Administrivia

How to Ace the Technical Interview Session today!
- 6-8pm
- Sieg 134

No BS CS Career Talk Thursday
- 5:30-6:30
- Bag 131
Shortest Paths

How does Google Maps figure out this is the fastest way to get to office hours?
Representing Maps as Graphs

How do we represent a map as a graph? What are the vertices and edges?
Representing Maps as Graphs
Shortest Paths

The **length** of a path is the sum of the edge weights on that path.

**Shortest Path Problem**

**Given:** a directed graph $G$ and vertices $s$ and $t$

**Find:** the shortest path from $s$ to $t$
Unweighted graphs

Let’s start with a simpler version: the edges are all the same weight (unweighted)

If the graph is unweighted, how do we find a shortest paths?
Unweighted Graphs

If the graph is unweighted, how do we find a shortest paths?

What’s the shortest path from s to s?
- Well....we’re already there.

What’s the shortest path from s to u or v?
- Just go on the edge from s

From s to w,x, or y?
- Can’t get there directly from s, if we want a length 2 path, have to go through u or v.
Unweighted Graphs: Key Idea

To find the set of vertices at distance k, just find the set of vertices at distance k-1, and see if any of them have an outgoing edge to an undiscovered vertex.

Do we already know an algorithm that does something like that?

Yes! BFS!

```java
bfsShortestPaths(graph G, vertex source)
    toVisit.enqueue(source)
    source.dist = 0
    while(toVisit is not empty){
        current = toVisit.dequeue()
        for (v : current.outNeighbors())
        {
            if (v is unknown){
                v.distance = current.distance + 1
                v.predecessor = current
                toVisit.enqueue(v)
                mark v as known
            }
        }
    }
```
Unweighted Graphs

Use BFS to find shortest paths in this graph.

```java
bfsShortestPaths(graph G, vertex source)
    toVisit.enqueue(source)
    source.dist = 0
    mark source as visited
    while(toVisit is not empty){
        current = toVisit.dequeue()
        for (v : current.outNeighbors()){)
            if (v is not yet visited){
                v.distance = current.distance + 1
                v.predecessor = current
                toVisit.enqueue(v)
                mark v as visited
            }
        }
    }
```
Unweighted Graphs

If the graph is unweighted, how do we find a shortest paths?

```java
bfsShortestPaths(graph G, vertex source)
    toVisit.enqueue(source)
    source.dist = 0
    while(toVisit is not empty){
        current = toVisit.dequeue()
        for (v : current.outNeighbors())
        {
            if (v is unknown){
                v.distance = current.distance + 1
                v.predecessor = current
                toVisit.enqueue(v)
                mark v as known
            }
        }
    }
```
What about the target vertex?

Shortest Path Problem

**Given:** a directed graph $G$ and vertices $s,t$

**Find:** the shortest path from $s$ to $t$.

BFS didn’t mention a target vertex...
It actually finds the shortest path from $s$ to every other vertex.
Weighted Graphs

Each edge should represent the “time” or “distance” from one vertex to another.

Sometimes those aren’t uniform, so we put a weight on each edge to record that number.

The length of a path in a weighted graph is the sum of the weights along that path.

We’ll assume all of the weights are positive
- For GoogleMaps that definitely makes sense.
- Sometimes negative weights make sense. Today’s algorithm doesn’t work for those graphs
- There are other algorithms that do work.
Weighted Graphs: Take 1

BFS works if the graph is unweighted.

Maybe it just works for weighted graphs too?
Weighted Graphs: Take 1

BFS works if the graph is unweighted. Maybe it just works for weighted graphs too?

What went wrong? When we found a shorter path from $s$ to $u$, we needed to update the distance to $v$ (and anything whose shortest path went through $u$) but BFS doesn’t do that.
Weighted Graphs: Take 2

Reduction (informally)
Using an algorithm for Problem B to solve Problem A.

You already do this all the time.

In Homework 3, you reduced implementing a hashset to implementing a hashmap.

Any time you use a library, you’re reducing your problem to the one the library solves.

Can we reduce finding shortest paths on weighted graphs to finding them on unweighted graphs?
Weighted Graphs Take 2

Given a weighted graph, how do we turn it into an unweighted one without messing up the path lengths?
Weighted Graphs: A Reduction

Transform Input

Unweighted Shortest Paths

Transform Output
Weighted Graphs: A Reduction

What is the running time of our reduction on this graph?

O(|V|+|E|) of the modified graph, which is...slow.

Does our reduction even work on this graph?

Ummm....

tl;dr: If your graph’s weights are all small positive integers, this reduction might work great. Otherwise we probably need a new idea.
Weighted Graphs: Take 3

So we can’t just do a reduction.

Instead figure out why BFS worked in the unweighted case, try to make the same thing happen in the weighted case.

How did we avoid this problem:
In BFS When we used a vertex $u$ to update shortest paths we already knew the exact shortest path to $u$.

So we never ran into the update problem

If we process the vertices in order of distance from $s$, we have a chance.
Weighted Graphs: Take 3

Goal: Process the vertices in order of distance from s

Idea:

Have a set of vertices that are “known”
- (we know at least one path from s to them).

Record an estimated distance
- (the best way we know to get to each vertex).

If we process only the vertex closest in estimated distance, we won’t ever find a shorter path to a processed vertex.
- This statement is the key to proving correctness.
- It’s nice if you want to practice induction/understand the algorithm better.
Dijkstra’s Algorithm

Dijkstra(Graph G, Vertex source)
initialize distances to $\infty$
mark source as distance 0
mark all vertices unprocessed
while(there are unprocessed vertices)
    let u be the closest unprocessed vertex
    foreach(edge (u,v) leaving u)
        if(u.dist+weight(u,v) < v.dist)
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
mark u as processed

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
<th>Predecessor</th>
<th>Processed</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm

Dijkstra(Graph G, Vertex source)
initialize distances to $\infty$
mark source as distance 0
mark all vertices unprocessed
while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    foreach(edge (u,v) leaving u){
        if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
        }
    }
    mark u as processed
}