2003

Directed Acyclic Graphs

Stephen B. Maurer , '67
Swarthmore College, smaurer1@swarthmore.edu

3.2 DIRECTED ACYCLIC GRAPHS

Stephen B. Maurer, Swarthmore College

Introduction

When a digraph has no directed cycles, it is called a directed acyclic graph, or a DAG. While being acyclic may seem to be a stringent condition, it arises quite naturally because vertices often have a natural ordering. For instance, vertices may represent events ordered in time or ordered by hierarchy. This ordering makes results and algorithms for DAGs relatively simple.

3.2.1 Examples and Basic Facts

DEFINITIONS

D1: A digraph is ***acyclic*** if it has no *directed* cycles.

D2: ***DAG*** is an acronym for directed acyclic graph.

D3: A ***source*** in a digraph is a vertex of indegree zero.

D4: A ***sink*** in a digraph is a vertex of outdegree zero.

D5: A ***basis*** of a digraph is a minimal set of vertices such that every other vertex can be reached from some vertex in this set by a directed path.

EXAMPLES

E1: *Operations Research.* A large project consists of many smaller tasks with a *precedence relation* — some tasks must be completed before certain others can begin. One graphical representation of such a project has a vertex for each task and an arc from u to v if task u must be completed before v can begin. For instance, in Figure 3.2.1, the food must be loaded and the cabin cleaned before passengers are loaded, but luggage unloading is independent of the timing of cabin activities. This model of a project will always be a DAG, because if there were a directed cycle, the project could not be done: every task on the cycle would have to be started before every other one on the cycle.

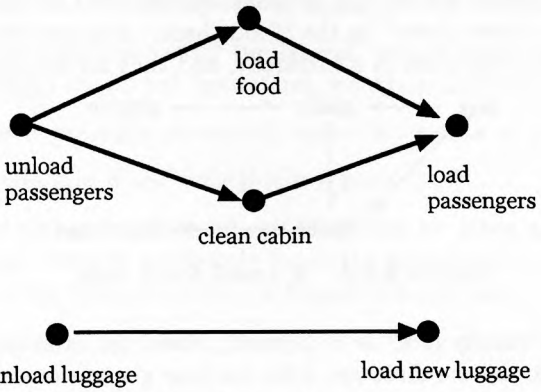

Figure 3.2.1 A digraph of precedence in an airplane stopover.

E2: *Sociology and Sociobiology.* A business (or army, or society, or ant colony) has a *hierarchical dominance* structure. The nodes are the employees (soldiers, citizens, ants) and there is an arc from u to v if u dominates v. If the chain of command is unique, with a single leader, and if only arcs representing immediate authority are included, then the result is a *rooted tree*, as in Figure 3.2.2. (Also see §3.2.2.)

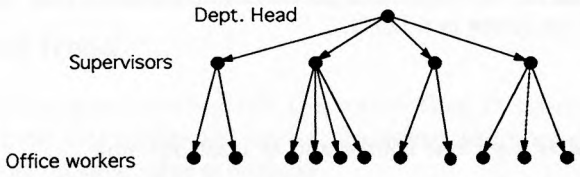

Figure 3.2.2 A corporate hierarchy.

E3: *Computer Software Design.* A large program consists of many subprograms, some of which can invoke others. Let the nodes of D be the subprograms, and let there be an arc from u to v if subprogram u can invoke subprogram v. Then this *call graph* D encapsulates all possible ways control can flow within the program. Must D be a DAG? No, but each directed cycle represents an indirect recursion and serves as a warning to the designer to ensure against infinite loops. See Figure 3.2.3, where Proc 2 can call itself indirectly. To determine if a digraph is a DAG or not, do a *topological sort* (§3.2.4).

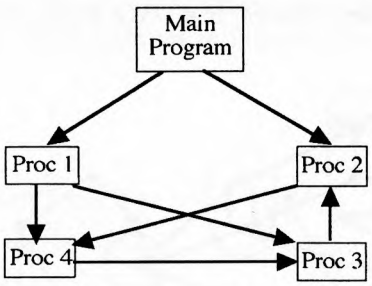

Figure 3.2.3 The call graph of a computer program.

E4: *Ecology.* A *food web* is a digraph in which nodes represent species and in which there is an arc from u to v if species u eats species v. Figure 3.2.4 shows a small food

web. In general, food webs are acyclic, because animals tend to eat smaller animals or animals in some way "lower down" in the "food chain." The very fact that phrases like this are used indicates that there is a hierarchy, and thus no directed cycles.

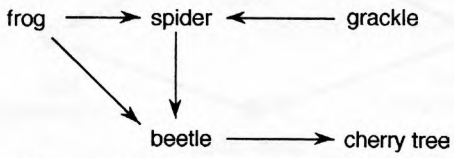

Figure 3.2.4 A small food web.

E5: *Genealogy.* A "family tree" is a digraph, where the orientation is traditionally given not by arrows but by the direction down for later generations. Despite the name, a family tree is usually not a tree, since people commonly marry distant cousins, knowingly or unknowingly. However, it is always a DAG, because if there were a cycle, everyone on it would be older than everyone else on the cycle.

E6: *State Diagrams.* Let the vertices of D be a set of states of some process, and let the arcs represent possible transitions. For instance, the process might be a board game, where the states are the configurations and each arc represents the transition of a single move. Then walks through D represent "histories" that the process/game can follow. If the game can never return to a previous configuration (e.g., as in tic-tac-toe), the state diagram of the game is a DAG.

FACTS

F1: Every DAG has at least one source and at least one sink.

F2: Every DAG has a unique basis, namely, the set of all its sources.

F3: Every subgraph of a DAG is a DAG.

F4: The transitive closure of a DAG is a DAG.

F5: A digraph is a DAG if and only if every walk in it is a path.

F6: A digraph is a DAG if and only if it is possible to order the vertices so that, in the adjacency matrix, all nonzero entries are above the main diagonal. (Topological sort in §3.2.4 finds the ordering.)

F7: The condensation of any digraph is a DAG. Figure 3.2.5 shows a digraph and its condensation.

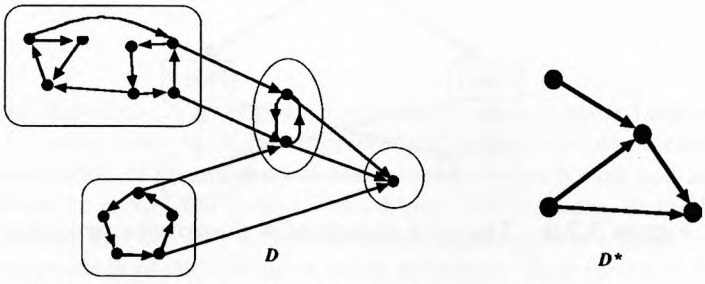

Figure 3.2.5 A digraph and its condensation.

F8: A digraph is a DAG if and only if it is isomorphic to its condensation.

F9: A digraph is strongly connected (unilateral, weakly connected) if and only if its condensation is strongly connected (unilateral, weakly connected).

F10: A DAG is never strongly connected, unless it consists of a single vertex.

F11: A DAG is unilateral if and only if it is a path.

F12: Every undirected graph without self-loops can be given an acyclic orientation, in fact, usually many. Namely, arbitrarily index the vertices as $v_1, v_2, \ldots, v_n$ and direct each edge from its lower indexed end to its higher indexed end.

REMARKS

R1: For more basic information on DAGs, see [Ha94, Ch. 16] and [Ro76, §2.2–3].

R2: Most of the acyclic orientations in Fact 12 are arbitrary and uninteresting, but occasionally an acyclic orientation is natural. In a tree, it is natural to orient edges away from a *root*; see §3.2.2. In a bipartite graph, it is natural to direct all edges from one side to the other. Still, most interesting orientations are already imposed by the nature of the problem, and the question is whether they are acyclic.

3.2.2 Rooted Trees

If the underlying graph of a digraph D is a tree, then D is certainly a DAG, because it doesn't even have any undirected cycles. However, the important tree DAGs have further restrictions on their edge directions.

For more on rooted trees, see [GrYe99, §3.2].

DEFINITIONS

D6: A ***directed tree*** is a digraph whose underlying graph is a tree.

D7: A ***rooted tree*** is a directed tree with a distinguished vertex r, called the ***root***, such that for every other vertex v, the unique path from r to v is a directed path from r to v.

CONVENTION: In drawing a rooted tree with the root marked, the arrows are usually omitted because the direction of each arc is always away from the root. In fact, if the direction is always down or left-to-right, as in Figure 3.2.6, it is not even necessary to indicate the root.

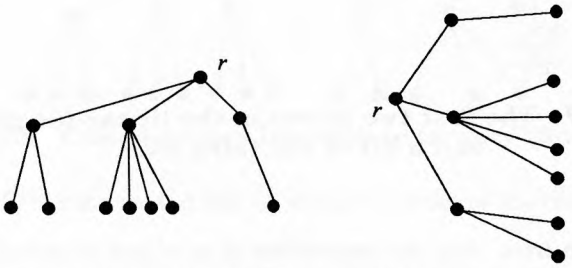

Figure 3.2.6 Two standard ways to draw a rooted tree.

D8: A rooted tree is also called an ***out-tree***. This alternative name is typically used when the arc directions are shown explicitly, for instance, when the tree is a spanning subgraph of a larger digraph.

D9: An ***in-tree*** is an out-tree with all the directions reversed, so that all paths are directed toward the root.

EXAMPLES

Previous Example 2 is about rooted trees. Here are some others.

E7: *Decision trees.* Any branching process leads to a rooted tree, where each node is a decision point, each arc from a node is an allowed decision, and the root is the start. For instance, the stages in a game may be represented this way. Figure 3.2.7 shows the first two moves in a game of tic-tac-toe, one by each player. Each node is represented by the way the board looks just *before* the decision. If we take into account symmetry, the figure is complete through the first two moves.

CONVENTION: In Figure 3.2.7 the two nodes on the bottom level (3rd move) illustrate that different nodes in the tree can represent the same state. While the board looks the same at these two nodes, the ordered sequence of decisions leading to these nodes are different. Thus in a decision tree, each node represents both a state and the complete history of how it was achieved. Compare with Example 6, where these nodes would be one, and the digraph would not be a tree.

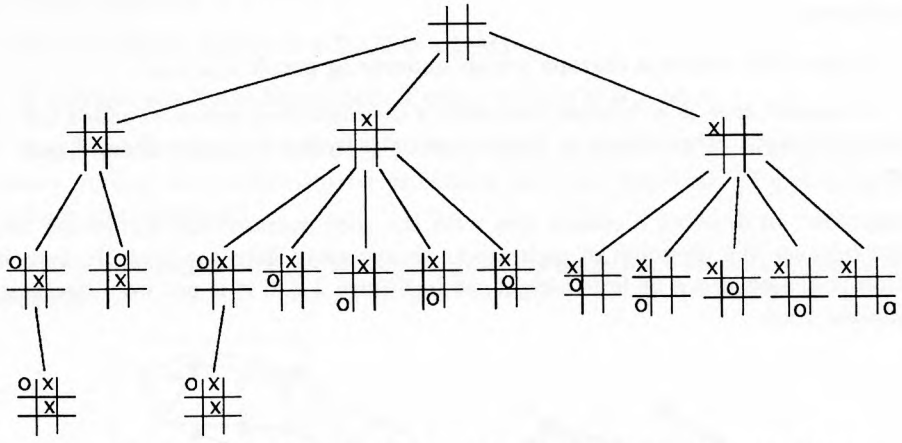

Figure 3.2.7 The first two moves in the tic-tac-toe game tree, and a bit of the third level.

E8: *Decomposition trees.* Any decomposition of an object or structure into finer and finer parts can be modeled with a rooted tree. Figure 3.2.8 shows an example of *sentence parsing*.

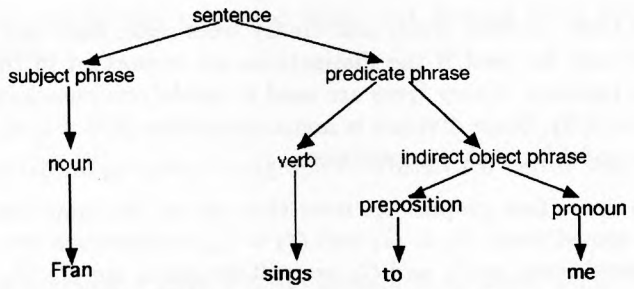

Figure 3.2.8 A sentence parse tree.

FACTS

F13: Every directed tree is a DAG.

F14: A digraph is a rooted tree if and only if its underlying graph is connected, exactly one vertex (the root) has indegree 0, and all others have indegree 1.

DEFINITIONS FOR ROOTED TREES

D10: The ***depth*** or ***level*** of a vertex v is its distance from the root, that is, the number of edges in the unique directed path from the root to v.

D11: The ***height*** of a rooted tree is the greatest depth of a vertex.

D12: If (u, v) is an edge, the u is the ***parent*** of v and v is the ***child*** of u.

D13: Vertices having the same parent are ***siblings***.

D14: If there is a directed path from vertex u to vertex v, then u is an ***ancestor*** of v and v is a ***descendant*** of u.

D15: A ***leaf*** is a vertex with outdegree 0 (no children).

D16: An ***internal vertex*** is a vertex that is not a leaf.

D17: An ***m-ary tree*** is a rooted tree in which every vertex has m or fewer children.

D18: A ***complete m-ary tree*** is an m-ary tree in which every internal vertex has exactly m children and all leaves are at the same level. See Figure 3.2.9.

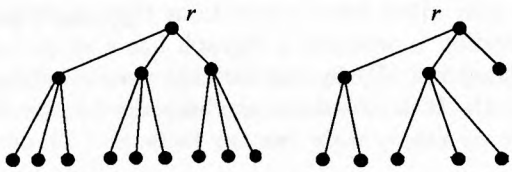

Figure 3.2.9 Complete and incomplete ternary (3-ary) trees.

D19: A ***ordered tree*** is a rooted tree in which the order of the children at each vertex makes a difference.

D20: A ***binary tree*** is an ordered 2-ary tree in which, even when a vertex has only one child, it makes a difference whether it is a ***left child*** or a ***right child***.

REMARKS

R3: Trees, rooted trees, ordered trees, and binary trees make finer and finer distinctions, which should only be used if the distinctions are important in the application being modeled. For instance, binary trees are used to model computations with binary operations, as in $3 \times (4/5)$. Since division is *noncommutative* $(4/5 \neq 5/4)$, binary trees are an appropriate model for such computations.

R4: Figure 3.2.10 shows four graphs. As trees they are all the same (that is, *isomorphic*). However, as rooted trees, $G_1 = G_2$ and $G_3 = G_4$, so there are two rooted trees. There are three ordered trees, as G_1 and G_2 are still the same, but G_3, G_4 are different. Finally, as binary trees they are all different. In G_1, vertex c is a right child; in G_2 it is a left child.

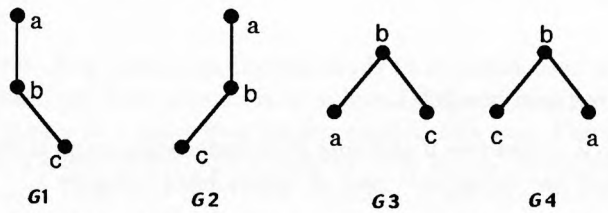

Figure 3.2.10 Four trees: the same and not the same.

FACTS

F15: An m-ary tree has at most m^k vertices at level k.

F16: Let T be an n-vertex m-ary tree of height h. Then

$$h + 1 \leq n \leq \frac{m^{h+1} - 1}{m - 1}.$$

The lower bound is attained if and only if T is a path. The upper bound is attained if and only if T is a complete m-ary tree.

Spanning Directed Trees

Since every connected graph has a spanning tree, every digraph has a spanning directed tree. In a graph, a spanning tree connects all the vertices, while using the minimum number of edges. However, in a digraph, a spanning directed tree may contain few directed *paths* and thus may allow fewer connections than the whole digraph does. So the more interesting question is whether a digraph has a spanning rooted tree. This question is answered algorithmically by the directed version of *depth first search*; see §10.1 and [GrYe99, §11.1]. It is answered algebraically by the directed matrix tree theorems; see §6.4. Here we simply state two key facts.

FACTS

F17: If digraph D has a spanning tree rooted at v, directed depth first search starting at v will find one.

F18: For every vertex of a digraph D there is a spanning tree rooted at that vertex if and only if D is strongly connected.

Functional Graphs

Closely related structurally to rooted trees, but devised for a different purpose, are functional graphs.

DEFINITION

D21: A ***functional graph*** is a digraph in which each vertex has outdegree one.

EXAMPLES

E9: For each function f from a finite domain U to itself, define a digraph D whose vertex set is U and for which (u, v) is an arc if and only if $f(u) = v$. By definition of a function, there is one such v for every $u \in U$. Hence, D is a functional graph (whence the name).

E10: Specifically, consider the doubling function on the positive integers, but consider only the effect on the ones digit. This function is completely described by its effect on the domain $\{0, 1, \ldots, 9\}$. Its functional graph is shown in Figure 3.2.11.

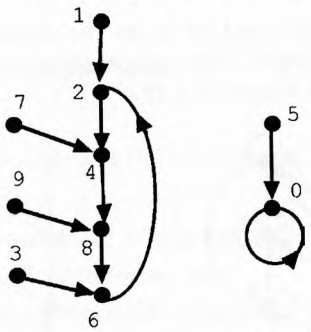

Figure 3.2.11 The functional graph for doubling (mod 10).

FACT

F19: Let D be a functional graph, and let G be the underlying undirected graph. Then each component of G contains exactly one cycle. In D this cycle is a directed cycle, and the removal of any arc in it turns that component into an in-tree.

3.2.3 DAGs and Posets

There is a very close connection between DAGs and posets. Every DAG represents a poset, and every poset can be represented by DAGs in several ways. For more information, see [Bo90, §7.1–2].

DEFINITIONS

D22: A ***partial order*** is a binary relation $\preceq$ on a set X that is

- *reflexive*: for all $x \in X$, $x \preceq x$;
- *antisymmetric*: for all $x, y \in X$, if $x \preceq y$ and $y \preceq x$, then $x = y$;
- *transitive*: for all $x, y, z \in X$, if $x \preceq y$ and $y \preceq z$, then $x \preceq z$.

D23: A ***poset***, or ***partially ordered set*** $P = (X, \preceq)$ is a pair consisting of a set X, called the ***domain***, and a partial order $\preceq$ on X.

D24: Elements x, y of P are ***comparable*** if either $x \preceq y$ or $y \preceq x$.

D25: Element x is ***less than*** element y, written $x \prec y$, if $x \preceq y$ and $x \neq y$.

D26: The ***comparability digraph*** of the poset $P = (X, \preceq)$ is the digraph with vertex set X such that there is an arc from x to y if and only if $x \preceq y$.

D27: The element y ***covers*** the element x in a poset if $x \prec y$ and there is no element z such that $x \prec z \prec y$.

D28: The ***cover graph*** of a poset $P = (X, \preceq)$ is the graph with vertex set X such that x, y are adjacent if and only if one of them covers the other.

D29: A ***Hasse diagram*** of poset P is a straight-line drawing of the cover graph such that the lesser element of each adjacent pair is lower in the drawing.

EXAMPLE

E11: Let $X = \{2, 4, 5, 8, 10, 20\}$ and let $\preceq$ be the *divisibility relation* on X. That is $x \preceq y$ if and only if y/x is an integer. The comparability digraph and the Hasse diagram for $P = (X, \preceq)$ are as shown in Figure 3.2.12.

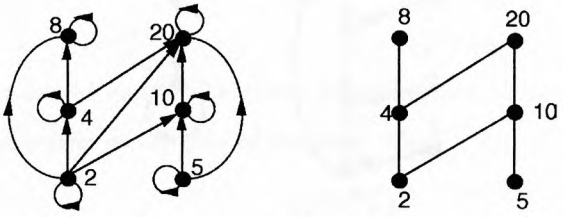

Figure 3.2.12 Comparability digraph and Hasse diagram for a poset.

FACTS

F20: If the loops are deleted, the comparability digraph of any poset is a DAG.

F21: Every Hasse diagram is a DAG if one considers all edges to be directed up (or all down).

F22: Every DAG D represents a poset in the following sense. The domain of P is the vertex set of D, and $x \preceq y$ if there is a directed path from x to y.

TERMINOLOGY NOTE: In passing from DAG D to poset P, null paths are included, so that $x \preceq x$ for all x. Alternatively, we obtain the poset by taking the transitive closure D^* of D. Then $x \prec y$ if and only if (x, y) is an arc of D^*.

3.2.4 Topological Sort and Optimization

In a DAG, the vertices can always be numbered consecutively so that all arcs go from lower to higher numbers. Using this numbering, many optimization problems can be solved by essentially the same algorithm, one that makes a single pass through the

vertices in numbered order. For more general digraphs, algorithms for these optimization problems are less efficient or at least more complicated to describe.

DEFINITIONS

D30: A ***linear extension ordering*** of a digraph is a consecutive numbering of the vertices as $v_1, v_2, \ldots, v_n$ so that all arcs go from lower-numbered to higher-numbered vertices.

D31: A ***topological sort***, or ***topsort***, is any algorithm that assigns a linear extension ordering to a digraph when it has one. (This name is traditional, but the relation to topology in the sense understood by topologists is obscure.) A simple topological sort algorithm is shown as Algorithm 3.2.1. See also [Ro84, §11.6.2].

FACTS

F23: A digraph has a linear extension ordering if and only if it is a DAG.

F24: Topological sort determines if a digraph is a DAG and finds a linear extension ordering if it is.

Algorithm 3.2.1: Topological sort

Input: a digraph D.
Output: A linear extension ordering if D is a DAG; failure otherwise

```
H := D; k = 1
while V_H ≠ φ    {vertex set of H non-empty}
  v_k := any vertex in H of indegree zero.
    {If no such vertex exists, exit: D is not a DAG}
  H := H − v_k    {New H is a DAG if old H was}
  k := k+1
```

REMARK

R5: Because of the close connection between DAGs and posets, this whole discussion of linear extensions and topological sort can just as well be stated in the poset context. For instance, every poset has a linear extension, which may be found by a topological sort. See [GrYe99, pp. 373–376].

Optimization

There are many computational problems about graphs, with important real-world applications, when the graphs have *weights* on their vertices and/or edges. For DAGs, many of these problems can be solved by essentially the same single-pass algorithm. This algorithm is the basic form of the sort of staged algorithm called *dynamic programming* in operations research circles [HiLi95, Ch. 10]. Algorithms 3.2.2 and 3.2.3 provide templates for two versions of this algorithm. The examples that follow fill in the templates by giving specific formulas for updating the functions they compute.

In Algorithm 3.2.2, topsort is done first, and then the function F is computed vertex by vertex in topsort order. In Algorithm 3.2.3, the topsort is done simultaneously with improving F on vertices not yet sorted.

Algorithm 3.2.2: Basic Dynamic Programming, First Version

Input: DAG D with vertices numbered $v_1, v_2, \ldots, v_n$ in topsort order; weights $w(v)$ on vertices or $w(v, u)$ on arcs, as needed.

Output: Correct values of desired function F.

Initialize $F(v_1)$
For $k = 2$ to n
  Determine $F(v_k)$ in terms of weights and $F(v_i)$ for $i < k$.

Algorithm 3.2.3: Basic Dynamic Programming, Second Version

Input: DAG D with n vertices and weights $w(v)$ on vertices or $w(v, u)$ on arcs, as needed.

Output: Correct values of desired function F.

Initialize $F(v)$ for all v.
$H := D$
For $k = 1$ to n
  $v_k :=$ a source in H {exists since H is a DAG}
  Update $F(u)$ for all u for which (v_k, u) is an edge in H.
  $H := H - v_k$

EXAMPLES

For simplicity in the formulas, in all examples below we assume that the DAGs have no multiple edges.

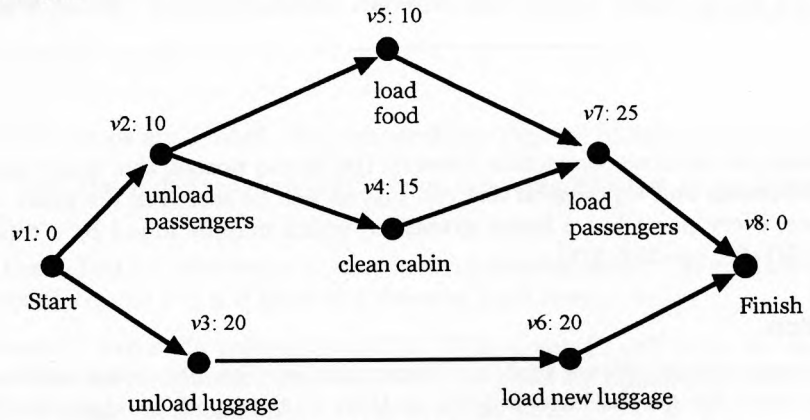

Figure 3.2.13 Airplane stopover as CPM graph.

E12: *Project Scheduling.* Consider Figure 3.2.13, which repeats Figure 3.2.1 with the following additions: Start and Finish vertices, a topsort ordering, and times for the tasks as weights on the vertices. Start and Finish, being merely marker vertices, take time 0. Recall that (u, v) is an arc if task u must be completed directly before task v begins, and that these tasks are the steps necessary to complete an airplane stopover. How quickly can the stopover be completed? The bottleneck is the *longest path* from

Start to Finish, where the length of a (directed) path is the sum of the weights on its vertices. Dynamic programming can answer this question as follows. Let

$F(u)$ = the length of the longest path (using vertex weights) from Start to u.

Then in Algorithm 3.2.2 use

Initialization: $F(v_1) = w(v_1) = 0$, (Note: v_1 = Start)
Update: $F(v_k) = w(v_k) + \max\{F(v_i) \mid (v_i, v_k) \text{ is an arc}\}$.

In Algorithm 3.2.3 use

Initialization: For all v, $F(v) = w(v)$,
Update: For all u such that (v_k, u) is an arc, $F(u) = \max\{F(u), F(v_k)+w(u)\}$.

For either algorithm, at termination the desired answer is F(Finish), that is, $F(v_n)$.

This method of finding the optimal schedule by iteratively finding the longest path is the essence of the *critical path method*, or *CPM* [HiLi95, Ch. 9]. This example uses the *activity on node* model, or *AoN*. See Example 13 for the *activity on arc* model, or *AoA*.

E13: *Project Scheduling, second model.* If *edges* represent subtasks, and tasks earlier on directed paths must be completed before those later are begun, then the longest path from the Start to Finish vertex is the shortest time in which the whole project can be completed, where now the length of a path is the sum of the weights on its edges. Let

$F(u)$ = the length of the longest path (using edge weights) from Start to u.

Then in Algorithm 3.2.2 use

Initialization: $F(v_1) = 0$,
Update: $F(v_k) = \max\{F(v_i) + w(v_i, v_k) \mid (v_i, v_k) \text{ is an arc}\}$.

In Algorithm 3.2.3 use

Initialization: For all v, $F(v) = 0$,
Update: For all u such that (v_k, u) is an arc,
$F(u) = \max\{F(u), F(v_k)+w(v_k, u)\}$.

For either algorithm, at termination the desired answer is F(Finish).

E14: *Shortest Paths.* What is the shortest directed path between two vertices u and u', where the length of a path is the sum of the weights on its edges? If a graph represents a road network, and the weights on the edges are the lengths of the road segments (or the travel times, or the toll on that segment), then shortest path means the shortest road distance (or least time, or lowest toll). If the graph is a DAG, and we make u the Start vertex (by eliminating earlier vertices in the topsort if necessary), then dynamic programming finds the shortest path as follows. Let

$F(u)$ = the length of the shortest path (using edge weights) from Start to u.

Then in Algorithm 3.2.2 use

Initialization: $F(v_1) = 0$,
Update: $F(v_k) = \min\{F(v_i) + w(v_i, v_k) \mid (v_i, v_k) \text{ is an arc}\}$.

In Algorithm 3.2.3 use

$$\begin{aligned} \text{Initialization:} \quad & F(u) = 0,\ F(v) = \infty \text{ for } v \neq u, \\ \text{Update:} \quad & \text{For all } v \text{ such that } (v_k, v) \text{ is an arc,} \\ & F(v) = \min\{F(v),\ F(v_k) + w(v_k, v)\}. \end{aligned}$$

For either algorithm, at termination the desired answer is the value of $F(u')$.

E15: What is the shortest directed path between two vertices, where the length of a path is the sum of the weights on its *vertices*? Dynamic programming solves this problem too for DAGs, with a slight change in the formulas in Example 14 (replace edge weights with vertex weights).

E16: *Counting Paths.* How many directed paths are there between a given pair of vertices? If the digraph is a DAG, and the vertices are Start and Finish, let

$$F(u) = \text{the number of directed paths from Start to } u.$$

Then in Algorithm 3.2.2 use

$$\begin{aligned} \text{Initialization:} \quad & F(v_1) = 1,\ (v_1 = \text{Start}) \\ \text{Update:} \quad & F(v_k) = \textstyle\sum\{F(v_i) \mid (v_i, v_k) \text{ is an arc}\}. \end{aligned}$$

In Algorithm 3.2.3 use

$$\begin{aligned} \text{Initialization:} \quad & F(\text{Start}) = 1,\ F(v) = 0 \text{ for } v \neq \text{Start}, \\ \text{Update:} \quad & \text{For all } v \text{ such that } (v_k, v) \text{ is an arc,} \\ & F(v) = F(v) + F(v_k). \end{aligned}$$

For either algorithm, at termination the desired answer is the value of $F(\text{Finish})$.

E17: *Maximin Paths.* What is the directed path between two vertices for which the minimum edge weight on that path is maximum among all paths between those two vertices? This is called the *maximin path* and that maximum value is called the *maximin value*. In Figure 3.2.14 the maximin path from v_1 to v_6 is $v_1v_3v_4v_6$ and the maximin value is 4. If the edges represent railroad segments, and each edge weight is the weight limit on that railroad segment, then this is the path between the two points over which the heaviest load can be shipped.

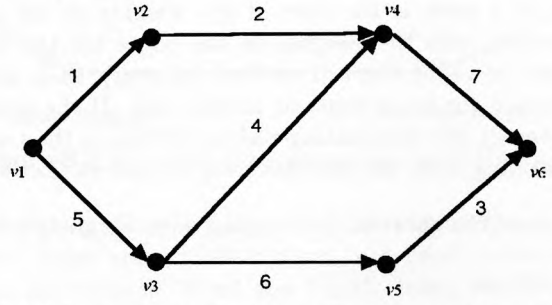

Figure 3.2.14 **The maximin path $v_1v_3v_4v_6$ has value 4 and the minimax path $v_1v_3v_5v_6$ has value 6.**

If the digraph is a DAG, and the vertices are Start and Finish, let

$F(u)$ = the maximin value for directed paths from from Start to u.

Then in Algorithm 3.2.2 use

Initialization: $F(v_1) = 0$, (v_1 = Start)

Update: $F(v_k) = \max\{\min\{F(v_i), w(v_i, v_k)\} \mid (v_i, v_k) \text{ is an arc}\}$.

In Algorithm 3.2.3 use

Initialization: $F(\text{Start}) = 0$, $F(v) = \infty$ for $v \neq$ Start,

Update: For all v such that (v_k, v) is an arc,

$F(v) = \max\{F(v), \min\{F(v_k), w(v_k, v)\}\}$.

For either algorithm, at termination the desired answer is the value of $F(\text{Finish})$.

E18: *Minimax Paths.* What is the directed path between two vertices for which the maximum edge weight on the path is minimum? This *minimax* question is relevant if the graph represents a pipeline network, and each edge weight is the maximum elevation on that segment, because the work necessary to push a fluid through a pipeline route is related to the maximum height to which the fluid must be raised along the way. In Figure 3.2.14 the minimax path from v_1 to v_6 is $v_1v_3v_5v_6$ and the minimax value is 6. Dynamic programming solutions to the minimax problem are found by interchanging the roles of min and max in the algorithms for Example 17. Also, in Algorithm 3.2.3, all $F(v)$ are initialized to 0.

FACTS

F25: Algorithms 3.2.2–3 each solve critical path problems and many other optimization and computation problems on DAGs. (See the examples above.)

F26: In project scheduling problems modeled by DAGs, the minimum completion time is the length of the longest path from the Start node to the Finish node.

F27: Any DAG may be augmented to have just one source and one sink (just create a new node named Start adjacent *to* all existing sources, and a new node named Finish adjacent *from* all existing sinks).

References

[Bo90] K. Bogart, *Introductory Combinatorics*, Harcourt Brace Jovanovich, 1990.

[GrYe99] J. L. Gross and J. Yellen, *Graph Theory and Its Applications*, CRC Press, 1999.

[Ha94] F. Harary, *Graph Theory*, Perseus, 1994 (reprint of original edition, Addison Wesley, 1969).

[HiLi95] F. Hillier and G. Lieberman, *Introduction to Operations Research*, Sixth Edition, McGraw-Hill, 1995.

[Ro76] F. Roberts, *Discrete Mathematical Models*, Prentice-Hall, 1976.

[Ro84] F. Roberts, *Applied Combinatorics*, Prentice-Hall, 1985.