Lecture 14

Critical Path Analysis

This lecture applies ideas about distance in weighted graphs to solve problems in the scheduling of large, complex projects.

Reading:
The topic is discussed in Section 3.5 of

> Dieter Jungnickel (2013), *Graphs, Networks and Algorithms*, 4th edition, which is available online via `SpringerLink`,

but it is such an important application that it is also treated in many other places.

14.1 Scheduling problems

Suppose you are planning a dinner that involves a number of dishes, some of which have multiple components or stages of preparation, say, a roast with sauces or a pie with pastry and filling that have to be prepared separately. Especially for a novice cook, it can be difficult to arrange the cooking so that all the food is ready at the same moment. Somewhat surprisingly, graph theory can help in this, as well as in many more complex scheduling problems.

The key abstraction is a certain kind of directed graph constructed from a list of tasks such as the one in Table 14.1, which breaks a large project down into a list of smaller tasks and, for each one, notes (a) how long it takes to complete and (b) which other tasks are its immediate prerequisites. Here, for example, task A might be "wash and peel all the vegetables" while D and E—which have A as a prerequisite—might be "assemble the salad" and "fry the garlic briefly over very high heat."

14.1.1 From tasks to weighted digraphs

Figure 14.1 shows a directed graph associated with the project summarised in Table 14.1. It has:

- a vertex for each task;

Task	Time to Complete	Prerequisites
A	10	None
B	12	None
C	15	None
D	6	A & C
E	3	A & B
F	5	B & C
G	7	D & F
H	6	D & E
I	9	E & F

Table 14.1: *Summary of a modestly-sized project. The first column lists various tasks required for the completion of the project, while the second column gives the time (in minutes) needed to complete each task and the third column gives each task's immediate prerequisites.*

- edges that run from prerequisites to the tasks that depend on them. Thus for example, there is a directed edge (A, D), as task D has task A as a prerequisite.
- There are also two extra vertices, one called S (for "start") that is a predecessor for all the tasks that have no prerequisites and another, Z, that corresponds to finishing the project and is a successor of all tasks that are not prerequisites for any other task.
- There are edge weights that correspond to the time it takes to complete the task at the *tail* vertex. Thus—as task A takes 10 minutes to complete—all the edges coming out of the vertex A have weight 10.

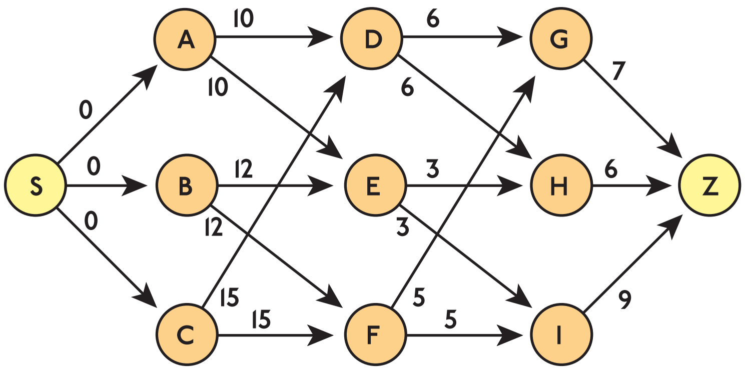

Figure 14.1: *The digraph associated with the scheduling problem from Table 14.1.*

Task	Prereq's
P: get a pink from	G
B: get a blue form	P
G: get a green form	B

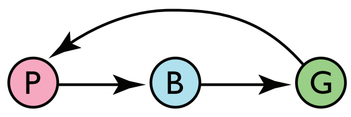

Figure 14.2: *An example showing why the digraph associated with a scheduling problem shouldn't contain cycles. It represents a bureaucratic nightmare in which one needs a pink form P in order to get a blue form B in order to get the green form G that one needs to get a pink form P.*

14.1.2 From weighted digraphs to schedules

Once we have a graph such as the one in Figure 14.1 we can answer a number of important questions about the project including:

(1) What is the shortest time in which we can complete the work?

(2) What is the earliest time (measured from the start of the project) at which we can start a given task?

(3) Are there any tasks whose late start would delay the whole project?

(4) For any tasks that don't need to be started as early as possible, how long can we delay their starts?

In the discussion that follows, we'll imagine that we have as many resources as we need (as many hands to help in the kitchen, as many employees and as much equipment as needed to pursue multiple tasks in parallel $\cdots$). In this setting, Lemma 14.1, proved below, provides a tool to answer all of these questions.

14.2 Graph-theoretic details

A directed graph representing a project that can actually be completed cannot contain any cycles. To see why, consider the graph in Figure 14.2. It tells us that we cannot start task G until we have completed its prerequisite, task B, which we cannot start before we complete its prerequisite, $P \cdots$ which we cannot start until we've completed G. This means we can never even start the project, much less finish it.

Thus any graph that describes a feasible project should be a directed, acyclic graph (often abbreviated *DAG*) with non-negative edge weights. From now on we'll restrict attention to such graphs and call them *task-dependency graphs*. We'll imagine that they are constructed from a project described by a list of tasks such as the one in Table 14.1 and that they look like the example in Figure 14.1. In particular, we'll require our graphs to have a *starting vertex* S which is the only vertex with $\deg_{in}(v) = 0$ and a *terminal vertex* Z, which is the only vertex with $\deg_{out}(v) = 0$.

Task	Time Needed	Prereq's
A	10 hours	None
B	6 hours	None

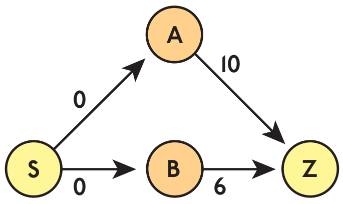

Figure 14.3: *An example showing why the shortest time in which one can complete a project corresponds to a* ***maximal-weight walk*** *from the start vertex S to the terminal vertex Z.*

14.2.1 Shortest times and maximal-weight paths

Now consider the very simple project illustrated in Figure 14.3. It involves just two tasks: A, which takes 10 hours to complete and B which takes 6 hours. Even if—as our assumptions allow—we start both tasks at the same time and work on them in parallel, the soonest we can possibly finish the project is 10 hours after we start. This is a special case of the following result, whose proof I'll only sketch briefly.

Lemma 14.1 (Shortest times and maximal weights)**.** *If $G(V, E, w)$ is a task-dependency graph that describes a scheduling problem, and if we start the work at $t = 0$, then the earliest time, t_v, at which we can start the task corresponding to vertex v is the weight of a maximal-weight walk from S to v.*

The proof of Lemma 14.1 turns on the observation that the times t_v satisfy equations that look similar to Bellman's Equations, except that they have a max() where Bellman's Equations have a min():

$$t_S = 0 \qquad \text{and} \qquad t_v = \max_{u \in P_v} (t_u + w(u, v)) \quad \forall\, v \neq S. \tag{14.1}$$

In the equation at right, P_v is v's predecessor list and $w(u, v)$ is the weight of the edge from u to v or, equivalently, the time it takes to complete the task corresponding to u.

Although the Bellman-like equations above provide an elegant characterisation of the t_v, they aren't necessarily all that practical as a way to *calculate* the t_v. The issue is that in order to use Eqn. (14.1) to compute t_v, we need t_u for all v's predecessors $u \in P_v$. And for each of them, we need t_w for $w \in P_u\ \cdots$, and so on. Fortunately this problem has a simple resolution in DAGs, as we'll see below. The idea is to find a clever way to organise the computations so that the results we need when computing t_v are certain to be available.

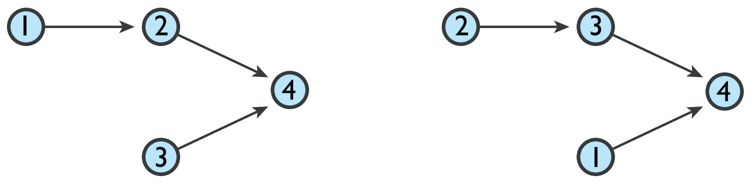

Figure 14.4: *A digraph with two distinct topological orderings.*

14.2.2 Topological ordering

Definition 14.2. *If $G(V, E)$ is a directed, acyclic graph with $|V| = n$, then a* ***topological ordering*** *(sometimes also called a* ***topological sorting****) of G is a map $\Phi : V \to \{1, 2, \dots, n\}$ with the properties that*

- $\Phi(v) = \Phi(u) \;\Rightarrow\; u = v$;
- $(u, v) \in E \;\Rightarrow\; \Phi(u) < \Phi(v)$.

In other words, a topological ordering is a way of numbering the vertices so that the graph's directed edges always point from a vertex with a smaller index to a vertex with a bigger one.

Topological orderings are not, in general, unique, as is illustrated in Figure 14.4, but as the following results show, a DAG always has at least one.

Lemma 14.3 (DAGs contain sink vertices)**.** *If $G(V, E)$ is a directed, acyclic graph then it contains at least one vertex v with $\deg_{out}(v) = 0$. Such a vertex is sometimes called a* ***sink vertex****.*

Proof of Lemma 14.3. Construct a walk through $G(V, E)$ as follows. First choose an arbitrary vertex $v_0 \in V$. If $\deg_{out}(v_0) = 0$ we are finished, but if not choose an arbitrary successor of v_0, $v_1 \in S_{v_0}$. If $\deg_{out}(v_1) = 0$ we are finished, but if not, choose an arbitrary successor of v_1, $v_2 \in S_{v_1} \cdots$ and so on. This construction can never revisit a vertex as G is acyclic. Further, as G has only finitely many vertices, the construction must come to a stop after at most $|V| - 1$ steps. But the only way for it to stop is to reach a vertex v_j such that $\deg_{out}(v_j) = 0$, which proves that such a vertex must exist. □

Theorem 14.4 (DAGs have topological orderings)**.** *A directed, acyclic graph $G(V, E)$ always has a topological ordering.*

Proof of Theorem 14.4. One can prove this by induction on the number of vertices. The base case is $|V| = 1$ and clearly, assigning the number 1 to the sole vertex gives a topological ordering.

Now suppose the result is true for all DAGs with $|V| \leq n_0$ and consider a DAG with $|V| = n_0 + 1$. Lemma 14.3 tells us that G contains a vertex w with $\deg_{out}(w) = 0$. Construct $G'(V', E') = G \backslash w$. It is a DAG (because G was one), but

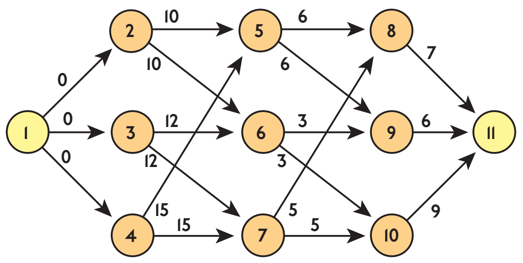

Figure 14.5: *A topological ordering for the digraph associated with the scheduling problem from Table 14.1 in which the vertex label v has been replaced by the value $\Phi(v)$ assigned by the ordering that's listed in Table 14.2.*

v	S	A	B	C	D	E	F	G	H	I	Z
$\Phi(v)$	1	2	3	4	5	6	7	8	9	10	11

Table 14.2: *The topological ordering illustrated in Figure 14.5.*

has only $|V'| = n_0$ vertices and so, by the inductive hypothesis, G' has a topological ordering $\Phi' : V' \to \{1, 2, \ldots, n_0\}$. We can extend this to a obtain a function $\Phi : V \to \{1, 2, \ldots, n_0 + 1\}$ by choosing

$$\Phi(v) = \begin{cases} \Phi'(v) & \text{if } v \neq w \\ n_0 + 1 & \text{if } v = w \end{cases}$$

Further, this Φ is clearly a topological ordering because all predecessors $u \in P_w$ of w have

$$\Phi(u) = \Phi'(u) \leq n_0 < n_0 + 1 = \Phi(w)$$

and, by construction, w has no successors. This concludes the inductive step and establishes that all DAGs have at least one topological ordering. □

14.3 Critical paths

Figure 14.5 shows a topological ordering for the graph from Figure 14.1. The reason we're interested in such orderings is that they provide a way to solve Eqns (14.1) in a task-dependency graph. By construction, the starting vertex S is the only one that has no predecessors and so any topological ordering must have $\Phi(S) = 1$. Similarly, the terminal vertex Z is the only one that has no successors, so for a project with n tasks, $\Phi(Z) = n + 2$.

By convention, we start the project at $t_S = 0$. If we then use Eqns (14.1) to compute the rest of the t_v, working through the vertex list in the order assigned by the topological ordering, it will always then be true that when we want to compute

$$t_v = \max_{u \in P_v} \left(t_u + w(u, v) \right),$$

we will have all the t_u for $u \in P_v$ available.

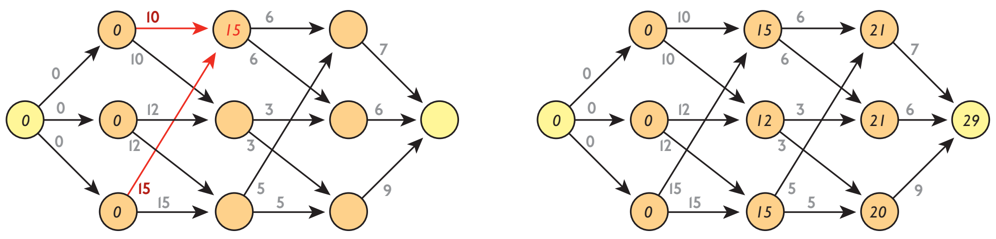

Figure 14.6: *In the graphs above the vertex labels have been replaced with values of t_j, the earliest times at which the corresponding task can start. The graph at left shows the edges that enter into Eqn.* (14.1) *for the computation of* t_4 *while the graph at right shows all the* t_j.

14.3.1 Earliest starts

For the digraph in Figure 14.5, we get

$$\begin{aligned}
t_S &= t_1 = 0 \\
t_A &= t_2 = t_1 + w(1,2) = 0 + 0 \\
&\vdots \\
t_D &= t_5 = \max(t_2 + w(2,5),\, t_4 + w(4,5)) = \max(0+10,\, 0+15) = 15 \qquad (14.2) \\
&\vdots
\end{aligned}$$

Figure 14.6 illustrates both the computation of t_4 and the complete set of t_j. As $t_Z = t_{11} = 29$, we can conclude that it takes a minimum of 29 minutes to complete the project.

14.3.2 Latest starts

The t_v that we computed in the previous section is the earliest time at which the task for vertx v *could* start, but it may be possible to delay the task without delaying the whole project. Consider, for example, task H in our main example. It could start as early as $t_H = t_9 = 21$, but since it only takes 6 minutes to complete, we could delay its start a bit without disrupting the project. If we define T_v to be the time by which task v *must* start if the project is not to be delayed, then it's clear that $T_H = T_Z - 6 = 23$. More generally, the latest time at which a task can start depends on the latest starts of its successors, so that

$$T_v = \min_{u \in S_v} \left(T_u - w(v,u) \right). \qquad (14.3)$$

This expression, along with the observation that $T_Z = t_Z$, allows us to find T_v for all tasks by working backwards through the DAG. Figure 14.7 illustrates this for our main example, while Table 14.3 lists t_v and T_v for all vertices $v \in V$.

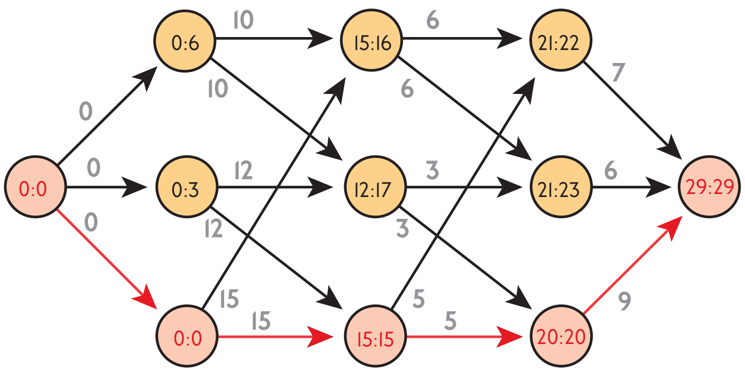

Figure 14.7: *Here a vertex v is labelled with a pair $t_v : T_v$ that shows both the earliest time t_v at which the corresponding task* ***could*** *start and T_v, the latest time by which the task* ***must*** *start if the whole project is not to be delayed. This project has only a single critical path, (S, C, F, I, Z), which is highlighted in red.*

v	S	A	B	C	D	E	F	G	H	I	Z
t_v	0	0	0	0	15	12	15	21	21	20	29
T_v	0	6	3	0	16	17	15	22	23	20	29

Table 14.3: *The earliest starts t_v and latest starts T_v for the main example.*

14.3.3 Critical paths

Notice that some of the vertices in Figure 14.7 have $t_v = T_v$. This happens because they lie on a maximal-weight path from S to Z and so a delay to any one of them will delay the whole project. Such maximal-weight paths play a crucial role in project management and so there is a term to describe them:

Definition 14.5 (Critical path)**.** *A maximal-weight path from S to Z in a task-dependency graph $G(V, E)$ is called a* ***critical path*** *and G may contain more than one of them.*

Vertices (and hence tasks) that lie off the critical path have $T_v > t_v$ and so do not require such keen supervision.