Graph Algorithms (16)

Michael L. Littman

October 27th, 1998

1 TOPOLOGICAL SORT

1.1 Scheduling

Last year, I went to a NASA conference and everyone was talking about planning (which is one of my areas of research), and scheduling (something I know less about).
Lots of applications (space-shuttle construction, science-mission planning, communications scheduling) used the basic machinery I'm about to describe.

1.2 Dressing

Data available (arrow means "must come before"):

- socks: can't have shoes already on
- shoes
- shirt: can't have tie or pants already on
- tie: can't have jacket already on
- jacket
- understuff: can't have pants already on
- pants: can't have shoes or jacket already on

Note: shirt $\rightarrow$ jacket, understuff $\rightarrow$ shoes already implied (transitive closure).
Represent restrictions as a graph.

1.3 Topological Sort

Fancy name. Makes it sound hard: label nodes with depth and then sort?

- Take a directed acyclic graph (DAG).
- Output a linear order of nodes such that all arrows are pointing to the right.

Essentially a linear schedule so there are no conflicts during dressing. Example.
Claim: A (finite) DAG always has at least one node with no incoming edges and one with no outgoing edges. Proof by contradiction.

> Assume there is no node with no incoming edges. That means that we can always go back one step in the graph. But, after doing this n times, we'll have to have repeated a node. The path traveled is a cycle, contradicting the assumption that we have an acyclic graph.

1.4 Data Structures and Notation

$G = (V, E)$ is a *graph*, with V as its *vertices* (or nodes), and E as its *edges* (or links).
By convention, $n = |V|$ and $m = |E|$. What's the largest m can be, assuming no edge appears more than once?

> $m \leq n^2$.

Often, we also have $m \geq n - 1$ (connected, undirected graph).
For $u \in V$, we write $Adj[u]$ to be the set of vertices to which u is directly connected (i.e., $v \in Adj[u]$ if and only if $(u, v) \in E$ (adjacency list).
Note: Matrices also useful for representing edges: $a_{u,v} = 1$ if $(u, v) \in E$ and 0 otherwise (adjacency matrix). If A is an adjacency matrix representing a graph G, which does A^T represent?

> A graph with all the edges reversed.

1.5 Topological Sort in Words

Each node gets a color.

- White nodes haven't been dealt with yet.
- Black nodes have been completely dealt with.
- Gray nodes are in progress.

Flow:

- Make all nodes white.

- Pick a node u to "visit" and make it gray.
- Pick a child of u to visit.
- Continue until we run out of white kids.
- Add the last visited node to a list and backtrack.

1.6 Algorithm

```
list<GraphVert*> * TopologicalSort( Graph G )
{
  for( unsigned int v=0; v < G.size(); v++ ) {

    G[v]->color = white;

    // make all edges G[u,v] = white; G[start,v] = white;
  }

  list<GraphVert*> * visited = new list<GraphVert*>();

  for( unsigned int v=0; v < G.size(); v++ ) {

    // G[start,v] = black;

    if( G[v]->color == white )
      Visit( v, G, visited );
  }

  return visited;
}

void Visit( int u, Graph G, list<GraphVert*> * visited )
{
  G[u]->color = gray;

  for( list<int>::iterator iter = G[u]->adj->begin();
       iter != G[u]->adj->end();
       iter++ )
  {
    int v = *iter;

    // G[u,v] = black;

    if( G[v]->color == white )
```

```
      Visit( v, G, visited );
  }

  visited->push_front( G[u] );
  G[u]->color = black;
}
```

1.7 Observations

Basically, a depth-first search of the graph (more on this later).
Also related to a post-order traversal of the appropriately defined depth-first tree.
Can also define the topological sort in terms of a breadth-first traversal. This is actually nicer if you want to output all possible topological sorts.

1.8 Correctness

Algorithm terminates because `Visit` only performs work on white nodes and instantly makes the visted node non-white. Thus, impossible to recurse forever.
Correctness proof by induction on the length of the constructed list.

- Inductive hypothesis: When a node is added to the list, all its children are already in the list. Therefore, all edges in the DAG point to the right.
- The first node added to the list will be one with no outgoing edges.
- Any node added to the list has no white kids. In fact, it has no gray kids, either (why?).

  All the gray nodes are ancestors of the current node. A gray kid implies a cycle.

- Therefore, all its kids are black and have already been added to the list.
- Therefore, it's ok to add it to the front of the list.

1.9 Running Time: Standard Analysis

Even though this is a recursive routine, it's not convenient to analyze it using recurrences. Side effects!
n is number of nodes (pieces of clothing, construction activities)
m is the number of edges (constraints).
Each node is visited only once (we never make a node white), and for each node we loop over all its outgoing edges once. Therefore, we visit each edge once.
Total work: $\Theta(n + m)$.

1.10 Potential Function Analysis

Intuition is that all the work happens in calls to `Visit` and there can't be too many such calls because nodes and edges are turning from white to gray to black.

- Operation is a call to `Visit`.
- $\Phi(G)$ is the total number of white edges in G:

$$\sum_{u \in V} \chi(G[\text{start}, u] = \text{WHITE}) + \sum_{v \in Adj[u]} \chi(G[u, v] = \text{WHITE}).$$

- Each time we enter `Visit`, an edge becomes black, decreasing the potential function by one and "paying for" the non-recursive work in the subroutine. How do we know that an edge isn't turned black twice?

  Because `visit` is only called when u (the "from" node) is gray.

- Maximum value of the potential function is one contribution from each edge and one from each node (because of "start"). So, running time is: $O(n + m)$.

1.11 Summary

Simple algorithm, very fast and practical. Comes up in all sorts of settings.

2 STRONGLY CONNECTED COMPONENTS

2.1 Airline Flight Planning

Given a route map of an airline, would like to know whether "you can't get there from here."

- Seattle (SEA): ORD
- Chicago (ORD): PIT, LAX, DFW
- Pittsburgh (PIT): PHL, ATL
- Philadelphia (PHL): PIT, RDU
- Los Angeles (LAX): SEA, DFW
- Dallas (DFW): ATL
- Atlanta (ATL): DFW, RDU

2.2 Definitions

A *path* in a graph $G = (V, E)$ is a sequence of nodes $u_1, \ldots, u_k$, such that for all $i \in [1, k-1]$, $(u_i, u_{i+1}) \in E$.
Graph G is *strongly connected* if, for every u and v in V, there is some path from u to v and some path from v to u.
A strongly connected *component* of a graph is a maximal subset of nodes (along with their associated edges) that is strongly connected. Nodes share a strongly connected component if they are interreachable.

2.3 More Formal Statement

Input is a graph $G = (V, E)$ (possibly with cycles) of n nodes and m directed edges.
Output is a set of components C, and a mapping $c : V \to C$ from nodes to components of interreachability, and edges D between components denoting one-way reachability.
Properties:

- If there is some path from u to v and from v to u, then $c(u) = c(v)$.
- If u and v are in different components and there is an edge from u to v, then $(c(u), c(v)) \in D$.
- The *component graph* $G' = (C, D)$ is a DAG.

How would you do it? Let's try an example.

2.4 Algorithm

```
vector<int> * StronglyConnectedComponents( Graph G )
{
  list<GraphVert*> * L = TopologicalSort( G );
  Graph G_t = *(transpose( G ));

  for( unsigned int v=0; v < G_t.size(); v++ )
    G_t[v]->color = white;

  vector<int> * components = new vector<int>( G.size() );
  int component = 1;

  for( list<GraphVert*>::iterator iter = L->begin();
       iter != L->end();
       iter++ )
  {
    list<GraphVert*> * l = new list<GraphVert*>();

    int v = (*iter)->num;
```

```
    if( G_t[v]->color == white )
    {
      Visit( v, G_t, l );

      for( list<GraphVert*>::iterator iter = l->begin();
   iter != l->end();
   iter++ )
(*components)[ (*iter)->num ] = component;

      component++;
    }
  }

  return components;
}
```

Here, `transpose(G)` is graph transpose or edge reversal.

2.5 Example

Note that component graph is given a topological numbering!
We left out the computation of the edges D of the component graph (HW).
Note what happens when we "topologically sort" a cyclic graph.

2.6 Running Time Analysis

This is really just two topological sorts: $\Theta(n + m)$.
The magic is in the fact that this works at all!
Terminology: forward reachable, backward reachable. If u and v are in the same strongly connected component, then v is both forward reachable and backward reachable from u (and vice versa).
Argue that G and G^T have the same strongly connected components.

> Two nodes share the same strongly connected component if and only if they are both forward and backward reachable from each other. This remains the same when we reverse edges.

What can we say about when to nodes in the same strongly connected component turn black (with respect to top-level calls to `Visit`)?

> They are visited in the same top-level call.

2.7 Analysis

Observations: If u is gray when v is blackened, then v is forward reachable from u. If u precedes v in L, then u was either white or gray when v was blackened. If white, then v is not forward reachable from u.
Let u be the first node in L (after the first call to `topological sort`).

- It is only forward reachable by nodes in its same strongly connected component. Why? Picture.
- So, any node v that is backward reachable fom u shares u's strongly connected component.
- Also, any v in u's strongly connected component is backward reachable from u (of course).
- `Visit(v, G_t, l)` returns all (and only) nodes backward reachable from u.
- So, we correctly identify (and blacken) component 1.
- Remove all the component 1 nodes from L and G^T.
- Repeat!

2.8 Summary

Like topological sort, fast and relatively simple.
Very useful for:

- analyzing graphs,
- breaking up computations.