. We colored the marked vertices.
- Now one crucial thing that's easy to mess up: we need to analyze the algorithm in terms of the *original* problem size.
- WhateverFirstSearch takes O(V + E) time. V = n^2 and E = O(n^2) (≤ 4 edges per vertex), so we took O(n^2) time.
- This would be enough to say we have an efficient algorithm for flood-fill, but there are a couple practical optimizations we can make. DON'T WORRY ABOUT THESE UNTIL YOU HAVE AN ASYMPTOTICALLY EFFICIENT ALGORITHM.
  - In an actual implementation, we don't need to build a separate graph. Instead, we work with the pixels directly as if they were the vertices. Listing adjacent pixels of the same color is easily done in O(1) time per pixel.
  - WhateverFirstSearch really only takes O(1 + E') time where E' is the number of edge's in s's connected component. Each pixel has up to four "edges" . So if there are k pixels to color, the algorithm really takes O(k) time. k may be much smaller than n^2.

- To end to today, I want to give you an opportunity to provide informal feedback for this class.
- On Monday, we'll look at a depth-first search variant of whatever-first search where we use a stack. The order in which it marks vertices is particularly useful for certain applications.