Graphs

See Chapter 14 of the text.

So far this semester we have talked about many specific data structures -- lists, queues, stacks, binary trees, heaps, etc.

We will now look at graphs, which are much more general.

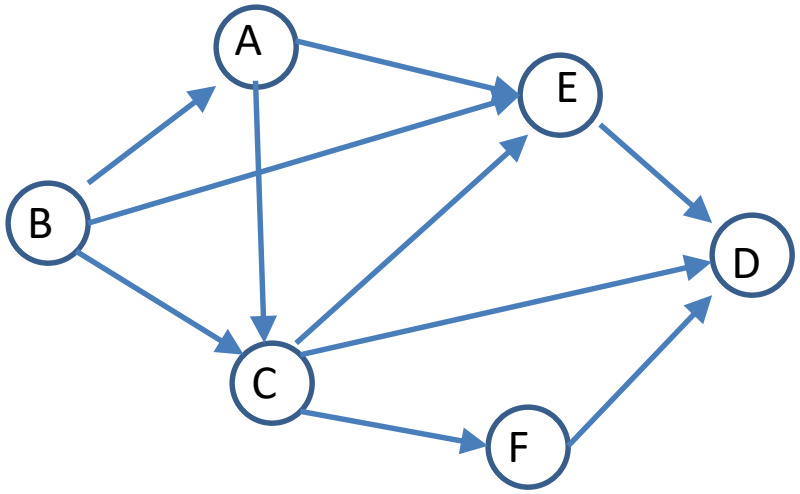
Many, many problems can be solved by creating a graph that represents the problem, processing the graph, and thereby creating a solution for the problem.

For example, suppose we are writing a scheduler for a collection of processes. Some processes get input from others. If process B gets input from A, we need to schedule A before we schedule B. Our task is to find an ordering of the processes so that we complete each before its output is needed by any other process.

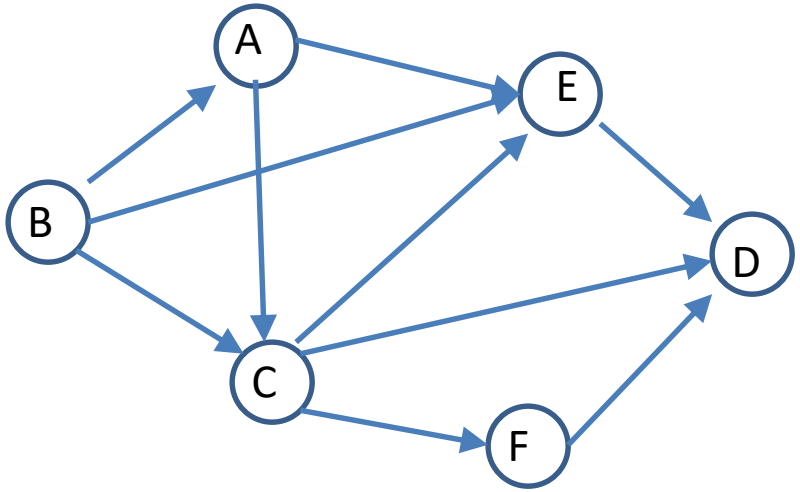
Here is a way to solve this problem:

First, make a graph where each process is represented by a node of the graph. Add an edge in the graph from node X to node Y if X needs to be processed before Y.

Our graph might look like this:

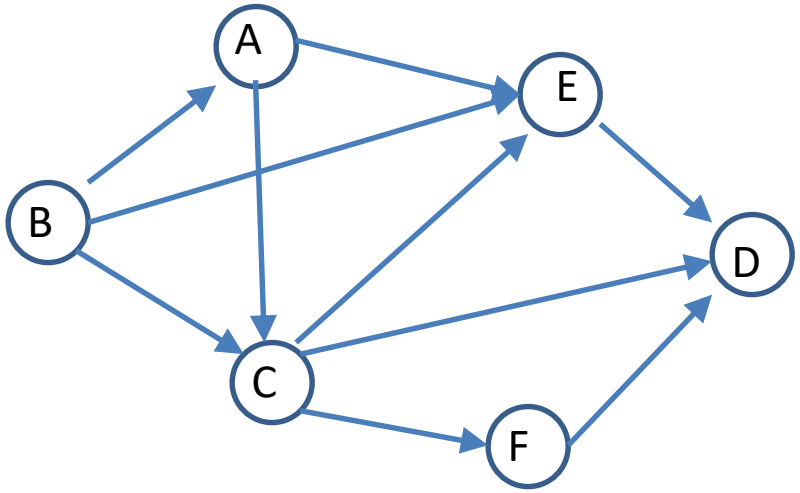


Clicker Q: Let's see if you understand the question. We want to process X before Y if there is an edge from X to Y. Here is a graph:



Which is a correct ordering?

A. B E D A C F
B. A B C D E F
C. B A C E F D
D. B A E C F D



To process this graph we will make a set of nodes we call our "WorkingSet". Initially our WorkingSet consists of all nodes that have no incoming edges.

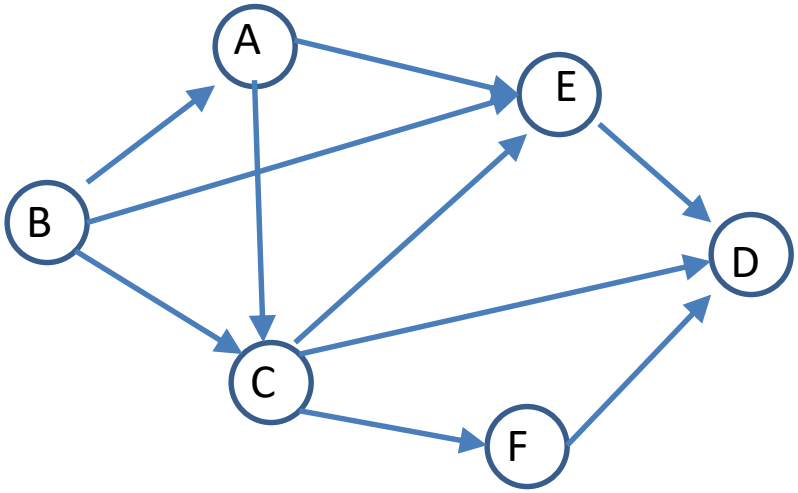
For the graph above WorkingSet = {B}

Here is the algorithm we use to process the graph. On each step:

a) Remove any one node from the WorkingSet. Call this node X.
b) Remove every edge from node X to any other node Y.
c) If node Y has no other incoming edges, add node Y to the WorkingSet.

Continue these steps until the WorkingSet is empty.

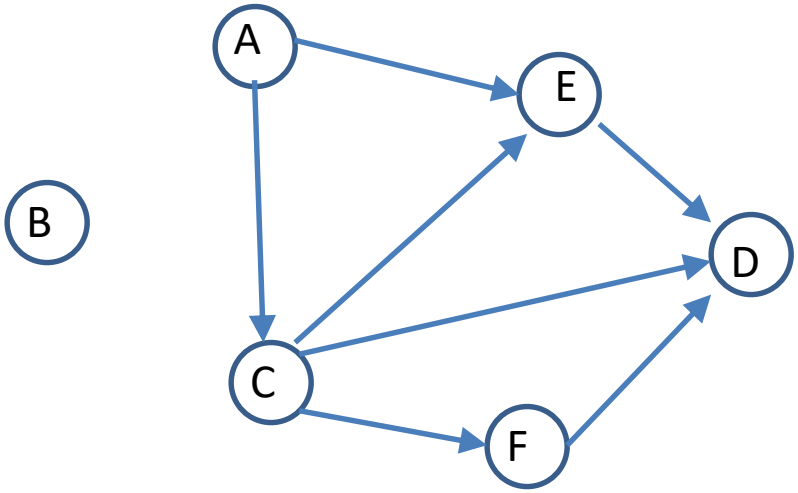
We start with this graph:



WorkingSet = {B}

Output = [ ]

On the first step we remove B from the WorkingSet, add it to the output, and we remove the edges from B to A, E, and C



WorkingSet = {}

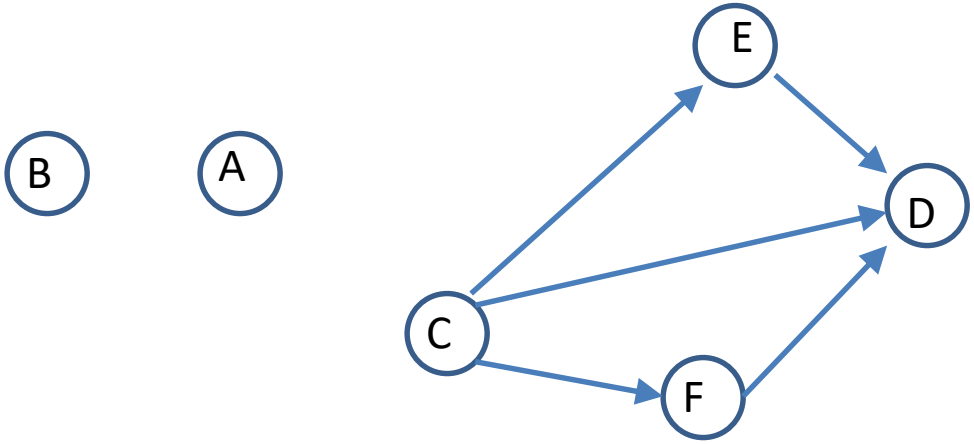
Output = [B]

We removed the only incoming edge for node A, so we add it to the WorkingSet:

WorkingSet = {A}

Output = [B]

For the next step we remove A from the WorkingSet, add it to the output, and remove its outgoing edges



WorkingSet = {}

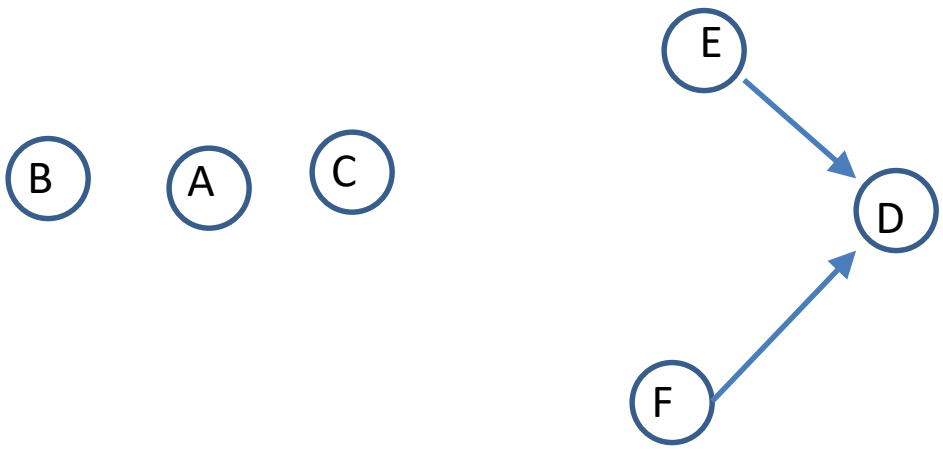
Output = [B, A]

C now has no incoming edges, so we add it to the WorkingSet:

WorkingSet = {C}

Output = [B, A]

On the next step we remove C and its edges; this leaves E and F with no incoming edges

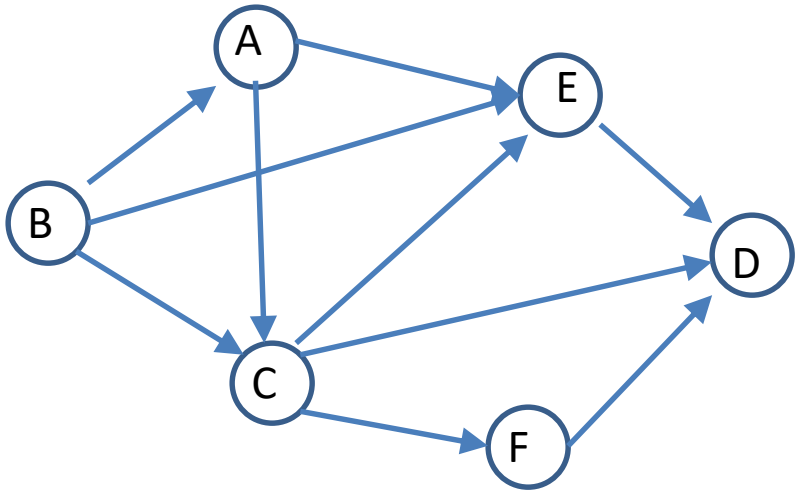


WorkingSet = {E, F}

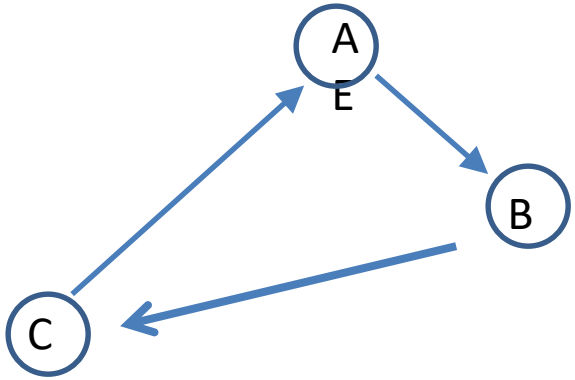
Output = [B, A, C]

On the next three steps the algorithm outputs E and F in either order, then D.

Our ordering is thus either [B, A, C, E, F, D] or [B, A,C, F, E, D]. If you compare these to the original graph you can see that no node appears in this list before any of its dependencies.

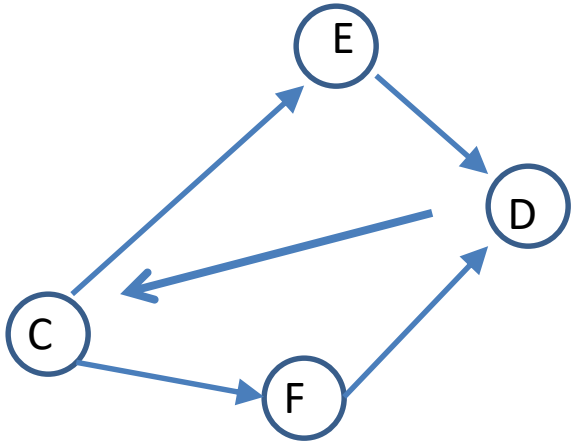


Clicker Q: Does our algorithm work for this graph?



A. Yes
B. No

Does our algorithm work for this graph?



A. Yes
B. No

When does our algorithm *not* work?

A. It never works

B. If the graph has too many nodes?

C. If the graph has a *cycle* -- a sequence of edges from node to node that eventually gets back to its starting point.

D. If the problem has no solution

This idea of mapping a problem to a graph and processing the graph to solve the problem has many applications. To consider any of these we need some terminology and we need to look at some ways to represent graphs.