Announcements

Remaining 3 questions of prelab 10 are due on Friday.
(Possible Negative) Weighted Shortest Paths

input: a graph $G=(V,E)$ with arbitrary edge costs $c_e$ and a start vertex $s$
goal: determine the cost of the least-cost path from $s$ to every vertex in $V$
(and while you’re at it, compute the path itself.)
The Bellman-Ford Algorithm

Recursive solution would be
\[ BF( v, k ) \]
- if \( k = 0 \) and \( v = s \), return 0
- if \( k = 0 \) (and \( v \) is not \( s \)) return \( \infty \)
- return \[ \min \{ BF(v,k-1), \min_{u : (u,v) \in E} \{ BF(u,k-1) + c(u,v) \} \} \]

But this solution could make an exponential number of recursive calls, blech!

Unwind it to its iterative solution and solve all vertices and all \( k \) at once:
\[ BF( ) \] – iteratively compute \( BF(v,k) \) for all \( v \) and all \( k=0,...,n \)
- create \( n \times n \) array \( BF \)
- initialize \( BF[s][0] = 0 \), for all other \( v \), \( BF[v][0] = \infty \)
- for \( k=1,2,...,n \)
  - for each vertex \( v \) in \( V \)
    - set \( BF[v][k] = \min \{ BF[v][k-1], \min_{u : (u,v) \in E} \{ BF[u][k-1] + c(u,v) \} \} \)
  - return \( BF[\ast][n-1] \)

RT: inner for loop runs in time \( \sum_v \deg(v) \in O(m) \), so whole alg is \( O(mn) \).
An natural number $k$ is either
- $0$, or
- $k'+1$, where $k'$ is a natural number

For a natural number $k$, let $P(k)$ denote the property that
- len of the shortest $s$-$v$ path on $\leq k$ edges equals $\text{BF}[v][k]$ for all vertices $v$

We will show $P(k)$ is true for $k=0, \ldots, n$, by induction on $k$.

(Why do we care? Well if $P(n)$ is true, then $\text{BF}[v][n]$ is the correct length of the shortest $s$-$v$ path on at most $n$ edges, and any simple path has at most $n-1$ edges!)

Step 0: try it on some examples (see the previous slide to check.)

Step 1: base case. $k=0$ .
- len of shortest $s$-$s$ path on 0 edges is $0 = \text{BF}[s][0]$.
- len of shortest $s$-$v$ path on 0 edges for $v \neq s$ is $\infty = \text{BF}[v][0]$

Step 2: inductive step. $k=k'+1$, where $k'$ is a natural number.

Want to show that $P(k)$ is true, given the hypothetical assumption that $P(k')$ is true.
Proof of Correctness

For the induction hypothesis, suppose P(k') is true, that is, for all vertices v, len of shortest s-v path on at most k' edges is BF[v][k']. We want to show that len of shortest s-v path on at most k=k'+1 edges is BF[v][k].

1. Connect left-hand-sides of P(k') and P(k):
The s.p. from s to v on at most k edges either
   • uses at most k-1 edges:
     \[ \text{len s.p. to v on } \leq k \text{ edges} = \text{len s.p. to v on } \leq k' \text{ edges} \]
   • uses up to k edges:
     \[ \text{len s.p. to v on } \leq k \text{ edges} = \min \{ \text{len s.p. to u on } \leq k' \text{ edges} + c(u,v) \} \]

2. Connect right-hand-sides of P(k') and P(k):
Our algorithm computes BF[v][k] as:
   \[ \text{set } BF[v][k] = \min \{ BF[v][k-1], \min \{ BF[u][k-1] + c(u,v) \} \} \]

   len s.p. to v on \leq k edges = \min\{ len s.p. to v on \leq k' edges, \]
   \[ \min\{ \text{len s.p. to u on } \leq k' \text{ edges} + c(u,v) \} \] (by 1)
   \[ = \min\{ BF[v][k-1], \min\{ BF[u][k-1] + c(u,v) \} \} \] (by IH)
   \[ = BF[v][k] \] (by 2)
Testing for Cycles, Undirected Graphs

input: an undirected graph $G=(V,E)$
goal: determine whether $G$ is acyclic (i.e. is it a forest?)

algorithm: modified BFS (or DFS)
when considering the front element $v$
if any of $v$’s neighbours, $w$ say, is already marked then there was already an
$s$-$w$ path that didn’t go through $v$, and now there is a second one.
Putting these two paths together will form a cycle somewhere among them.

ex:
input: a directed graph $G=(V,E)$
goal: determine whether $G$ is acyclic (is it a DAG, a directed acyclic graph?)

Q: does BFS still work?
A: no. We’ll say there are cycles even when there are not.

To determine whether a graph is a DAG, we solve a related problem.

Def: A topological ordering/sort of a DAG is an ordering of its nodes such that if there is a directed path from $u$ to $v$ then $u$ precedes $v$ in the order.

Ex. Suppose nodes are courses, and you put an edge from course $A$ to course $B$ if $A$ is a prerequisite of $B$. Then a topological ordering is a valid ordering of the courses (so that you don’t violate any prerequisites constraints.)

Claim: a directed graph $G$ is a DAG if and only if $G$ has a topological ordering. Proof later. But this means that we can focus on solving topological ordering.
Topological Sort

input: a directed graph G=(V,E)
goal: find a topological ordering of G if it exists (o.w. output “doesn’t exist”)

TopologicalSort( G ) – output a topological ordering of G
• Initialize list L to empty
• while there is some vertex v with no in-edges
  • add v to L
  • remove v and all its out- and in-edges from G
• if vertices remain (but with in-edges) output NO GO
• otherwise, return L as the topological ordering of G

RT: keep list of in-deg 0 nodes. As you add v to L, you need to change the
degree of all of its neighbours, possibly adding them to the in-deg-0 list.
This is \[ \sum_v deg(v) \in O(m) \] and since the init is O(n), algorithm is O(m+n)

Ex.
Proof of Correctness

To prove that we correctly find a topological sort, we’ll prove the following:
Claim: G has a topological ordering if and only if G is a DAG.

Proof.

Suppose that G has a topological ordering O.
By way of contradiction, suppose that G is NOT a DAG.
Then G has a cycle w₁, w₂, ..., wk, w₁ in G.
But then b/c there is a w₁-wk path, w₁ must appear before wk in O.
But b/c there is a wk-w₁ path, wk must appear before w₁ in O.
These cannot both happen, thus we have a contradiction.
Thus our supposition that G is not a DAG cannot be true, and G is thus a DAG.

Suppose G is a DAG. We’ll show that the algorithm finds a topological sort.
In particular, we’ll show that if G is a DAG, there is always a in-deg 0 vertex v.
By way of contraction, suppose every node has in-deg at least 1. Construct C.
Start at any node u. It has an incoming edges (t,u), follow it backwards to t; t also has incoming edge, follow it, etc. But sooner or later have to repeat a node
Therefore, whenever $G$ is a DAG, there is a topological sort, and the algorithm finds it because there is always a vertex with in-deg 0.

And whenever $G$ is not a DAG, at some point we’ll be left with vertices of in-deg $\geq 1$.

So the topological sort algorithm works to (a) find a topological ordering of the vertices, if it exists, and (b) determine whether $G$ is a DAG.