10 Directed graphs & Partial Orders

Directed graphs, called *digraphs* for short, provide a handy way to represent how things are connected together and how to get from one thing to another by following those connections. They are usually pictured as a bunch of dots or circles with arrows between some of the dots, as in Figure 10.1. The dots are called *nodes* or *vertices* and the lines are called *directed edges* or *arrows*; the digraph in Figure 10.1 has 4 nodes and 6 directed edges.

Digraphs appear everywhere in computer science. For example, the digraph in Figure 10.2 represents a communication net, a topic we'll explore in depth in Chapter 11. Figure 10.2 has three “in” nodes (pictured as little squares) representing locations where packets may arrive at the net, the three “out” nodes representing destination locations for packets, and the remaining six nodes (pictured with little circles) represent switches. The 16 edges indicate paths that packets can take through the router.

Another place digraphs emerge in computer science is in the hyperlink structure of the World Wide Web. Letting the vertices $x_1, \ldots, x_n$ correspond to web pages, and using arrows to indicate when one page has a hyperlink to another, results in a digraph like the one in Figure 10.3—although the graph of the real World Wide Web would have n be a number in the billions and probably even the trillions. At first glance, this graph wouldn't seem to be very interesting. But in 1995, two students at Stanford, Larry Page and Sergey Brin, ultimately became multibillionaires from the realization of how useful the structure of this graph could be in building a search engine. So pay attention to graph theory, and who knows what might happen!

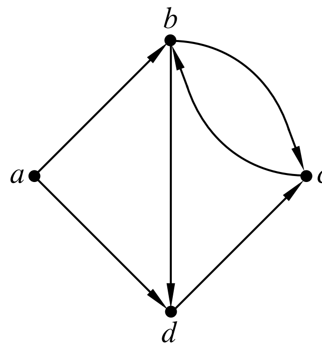

Figure 10.1 A 4-node directed graph with 6 edges.

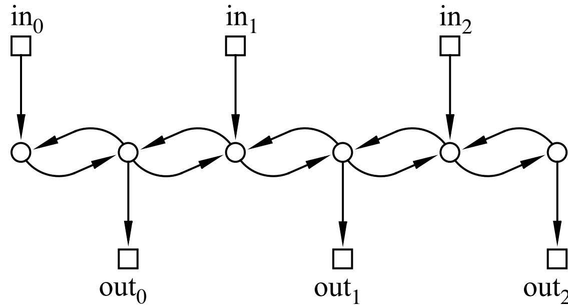

Figure 10.2 A 6-switch packet routing digraph.

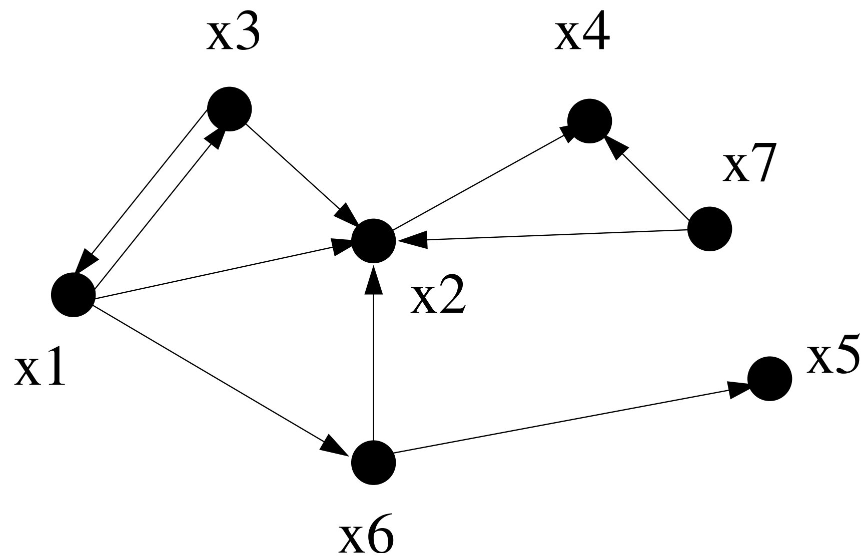

Figure 10.3 Links among Web Pages.

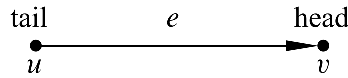

Figure 10.4 A directed edge $e = \langle u \to v \rangle$. The edge e starts at the tail vertex u and ends at the head vertex v.

Definition 10.0.1. A *directed graph* G consists of a nonempty set $V(G)$, called the *vertices* of G, and a set $E(G)$, called the *edges* of G. An element of $V(G)$ is called a *vertex*. A vertex is also called a *node*; the words "vertex" and "node" are used interchangeably. An element of $E(G)$ is called a *directed edge*. A directed edge is also called an "arrow" or simply an "edge." A directed edge *starts* at some vertex u called the *tail* of the edge, and *ends* at some vertex v called the *head* of the edge, as in Figure 10.4. Such an edge can be represented by the ordered pair (u, v). The notation $\langle u \to v \rangle$ denotes this edge.

There is nothing new in Definition 10.0.1 except for a lot of vocabulary. Formally, a digraph G is the same as a binary relation on the set, $V = V(G)$—that is, a digraph is just a binary relation whose domain and codomain are the same set V. In fact, we've already referred to the arrows in a relation G as the "graph" of G. For example, the divisibility relation on the integers in the interval [1..12] could be pictured by the digraph in Figure 10.5.

10.1 Vertex Degrees

The *in-degree* of a vertex in a digraph is the number of arrows coming into it, and similarly its *out-degree* is the number of arrows out of it. More precisely,

Definition 10.1.1. If G is a digraph and $v \in V(G)$, then

$$\text{indeg}(v) ::= |\{e \in E(G) \mid \text{head}(e) = v\}|$$
$$\text{outdeg}(v) ::= |\{e \in E(G) \mid \text{tail}(e) = v\}|$$

An immediate consequence of this definition is

Lemma 10.1.2.

$$\sum_{v \in V(G)} \text{indeg}(v) = \sum_{v \in V(G)} \text{outdeg}(v).$$

Proof. Both sums are equal to $|E(G)|$. ■

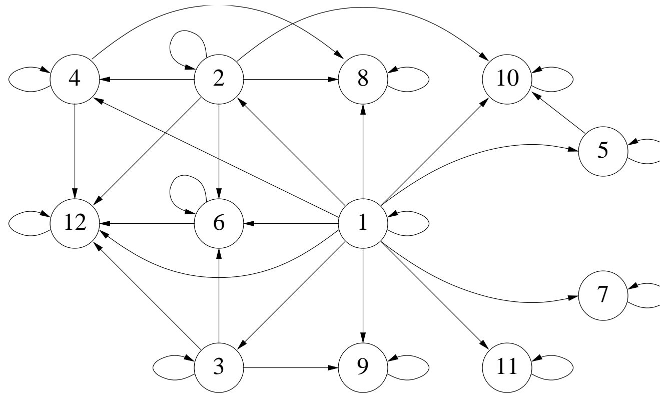

Figure 10.5 The Digraph for Divisibility on $\{1, 2, \ldots, 12\}$.

10.2 Walks and Paths

Picturing digraphs with points and arrows makes it natural to talk about following successive edges through the graph. For example, in the digraph of Figure 10.5, you might start at vertex 1, successively follow the edges from vertex 1 to vertex 2, from 2 to 4, from 4 to 12, and then from 12 to 12 twice (or as many times as you like). The sequence of edges followed in this way is called a *walk* through the graph. A *path* is a walk which never visits a vertex more than once. So following edges from 1 to 2 to 4 to 12 is a path, but it stops being a path if you go to 12 again.

The natural way to represent a walk is with the sequence of sucessive vertices it went through, in this case:

$$1 \ \ 2 \ \ 4 \ \ 12 \ \ 12 \ \ 12.$$

However, it is conventional to represent a walk by an alternating sequence of successive vertices and edges, so this walk would formally be

$$1 \ \langle 1 \to 2 \rangle \ 2 \ \langle 2 \to 4 \rangle \ 4 \ \langle 4 \to 12 \rangle \ 12 \ \langle 12 \to 12 \rangle \ 12 \ \langle 12 \to 12 \rangle \ 12. \tag{10.1}$$

The redundancy of this definition is enough to make any computer scientist cringe, but it does make it easy to talk about how many times vertices and edges occur on the walk. Here is a formal definition:

Definition 10.2.1. A *walk in a digraph* is an alternating sequence of vertices and edges that begins with a vertex, ends with a vertex, and such that for every edge $\langle u \to v \rangle$ in the walk, vertex u is the element just before the edge, and vertex v is the next element after the edge.

So a walk $\mathbf{v}$ is a sequence of the form

$$\mathbf{v} ::= v_0 \ \langle v_0 \to v_1 \rangle \ v_1 \ \langle v_1 \to v_2 \rangle \ v_2 \ \ldots \ \langle v_{k-1} \to v_k \rangle \ v_k$$

where $\langle v_i \to v_{i+1} \rangle \in E(G)$ for $i \in [0..k)$. The walk is said to *start* at v_0, to *end* at v_k, and the *length* $|\mathbf{v}|$ of the walk is defined to be k.

The walk is a *path* iff all the v_i's are different, that is, if $i \neq j$, then $v_i \neq v_j$.

A *closed walk* is a walk that begins and ends at the same vertex. A *cycle* is a positive length closed walk whose vertices are distinct except for the beginning and end vertices.

Note that a single vertex counts as a length zero path that begins and ends at itself. It also is a closed walk, but does not count as a cycle, since cycles by definition must have positive length. Length one cycles are possible when a node has an arrow leading back to itself. The graph in Figure 10.1 has none, but every vertex in the divisibility relation digraph of Figure 10.5 is in a length one cycle. Length one cycles are sometimes called *self-loops*.

Although a walk is officially an alternating sequence of vertices and edges, it is completely determined just by the sequence of successive vertices on it, or by the sequence of edges on it. We will describe walks in these ways whenever it's convenient. For example, for the graph in Figure 10.1,

- (a, b, d), or simply abd, is a (vertex-sequence description of a) length two path,
- $(\langle a \to b \rangle, \langle b \to d \rangle)$, or simply $\langle a \to b \rangle \langle b \to d \rangle$, is (an edge-sequence description of) the same length two path,
- $abcbd$ is a length four walk,
- $dcbcbd$ is a length five closed walk,
- $bdcb$ is a length three cycle,
- $\langle b \to c \rangle \langle c \to b \rangle$ is a length two cycle, and
- $\langle c \to b \rangle \langle b \leftarrow a \rangle \langle a \to d \rangle$ is *not* a walk. A walk is not allowed to follow edges in the wrong direction.

If you walk for a while, stop for a rest at some vertex, and then continue walking, you have broken a walk into two parts. For example, stopping to rest after following two edges in the walk (10.1) through the divisibility graph breaks the walk into the first part of the walk

$$1 \ \langle 1 \to 2 \rangle \ 2 \ \langle 2 \to 4 \rangle \ 4 \tag{10.2}$$

from 1 to 4, and the rest of the walk

$$4 \ \langle 4 \to 12 \rangle \ 12 \ \langle 12 \to 12 \rangle \ 12 \ \langle 12 \to 12 \rangle \ 12. \tag{10.3}$$

from 4 to 12, and we'll say the whole walk (10.1) is the *merge* of the walks (10.2) and (10.3). In general, if a walk $\mathbf{f}$ ends with a vertex v and a walk $\mathbf{r}$ starts with the same vertex v we'll say that their *merge* $\mathbf{f}\,\widehat{\ }\,\mathbf{r}$ is the walk that starts with $\mathbf{f}$ and continues with $\mathbf{r}$.[1] Two walks can only be merged if the first walk ends at the same vertex v with which the second one walk starts. Sometimes it's useful to name the node v where the walks merge; we'll use the notation $\mathbf{f}\,\widehat{v}\,\mathbf{r}$ to describe the merge of a walk $\mathbf{f}$ that ends at v with a walk $\mathbf{r}$ that begins at v.

A consequence of this definition is that

Lemma 10.2.2.

$$|\mathbf{f}\,\widehat{\ }\,\mathbf{r}| = |\mathbf{f}| + |\mathbf{r}|.$$

In the next section we'll get mileage out of walking this way.

10.2.1 Finding a Path

If you were trying to walk somewhere quickly, you'd know you were in trouble if you came to the same place twice. This is actually a basic theorem of graph theory.

Theorem 10.2.3. *A shortest walk from one vertex to another is a path.*

Proof. If there is a walk from vertex u to another vertex $v \neq u$, then by the Well Ordering Principle, there must be a minimum length walk $\mathbf{w}$ from u to v. We claim $\mathbf{w}$ is a path.

To prove the claim, suppose to the contrary that $\mathbf{w}$ is not a path, meaning that some vertex x occurs twice on this walk. That is,

$$\mathbf{w} = \mathbf{e}\,\widehat{x}\,\mathbf{f}\,\widehat{x}\,\mathbf{g}$$

for some walks $\mathbf{e}, \mathbf{f}, \mathbf{g}$ where the length of $\mathbf{f}$ is positive. But then "deleting" $\mathbf{f}$ yields a strictly shorter walk

$$\mathbf{e}\,\widehat{x}\,\mathbf{g}$$

from u to v, contradicting the minimality of $\mathbf{w}$. ■

Definition 10.2.4. The *distance*, $\text{dist}(u, v)$, in a graph from vertex u to vertex v is the length of a shortest path from u to v.

[1] It's tempting to say the *merge* is the concatenation of the two walks, but that wouldn't quite be right because if the walks were concatenated, the vertex v would appear twice in a row where the walks meet.

As would be expected, this definition of distance satisfies:

Lemma 10.2.5. *[The Triangle Inequality]*

$$dist(u, v) \le dist(u, x) + dist(x, v)$$

for all vertices u, v, x with equality holding iff x is on a shortest path from u to v.

Of course, you might expect this property to be true, but distance has a technical definition and its properties can't be taken for granted. For example, unlike ordinary distance in space, the distance from u to v is typically different from the distance from v to u. So, let's prove the Triangle Inequality

Proof. To prove the inequality, suppose $\mathbf{f}$ is a shortest path from u to x and $\mathbf{r}$ is a shortest path from x to v. Then by Lemma 10.2.2, $\mathbf{f} \,\widehat{x}\, \mathbf{r}$ is a walk of length $\text{dist}(u, x) + \text{dist}(x, v)$ from u to v, so this sum is an upper bound on the length of the shortest path from u to v by Theorem 10.2.3.

Proof of the "iff" is in Problem 10.3. ■

Finally, the relationship between walks and paths extends to closed walks and cycles:

Lemma 10.2.6. *The shortest positive length closed walk through a vertex is a cycle through that vertex.*

The proof of Lemma 10.2.6 is essentially the same as for Theorem 10.2.3; see Problem 10.4.

10.3 Adjacency Matrices

If a graph G has n vertices $v_0, v_1, \ldots, v_{n-1}$, a useful way to represent it is with an $n \times n$ matrix of zeroes and ones called its *adjacency matrix* A_G. The ijth entry of the adjacency matrix, $(A_G)_{ij}$, is 1 if there is an edge from vertex v_i to vertex v_j and 0 otherwise. That is,

$$(A_G)_{ij} ::= \begin{cases} 1 & \text{if } \langle v_i \to v_j \rangle \in E(G), \\ 0 & \text{otherwise.} \end{cases}$$

For example, let H be the 4-node graph shown in Figure 10.1. Its adjacency matrix A_H is the 4×4 matrix:

$$A_H = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 1 & 1 \\ c & 0 & 1 & 0 & 0 \\ d & 0 & 0 & 1 & 0 \end{array}$$

A payoff of this representation is that we can use matrix powers to count numbers of walks between vertices. For example, there are two length two walks between vertices a and c in the graph H:

$$a\ \langle a \to b \rangle\ b\ \langle b \to c \rangle\ c$$
$$a\ \langle a \to d \rangle\ d\ \langle d \to c \rangle\ c$$

and these are the only length two walks from a to c. Also, there is exactly one length two walk from b to c and exactly one length two walk from c to c and from d to b, and these are the only length two walks in H. It turns out we could have read these counts from the entries in the matrix $(A_H)^2$:

$$(A_H)^2 = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & 0 & 0 & 2 & 1 \\ b & 0 & 1 & 1 & 0 \\ c & 0 & 0 & 1 & 1 \\ d & 0 & 1 & 0 & 0 \end{array}$$

More generally, the matrix $(A_G)^k$ provides a count of the number of length k walks between vertices in any digraph G as we'll now explain.

Definition 10.3.1. The length-k *walk counting matrix* for an n-vertex graph G is the $n \times n$ matrix C such that

$$C_{uv} ::= \text{the number of length-}k \text{ walks from } u \text{ to } v. \tag{10.4}$$

Notice that the adjacency matrix A_G is the length-1 walk counting matrix for G, and that $(A_G)^0$, which by convention is the identity matrix, is the length-0 walk counting matrix.

Theorem 10.3.2. *If C is the length-k walk counting matrix for a graph G, and D is the length-m walk counting matrix, then CD is the length $k + m$ walk counting matrix for G.*

According to this theorem, the square $(A_G)^2$ of the adjacency matrix is the length two walk counting matrix for G. Applying the theorem again to $(A_G)^2 A_G$ shows that the length-3 walk counting matrix is $(A_G)^3$. More generally, it follows by induction that

Corollary 10.3.3. *The length-k counting matrix of a digraph G is $(A_G)^k$, for all $k \in \mathbb{N}$.*

In other words, you can determine the number of length k walks between any pair of vertices simply by computing the kth power of the adjacency matrix!

That may seem amazing, but the proof uncovers this simple relationship between matrix multiplication and numbers of walks.

Proof of Theorem 10.3.2. Any length $(k+m)$ walk between vertices u and v begins with a length k walk starting at u and ending at some vertex w followed by a length m walk starting at w and ending at v. So the number of length $(k+m)$ walks from u to v that go through w at the kth step equals the number C_{uw} of length k walks from u to w, times the number D_{wv} of length m walks from w to v. We can get the total number of length $(k+m)$ walks from u to v by summing, over all possible vertices w, the number of such walks that go through w at the kth step. In other words,

$$\#\text{length } (k+m) \text{ walks from } u \text{ to } v = \sum_{w \in V(G)} C_{uw} \cdot D_{wv} \tag{10.5}$$

But the right-hand side of (10.5) is precisely the definition of $(CD)_{uv}$. Thus, CD is indeed the length-$(k+m)$ walk counting matrix. ■

10.3.1 Shortest Paths

The relation between powers of the adjacency matrix and numbers of walks is cool—to us math nerds at least—but a much more important problem is finding shortest paths between pairs of nodes. For example, when you drive home for vacation, you generally want to take the shortest-time route.

One simple way to find the lengths of all the shortest paths in an n-vertex graph G is to compute the successive powers of A_G one by one up to the $n-1$st, watching for the first power at which each entry becomes positive. That's because Theorem 10.3.2 implies that the length of the shortest path, if any, between u and v, that is, the distance from u to v, will be the smallest value k for which $(A_G)^k_{uv}$ is nonzero, and if there is a shortest path, its length will be $\leq n-1$. Refinements of this idea lead to methods that find shortest paths in reasonably efficient ways. The methods apply as well to weighted graphs, where edges are labelled with weights or costs and the objective is to find least weight, cheapest paths. These refinements

are typically covered in introductory algorithm courses, and we won't go into them any further.

10.4 Walk Relations

A basic question about a digraph is whether there is a way to get from one particular vertex to another. So for any digraph G we are interested in a binary relation G^*, called the *walk relation* on $V(G)$, where

$$u \; G^* \; v ::= \text{there is a walk in } G \text{ from } u \text{ to } v. \tag{10.6}$$

Similarly, there is a *positive walk relation*

$$u \; G^+ \; v ::= \text{there is a positive length walk in } G \text{ from } u \text{ to } v. \tag{10.7}$$

Definition 10.4.1. When there is a walk from vertex v to vertex w, we say that w is *reachable* from v, or equivalently, that v is *connected* to w.

10.4.1 Composition of Relations

There is a simple way to extend composition of functions to composition of relations, and this gives another way to talk about walks and paths in digraphs.

Definition 10.4.2. Let $R : B \to C$ and $S : A \to B$ be binary relations. Then the composition of R with S is the binary relation $(R \circ S) : A \to C$ defined by the rule

$$a \; (R \circ S) \; c ::= \; \exists b \in B. \, (a \; S \; b) \text{ AND } (b \; R \; c). \tag{10.8}$$

This agrees with the Definition 4.3.1 of composition in the special case when R and S are functions.[2]

Remembering that a digraph is a binary relation on its vertices, it makes sense to compose a digraph G with itself. Then if we let G^n denote the composition of G with itself n times, it's easy to check (see Problem 10.11) that G^n is the *length-n walk relation*:

$$a \; G^n \; b \quad \text{iff} \quad \text{there is a length } n \text{ walk in } G \text{ from } a \text{ to } b.$$

[2]The reversal of the order of R and S in (10.8) is not a typo. This is so that relational composition generalizes function composition. The value of function f composed with function g at an argument x is $f(g(x))$. So in the composition $f \circ g$, the function g is applied first.

This even works for $n = 0$, with the usual convention that G^0 is the *identity relation* $\text{Id}_{V(G)}$ on the set of vertices.[3] Since there is a walk iff there is a path, and every path is of length at most $|V(G)| - 1$, we now have[4]

$$G^* = G^0 \cup G^1 \cup G^2 \cup \ldots \cup G^{|V(G)|-1} = (G \cup G^0)^{|V(G)|-1}. \tag{10.9}$$

The final equality points to the use of repeated squaring as a way to compute G^* with $\log n$ rather than $n - 1$ compositions of relations.

10.5 Directed Acyclic Graphs & Scheduling

Some of the prerequisites of MIT computer science subjects are shown in Figure 10.6. An edge going from subject s to subject t indicates that s is listed in the catalogue as a direct prerequisite of t. Of course, before you can take subject t, you have to take not only subject s, but also all the prerequisites of s, and any prerequisites of those prerequisites, and so on. We can state this precisely in terms of the positive walk relation: if D is the direct prerequisite relation on subjects, then subject u has to be completed before taking subject v iff $u \; D^+ \; v$.

Of course it would take forever to graduate if this direct prerequisite graph had a positive length closed walk. We need to forbid such closed walks, which by Lemma 10.2.6 is the same as forbidding cycles. So, the direct prerequisite graph among subjects had better be *acyclic*:

Definition 10.5.1. A *directed acyclic graph (DAG)* is a directed graph with no cycles.

DAGs have particular importance in computer science. They capture key concepts used in analyzing task scheduling and concurrency control. When distributing a program across multiple processors, we're in trouble if one part of the program needs an output that another part hasn't generated yet! So let's examine DAGs and their connection to scheduling in more depth.

[3] The *identity relation* Id_A on a set A is the equality relation:

$$a \; \text{Id}_A \; b \quad \text{iff} \quad a = b,$$

for $a, b \in A$.

[4] Equation (10.9) involves a harmless abuse of notation: we should have written

$$\text{graph}(G^*) = \text{graph}(G^0) \cup \text{graph}(G^1) \ldots.$$

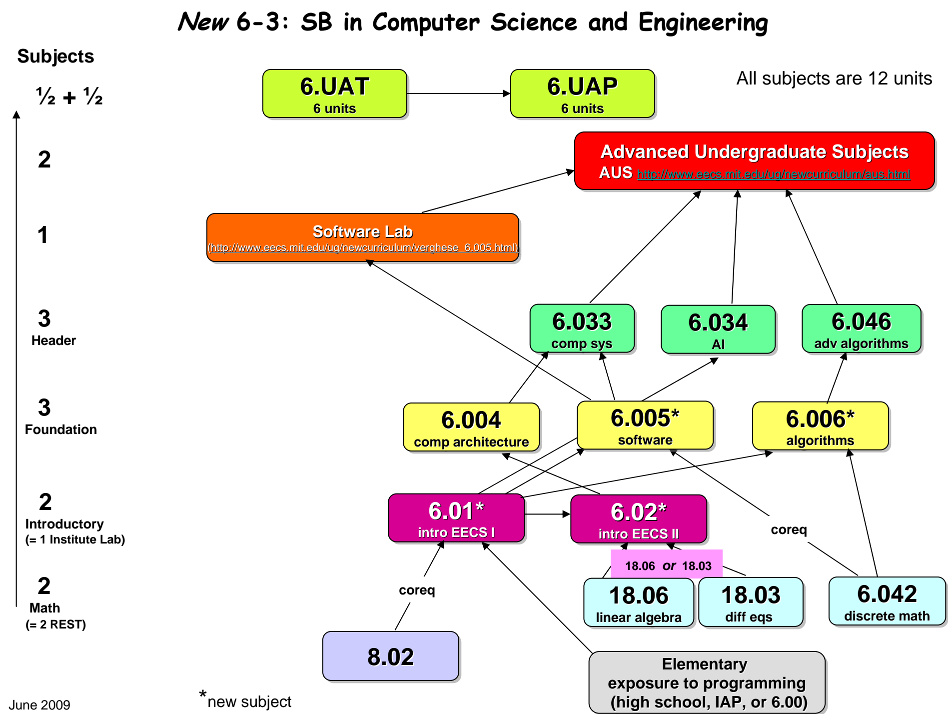

Figure 10.6 Subject prerequisites for MIT Computer Science (6-3) Majors.

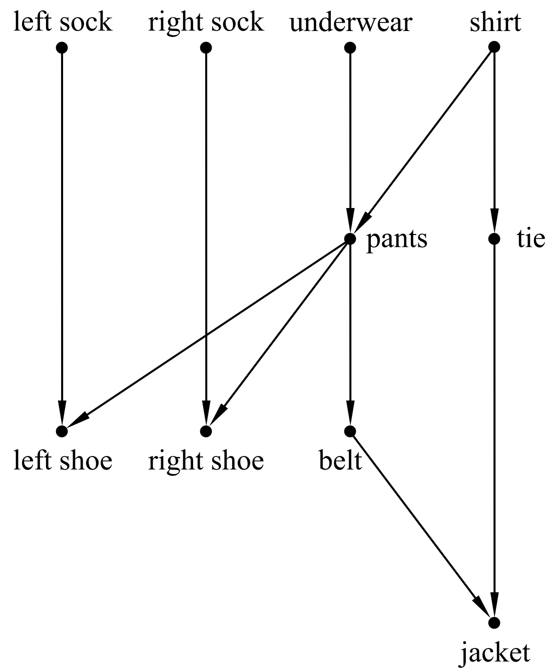

Figure 10.7 DAG describing which garments to put on before others.

10.5.1 Scheduling

In a scheduling problem, there is a set of tasks, along with a set of constraints specifying that starting certain tasks depends on other tasks being completed beforehand. We can map these sets to a digraph, with the tasks as the nodes and the direct prerequisite constraints as the edges.

For example, the DAG in Figure 10.7 describes how a man might get dressed for a formal occasion. As we describe above, vertices correspond to garments and edges specify which garments have to be put on before others.

When faced with a set of prerequisites like this one, the most basic task is finding an order in which to perform all the tasks, one at a time, while respecting the dependency constraints. Ordering tasks in this way is known as *topological sorting*.

Definition 10.5.2. A *topological sort* of a finite DAG is a list of all the vertices such that each vertex v appears earlier in the list than every other vertex reachable from v.

There are many ways to get dressed one item at a time while obeying the constraints of Figure 10.7. We have listed two such topological sorts in Figure 10.8. In

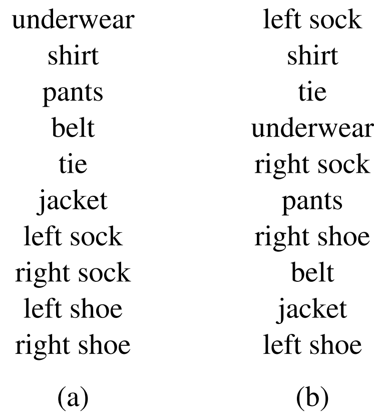

Figure 10.8 Two possible topological sorts of the prerequisites described in Figure 10.7

.

fact, we can prove that *every* finite DAG has a topological sort. You can think of this as a mathematical proof that you can indeed get dressed in the morning.

Topological sorts for finite DAGs are easy to construct by starting from *minimal* elements:

Definition 10.5.3. An vertex v of a DAG D is *minimum* iff every other vertex is reachable from v.

A vertex v is *minimal* iff v is not reachable from any other vertex.

It can seem peculiar to use the words "minimum" and "minimal" to talk about vertices that start paths. These words come from the perspective that a vertex is "smaller" than any other vertex it connects to. We'll explore this way of thinking about DAGs in the next section, but for now we'll use these terms because they are conventional.

One peculiarity of this terminology is that a DAG may have no minimum element but lots of minimal elements. In particular, the clothing example has four minimal elements: leftsock, rightsock, underwear, and shirt.

To build an order for getting dressed, we pick one of these minimal elements—say, shirt. Now there is a new set of minimal elements; the three elements we didn't chose as step 1 are still minimal, and once we have removed shirt, tie becomes minimal as well. We pick another minimal element, continuing in this way until all elements have been picked. The sequence of elements in the order they were picked will be a topological sort. This is how the topological sorts above were constructed.

So our construction shows:

Theorem 10.5.4. *Every finite DAG has a topological sort.*

There are many other ways of constructing topological sorts. For example, instead of starting from the minimal elements at the beginning of paths, we could build a topological sort starting from *maximal* elements at the end of paths. In fact, we could build a topological sort by picking vertices arbitrarily from a finite DAG and simply inserting them into the list wherever they will fit.[5]

10.5.2 Parallel Task Scheduling

For task dependencies, topological sorting provides a way to execute tasks one after another while respecting those dependencies. But what if we have the ability to execute more than one task at the same time? For example, say tasks are programs, the DAG indicates data dependence, and we have a parallel machine with lots of processors instead of a sequential machine with only one. How should we schedule the tasks? Our goal should be to minimize the total *time* to complete all the tasks. For simplicity, let's say all the tasks take the same amount of time and all the processors are identical.

So given a finite set of tasks, how long does it take to do them all in an optimal parallel schedule? We can use walk relations on acyclic graphs to analyze this problem.

In the first unit of time, we should do all minimal items, so we would put on our left sock, our right sock, our underwear, and our shirt.[6] In the second unit of time, we should put on our pants and our tie. Note that we cannot put on our left or right shoe yet, since we have not yet put on our pants. In the third unit of time, we should put on our left shoe, our right shoe, and our belt. Finally, in the last unit of time, we can put on our jacket. This schedule is illustrated in Figure 10.9.

The total time to do these tasks is 4 units. We cannot do better than 4 units of time because there is a sequence of 4 tasks that must each be done before the next. We have to put on a shirt before pants, pants before a belt, and a belt before a jacket. Such a sequence of items is known as a *chain*.

Definition 10.5.5. Two vertices in a DAG are *comparable* when one of them is reachable from the other. A *chain* in a DAG is a set of vertices such that any two of them are comparable. A vertex in a chain that is reachable from all other vertices in the chain is called a *maximum element* of the chain. A finite chain is said to *end at* its maximum element.

[5]Topolgical sorts can be generalized and shown to exist even for infinite DAGs, but you'll be relieved to know that we have no need to go into this.

[6]Yes, we know that you can't actually put on both socks at once, but imagine you are being dressed by a bunch of robot processors and you are in a big hurry. Still not working for you? Ok, forget about the clothes and imagine they are programs with the precedence constraints shown in Figure 10.7.

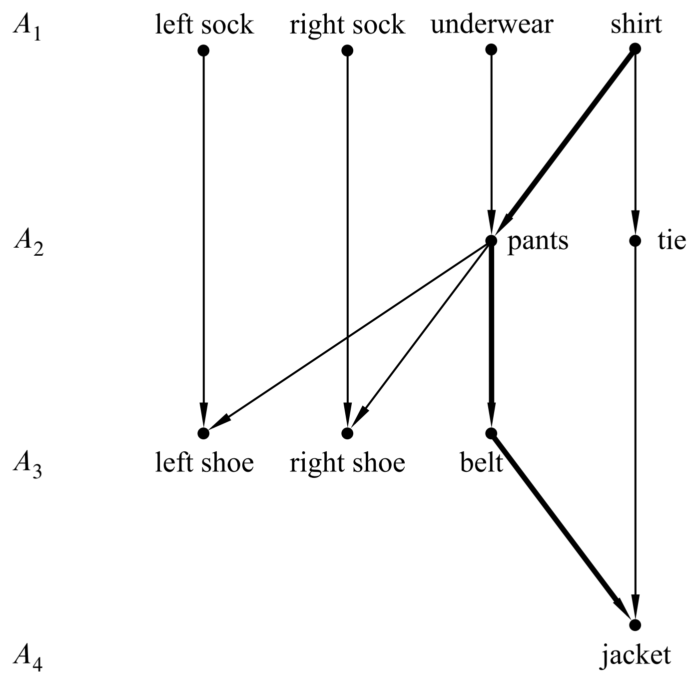

Figure 10.9 A parallel schedule for the tasks-getting-dressed digraph in Figure 10.7. The tasks in A_i can be performed in step i for $1 \leq i \leq 4$. A chain of 4 tasks (the critical path in this example) is shown with bold edges.

The time it takes to schedule tasks, even with an unlimited number of processors, is at least as large as the number of vertices in any chain. That's because if we used less time than the size of some chain, then two items from the chain would have to be done at the same step, contradicting the precedence constraints. For this reason, a *largest* chain is also known as a *critical path*. For example, Figure 10.9 shows the critical path for the getting-dressed digraph.

In this example, we were able to schedule all the tasks with t steps, where t is the size of the largest chain. A nice feature of DAGs is that this is always possible! In other words, for any DAG, there is a legal parallel schedule that runs in t total steps.

In general, a *schedule* for performing tasks specifies which tasks to do at successive steps. Every task a has to be scheduled at some step, and all the tasks that have to be completed before task a must be scheduled for an earlier step.

Let's be precise about the definition of schedule.

Definition 10.5.6. A *partition* of a set A is a set of nonempty subsets of A called the *blocks*[7] of the partition, such that every element of A is in exactly one block.

For example, one possible partition of the set $\{a, b, c, d, e\}$ into three blocks is

$$\{a, c\} \qquad \{b, e\} \qquad \{d\}.$$

Definition 10.5.7. A *parallel schedule* for a DAG D is a partition of $V(D)$ into blocks $A_0, A_1, \ldots$, such that when $j < k$, no vertex in A_j is reachable from any vertex in A_k. The block A_k is called the set of elements *scheduled at step* k, and the *time* of the schedule is the number of blocks. The maximum number of elements scheduled at any step is called the *number of processors* required by the schedule.

A *largest* chain ending at an element a is called a *critical path* to a, and the number of elements less than a in the chain is called the *depth* of a. So in any possible parallel schedule, there must be at least depth (a) steps before task a can be started. In particular, the minimal elements are precisely the elements with depth 0.

There is a very simple schedule that completes every task in its minimum number of steps: just use a "greedy" strategy of performing tasks as soon as possible. Schedule all the elements of depth k at step k. That's how we found the above schedule for getting dressed.

Theorem 10.5.8. *A minimum time schedule for a finite DAG D consists of the sets $A_0, A_1, \ldots$, where*

$$A_k ::= \{a \in V(D) \mid \text{depth}\,(a) = k\}.$$

[7] We think it would be nicer to call them the *parts* of the partition, but "blocks" is the standard terminology.

We'll leave to Problem 10.25 the proof that the sets A_k are a parallel schedule according to Definition 10.5.7. We can summarize the story above in this way: with an unlimited number of processors, the parallel time to complete all tasks is simply the size of a critical path:

Corollary 10.5.9. *Parallel time = size of critical path.*

Things get more complex when the number of processors is bounded; see Problem 10.26 for an example.

10.5.3 Dilworth's Lemma

Definition 10.5.10. An *antichain* in a DAG is a set of vertices such that *no* two elements in the set are comparable—no walk exists between any two different vertices in the set.

Our conclusions about scheduling also tell us something about antichains.

Corollary 10.5.11. *In a DAG D if the size of the largest chain is t, then $V(D)$ can be partitioned into t antichains.*

Proof. Let the antichains be the sets $A_k ::= \{a \in V(D) \mid \text{depth}(a) = k\}$. It is an easy exercise to verify that each A_k is an antichain (Problem 10.25). ■

Corollary 10.5.11 implies[8] a famous result about acyclic digraphs:

Lemma 10.5.12 (Dilworth)**.** *For all $t > 0$, every DAG with n vertices must have either a chain of size greater than t or an antichain of size at least n/t.*

Proof. Assume that there is no chain of size greater than t. Let ℓ be the size of the largest antichain. If we make a parallel schedule according to the proof of Corollary 10.5.11, we create a number of antichains equal to the size of the largest chain, which is less than or equal t. Each element belongs to exactly one antichain, none of which are larger than ℓ. So the total number of elements is at most ℓ times t—that is, $\ell t \geq n$. Simple division implies that $\ell \geq n/t$. ■

Corollary 10.5.13. *Every DAG with n vertices has a chain of size greater than $\sqrt{n}$ or an antichain of size at least $\sqrt{n}$.*

Proof. Set $t = \sqrt{n}$ in Lemma 10.5.12. ■

Example 10.5.14. When the man in our example is getting dressed, $n = 10$.

Try $t = 3$. There is a chain of size 4.

Try $t = 4$. There is no chain of size 5, but there is an antichain of size $4 \geq 10/4$.

[8] Lemma 10.5.12 also follows from a more general result known as Dilworth's Theorem, which we will not discuss.