Lecture 16:
• Transitive (edge) closure
• Transitive (edge) reduction

Announcements
• HW-3 due
• PS-5 out, due Tues 11/1
• HW-4 out, due Thur 11/3

Transitive Edge Closure

Basic Idea:
• Pre-compute “paths” (by adding edges) to speed-up reachability questions
• Becomes easy to determine if there is a path from \( u \) to \( v \)

An example of a trade-off of space (“path” edges) for time
Transitive Edge Closure (cont)

Definition:
- Let $G = (V, E)$ be a directed graph
- $G$’s transitive closure is $G^+ = (V, E^+)$ such that:
  $$E^+ = \{(u, v) \mid \text{there is a path from } u \text{ to } v \text{ in } G\}$$

Check In: How can we compute closure using DFS? ... “naive” approach
- Copy vertices from $G$ into $G^+$
- For each vertex $u$ in $G$
  - Find all vertices $v$ connected to $u$ using DFS (over $G$)
  - Add each edge $(u, v)$ to $E^+$

Check In: What is the cost?
- $O(V(V + E))$ since we compute DFS for each vertex $v \in V$

If $G$ is a DAG, can modify topological sort to compute the closure:
- When “dead end” $y$ reached from $x$, add $y$’s out-nodes to $x$’s in $E^+$
- Cost is $O(V + E)$ for topological sort plus $O(V^2)$ to add all the edges

IF $G$ has cycles, can leverage (strongly) connected components:
- Compute the meta-graph of $G$ … $O(V + E)$
- Compute the closure of the meta-graph … $C$ components
- Apply the resulting meta-graph closure to $G$ … (*)

Where $C$ is ideally much smaller than $V$ (and similarly for meta-graph edges)

(*) Notes:
- Add edges between all nodes in each strongly connected component
- If edge $(C_1, C_2)$ in meta-graph, add all edges $(x, y)$ for $x \in C_1$ and $y \in C_2$
Transitive (Edge) Reduction

**Goal:** Reduce number of edges while maintaining reachability
- if $v$ reachable from $u$ in $G$, then $v$ still reachable in reduced $G$
- transitive closure can decrease reachability query time (vs storage)
- transitive reduction can decrease graph storage (vs query time)

**Basic Definition:** Let $G = (V, E)$ be the original directed graph
- $G^- = (V, E^-)$ is a “minimal” directed graph ... no “extra” edges
- there is a path $i$ to $j$ in $G$ iff a path $i$ to $j$ is in $G^-$

Note that the closure of the reduced graph $G^-$ is the closure of the graph $G$
- closure(reduction($G$)) $\equiv$ closure($G$)

Transitive (Edge) Reduction

**Two variants:**
1. only include original edges in $G$
2. allow edges not in $G$ to be used

**Variant (1): Only include original edges in $G$**

$G^-$ a subgraph of $G$

**A Simple Approach:**
- Remove edges $(u, v)$ from $G$ that don’t change reachability from $u$ to $v$
- The minimal subgraph obtained is an irreducible kernel of $G$
- A subgraph with the fewest edges is a minimum equivalent graph of $G$

**Questions:**
- What is the cost?
- Does the algorithm always produce minimum equivalent subgraphs?
Transitive Reduction – Irreducible Kernel Approach

Finding irreducible kernel’s can be performed in $O(E(V + E))$ time

1. $E^- \leftarrow E$
2. for each $(u, v)$ in $E$
3. use shortest path on $E^- - (u, v)$ to check $v$ reachable from $u$
4. if $v$ is still reachable from $u$
5. delete $(u, v)$ from $E^-$

The result, however, might not be a minimum equivalent graph ...

Finding minimum equivalent graphs:
- NP-complete for connected components (thus, not practical in general)
- For DAGs, irreducible kernel finds minimum equivalent subgraphs

Transitive (Edge) Reduction

Variant (2): Relax subgraph requirement (allow edges not in $E$)

This variant is typically what is meant by “transitive reduction”

Note that a simple cycle contains unique vertices
- Versus a non-simple cycle
- Similar idea for simple vs non-simple paths

Basic Algorithm:
1. compute $G$’s strongly connected components $C_1, C_2, \ldots, C_n$
2. for each component $C_i$
3. create a simple cycle connecting each vertex in $C_i$
4. keep one edge per connected component pair as a “bridge”
5. compute the irreducible kernel over the meta-graph (a DAG)