CS 561, Lecture 22

Jared Saia
University of New Mexico

Today's Outline

- BFS and DFS Wrapup
- Shortest Paths

Generic Traverse

```
Traverse(s){
  put (nil,s) in bag;
  while (the bag is not empty){
    take some edge (p,v) from the bag
    if (v is unmarked)
      mark v;
      parent(v) = p;
      for each edge (v,w) incident to v{
        put (v,w) into the bag;
      }
    }
  }
}
```


DFS and BFS

- If we implement the "bag" by using a stack, we have *Depth First Search*
- If we implement the "bag" by using a queue, we have *Breadth First Search*

Analysis

- Note that if we use adjacency lists for the graph, the overhead for the "for" loop is only a constant per edge (no matter how we implement the bag)
- If we implement the bag using either stacks or queues, each operation on the bag takes constant time
- Hence the overall runtime is $O(|V| + |E|) = O(|E|)$

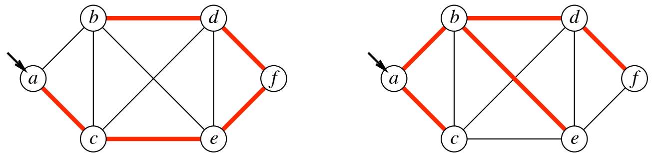
DFS vs BFS

- Note that DFS trees tend to be long and skinny while BFS trees are short and fat
- In addition, the BFS tree contains *shortest paths* from the start vertex s to every other vertex in its connected component. (here we define the length of a path to be the number of edges in the path)

Final Note

- Now assume the edges are weighted
- If we implement the "bag" using a *priority queue*, always extracting the minimum weight edge from the bag, then we have a version of Prim's algorithm
- Each extraction from the "bag" now takes $O(|E|)$ time so the total running time is $O(|V| + |E| \log |E|)$

Example



A depth-first spanning tree and a breadth-first spanning tree of one component of the example graph, with start vertex a.

Searching Disconnected Graphs

If the graph is disconnected, then Traverse only visits nodes in the connected component of the start vertex s. If we want to visit all vertices, we can use the following "wrapper" around Traverse

```
TraverseAll(){
  for all vertices v{
    if (v is unmarked){
      Traverse(v);
    }
  }
}
```


DFS and BFS

- Note that we can do DFS and BFS equally well on undirected and directed graphs
- If the graph is undirected, there are two types of edges in G: edges that are in the DFS or BFS tree and edges that are not in this tree
- If the graph is directed, there are several types of edges

DFS in Directed Graphs

- *Tree edges* are edges that are in the tree itself
- *Back edges* are those edges (u, v) connecting a vertex u to an ancestor v in the DFS tree
- *Forward edges* are nontree edges (u, v) that connect a vertex u to a descendant in a DFS tree
- *Cross edges* are all other edges. They go between two vertices where neither vertex is a descendant of the other

Acyclic graphs

- Useful Fact: A directed graph G is acyclic if and only if a DFS of G yeilds no back edges
- Challenge: Try to prove this fact.

Take Away

- BFS and DFS are two useful algorithms for exploring graphs
- Each of these algorithms is an instantiation of the Traverse algorithm. BFS uses a queue to hold the edges and DFS uses a stack
- Each of these algorithms constructs a spanning tree of all the nodes which are reachable from the start node s

Shortest Paths Problem

- Another interesting problem for graphs is that of finding shortest paths
- Assume we are given a weighted *directed* graph $G = (V, E)$ with two special vertices, a source s and a target t
- We want to find the shortest directed path from s to t
- In other words, we want to find the path p starting at s and ending at t minimizing the function

$$w(p) = \sum_{e \in p} w(e)$$

Example

- Imagine we want to find the fastest way to drive from Albuquerque,NM to Seattle,WA
- We might use a graph whose vertices are cities, edges are roads, weights are driving times, s is Albuquerque and t is Seattle
- The graph is directed since driving times along the same road might be different in different directions (e.g. because of construction, speed traps, etc)

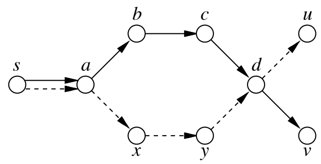
SSSP

- Every algorithm known for solving this problem actually solves the following more general *single source shortest paths* or SSSP problem:
- Find the shortest path from the source vertex s to *every* other vertex in the graph
- This problem is usually solved by finding a *shortest path tree* rooted at s that contains all the desired shortest paths

Shortest Path Tree

- It's not hard to see that if the shortest paths are unique, then they form a tree
- To prove this, we need only observe that the sub-paths of shortest paths are themselves shortest paths
- If there are multiple shotest paths to the same vertex, we can always choose just one of them, so that the union of the paths is a tree
- If there are shortest paths to two vertices u and v which diverge, then meet, then diverge again, we can modify one of the paths so that the two paths diverge once only.

Example

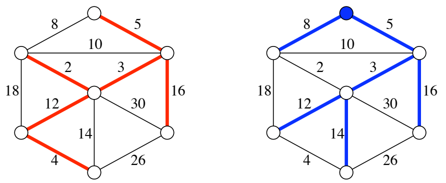


If $s \to a \to b \to c \to d \to v$ and $s \to a \to x \to y \to d \to u$ are both shortest paths,
then $s \to a \to b \to c \to d \to u$ is also a shortest path.

MST vs SPT

- Note that the minimum spanning tree and shortest path tree can be different
- For one thing there may be only one MST but there can be multiple shortest path trees (one for every source vertex)

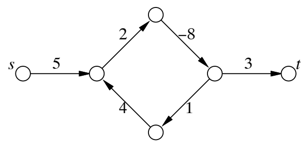
Example



A minimum spanning tree (left) and a shortest path tree rooted at the topmost vertex (right).

Negative Weights

- We'll actually allow negative weights on edges
- The presence of a negative cycle might mean that there is no shortest path
- A shortest path from s to t exists if and only if there is *at least one* path from s to t but no path from s to t that touches a negative cycle
- In the following example, there is no shortest path from s to t



SSSP Algorithms

- We'll now go over some algorithms for SSSP on directed graphs.
- These algorithms will work for undirected graphs with slight modification
- In particular, we must specifically prohibit alternating back and forth across the same undirected negative-weight edge
- Like for graph traversal, all the SSSP algorithms will be special cases of a single generic algorithm

SSSP Algorithms

Each vertex v in the graph will store two values which describe a *tentative* shortest path from s to v

- $dist(v)$ is the length of the tentative shortest path between s and v
- $pred(v)$ is the predecessor of v in this tentative shortest path
- The predecessor pointers automatically define a tentative shortest path tree

Defns

Initially we set:

- $dist(s) = 0$, $pred(s) = NULL$
- For every vertex $v \neq s$, $dist(v) = \infty$ and $pred(v) = NULL$

Relaxation

- We call an edge (u, v) *tense* if $dist(u) + w(u, v) < dist(v)$
- If (u, v) is tense, then the tentative shortest path from s to v is incorrect since the path s to u and then (u, v) is shorter
- Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it
- If there are no tense edges, our algorithm is finished and we have our desired shortest path tree

Relax

```
Relax(u,v){
  dist(v) = dist(u) + w(u,v);
  pred(v) = u;
}
```


Correctness

- The correctness of the relaxation algorithm follows directly from three simple claims
- The run time of the algorithm will depend on the way that we make choices about which edges to relax

Claim 1

- If $dist(v) \neq \infty$, then $dist(v)$ is the total weight of the predecessor chain ending at v:

$$s \rightarrow \cdots \rightarrow pred(pred(v)) \rightarrow pred(v) \rightarrow v.$$

- This is easy to prove by induction on the number of edges in the path from s to v. (left as an exercise)

Claim 2

- If the algorithm halts, then $dist(v) \leq w(s \rightsquigarrow v)$ for *any* path $s \rightsquigarrow v$.
- This is easy to prove by induction on the number of edges in the path $s \rightsquigarrow v$. (left as an exercise)

Claim 3

- The algorithm halts if and only if there is no negative cycle reachable from s.
- The 'only if' direction is easy—if there is a reachable negative cycle, then after the first edge in the cycle is relaxed, the cycle *always* has at least one tense edge.
- The 'if' direction follows from the fact that every relaxation step reduces either the number of vertices with $dist(v) = \infty$ by 1 or reduces the sum of the finite shortest path lengths by some positive amount.

Generic SSSP

- We haven't yet said how to detect which edges can be relaxed or what order to relax them in
- The following Generic SSSP algorithm answers these questions
- We will maintain a "bag" of vertices initially containing just the source vertex s
- Whenever we take a vertex u out of the bag, we scan all of its outgoing edges, looking for something to relax
- Whenever we successfully relax an edge (u, v), we put v in the bag

InitSSSP

```
InitSSSP(s){
  dist(s) = 0;
  pred(s) = NULL;
  for all vertices v != s{
    dist(v) = infinity;
    pred(v) = NULL;
  }
}
```

GenericSSSP

```
GenericSSSP(s){
  InitSSSP(s);
  put s in the bag;
  while the bag is not empty{
    take u from the bag;
    for all edges (u,v){
      if (u,v) is tense{
        Relax(u,v);
        put v in the bag;
      }
    }
  }
}
```


Generic SSSP

- Just as with graph traversal, using different data structures for the bag gives us different algorithms
- Some obvious choices are: a stack, a queue and a heap
- Unfortunately if we use a stack, we need to perform $\Theta(2^{|E|})$ relaxation steps in the worst case (an exercise for the diligent student)
- The other possibilities are more efficient

Diskstra's Algorithm

- If we implement the bag as a heap, where the key of a vertex v is $dist(v)$, we obtain Dijkstra's algorithm
- Dijkstra's algorithm does particularly well if the graph has no negative-weight edges
- In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from s
- It follows that each vertex is scanned at most once, and thus that each edge is relaxed at most once

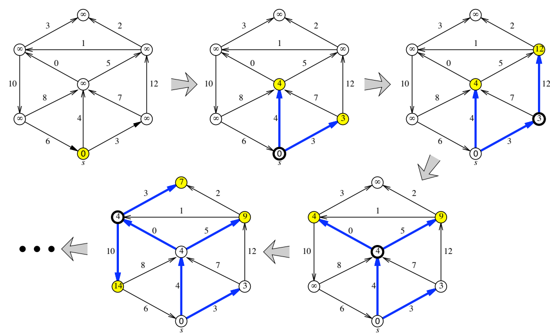
Dijktra's Algorithm

- Since the key of each vertex in the heap is its tentative distance from s, the algorithm performs a DecreaseKey operation every time an edge is relaxed
- Thus the algorithm performs at most $|E|$ DecreaseKey's
- Similarly, there are at most $|V|$ Insert and ExtractMin operations
- Thus if we store the vertices in a Fibonacci heap, the total running time of Dijkstra's algorithm is $O(|E| + |V| \log |V|)$

Negative Edges

- This analysis assumes that no edge has negative weight
- The algorithm given here is still correct if there are negative weight edges but the worst-case run time could be exponential
- The algorithm in our text book gives incorrect results for graphs with negative edges (which they make clear)

Example



Four phases of Dijkstra's algorithm run on a graph with no negative edges. At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.