Topological Sorting Acyclic Directed Graphs

April 06, 2012 / Home (/)

The actual code written when developing a typical web application, ranks pretty lowly on the complexity scale, when compared to the rest of the software engineering industry. Of course, web development has its own set of interesting complexities around architecture and scaling, but the application code itself is relatively simple. Glue one library to another, update a row in a database, print some more rows out, send out an email, and so on.

As a web developer, I always enjoy the chance to dig deeper into more formal data structures and algorithms when the opportunity arises. At Fairfax (http://fairfax.com.au) I've recently had two distinct situations come up where I've needed to work through sets of data, with each of the items in the set containing references to other items that they depends on. The desired outcome was to sort the items so that I could process each item, knowing that each time I reached an item with dependencies, those dependent items would already have been processed.

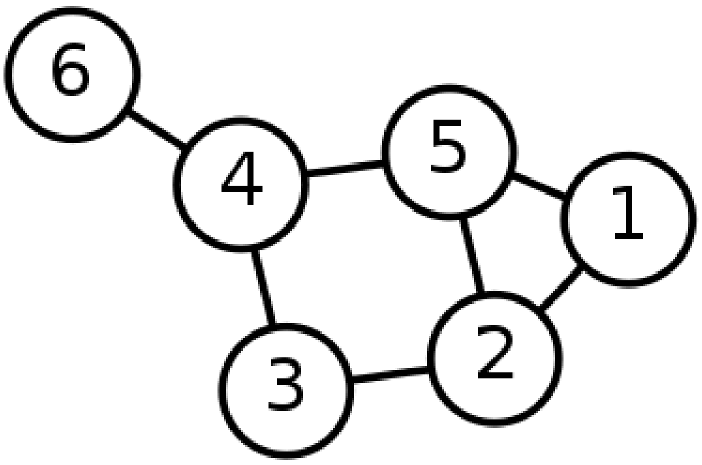
The first scenario was one with serialised Django models (https://docs.djangoproject.com/en/dev/topics/db/models/). I had a number of model instances that would be instantiated and serialised, with the aim of persisting them to the database at a later point. These models contained relationships with other models that were also being serialised. Temporary primary keys were used in order to build their relationships, allowing all of the models to then be serialised with their relationships intact.

The problem then arose after deserialising these instances, and persisting them to the database. Each model with a foreign key to another couldn't be processed, until the model it was related to had been persisted to the database first, as real primary keys are required for foreign key relationships to be defined. My first solution was a quick hack that manually sorted the objects correctly, knowing in advance which classes of models I was dealing with. This of course meant the code couldn't be reused in a general manner, and as more classes of models were added to the mix down the track, this would need to be addressed.

The second case like this came up in an entirely different situation. At Fairfax we're building an impressive distributed system built on many RESTful APIs (http://en.wikipedia.org/wiki/Representational_state_transfer). Each of these APIs contains its own schema, and we set out to build a tool that could introspect these schemas, and generate client objects for interacting with them. These schemas also contain relationships between resources, and these resource relationships are also modelled in our client objects. Again this situation was one where I needed to process an unordered set of resources in the correct order, so that their relationships could correctly reference resources already handled.

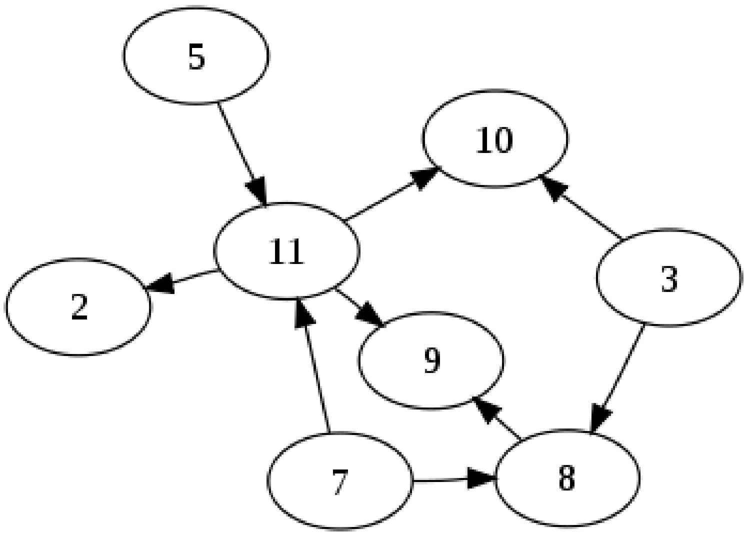
Shyly fool my bitten shame twice, or something like that. When this problem came around the second time, I didn't have the luxury of knowing up front what the exact classes of data I'd be dealing with were, as they were due to change quite quickly as we iterated. I then set out to implement the correct solution for the problem, that could be applied to both of these situations.

A quick Google search for "dependency resolution" brought me to the Wikipedia page for topological sorting (http://en.wikipedia.org/wiki/Topological_sorting), which was the solution I was looking for. In both cases, we have graphs (http://en.wikipedia.org/wiki/Graph_(mathematics)). Think of a graph as points on a map called nodes, with each node connected by lines called edges, to other nodes in the graph.



A simple graph

A directed graph (http://en.wikipedia.org/wiki/Directed_graph) is one where each of the edges contain a direction, so each of the lines contain an arrow pointing one way or the other. An acyclic directed graph (http://en.wikipedia.org/wiki/Directed_acyclic_graph) is where each of the edges only point in one direction, so that it's not possible to follow the edges from one node to another, returning back to the original node. If this last condition is not satisfied, then the graph is said to contain directed cycles (http://en.wikipedia.org/wiki/Directed_cycle), that is, a path can be followed from one node to others, and back to the original node again.



An acyclic directed graph

In both scenarios I faced, my graph structure was different from those commonly found in examples of topological sorting. I had the outgoing edges defined, rather than incoming edges. The gist of the topological sort I needed, is to repeatedly go through all of the nodes in the graph, moving each of the nodes that has all of its edges resolved, onto a sequence that forms our sorted graph. A node has all of its edges resolved and can be moved, once all the nodes its edges point to, have been moved from the unsorted graph onto the sorted one.

Consider the graph above from left to right, as pairs of nodes and their outgoing edges:

```
graph_unsorted = [(2, []),
                  (5, [11]),
                  (11, [2, 9, 10]),
                  (7, [11, 8]),
                  (9, []),
                  (10, []),
                  (8, [9]),
                  (3, [10, 8])]
```

Our expected output from a topological sort function would be as follows, with no nodes containing edges pointing to nodes before themselves:

```
>>> from pprint import pprint
>>> pprint(topolgical_sort(graph_unsorted))
[(2, []),
 (9, []),
 (10, []),
 (11, [2, 9, 10]),
 (5, [11]),
 (8, [9]),
 (3, [10, 8]),
 (7, [11, 8])]
```

Note that the ordering need not be precisely the same each time. In the result above, node 9 could come before node 2, as neither of these contain edges, so they equally belong first in the sorted graph.

Here’s an implementation of my topological sort in Python:

```python
def topolgical_sort(graph_unsorted):
    """
    Repeatedly go through all of the nodes in the graph, moving each of
    the nodes that has all its edges resolved, onto a sequence that
    forms our sorted graph. A node has all of its edges resolved and
    can be moved once all the nodes its edges point to, have been moved
    from the unsorted graph onto the sorted one.
    """

    # This is the list we'll return, that stores each node/edges pair
    # in topological order.
    graph_sorted = []

    # Convert the unsorted graph into a hash table. This gives us
    # constant-time lookup for checking if edges are unresolved, and
    # for removing nodes from the unsorted graph.
    graph_unsorted = dict(graph_unsorted)

    # Run until the unsorted graph is empty.
    while graph_unsorted:

        # Go through each of the node/edges pairs in the unsorted
        # graph. If a set of edges doesn't contain any nodes that
        # haven't been resolved, that is, that are still in the
        # unsorted graph, remove the pair from the unsorted graph,
        # and append it to the sorted graph. Note here that by using
        # using the items() method for iterating, a copy of the
        # unsorted graph is used, allowing us to modify the unsorted
        # graph as we move through it. We also keep a flag for
        # checking that that graph is acyclic, which is true if any
        # nodes are resolved during each pass through the graph. If
        # not, we need to bail out as the graph therefore can't be
        # sorted.
        acyclic = False
        for node, edges in graph_unsorted.items():
            for edge in edges:
                if edge in graph_unsorted:
                    break
            else:
                acyclic = True
                del graph_unsorted[node]
                graph_sorted.append((node, edges))

        if not acyclic:
            # Uh oh, we've passed through all the unsorted nodes and
            # weren't able to resolve any of them, which means there
            # are nodes with cyclic edges that will never be resolved,
            # so we bail out with an error.
            raise RuntimeError("A cyclic dependency occurred")

    return graph_sorted
```

I imagine some other uses for topological sorting would be task queues, where certain tasks are dependent on other tasks being completed first. It could also be used by package managers that install software libraries, to ensure each library has its dependencies met before it’s installed.

Home (/)

By Stephen McDonald / steve@jupo.org (mailto:steve@jupo.org) / LinkedIn (http://www.linkedin.com/in/stephenmcd) / GitHub (http://github.com/stephenmcd) / Twitter (http://twitter.com/stephen_mcd)