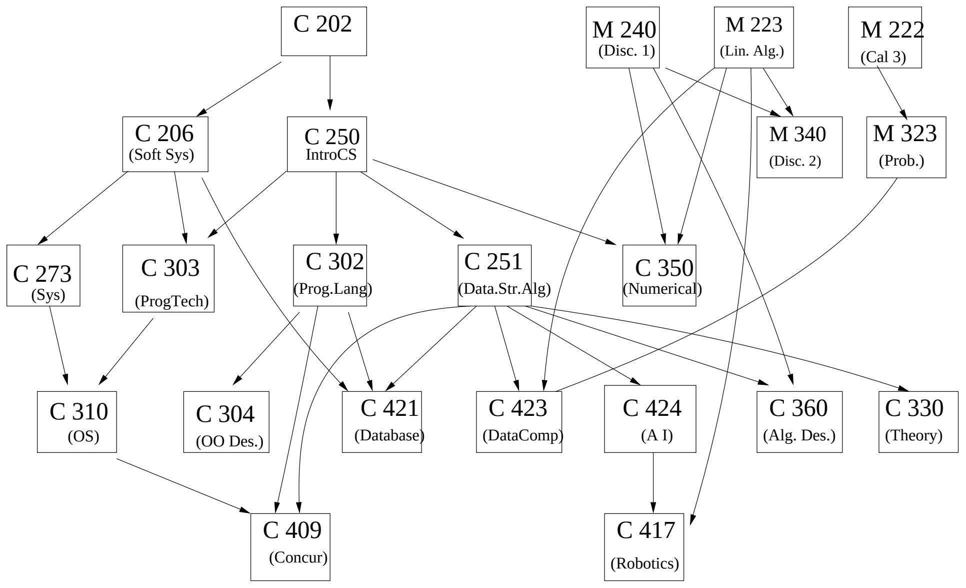
Directed acyclic graphs

Let's suppose we have a directed graph with no cycles in it. This is called a *directed acyclic graph* or DAG.

An example of a DAG is the prerequisite structure of courses that you can take as an undergraduate computer science student at McGill. [1] There are also 500 level courses that you can take, but I'll just keep the graph simple.



A DAG must have at least one vertex that has no incoming edges. That is, there must be at least one vertex with indegree 0. Why? Suppose it were not true, namely you have some DAG and all vertices have degree greater than 0. Take any vertex `w` in the DAG. Since it has indegree greater than zero, there must be an incoming edge `(v,w)`. Similarly, there must be an incoming edge to `v`, call it `(u,v)`, and so on. But since there are only finitely many vertices in the graph (we are only talking about finite graphs), eventually we will have an edge whose first vertex is one we have seen before. But this defines a cycle. But DAGs have no cycles. Thus, our assumption that we have a DAG with no vertices with in-degree 0 is wrong. This completes the proof (by contradiction).

Topological sorting (of a DAG)

In the above example, let's say you are a student in computer science and you need to choose a sequence of courses that fits into your time schedule and fits in with the other courses (electives etc) that you wish to take. Suppose that you can only take one of the above courses per semester, but let's say you want to take all those course. You therefore need to find an ordering of the courses that is consistent with the prequisites i.e. edges in the graph. This is the problem of *topological sorting* (of a DAG).

[1] While I have your attention, let me warn you that, although COMP 240 is not officially a prerequisite for COMP 251, it should be. I *strongly* recommend that you take MATH 240 first.

Consider the problem of choosing an ordering of all the vertices in a DAG such that, for every edge (u, v) in the DAG, the vertex u appears before the vertex v in the ordering. By ordering, I mean a labelling of the vertices $v_1, v_2, \ldots, v_{|V|}$, such that $v_i < v_j$ if and only if $i < j$. Another way to think about this constraint is in terms of the adjacency matrix. A *topological sorting* of the vertices in the graph labels the vertices $v_1, v_2, \ldots, v_{|V|}$, such that all 1's in the adjacency matrix are to the right of the main diagonal.

```
          01234   TO
          -----
      0 | 00010
      1 | 00100
      2 | 00000
FROM  3 | 00001
      4 | 00000
```

You might expect that you can solve the problem by using either a depth first or breadth first search. However, one needs to be careful. We cannot just define an ordering by the order of nodes traversed either in breadth first or depth first since, since that doesn't always work. For example, take a DAG with three vertices `a,b,c` and three edges `(a,b), (a,c), (c,b)`. If you do either a breadth-first or depth-first search, then the vertices are visited in order `(a,b,c)`. That is not a valid *topological sorting* since there is an edge `(c,b)`. The only valid topological ordering is `(a,b,c)`.

Below is an algorithm that uses depth first search,[2] but we choose the ordering in a slightly different way than by the order of nodes visited, namely we keep track of the order in which vertices are popped from the stack.

```
TopologicalSort(){                //  of a DAG  G = (V,E)
  initialize empty list
  for each vertex v in graph
    w.visited == false
  for each vertex v in DAG with (v.inDegree == 0) {
    initialize empty stack s
    s.push(v)
    v.visited = true
    while ( !s.empty )
      if ((w.visited == false) for some w in adjList(s.top)){
        select first w in adjList(s.top) such that w.visited == false
        w.visited = true
        s.push(w)}
      }
      else
        list.addfirst( s.pop() );
      }
    }
}
```

[2]recall non-recursive depth-first-search from lecture 32 – the algorithm there was modified on April 6 so be sure you have the most up-to-date version.

Correctness of algorithm

For the algorithm to be correct, we need the following: If `(v,w)` $\in E$, then `v` $<$ `w`, that is, `v` comes before `w` in the `list` that is produced by the algorithm.

How do we prove that the algorithm is correct. We break up our analysis into various cases:

1. The vertex `w` is pushed onto the stack while `v` is currently in the stack (`v` may not be on top).

   In this case, we are guarenteed that `v < w`, since `w` will be pushed from the stack first, and hence will enter the (front of the) list first. i.e. Later, when `v` in added to the front of the list, it will be placed in front of `w`.

2. The vertex `v` is pushed onto the stack while `w` is currently in the stack. This cannot happen, since the path from `w` to `v` defined by the stack would define a cycle (if we were to add an edge `(v,w)` to that path. But the graph is a DAG so no such cycles can exist.

   (In particular, notice that if `(v,w)` $\in E$, then there cannot be an edge `(w,v)` in the DAG, because that we would imply that the graph has a cycle.)

3. When `w` is pushed onto the stack, `v` has already been pushed and popped from the stack.

   This is also impossible. Since there is an edge `(v,w)`, the vertex `v` will not be popped until `w` is visited (pushed).

4. `v` is push on the stack after `w` has been pushed and popped from the stack.

   This is fine since, like case 1, `w` will be pushed from the stack first, and hence will enter the (front of the) list first. Later, when `v` in added to the front of the list, it will be put in front of `w`.