Lecture 7 — January 18, 2010

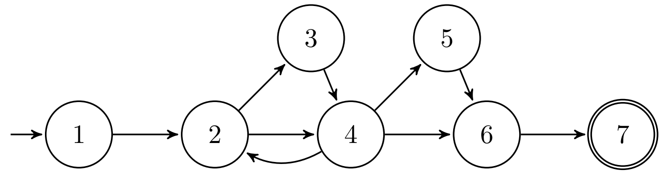
Patrick Lam *version 2*

Input: Directed graph G
Output: List of prime paths in G, primePaths

```
nonextendablePaths ← ∅;
primePaths ← ∅;
worklist ← all paths of length 0, i.e. nodes;
while worklist ≠ ∅ do
    p ← worklist.removeFirst();
    p_i ← initial node of p;
    p_f ← final node of p;
    wasExtended ← false;
    if p_f has no outgoing edges then
        nonextendablePaths += p;
    else
        foreach p'_f such that (p_f, p'_f) is an edge in G do
            if p'_f does not appear in p then
                worklist += p++pf';
                wasExtended ← true;
            else
                if p'_f = p_i then
                    nonextendablePaths += p++pf';
                    wasExtended ← true;
                end
            end
        end
        if not wasExtended then
            nonextendablePaths += p;
        end
    end
end
primePaths ← ∅;
foreach p ∈ nonextendablePaths do
    // (p could only be a suffix of a non-extendable path; I'd use a tree:)
    if p is not a proper subpath of any simple path then
        primePaths+ = p
    end
end
```

Bonus: What is the best (asymptotic) bound you can find for the number of prime paths? Is it tight?
Next problem: finding test paths to tour all prime paths (to achieve prime path coverage). Usually you need far fewer test paths than prime paths. Book doesn't present an algorithm, but suggests extending prime paths, starting from the longest prime paths.

Prime paths example.



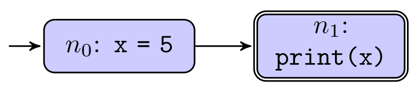
Data flow Criteria

So far we've seen structure-based criteria which imposed test requirements solely based on the nodes and edges of a graph. These criteria have been oblivious to the contents of the nodes.

However, programs mostly move data around, so it makes sense to propose some criteria based on the flow of data around a program. We'll be talking about *du*-pairs, which connect definitions and uses of variables.

du-pair
```
x = 5            // def(x)
⋮
print(x)         // use(x)
```

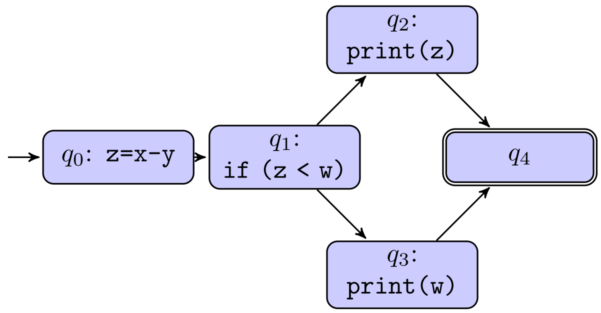
Let's look at some graphs.



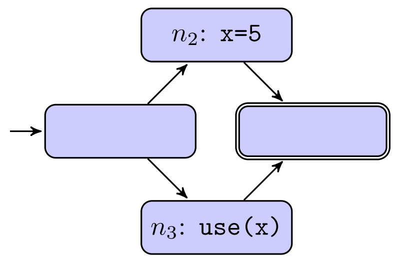
We write

Note that edges can also have defs and uses, for instance in a graph corresponding to a finite state machine. In that case, we could write:

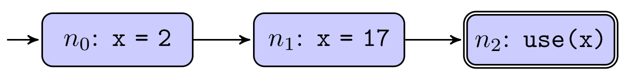
Here's another example.



A particular def d of variable x may (or may not) *reach* a particular use u. If a def may reach a particular use, then there exists a path from d to u which is free of redefinitions of x. In the following graph, the def at n_2 does not reach the use at n_3, since no path goes from n_2 to n_3.



Another example of a definition which does not reach:



We say that the definition at n_1 *kills* the definition at n_0, so that def(n_0) does not reach n_2. We are therefore looking for def-clear paths.

Definition 1 *A path p from ℓ_1 to ℓ_m is* def-clear *with respect to variable v if for every node n_k and every edge e_k on p from ℓ_1 to ℓ_m, where $k \neq 1$ and $k \neq m$, then v is not in def(n_k) or in def(e_k).*

That is, nothing on the path p from location ℓ_1 to location ℓ_m redefines v. (Locations are edges or nodes.)

Definition 2 *A def of v at ℓ_i reaches a use of v at ℓ_2 if there exists a def-clear path from ℓ_i to ℓ_j with respect to v.*

Quick poll: does the def at n_0 reach the use at n_5?