COMP 3760

Algorithm Analysis and Design

Lesson 15: Topological Sort

Rob Neilson

rneilson@bcit.ca

Homework and Reading

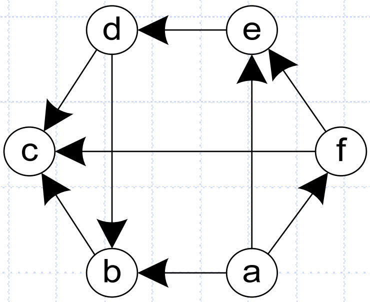
- Due at start of lab in week of Nov 10-14
  - Read chapter 5.3
  - Answer questions 1 and 5 (page 176)
- Note: no lab for set 3B or 3F next week; I will set up a webct assignment so that you can submit your homework by start of your regularly scheduled lab.

This Lesson's Agenda

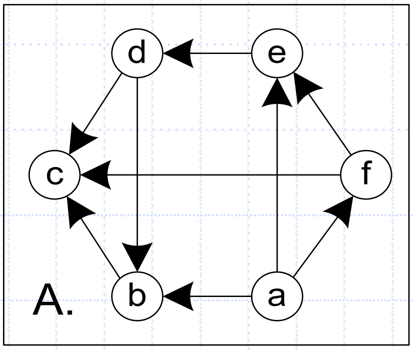
- Topological Sort

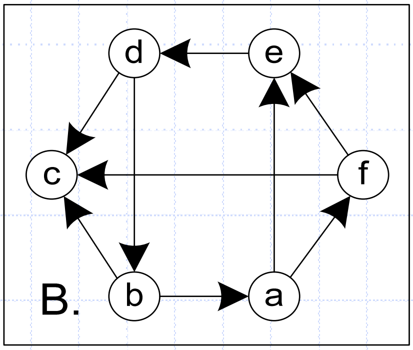
DAG's (Directed Acyclic Graphs)

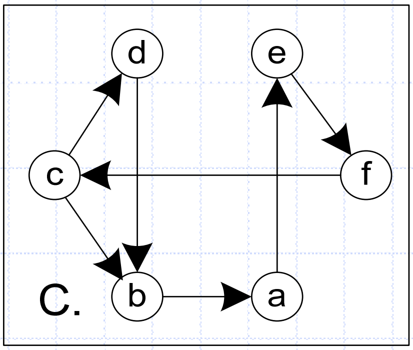
- recall that a ***directed graph*** is a graph that uses arrows to show direction
  - for example:

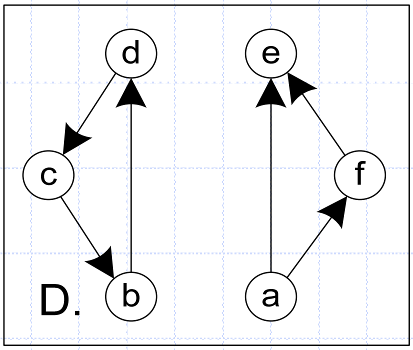


- a ***directed acyclic graph***, aka ***DAG***, is a directed graph that contains no cycles
  - which of the following are DAG's?









Topological Sort

Problem: given a set of inter-dependent items, find a linear ordering that satisfies all dependencies

- ie: if an arbitrary item i_x depends on another item i_y, then i_y appears before i_x in the sorted order

- eg: work tasks: you are trying to schedule n tasks.
- The tasks are not independent and the execution of one task is only possible if other tasks have already been completed.

- *Input might look like this ….*

6 ← (there are 6 tasks in total)
2 1 ← (task 2 must be done before task 1)
4 3 ← (task 4 must be done before task 3)
1 4 ← (task 1 must be done before task 4)
5 2 ← (task 5 must be done before task 2)

one possible solution (topologically sorted order):

- 5 2 1 4 3 6

Topo Sort Algo 1: DFS

- to obtain a topological sort order for a set of items:

  1. represent the items as a directed graph G such that:
     a. vertices are the items that are interdependent
     b. edges are the dependencies (constraints) between items
        - an edge from v to w (eg: $v \rightarrow w$) means that v is *dependent* on w ... ie ... v *must be done before* w
  2. apply the DFS algorithm to G
  3. the *order in which vertices become dead ends* gives the reverse topological sort order

  recall: the DFS implementation uses a stack
  - the "order in which vertices become dead ends" is given by the "order in which vertices are popped off the stack"

  Note: Topological Sort produces no solution if the graph contains a cycle

Example 2: Work Tasks (from lab 2)

2 1 ← (2 before 1)	4 3 ← (4 before 3)
1 4 ← (1 before 4)	5 2 ← (5 before 2)
2 3 ← (2 before 3)	5 1 ← (5 before 1)
5 6 ← (5 before 6)	6 3 ← (6 before 3)
2 4 ← (2 before 4)	6 2 ← (6 before 2)

Step 1: draw the graph (and verify it is a DAG)

Step 2: apply DFS

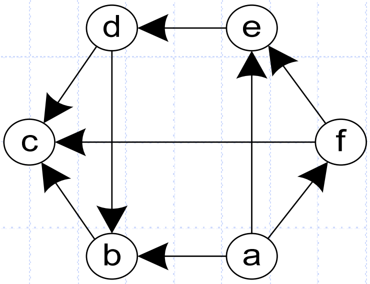
Step 3: find the order vertices were removed from stack, and reverse this order to get topological sort order

Example 2: Different Work Tasks

- Assume you have a set of 6 tasks (a, b, c, d, e, f) with the following dependencies:
  - a must be done before b, e, f
  - b must be done before c
  - d must be done before b and c
  - e must be done before d
  - f must be done before c and e

Example 2 (cont)

- we can draw a directed graph showing these dependencies as follows:



- and apply DFS …

Cycles and DAGs

- Topo only works on Directed Acyclic Graphs (DAGs)
- We need to check if a given problem has a feasible solution (ie: it is a DAG)

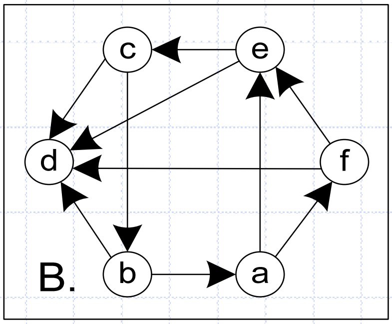
How to tell if there is a cycle?

Define:

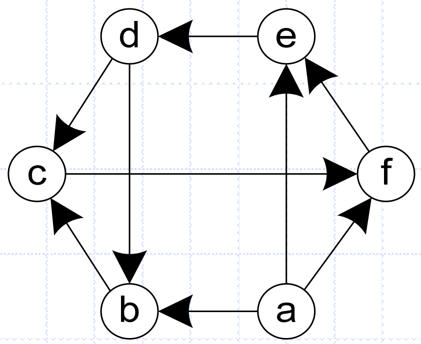
- dfs tree edge : edge in the dfs tree
- dfs back edge : edge from v to ancestor of v
- dfs forward edge : edge from v to descendent of v other than child
- dfs cross edge : any other type of edge not in the dfs tree

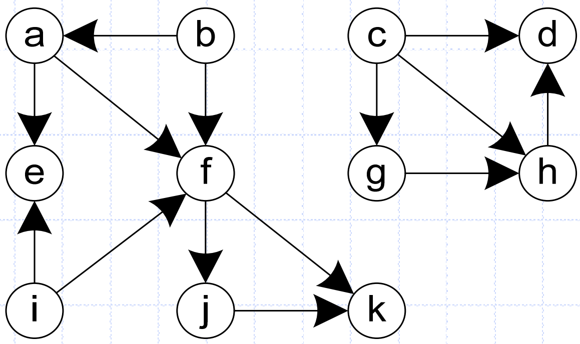
Then:

- if the DFS tree (forest) contains a back edge, the graph is not a DAG, and no feasible solution exists



More Examples





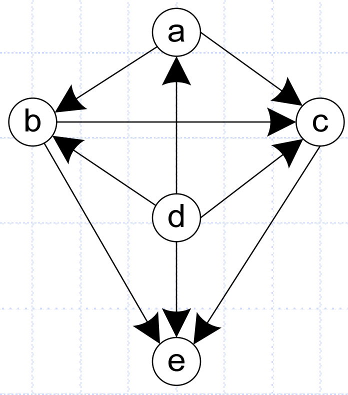
Topo Sort Algo 2: Decrease by One

Observe:

- if a vertex v in the dependency graph G has no incoming arrows (ie: in-degree(v) == 0), then v does not have any dependencies
- it follows that any v that does not have dependencies is a candidate to be visited next in topographical order

A Decrease-by-One approach:

- identify a $v \in V$ that has in-degree = 0
- delete v and all of its edges
- when all vertices have been deleted, the topo sort order is given by the order of deletion
- if there are $v \in V$, but no v has in-degree = 0, the graph G is not a DAG (no feasible solution exists)



Topo 2: Dec by One: More Details

More detailed algorithm:

- need a set to store the candidate v's (in-degree = 0)
  - I will use a stack. Any set will do.
- need an ordered list to store the delete order
  - I will use a queue. Any ordered list will do.

Then the algorithm is:

```
topo(G)
   create an empty queue Q
   create stack S
   add all v with inDegree=0 to S
   while S is not empty
      v ← S.pop()
      add v to Q
      for each vertex w adjacent to v
         remove edge (v,w) from G
         if w has inDegree=0
            push w on S
      remove vertex v from G
   if there are vertices remaining in G
      no feasible solution exists
   else
      solution is in Q
```

Examples: with S and Q