5 Directed Acyclic Graphs

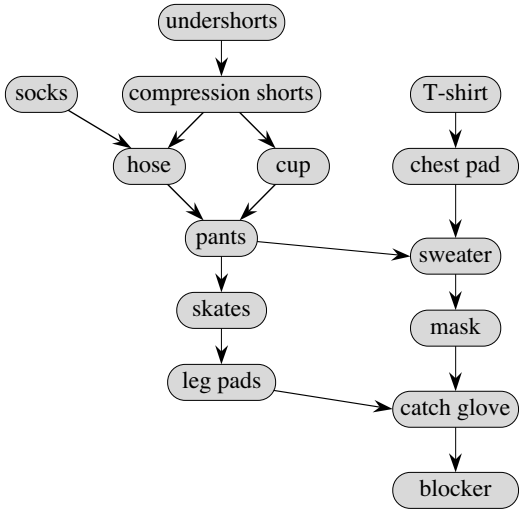
Recall the footnote on page 1, where I revealed that I used to play hockey. For several years, I was a goalie, but eventually my game deteriorated to the point that I couldn't stand watching myself play. It seemed as though every shot found its way to the back of the net. Then, after a hiatus of over seven years, I got back between the pipes (that is, I resumed playing goal) for a couple of games.

My biggest concern wasn't whether I'd be any good—I knew I was going to be terrible—but rather whether I'd remember how to put on all the goalie equipment. In ice hockey, goalies wear a lot of gear (35 to 40 pounds of it), and when dressing for a game, I have to put it on in the right order. For example, because I am right-handed, I wear on my left hand an oversized mitt for catching pucks; it's called a catch glove. Once I've got the catch glove on, my left hand has no dexterity, and I cannot get an upper-body garment of any sort over it.

When I was preparing to don the goalie gear, I made myself a diagram showing which items had to be put on before other items. The diagram appears on the next page. An arrow from item A to item B indicates a constraint that A must be put on before B. For example, I have to put on the chest pad before the sweater. Of course, the "must be put on before" constraint is ***transitive***: if item A must be put on before item B, and item B must be put on before item C, then item A must be put on before item C. Therefore, I must put on the chest pad before the sweater, mask, catch glove, and blocker.

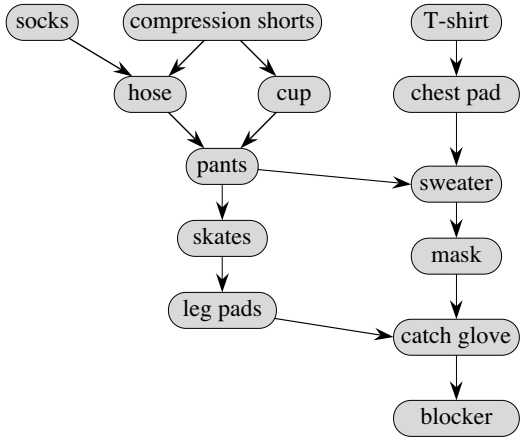
For some pairs of items, however, it doesn't matter in which order I put them on. I can put socks on either before or after the chest pad, for example.

I needed to determine an order in which to get dressed. Once I had my diagram, I had to come up with a list containing all the items I had to don, in a single order that did not violate any of the "must be put on before" constraints. I found that several orders worked; below the diagram are three of them.

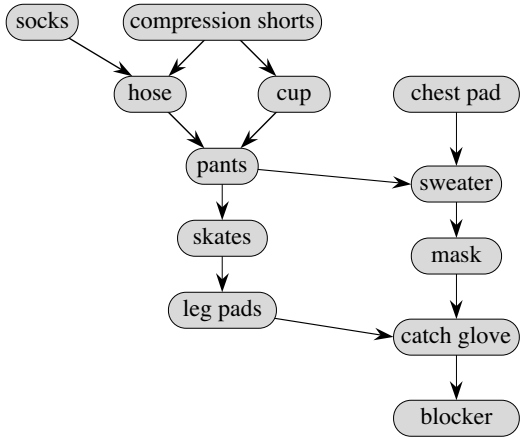


Order #1	Order #2	Order #3
undershorts	undershorts	socks
compression shorts	T-shirt	T-shirt
cup	compression shorts	undershorts
socks	cup	chest pad
hose	chest pad	compression shorts
pants	socks	hose
skates	hose	cup
leg pads	pants	pants
T-shirt	sweater	skates
chest pad	mask	leg pads
sweater	skates	sweater
mask	leg pads	mask
catch glove	catch glove	catch glove
blocker	blocker	blocker

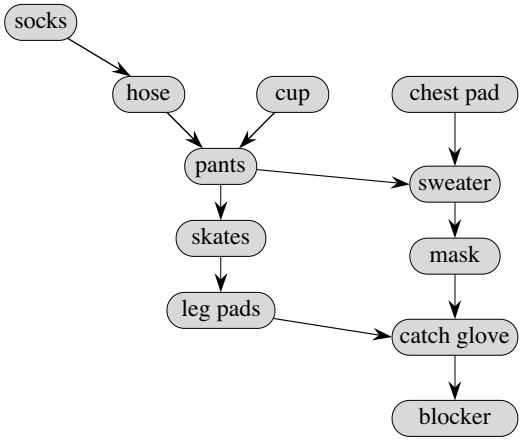
How did I arrive at these orders? Here's how I came up with order #2. I looked for an item that had no incoming arrows, because such an item need not be put on after any other item. I chose undershorts to be the first item in the order, and then, having (conceptually) put on the undershorts, I removed them from the diagram, resulting in the diagram at the top of the next page.



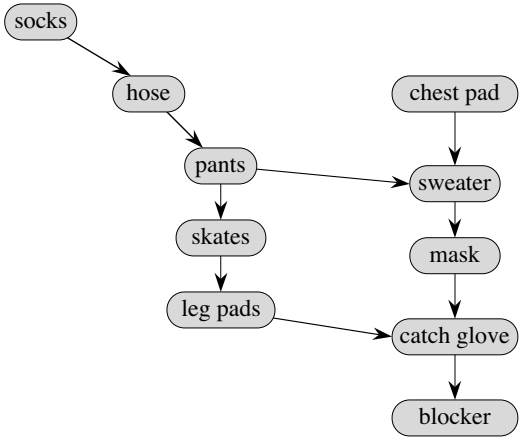
Then, again, I chose an item with no incoming arrows, this time T-shirt. I added it to the end of the order and removed it from the diagram, resulting in this diagram:



Once again, I chose an item with no incoming arrows—compression shorts—and then I added it to the end of the order and removed it from the diagram, resulting in the diagram at the top of the next page.



Next, I chose cup:



I kept going in this way—choosing an item with no incoming arrows, adding it to the end of the order, and removing the item from the diagram—until no items remained. The three orders shown on page 72 result from making various choices for the item with no incoming arrows, starting from the diagram on page 72.

Directed acyclic graphs

These diagrams are specific examples of ***directed graphs***, which are made of ***vertices*** (singular: ***vertex***), corresponding to the items of goalie equipment, and ***directed edges***, shown by arrows. Each directed edge is an ordered pair of the form (u, v), where u and v are vertices. For exam-

ple, the leftmost edge in the directed graph on page 72 is (socks, hose). When a directed graph contains a directed edge (u, v), we say that v is ***adjacent*** to u and that (u, v) ***leaves*** u and ***enters*** v, so that the vertex labeled hose is adjacent to the vertex labeled socks and the edge (socks, hose) leaves the vertex labeled socks and enters the vertex labeled hose.

The directed graphs that we have seen have another property: there is no way to get from a vertex back to itself by following a sequence of one or more edges. We call such a directed graph a ***directed acyclic graph***, or ***dag***. It's acyclic because there is no way to "cycle" from a vertex back to itself. (We'll see a more formal definition of a cycle later in this chapter.)

Dags are great for modeling dependencies where one task must occur before another. Another use for dags arises when planning projects, such as building a house: for example, the framing must be in place before the roof goes on. Or, in cooking, certain steps must occur in certain orders, but for some steps it doesn't matter in which order they happen; we'll see an example of a dag for cooking later in this chapter.

Topological sorting

When I needed to determine a single, linear order in which to put on the goalie equipment, I needed to perform a "topological sort." Put more precisely, a ***topological sort*** of a dag produces a linear order such that if (u, v) is an edge in the dag, then u appears before v in the linear order. Topological sorting differs from sorting in the sense that we used in Chapters 3 and 4.

The linear order produced by a topological sort is not necessarily unique. But you know that already, since each of the three orders for donning goalie equipment on page 72 could be produced by a topological sort.

Another use for topological sorting occurred at a programming job I had a long time ago. We were creating computer-aided design systems, and our systems could maintain a library of parts. Parts could contain other parts, but no circular dependencies were allowed: a part could not eventually contain itself. We needed to write out the part designs to tape (I *said* that the job was a long time ago) so that each part preceded any other parts that contained it. If each part is a vertex and an edge (u, v) indicates that part v contains part u, then we needed to write the parts according to a topologically sorted linear order.

What vertex would be a good candidate to be the first one in the linear order? Any vertex with no entering edges would do. The number of edges entering a vertex is the vertex's ***in-degree***, and so we could start with any vertex whose in-degree is 0. Fortunately, every dag must have at least one vertex with in-degree 0 and at least one vertex with ***out-degree*** 0 (no edges leaving the vertex), for otherwise there would be a cycle.

So suppose we choose any vertex with in-degree 0—let's call it vertex u—and put it at the beginning of the linear order. Because we have taken care of vertex u first, all other vertices will be placed after u in the linear order. In particular, any vertex v that is adjacent to u must appear somewhere after u in the linear order. Therefore, we can safely remove u and all edges leaving u from the dag, knowing that we've taken care of the dependencies that these edges define. When we remove a vertex and the edges that leave it from a dag, what are we left with? Another dag! After all, we cannot create a cycle by removing a vertex and edges. And so we can repeat the process with the dag that remains, finding some vertex with in-degree 0, placing it after vertex u in the linear order, removing edges, and so on.

The procedure on the next page for topological sorting uses this idea, but instead of actually removing vertices and edges from the dag, it just keeps track of the in-degree of each vertex, decrementing the in-degree for each entering edge that we conceptually remove. Since array indices are integers, let's assume that we identify each vertex by a unique integer in the range 1 to n. Because the procedure needs to quickly identify some vertex with in-degree 0, it maintains the in-degree of each vertex in an array *in-degree* indexed by the vertices, and it maintains a list *next* of all the vertices with in-degree 0. Steps 1–3 initialize the *in-degree* array, step 4 initializes *next*, and step 5 updates *in-degree* and *next* as vertices and edges are conceptually removed. The procedure can choose any vertex in *next* as the next one to put into the linear order.

Let's see how the first few iterations of step 5 work on the dag for putting on goalie equipment. In order to run the TOPOLOGICAL-SORT procedure on this dag, we need to number the vertices, as shown on page 78. Only vertices 1, 2, and 9 have in-degree 0, and so as we enter the loop of step 5, the list *next* contains only these three vertices. To get order #1 on page 72, the order of the vertices in *next* would be 1, 2, 9. Then, in the first iteration of step 5's loop, we choose vertex 1 (undershorts) as vertex u, delete it from *next*, add this vertex to the

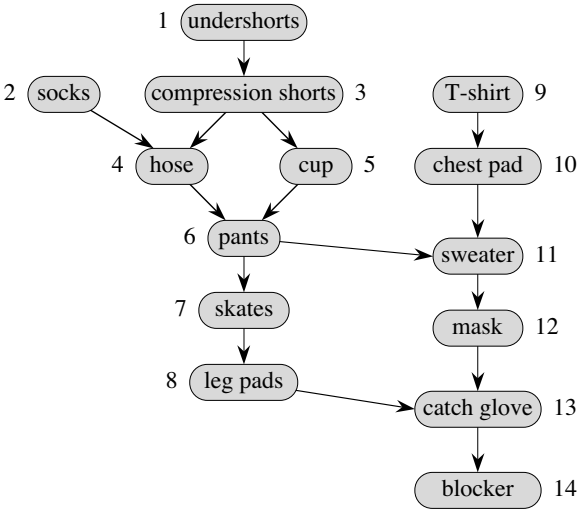
Procedure TOPOLOGICAL-SORT(G)

Input: G: a directed acyclic graph with vertices numbered 1 to n.

Output: A linear order of the vertices such that u appears before v in the linear order if (u, v) is an edge in the graph.

1. Let *in-degree*$[1\,..\,n]$ be a new array, and create an empty linear order of vertices.
2. Set all values in *in-degree* to 0.
3. For each vertex u:
   A. For each vertex v adjacent to u:
      i. Increment *in-degree*$[v]$.
4. Make a list *next* consisting of all vertices u such that *in-degree*$[u] = 0$.
5. While *next* is not empty, do the following:
   A. Delete a vertex from *next*, and call it vertex u.
   B. Add u to the end of the linear order.
   C. For each vertex v adjacent to u:
      i. Decrement *in-degree*$[v]$.
      ii. If *in-degree*$[v] = 0$, then insert v into the *next* list.
6. Return the linear order.

end of the initially empty linear order, and then decrement *in-degree*[3] (compression shorts). Because that operation takes *in-degree*[3] down to 0, we insert vertex 3 into *next*. Let's assume that when we insert a vertex into *next*, we insert it as the first vertex on the list. Such a list, where we always insert and delete at the same end, is known as a ***stack***, because it's like a stack of plates, where you always take a plate from the top and place a new plate at the top. (We call this order ***last in, first out***, or ***LIFO***.) Under this assumption, *next* becomes 3, 2, 9 and in the next loop iteration, we choose vertex 3 as vertex u. We delete it from *next*, add it to the end of the linear order, so that the linear order now reads "undershorts, compression shorts," and we decrement *in-degree*[4] (from 2 down to 1) and *in-degree*[5] (from 1 down to 0). We insert vertex 5 (cup) into *next*, resulting in *next* becoming 5, 2, 9. In the next iteration, we choose vertex 5 as vertex u, delete it from *next*, add it to the end of the linear order (now "undershorts, compression



shorts, cup"), and decrement *in-degree*[6], taking it down from 2 to 1. No vertices are added to *next* this time, and so in the next iteration, we choose vertex 2 as vertex u, and so on.

In order to analyze the TOPOLOGICAL-SORT procedure, we first have to understand how to represent a directed graph and a list such as *next*. When representing a graph, we won't require it to be acyclic, because the absence or presence of cycles has no effect on how we represent a graph.

How to represent a directed graph

In a computer, we can represent a directed graph in a few ways. Our convention will be that a graph has n vertices and m edges. We continue to assume that each vertex has its own number from 1 to n, so that we can use a vertex as an index into an array, or even as the row or column number of a matrix.

For now, we just want to know which vertices and edges are present. (Later on, we'll also associate a numeric value with each edge.) We could use an $n \times n$ ***adjacency matrix*** in which each row and each column corresponds to one vertex, and the entry in the row for vertex u and the column for vertex v is either 1 if the edge (u, v) is present or 0 if the graph does not contain edge (u, v). Since an adjacency matrix has n^2 entries, it must be true that $m \leq n^2$. Alternatively, we could just keep a list of all m edges in the graph in no particular order. As

a hybrid between an adjacency matrix and an unordered list, we have the ***adjacency-list representation***, with an n-element array indexed by the vertices in which the array entry for each vertex u is a list of all the vertices adjacent to u. Altogether, the lists have m vertices, since there's one list item for each of the m edges. Here are the adjacency-matrix and adjacency-list representations for the directed graph on page 78:

Adjacency matrix

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Adjacency lists

1	3
2	4
3	4, 5
4	6
5	6
6	7, 11
7	8
8	13
9	10
10	11
11	12
12	13
13	14
14	(none)

The unordered list of edges and the adjacency-list representation lead to the question of how to represent a list. The best way to represent a list depends on what types of operations we need to perform on the list. For unordered edge lists and adjacency lists, we know in advance how many edges will be in each list, and the contents of the lists won't change, and so we can store each list in an array. We can also use an array to store a list even if the list's contents change over time, as long as we know the maximum number of items that will ever be in the list at any one time. If we don't need to insert an item into the middle of the list or delete an item from the middle of the list, representing a list by an array is as efficient as any other means.

If we do need to insert into the middle of the list, then we can use a ***linked list***, in which each list item includes the location of its successor item in the list, making it simple to splice in a new item after a given item. If we also need to delete from the middle of the list, then each item in the linked list should also include the location of its predecessor item, so that we can quickly splice out an item. From now on, we will assume that we can insert into or delete from a linked list in constant

time. A linked list that has only successor links is a ***singly linked list***. Adding predecessor links makes a ***doubly linked list***.

Running time of topological sorting

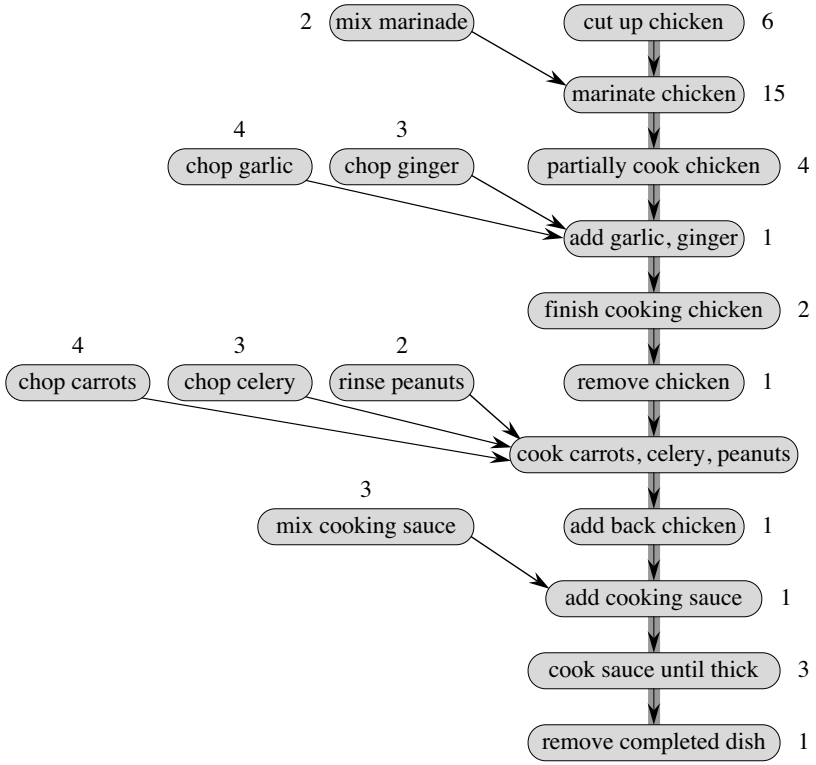
If we assume that the dag uses the adjacency-list representation and the *next* list is a linked list, then we can show that the TOPOLOGICAL-SORT procedure takes $\Theta(n+m)$ time. Since *next* is a linked list, we can insert into it or delete from it in constant time. Step 1 takes constant time, and because the *in-degree* array has n elements, step 2 initializes the array to all 0s in $\Theta(n)$ time. Step 3 takes $\Theta(n+m)$ time. The $\Theta(n)$ term in step 3 arises because the outer loop examines each of the n vertices, and the $\Theta(m)$ term is because the inner loop of step 3A visits each of the m edges exactly once over all iterations of the outer loop. Step 4 takes $O(n)$ time, since the *next* list starts with at most n vertices. Most of the work occurs in step 5. Because each vertex is inserted into *next* exactly once, the main loop iterates n times. Steps 5A and 5B take constant time in each iteration. Like step 3A, the loop of step 5C iterates m times altogether, once per edge. Steps 5Ci and 5Cii take constant time per iteration, so that all iterations together of step 5C take $\Theta(m)$ time, and therefore the loop of step 5 takes $\Theta(n+m)$ time. Of course, step 6 takes constant time, and so when we add up the time for all the steps, we get $\Theta(n+m)$.

Critical path in a PERT chart

I like to relax after a day at work by cooking, and I always enjoy cooking and eating kung pao chicken. I have to prepare the chicken, chop vegetables, mix a marinade and cooking sauce, and cook the dish. Just as when I put on goalie equipment, some steps must occur before others, and so I can use a dag to model the procedure for cooking kung pao chicken. The dag appears on the next page.

Next to each vertex in the dag appears a number, indicating how many minutes I need to perform the task corresponding to the vertex. For example, I take four minutes to chop the garlic (because I peel each clove first, and I use a *lot* of garlic). If you add up the times for all the tasks, you can see that if I were to perform them in sequence, it would take me an hour to make kung pao chicken.

If I had help, however, we could perform several of the tasks simultaneously. For example, one person could mix the marinade while some-



one else cut up the chicken. Given enough people helping, and sufficient space, knives, cutting boards, and bowls, we could perform many of the tasks simultaneously. If you look at any two tasks in the dag and find that there is no way to follow arrows to get from one to the other, then I could assign each of the tasks to a different person and have them done simultaneously.

Given unlimited resources (people, space, cooking equipment) to perform tasks simultaneously, how quickly can we make kung pao chicken? The dag is an example of a ***PERT chart***, an acronym for "program evaluation and review technique." The time to complete the entire job, even with as many tasks performed simultaneously as possible, is given by the "critical path" in the PERT chart. To understand what a critical path is, we first have to understand what a path is, and then we can define a critical path.

A ***path*** in a graph is a sequence of vertices and edges that allow you to get from one vertex to another (or back to itself); we say that the path

contains both the vertices on it and the edges traversed. For example, one path in the dag for kung pao chicken has, in order, the vertices labeled "chop garlic," "add garlic, ginger," "finish cooking chicken," and "remove chicken," along with the edges connecting these vertices. A path from a vertex back to itself is a ***cycle***, but of course dags do not have cycles.

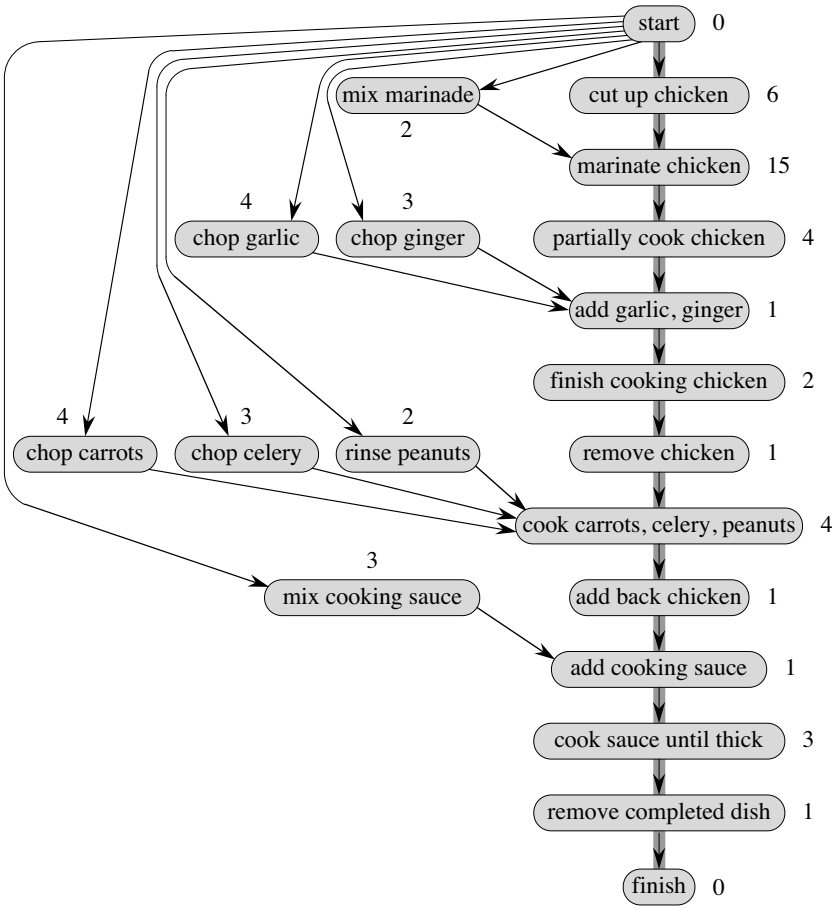
A ***critical path*** in a PERT chart is a path for which the sum of the task times is maximum over all paths. The sum of the task times along a critical path gives the minimum possible time for the entire job, no matter how many tasks are performed simultaneously. I shaded the critical path in the PERT chart for cooking kung pao chicken. If you add up the task times along the critical path, you'll see that no matter how much help I have, it takes me at least 39 minutes to make kung pao chicken.[1]

Assuming that all task times are positive, a critical path in a PERT chart must start at some vertex with in-degree 0 and end at some vertex with out-degree 0. Rather than checking paths between all pairs of vertices in which one has in-degree 0 and one has out-degree 0, we can just add two "dummy" vertices, "start" and "finish," as in the figure on the next page. Because these are dummy vertices, we give them task times of 0. We add an edge from start to each vertex with in-degree 0 in the PERT chart, and we add an edge from each vertex with out-degree 0 to finish. Now the only vertex with in-degree 0 is start, and the only vertex with out-degree 0 is finish. A path from start to finish with the maximum sum of task times on its vertices (shaded) gives a critical path in the PERT chart—minus the dummy vertices start and finish, of course.

Once we have added the dummy vertices, we find a critical path by finding a shortest path from start to finish, based on the task times. At this point, you might think I made an error in the previous sentence, because a critical path should correspond to a longest path, not a shortest path. Indeed, it does, but because a PERT chart has no cycles, we can alter the task times so that a shortest path gives us a critical path. In particular, we negate each task time and find a path from start to finish with the *minimum* sum of task times.

Why negate task times and find a path with the minimum sum of task times? Because solving this problem is a special case of finding short-

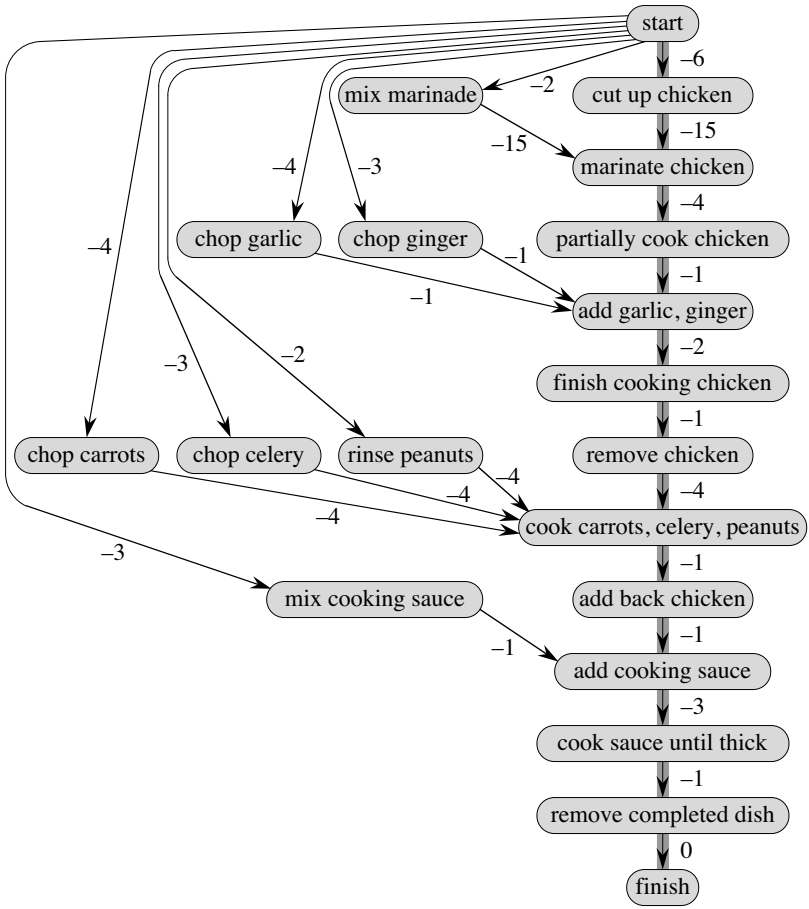
[1] If you're wondering why Chinese restaurants can turn out an order of kung pao chicken in much less time, it's because they prepare many of the ingredients in advance, and their commercial stoves can cook faster than my residential-grade stove.



est paths, and we have plenty of algorithms for finding shortest paths. When we talk about shortest paths, however, the values that determine path lengths are associated with edges, not with vertices. We call the value that we associate with each edge its ***weight***. A directed graph with edge weights is a ***weighted directed graph***. "Weight" is a generic term for values associated with edges. If a weighted directed graph represents a road network, each edge represents one direction of a road between two intersections, and the weight of an edge could represent the road's length, the time required to travel the road, or the toll a vehicle pays to use the road. The ***weight of a path*** is the sum of the weights of the edges on the path, so that if edge weights represent road distances, the weight of a path might indicate the total distance traveled along the roads on

the path. A ***shortest path*** from vertex u to vertex v is a path whose sum of edge weights is minimum over all paths from u to v. Shortest paths are not necessarily unique, as a directed graph from u to v could contain multiple paths whose weights achieve the minimum.

To convert a PERT chart with negated task times into a weighted directed graph, we push the negated task time for each vertex onto each of its entering edges. That is, if vertex v has a (non-negated) task time of t, we set the weight of each edge (u, v) entering v to be $-t$. Here's the dag we get, with edge weights appearing next to their edges:



Now we just have to find a shortest path (shaded) from start to finish in this dag, based on these edge weights. A critical path in the original

PERT chart will correspond to the vertices on the shortest path we find, minus start and finish. So let's see how to find a shortest path in a dag.

Shortest path in a directed acyclic graph

There is another advantage to learning how to find a shortest path in a dag: we'll lay the foundations for finding shortest paths in arbitrary directed graphs that may have cycles. We'll examine this more general problem in Chapter 6. As we did for topologically sorting a dag, we'll assume that the dag is stored with the adjacency-list representation, and that with each edge (u, v) we have also stored its weight as $weight(u, v)$.

In a dag that we derive from a PERT chart, we want a shortest path from the ***source vertex***, which we call "start," to a specific ***target vertex***, "finish." Here, we'll solve the more general problem of finding ***single-source shortest paths***, where we find shortest paths from a source vertex to *all* other vertices. By convention, we will name the source vertex s, and we want to compute two things for each vertex v. First, the weight of a shortest path from s to v, which we denote by $sp(s, v)$. Second, the ***predecessor*** of v on a shortest path from s to v: a vertex u such that a shortest path from s to v is a path from s to u and then a single edge (u, v). We will number the n vertices from 1 to n, so that our algorithms for shortest paths here and in Chapter 6 can store these results in arrays $shortest[1..n]$ and $pred[1..n]$, respectively. As the algorithms unfold, the values in $shortest[v]$ and $pred[v]$ might not be their correct final values, but when the algorithms are done, they will be.

We need to handle a couple of cases that can arise. First, what if there is no path at all from s to v? Then we define $sp(s, v) = \infty$, so that $shortest[v]$ should come out to ∞. Since v would have no predecessor on a shortest path from s, we also say that $pred[v]$ should be the special value NULL. Moreover, all shortest paths from s start with s, and so s has no predecessor, either; thus we say that $pred[s]$ should also be NULL. The other case arises only in graphs that have both cycles and negative edge weights: what if the weight of a cycle is negative? Then we could go around the cycle forever, decreasing the path weight each time around. If we can get from s to a negative-weight cycle and then to v, then $sp(s, v)$ is undefined. For now, however, we're concerned only with acyclic graphs, and so there are no cycles, much less negative-weight cycles, to worry about.

To compute shortest paths from a source vertex s, we start off with $shortest[s] = 0$ (since we don't have to go anywhere to get from a vertex

to itself), $shortest[v] = \infty$ for all other vertices v (since we don't know in advance which vertices can be reached from s), and $pred[v] = \text{NULL}$ for all vertices v. Then we apply a series of ***relaxation steps*** to the edges of the graph:

Procedure RELAX(u, v)

Inputs: u, v: vertices such that there is an edge (u, v).

Result: The value of $shortest[v]$ might decrease, and if it does, then $pred[v]$ becomes u.

1. If $shortest[u] + weight(u, v) < shortest[v]$, then set $shortest[v]$ to $shortest[u] + weight(u, v)$ and set $pred[v]$ to u.

When we call RELAX(u, v), we are determining whether we can improve upon the current shortest path from s to v by taking (u, v) as the last edge. We compare the weight of the current shortest path to u plus the weight of edge (u, v) with the weight of the current shortest path to v. If it's better to take edge (u, v), then we update $shortest[v]$ to this new weight and we set v's predecessor on a shortest path to be u.

If we relax edges along a shortest path, in order, we get the right results. You might wonder how we can be assured of relaxing the edges in order along a shortest path when we don't even know what the path is—after all, that's what we're trying to find out—but it will turn out to be easy for a dag. We're going to relax all the edges in the dag, and the edges of each shortest path will be interspersed, in order, as we go through all the edges and relax each one.

Here's a more precise statement of how relaxing edges along a shortest path works, and it applies to any directed graph, with or without cycles:

> Start with $shortest[u] = \infty$ and $pred[u] = \text{NULL}$ for all vertices, except that $shortest[s] = 0$ for the source vertex s.
>
> Then relax the edges along a shortest path from s to any vertex v, *in order, starting from the edge leaving s and ending with the edge entering v*. Relaxations of other edges may be interspersed freely with the relaxations along this shortest path, but only relaxations may change any *shortest* or *pred* values.
>
> After the edges have been relaxed, v's *shortest* and *pred* values are correct: $shortest[v] = sp(s, v)$ and $pred[v]$ is the vertex preceding v on some shortest path from s.

It's pretty easy to see why relaxing the edges along a shortest path, in order, works. Suppose that a shortest path from s to v visits the vertices $s, v_1, v_2, v_3, \ldots, v_k, v$, in that order. After edge (s, v_1) has been relaxed, $shortest[v_1]$ must have the correct shortest-path weight for v_1, and $pred[v_1]$ must be s. After (v_1, v_2) has been relaxed, $shortest[v_2]$ and $pred[v_2]$ must be correct. And so on, up through relaxing (v_k, v), after which $shortest[v]$ and $pred[v]$ have their correct values.

This is great news. In a dag, it's really easy to relax each edge exactly once yet relax the edges along *every* shortest path, in order. How? First, topologically sort the dag. Then consider each vertex, taken in the topologically sorted linear order, and relax all the edges leaving the vertex. Since every edge must leave a vertex earlier in the linear order and enter a vertex later in the order, every path in the dag must visit vertices in an order consistent with the linear order.

Procedure DAG-SHORTEST-PATHS(G, s)

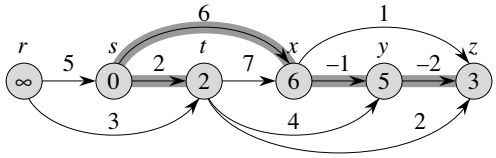
Inputs:

- G: a weighted directed acyclic graph containing a set V of n vertices and a set E of m directed edges.
- s: a source vertex in V.

Result: For each non-source vertex v in V, $shortest[v]$ is the weight $sp(s, v)$ of a shortest path from s to v and $pred[v]$ is the vertex preceding v on some shortest path. For the source vertex s, $shortest[s] = 0$ and $pred[s] =$ NULL. If there is no path from s to v, then $shortest[v] = \infty$ and $pred[v] =$ NULL.

1. Call TOPOLOGICAL-SORT(G) and set l to be the linear order of vertices returned by the call.
2. Set $shortest[v]$ to ∞ for each vertex v except s, set $shortest[s]$ to 0, and set $pred[v]$ to NULL for each vertex v.
3. For each vertex u, taken in the order given by l:
   A. For each vertex v adjacent to u:
      i. Call RELAX(u, v).

The next page shows a dag with weights appearing next to the edges. The *shortest* values from running DAG-SHORTEST-PATHS from source vertex s appear inside the vertices, and shaded edges indicate the *pred* values. The vertices are laid out left to right in the linear order returned by the topological sort, so that all edges go from left to right. If an



edge (u, v) is shaded, then $pred[v]$ is u and $shortest[v] = shortest[u] + weight(u, v)$; for example, since (x, y) is shaded, $pred[y] = x$ and $shortest[y]$ (which is 5) equals $shortest[x]$ (which is 6) + $weight(x, y)$ (which is -1). There is no path from s to r, and so $shortest[r] = \infty$ and $pred[r] =$ NULL (no shaded edges enter r).

The first iteration of the loop of step 3 relaxes edges (r, s) and (r, t) leaving r, but because $shortest[r] = \infty$, these relaxations do not change anything. The next iteration of the loop relaxes edges (s, t) and (s, x) leaving s, causing $shortest[t]$ to be set to 2, $shortest[x]$ to be set to 6, and both $pred[t]$ and $pred[x]$ to be set to s. The following iteration relaxes edges (t, x), (t, y), and (t, z) leaving t. The value of $shortest[x]$ does not change, since $shortest[t] + weight(t, x)$, which is $2 + 7 = 9$, is greater than $shortest[x]$, which is 6; but $shortest[y]$ becomes 6, $shortest[z]$ becomes 4, and both $pred[y]$ and $pred[z]$ are set to t. The next iteration relaxes edges (x, y) and (x, z) leaving x, causing $shortest[y]$ to become 5 and $pred[y]$ to be set to x; $shortest[z]$ and $pred[z]$ remain unchanged. The final iteration relaxes edge (y, z) leaving y, causing $shortest[z]$ to become 3 and $pred[z]$ to be set to y.

You can easily see how DAG-SHORTEST-PATHS runs in $\Theta(n + m)$ time. As we saw, step 1 takes $\Theta(n + m)$ time, and of course step 2 initializes two values for each vertex and therefore takes $\Theta(n)$ time. As we've seen before, the outer loop of step 3 examines each vertex exactly once, and the inner loop of step 3A examines each edge exactly once over all iterations. Because each call of RELAX in step 3Ai takes constant time, step 3 takes $\Theta(n + m)$ time. Adding up the running times for the steps gives us the $\Theta(n + m)$ for the procedure.

Going back to PERT charts, it's now easy to see that finding a critical path takes $\Theta(n + m)$ time, where the PERT chart has n vertices and m edges. We add the two vertices, start and finish, and we add at most m edges leaving start and at most m edges entering finish, for a total of at most $3m$ edges in the dag. Negating the weights and pushing them from the vertices to the edges takes $\Theta(m)$ time, and then finding a shortest path through the resulting dag takes $\Theta(n + m)$ time.