More Graph Algorithms

Henry Z. Lo

July 16, 2014

1 Transitive Closure

Problem 1. *Given a list of train tickets, each of which contain a source and destination airport, construct another graph where an edge between a and b denotes that b is reachable from a.*

By reachable, we mean that there is some path from a to b. As an example, suppose we are given the following list:

Source	Destination
San Antonio	Dallas
Dallas	San Antonio, Austin
Austin	Dallas, Houston
Houston	Austin
Boston	New York
New York	Philadelphia, Boston
Philadelphia	Boston

This is a graph (figure 1) in the form of an adjacency list. We model the problem as an undirected graph, though the solution is similar for directed graphs. The graph and its transitive closure can be visualized in figure 2.

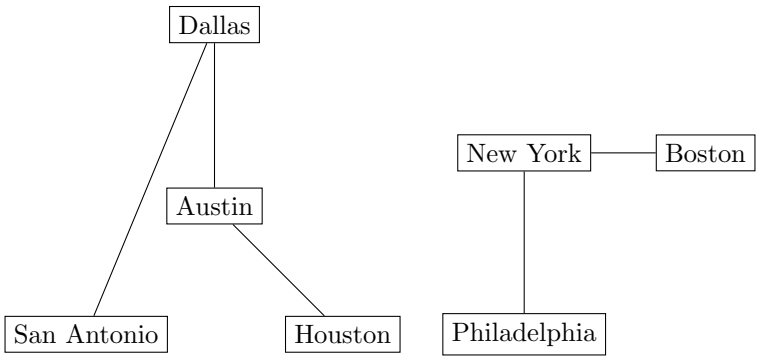


Figure 1: Undirected graph representing train routes.

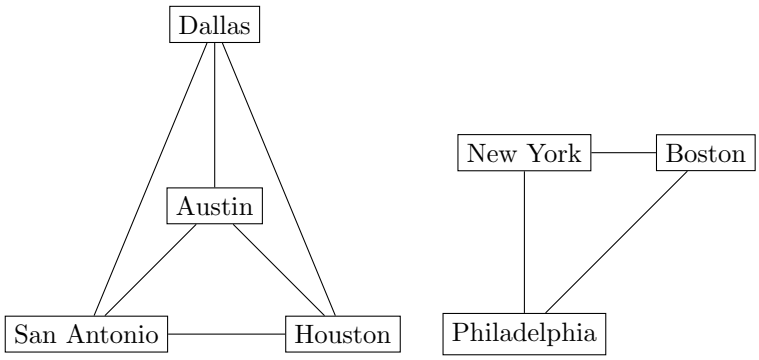


Figure 2: Transitive closure for figure 1.

1.1 Algorithm

The idea behind the Warshall algorithm is to add edges between nodes reachable by one intermediate node. Once we do this, another pair of nodes will be reachable by one intermediate node. By doing this over and over, we can connect all nodes which have paths to each other.

The algorithm:

```
function warshall():
  for k from 1 to V
    for i from 1 to V
      for j from 1 to V
        if edge exists between i,k and edge exists between k,j
          add edge i,j
```

Why does this work? In the first iteration of the outer loop, the algorithm connects all nodes which have node 1 as an intermediate node. Next, it connects all nodes which have node 1 or 2 as intermediate nodes. Eventually, it connects all nodes which have nodes 1 to V as intermediate nodes.

The algorithm here is a triple for loop, so runtime is simply $O(V^3)$.

1.2 Example

If we go in the order of the adjacency list, then the Warshall algorithm will do the following in each iteration of the outer loop:

1. San Antonio is not an intermediate node for any pair, so nothing happens.
2. Dallas is an intermediate between Austin and San Antonio, so we add an edge between them.
3. Due to the last step, Austin bridges San Antonio and Houston, so we add an edge between those two. Also add an edge between Dallas and Houston.

4. Houston bridges nodes which already have an edge, so we do nothing.

5. Boston does not bridge nodes, so do nothing.

6. New York bridges Boston and Philadelphia, add edge.

7. Philadelphia adds nothing new.

In short, the following edges are added to the graph in figure 1 in order:

Austin → San Antonio
San Antonio → Houston
Dallas → Houston
Boston → Philadelphia

2 Topological Sorting

Problem 2. *Given a list of classes and their prerequisites, order the classes based on the earliest semester (first, second, third) that you can take them.*

As an example, consider the computer science degree requirements at the UMass Boston:

Course	Prerequisites
CS110	
CS285	
MATH140	
CS240	CS110
CS210	CS110
MATH260	MATH140
CS310	CS210, CS240
CS320	CS110, MATH260
CS341	CS240
CS410	CS310, CS320
CS420	CS320
CS444	CS310, CS341
CS450	CS310, CS320
CS451	CS310, CS420

We can think of courses as nodes, and a course's prerequisites as directed edges pointing to that course. Thus, we have a directed acyclic graph (DAG).

Given this graph, we want to first find the set of nodes which have no dependencies, then the nodes which only have those dependencies. Putting the nodes in this order gives us a *topological sort*. This ordering reflects the "levels" in the graph shown in figure 3.

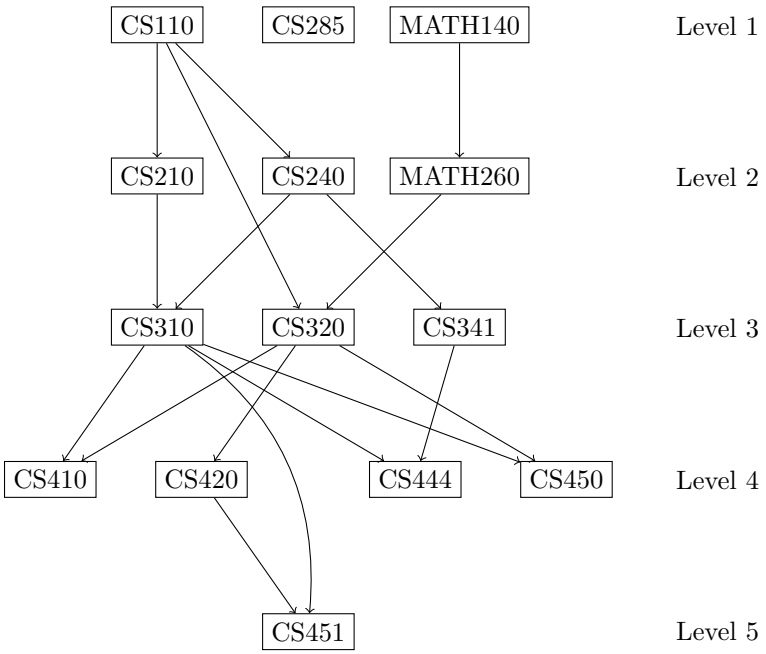


Figure 3: Directed acyclic graph representing the class requirements for a computer science major at UMass Boston.

2.1 Algorithm

The idea behind the algorithm is to first find the nodes without incoming edges. In a DAG, there is always at least one "ancestor" node. These ancestor nodes belong to the same "level". Those nodes which are only dependent on the ancestor nodes form the next level. The third level of nodes depends only on the first and second level, and so on.

The key insight is that when we remove the ancestor nodes, the new ancestors are the next topological level. We can repeatedly find and remove ancestors until we go through all nodes. If we run out of ancestor nodes without going through all nodes, then there must have been a cycle.

The algorithm for topological sort:

```
function toposort():
  out_q = empty queue
  node_q = queue of nodes with nodes with no parents
  while node_q is not empty
    remove node n1 from node_q
    queue n1 onto out_q
    for each child node n2
      remove edge from n1 to n2
      if n2 has no more parents, add to node_q
  return out_q
```

2.2 Analysis

We assume that getting a node's parents or children can be done in constant time. This is possible if we use two adjacency lists: one to map from the node to its children, and one to map to its parents. Line 2 then takes $O(V)$ time.

The code within the while loop also runs in $O(V)$ time.

The for loop runs once for each child of the current node. In total, it runs exactly once for each edge, since iterating through all children of all nodes is equivalent to traversing all edges. Thus, even though it is within another loop, it must be noted that it runs a *total* of $O(E)$ times.

In total, this topological sorting algorithm runs in $O(V+E)$, or linear, time.

2.3 Example

Going back to the problem of finding the order for classes in figure 3, the first few iterations of the topological sorting algorithm will look like this:

1. Initially, the node queue will contain CS110, CS285, and MATH140.
2. We take out CS110 and its edges, and put CS110 in the output queue.
3. Then we check to see if CS110's children (CS210, CS240, CS240) have any parents left. CS210 and CS240 do not, so they are put onto the node queue.

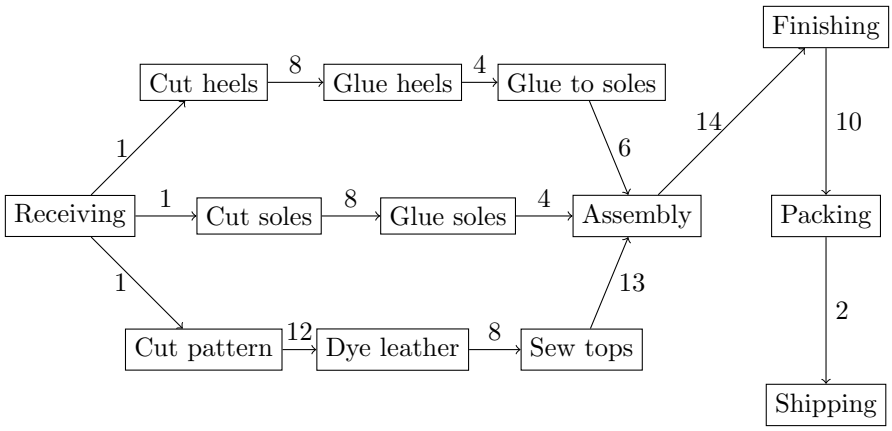


Figure 4: PERT chart for shoe construction workflow. Edge weights represent days needed for the given task.

4. Then we take out CS285. It has no edges, so we continue.
5. We take out MATH140 from the node queue. Its edges are removed, leaving MATH260 without parents. We put MATH260 onto our topological ordering.
6. Now we have CS210. We remove its edges. CS310, its child, still has a parent, so we don't do anything else.
7. We remove CS240 and its nodes. This orphans CS310 and CS341, which we then put on the queue.

We continue until done. The resulting topological ordering will look like:

CS110 → CS285 → MATH140 → CS210 → CS240 → MATH260 → CS310 → CS320 → CS341 → CS410 → CS420 → CS444 → CS450 → CS451

3 Longest Path

Problem 3. *Given a set of tasks, their completion times, and their dependencies on other tasks, compute the total time it takes to finish all tasks.*

We can model this type of task set with a PERT chart (see figure 4), a graphical model which contains tasks as nodes, and dependencies as edges. Edge weights represent the time it takes to complete the parent task.

In PERT charts we are interested in the critical path, the longest path in the graph. This path in figure 4 is:

Receiving → Cut pattern → Dye leather → Sew tops → Assembly → Finishing → Packing → Shipping

The critical path here has a length of 60 days, which is the time it takes to complete all tasks.

3.1 Longest path problem

In general, the longest path problem is easy to compute, but not fast. The obvious solution is to use a breadth-first or depth-first search.

These traversal algorithms run in linear time for finding a certain node, because we only visit each node exactly once. We do not traverse edges already traversed.

However, here we are looking for a *path*. Even though we should not encounter a node twice in the same path, we can encounter the node multiple times in different paths. This makes the corresponding *path* tree much larger than the BFS/DFS tree.

See figure 5 for an example. Note that to solve the longest path problem, we actually need to initialize the path from every single node. The number of possible paths grows factorially with the number of nodes. Hence, unlike the $O(V)$ BFS/DFS tree, the path tree is $O(V!)$.

There is actually no known solution for finding the longest path in a general graph which is reasonably fast (in polynomial time). We will discuss this in the next set of notes. However, for directed acyclic graphs, there is an extremely efficient algorithm.

3.2 Longest path in a DAG

Despite the general difficulty of finding the longest path, we can see very quickly in figure 4 that there are only 3 paths to consider:

- receiving $\rightarrow$ cut pattern $\rightarrow \ldots \rightarrow$ shipping
- receiving $\rightarrow$ cut heels $\rightarrow \ldots \rightarrow$ shipping
- receiving $\rightarrow$ cut soles $\rightarrow \ldots \rightarrow$ shipping.

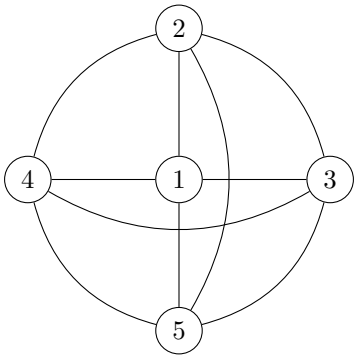
We only consider these three because we know that any longest path must start at a parentless node and end at a childless node. If it didn't, then we can always make a longer path by including a parent or child.

For example, cut soles $\rightarrow \ldots \rightarrow$ packing could not possibly be a longest path, because we can add receiving to the front of this path, and shipping to the end to make it longer.

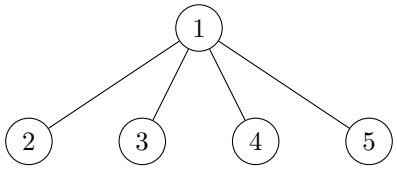
This suggests that we can use topological ordering to help find the longest path.

3.3 Algorithm

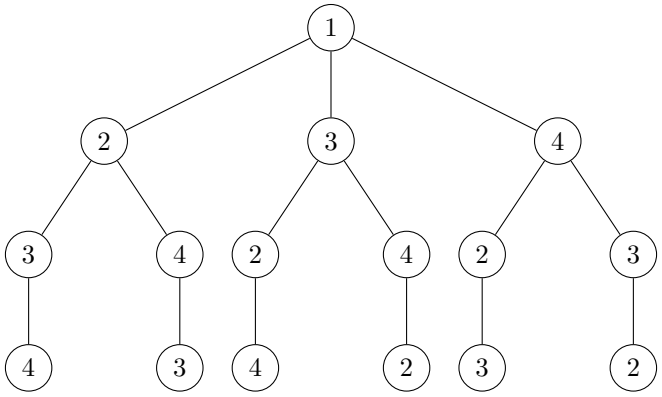
The idea behind finding the longest path in a DAG is first topologically sort all nodes, then iterate through this list. For each node, only consider the *longest* path from the ancestor nodes, and use that as its current value. Note that if



(a) Arbitrary undirected graph.



(b) BFS tree for this graph, starting from 1.



(c) Path tree, starting from 1.

Figure 5: BFS and path tree for a complete graph, starting at node 1.

we perform computations in this order, by the time we process a node, we have already processed its parents.

The algorithm:

```
function longestDAGpath():
  q = toposort(nodes)
  m = map from nodes to integer
  for node in q
    if node has no parents
      m[node] = 0
    else
      for each parent p of node
        m[node] = max(m[node], m[p] + edge value of p->node)
  len = infinity
  for n in set of childless nodes
    len = min(len, m[n])
  return len
```

Compare this with Dijkstra's algorithm.

Topological sorting, as mentioned, takes $O(V + E)$. Just as in topological sorting, the outer loop in this algorithm occurs $O(V)$ times, and the inner loop occurs exactly $O(E)$ times in total. Finding childless nodes and iterating through them is also $O(V)$. Hence, the entire algorithm is linear, $O(V + E)$.

3.4 Example

In the PERT chart example, `m` changes as follows:

1. `m[receiving]` $= 0$
2. `m[cut heels]` $= 1$
3. `m[cut soles]` $= 1$
4. `m[cut pattern]` $= 1$
5. `m[glue heels]` $= 8 + 1 = 9$
6. `m[glue soles]` $= 8 + 1 = 9$
7. `m[dye leather]` $= 12 + 1 = 13$
8. `m[glue to soles]` $= 4 + 9 = 13$
9. `m[sew tops]` $= 8 + 13 = 21$
10. `m[assembly]` $= max(13 + 6, 9 + 4, 21 + 13) = 34$

After step 10, everything else is pretty linear. Note that we only consider the longest path to get to assembly. Everything after assembly uses this time. Thus, by the time we get to shipping, we will be using the longest path length to get to shipping, which was through the cut patterns route.