Scheduling

Introduction

It is natural to think that effort estimates for each task are all that are necessary to determine how long a project will take to complete. However, a little additional thought makes it readily apparent that this is not the case.

Units Matter

It is easy to be careless about units, especially since few programming languages take them into account and, as a result, programmers tend to just think about unitless numbers. However, it is important to realize that effort and time are not measured in the same units; effort is commonly measured in person-days or person-months and time is commonly measured in days or months.

Thinking only about the units, we can ask what needs to be done to convert person-days to days and the answer is obvious -- divide by the number of people working on the task. That is, thinking only about units

$$T = E / N$$

where T denote the time, E denotes the effort, and N denotes the number of people.

Task Details Matter

Unfortunately, the relationship between E and T described above does not hold in general. To see why, consider an extreme example unrelated to software engineering. Specifically, suppose the task is to thread a nut onto a that is 1 meter long, and suppose that the nut can be spun at a rate of 1 centimeter per second. Then, the effort required to complete the task is 100 person-seconds. In other words, one person would have to work for 100 seconds to complete the task. Now suppose 20 people are assigned to the task. The relationship above says that the task can be completed in 100 person-seconds / 20 persons = 5 seconds, but this is obviously ridiculous. There is no way 20 people could work on this task at the same time. Indeed, it's hard to imagine more than one person working on this task at a time. So, clearly, the details of the task matter.

Very small tasks can typically only be worked on by one person at a time. Hence, the relationship above is applicable only for $N = 1$, in which case the effort required to complete the task (in person-days) is equal to the time required to complete the task (in days).

Larger sized tasks often exhibit the proportional relationship described above for small value of N. However, for larger values of N the benefit of additional people declines and ultimately become negative. In other words, throwing more people at a task can actually make things worse.

Task Dependencies Matter

Even when we know all of the tasks and have a clear relationship between E and T for each one it is not easy to calculate the time required to complete a project that consists of multiple tasks dependencies between tasks. This can easily be illustrated using two examples.

At one extreme, it might be possible to work on all of the tasks simultaneously. In this case, if there are enough people available, the project will only take as long to complete as the longest task will take to complete. For example, if there are five tasks, requiring efforts of 4, 1, 3, 1, and 2 person months, and there are five people available to work on the project, then the project will take 4 months to complete. In addition, task 2 can slip 3 months (since it only requires 1 person-month of

effort and the longest task requires 4 person-months), task 2 can slip 1 month, task three can slip 3 months, and task 5 can slip 3 months, all without the project being delayed.

At the other extreme, it might not be possible to complete task $i + 1$ until task i has been completed. In this case, regardless of the number of people available, the project will take as long to complete as the sum of the times required for each task. Continuing with the same example, regardless of the number of people available, the project will take 4 + 1 + 3 + 1 + 2 = 11 months to complete. In addition, if any of the tasks slips, the project will be delayed by the length of the slip.

Generalizing from these two extremes, it is clear that the amount of time required to complete a project depends on both the effort required for each task, and the dependencies between tasks. Current and former college students have some experience with this kind of problem already. Specifically, think about the problem of determining how long it will take to complete a particular major. The answer depends on the required courses, the effort required to complete each course (e.g., one person-semester), and the prerequisite structure (i.e., which courses must be completed before which other courses can be attempted).

Personnel Capabilities Matter

The dependencies discussed above were intrinsic to the tasks themselves. For example, the unit testing of a particular module cannot possibly begin until after the unit tests have been constructed and the module has been constructed. (In test-driven development the unit tests are constructed first and in traditional development the module is constructed first, but in neither case can the actual testing be conducted until both of the other tasks have been completed.) However, the capabilities of team members can also place constraints on the order in which tasks can be completed. For example, a team may have only one person who has expertise with GUI construction technologies. Hence, GUI construction tasks must be worked on serially, they cannot be worked on in parallel.

The Problem to be Considered

We assume that relationship between effort and time is known. Most commonly, this means that the number of people assigned to a task is appropriate for the task (e.g., tasks are very small and assigned to a single individual). We also assume that personnel capabilities have been captured by the task dependencies (e.g., if tasks i and j can only be completed by person p then there is a task dependency between i and j).

Given these two assumptions, the basic data needed to calculate the time required to complete a project can be summarized in a table that includes the **duration** (i.e., the time required to complete each task) and the **dependencies** (i.e., which must be completed before a specific task of interest can be started). For example, consider the following table describing a product that requires the finish of 14 distinct tasks.

Task Number	Duration (Days)	Prerequisite Tasks
1	6	—
2	5	1
3	2	1
4	6	1

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

For example, task 1 has a duration of 6 days and has no prerequisite tasks. As another example, task 5 has a duration of 4 days and can't be initiated until after both tasks 2 and 3 have been completed.

While this table completely describes the tasks required to complete the product, it is sometimes useful to look at the dependency data in another way. Specifically, instead of thinking about the prerequisite tasks we can think about the subsequent tasks, which are summarized in the following table.

Task Number	Subsequent Tasks
1	2, 3, 4
2	5, 9
3	5
4	6, 7, 8
5	10
6	10, 13
7	11
8	11
9	12
10	12
11	13
12	14
13	14
14	—

For example, this table shows that task 1 (and perhaps others) must be completed before tasks 2, 3 and 4 can be initiated and that task 7 (and perhaps others) must be completed before task 11 can be initiated.'.

Note that, *tasks must be numbered/ordered in such a way that lower numbered tasks do not depend on higher numbered tasks* (i.e., tasks with a higher number are not prerequisites for tasks with a lower number). This is necessary to ensure that there are no dependency cycles, which would make completion of all tasks impossible. Obviously, the data about tasks times and task dependencies can be combined into one table but, especially at first, it is usually clearer to use two.

Note also that not all projects have a natural first task. In other words, it may be possible at the outset of a project to initiate several tasks simultaneously. In such cases, it is sometimes convenient

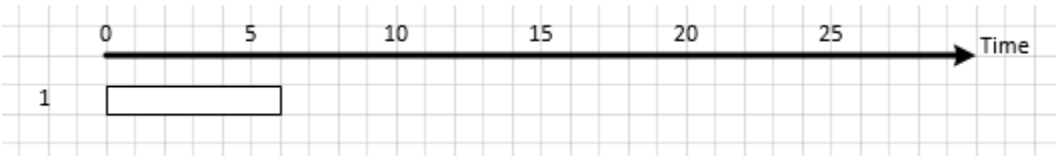
to create a *dummy* task that has no duration that is imagined to be a prerequisite for these tasks. Similarly, not all projects have a natural last task. In other words, it may be possible at the end of a project to complete several tasks simultaneously. In such cases, it is again sometimes convenient to create a dummy task that has no duration that is imagined to be subsequent to these tasks.

With these data in hand we can consider the problem of determining how long it will take to complete a project. The techniques we consider all involve the identification of the **critical tasks**—the tasks that determines the minimum amount of time required to complete the project. They also involve the calculation of **slack** (sometimes called **float**)—the amount of time that a task can slip without the project falling behind.

Gantt Charts

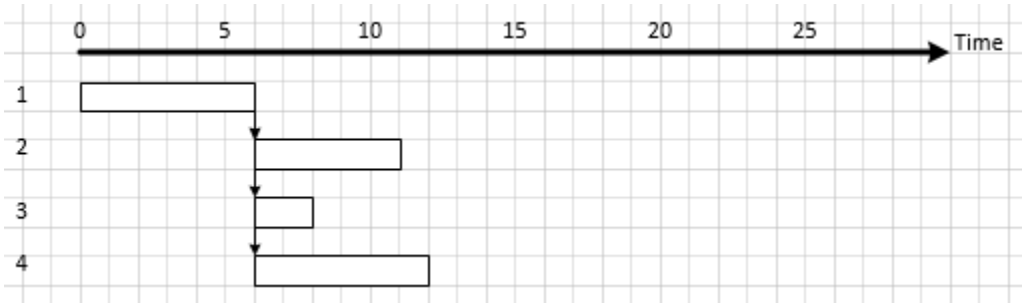
Gantt charts have a long history, having been first developed in the 1910s. In a Gantt chart, tasks are represented as rectangles and dependencies are represented as arrows. Time increases in the horizontal direction (traditionally, from left to right) and tasks are added in the vertical direction (with the first task at the top). The left side of each task is positioned at its earliest possible start time. A task's horizontal extent is determined by the time required to complete it, its vertical extent is arbitrary.

So, continuing with the example above, since task 1 starts at time 0 and requires 6 time units to complete, it is represented as follows

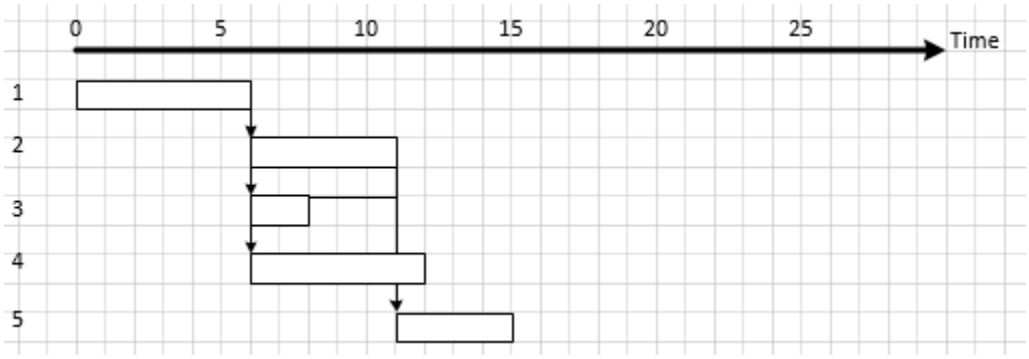


Task 2 has task 1 as a prerequisite so the earliest it can start is at time 6. Further, it requires 5 time units to complete, so it is represented as follows.

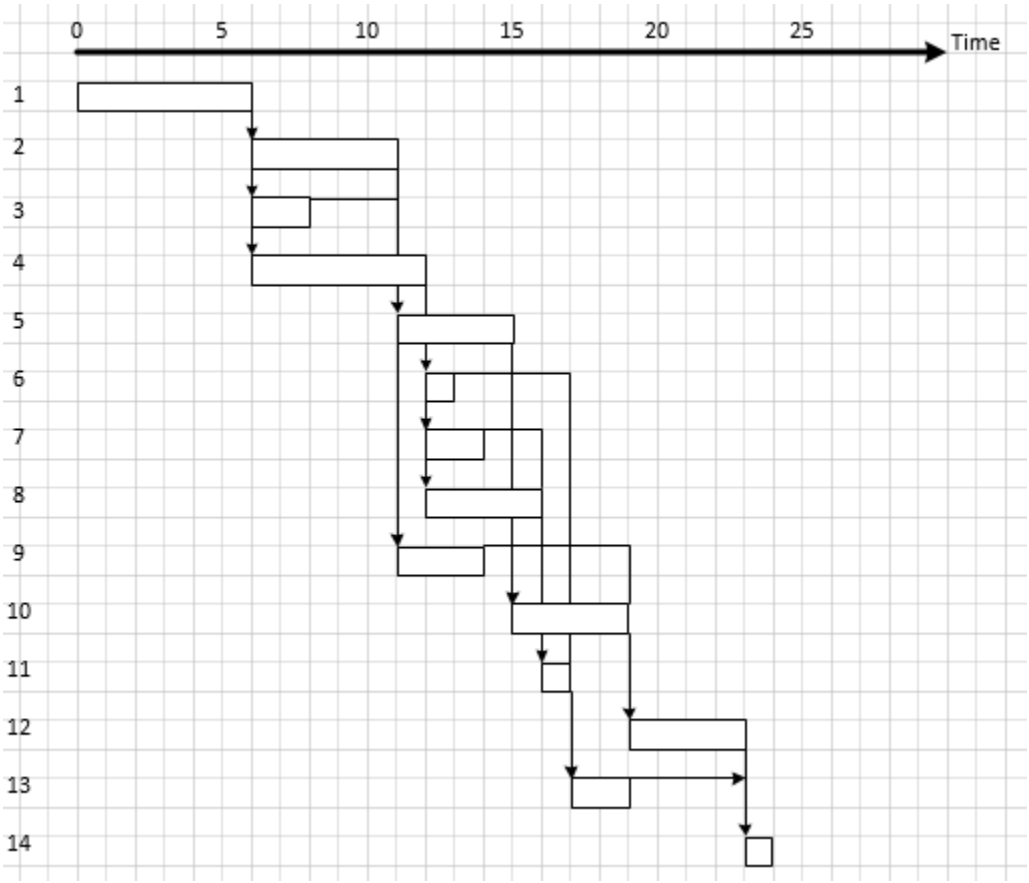
The arrow from the end of task 1 to the start of task 2 indicates that task 2 cannot start until after task 1 has been completed (i.e., that task 2 depends on task 1). Similarly, tasks 3 and 4 have task 1 as a prerequisite and require 2 and 6 days respectively. Hence, they are represented as follows



Since task 5 has both tasks 2 and 3 as prerequisites, it can't be started until after both are completed. This means that the earliest it can start is after task 2 (i.e., at time 11). Hence, it is added as follows.

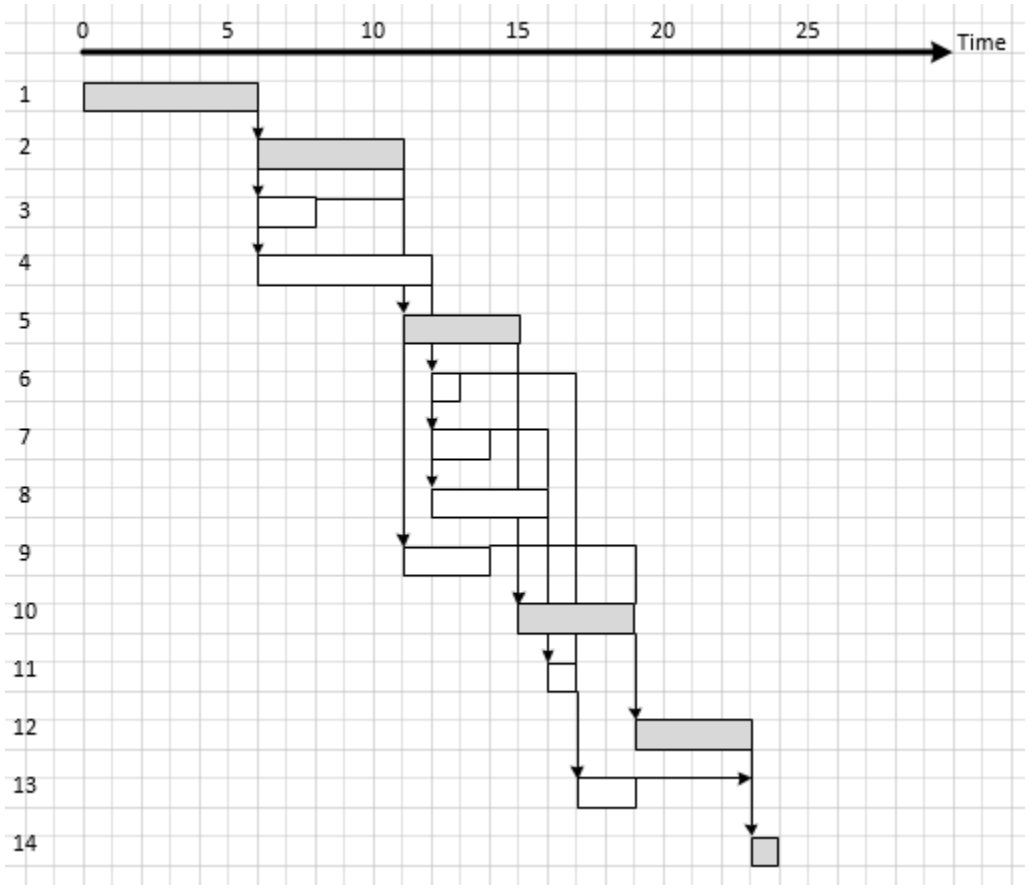


Continuing in this fashion, the other tasks are added as follows.

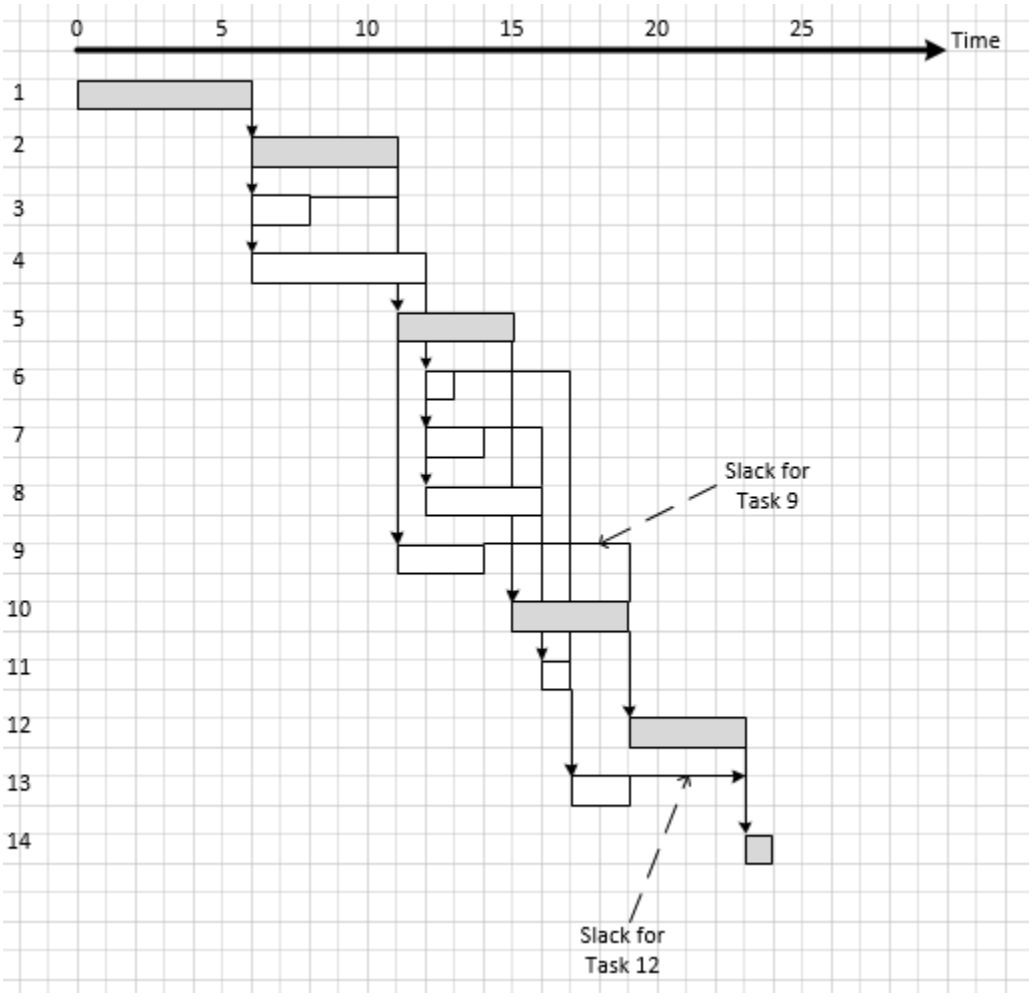


The critical tasks can be found by working backwards from the task with the latest finish time. Then, the predecessor that is part of the the critical sequence is the predecessor that itself has the latest end time.

For this example, task 14 is clearly critical. Then, looking at its two predecessors (12 and 13), task 12 has the later end time, so it is critical. Task 12 has two predecessors, tasks 10 and 11. Since task 10 has the later end time, it too is critical. Task 10 has two predecessors, tasks 5 and 6. Since task 5 has the later end time, it is critical. Continuing in this fashion, task 5 has two predecessors, task 2 having the later end time. Finally, task 2 has only one predecessor. The critical sequence of tasks can be represented as follows.

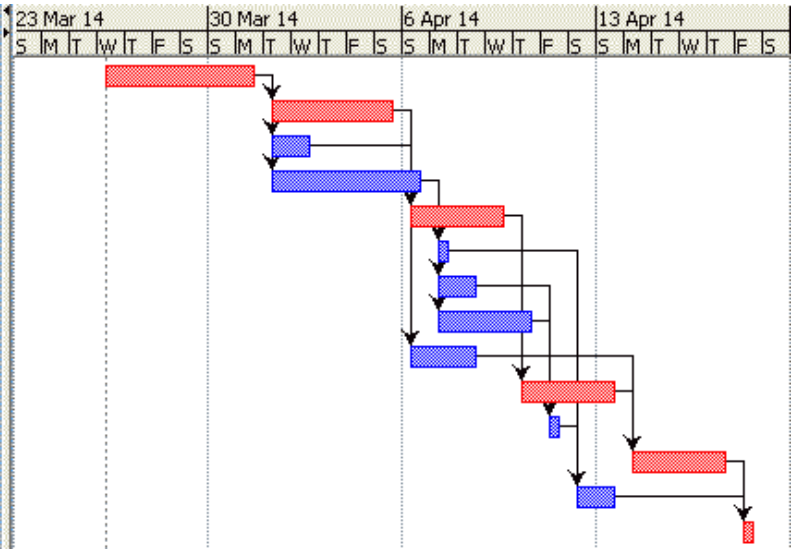


Once the critical tasks have been identified, it is an easy matter to identify the slack times. For example task 13 can slip 4 time units without causing task 14 to be delayed. Similarly, task 9 can slip five time units without causing task 12 to be delayed. This can be illustrated as follows.



Not surprisingly, there is no reason to create Gantt charts by hand; many software products exist that can create them. One example is shown below.

Name	Duration	Predecessors
1	6 days	
2	5 days	1
3	2 days	1
4	6 days	1
5	4 days	2;3
6	1 day	4
7	2 days	4
8	4 days	4
9	3 days	2
10	4 days	5;6
11	1 day	7;8
12	4 days	9;10
13	2 days	6;11
14	1 day	12;13



Note that the software product used to create this Gantt chart shortens tasks slightly so that the dependency arrow can extend horizontally from the prerequisite task even when there is no slack. Thus, task durations should not be measured from the chart, they should be read from the textual description.

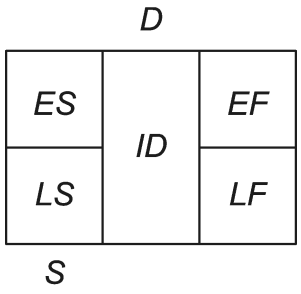
The Critical Path Method (CPM)

In the 1950s, a considerable amount of attention was devoted to the use of graph theoretic algorithms (e.g., path finding algorithms) and numerical optimization algorithms (e.g., the Simplex Method) to solve planning the problems of various kinds, including scheduling and assignment problems. On such effort resulted in the development of the Program Evaluation and Review Technique (PERT). In PERT, the data from the task table and dependency table are used to construct a graph in which the vertices represent **milestones** (i.e., events that can be used to mark the progress of the project) and the edges represent tasks (each edge having a weight that corresponds to the duration of the task). The critical sequence of tasks is determined by finding the longest path from the first event to the last event. As a result, the critical sequence of tasks is usually called the critical path.

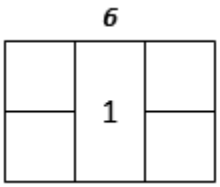
At around the same time, another group of researchers developed a technique that uses both time and cost estimates. Specifically, it constructs a time-cost curve for each task based on the ***normal time*** and the ***crash time*** for that task (to account for the relationship between effort and time/cost). This technique, which was called the Critical Path Method (CPM), requires the solution of a constrained linear optimization problem to identify a schedule that results in the minimum cost way to complete the project at a particular time.

Subsequently, a variety of other methods have been developed, some of which solve different variants of the problem and others of which use different graph representations (e.g., some represent the tasks as edges and some as vertices, some use dummy edges and vertices for different purposes), all of which make use of the term "critical path." We will consider one such Critical Path Method (CPM).

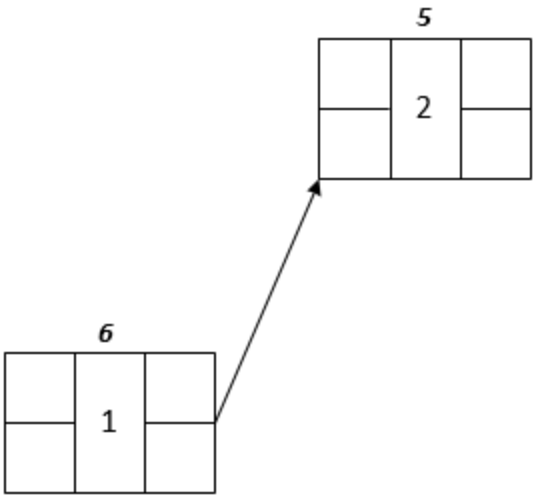
In this version of the CPM, the data from the duration column and dependency column are used to construct a graph in which the vertices represent the tasks and the directed edges represent the dependencies. Each task is visualized as a rectangle that contains seven pieces of information, the task identifier (*ID*), task duration (*D*), earliest start time (*ES*), earliest finish time (*EF*), latest start time (*LS*) and latest finish time (*LF*), and slack (*S*) as follows.



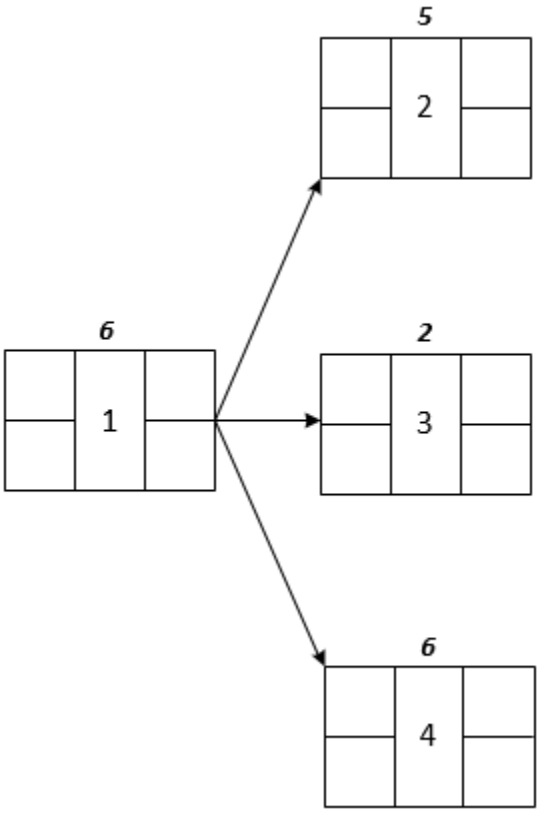
The duration is taken directly from the table, and the five other pieces of information, which are described below, are calculated in an iterative fashion. So, before any iterations are performed, task 1 is represented as follows.



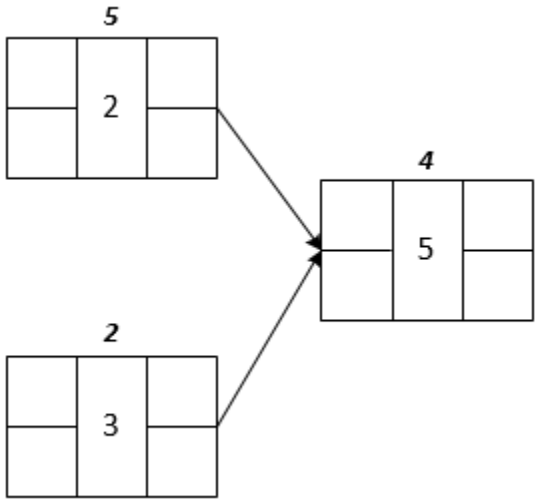
The dependency column is used to construct the directed edges in the graph. Specifically, if task *i* must be completed before task *j* then the graph includes a directed edge from *i* to *j*. For example, since task 2 cannot be completed until after task 1, there is a directed edge from vertex 1 to vertex 2.



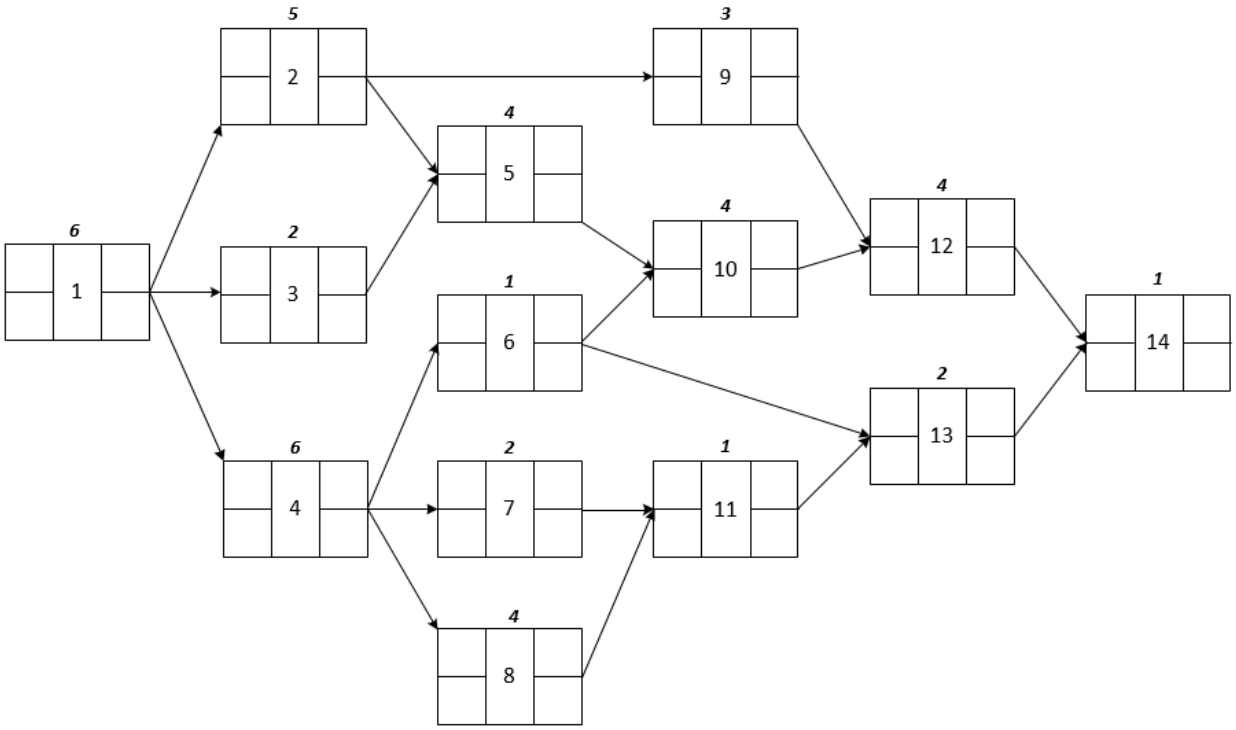
Similarly, since tasks 3 and 4 also can't be completed until after task 1, there are directed edges from task 1 to both tasks 3 and 4 as follows.



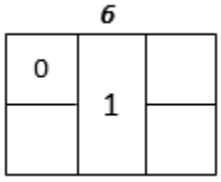
Now, since task 5 cannot be completed until after tasks 2 and task 3, there is one directed edge from 2 to 5 and another from 3 to 5. In other words, task 5 has two inbound edges as follows.



The entire graph showing all of the information from the durations and dependencies table is shown below.



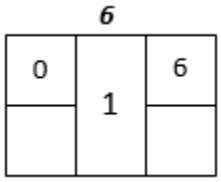
The **earliest start time** for task *i*, denoted by ES_i, is the start time for task *i* assuming the preceding tasks are completed as early as possible. The calculation of these values starts by assigning the initial tasks (the tasks that have no prerequisites) an earliest start time of 0. In this example, this means $ES_1 = 0$.



The **earliest finish time** for task *i*, denoted by EF_i,is the sum of the earliest start time and the duration. That is,

$$EF_i = ES_i + D_i$$

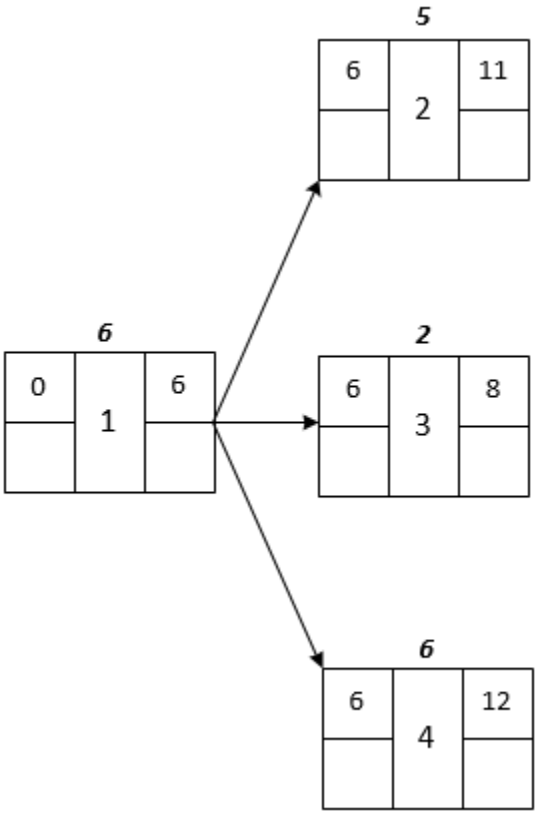
So, the earliest finish time for task 1 is $EF_1 = ES_1 + D_1 = 0 + 6 = 6$.



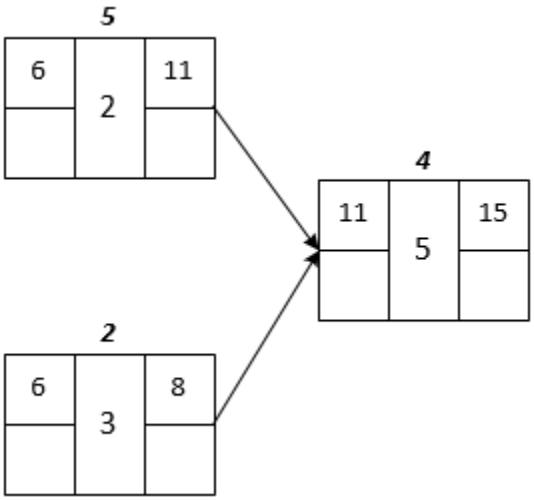
The earliest start time for each task other than the initial tasks is the maximum of the earliest finish time for all of its prerequisite tasks. So, letting $\mathcal{P}_j$ denote the set of prerequisite tasks of task j.

$$ES_j = \max_{i \in \mathcal{P}_j} EF_i$$

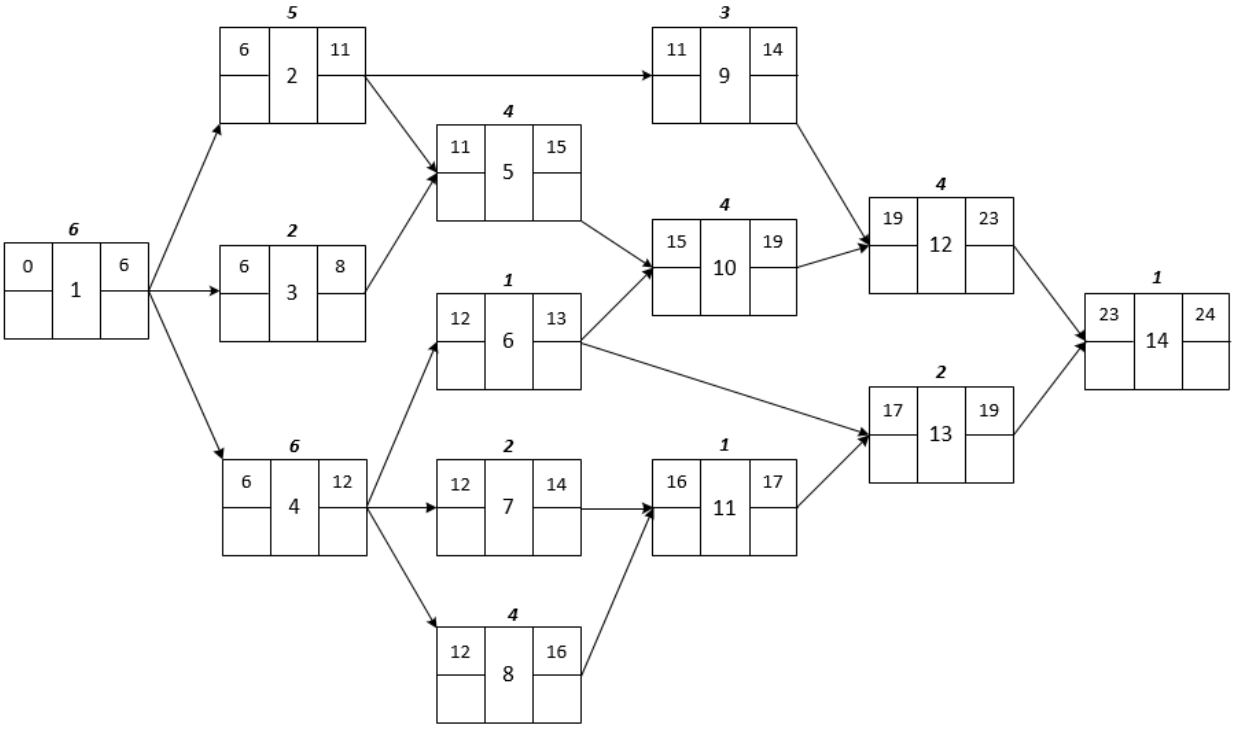
So, for example, consider task 2. It has only one prerequisite (task 1), so $\mathcal{P}_2 = \{1\}$ and, hence, $ES_2 = 6$. That is, since task 2 can't start until after task 1, it's earliest start time is task 1's earliest finish time (i.e., time 6). It's earliest finish time is simply $EF_2 = ES_2 + D_2 = 6 + 5 = 11$. Similarly, tasks 3 and 4 have earliest start times of 6 and earliest finish times of 6 + 2 = 8 and 6 + 6 = 12, respectively.



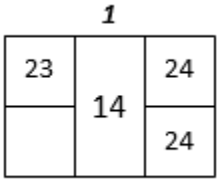
Now consider task 5.Since task 5 can't be completed until after tasks 2 and 3 (i.e., tasks 2 and 3 are prerequisites) $\mathcal{P}_5 = \{2, 3\}$. So, it's earliest start time is the latest finish time for those two tasks. Specifically, since the earliest finish time for 2 is 11 and the earliest finish time for 3 is 8, the earliest start time for 5 is 11.



Continuing in this fashion yields the following.



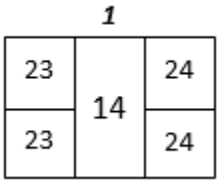
The **latest finish time** for task *i*, denoted by LF_i, is the last time the task can be completed without delaying the project beyond its earliest time. For the last task on the critical path the latest finish time is the same as the earliest finish time. So, in this example $LF_{14} = EF_{14} = 24$ as illustrated below.



The **latest start time** is the latest finish time minus the duration. That is

$$LS_i = LF_i - T_i$$

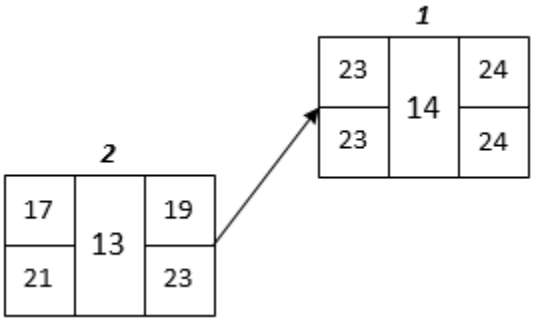
So, for task 14, $LS_{14} = LF_{14} - T_{14} = 24 - 1 = 23$ as shown below.



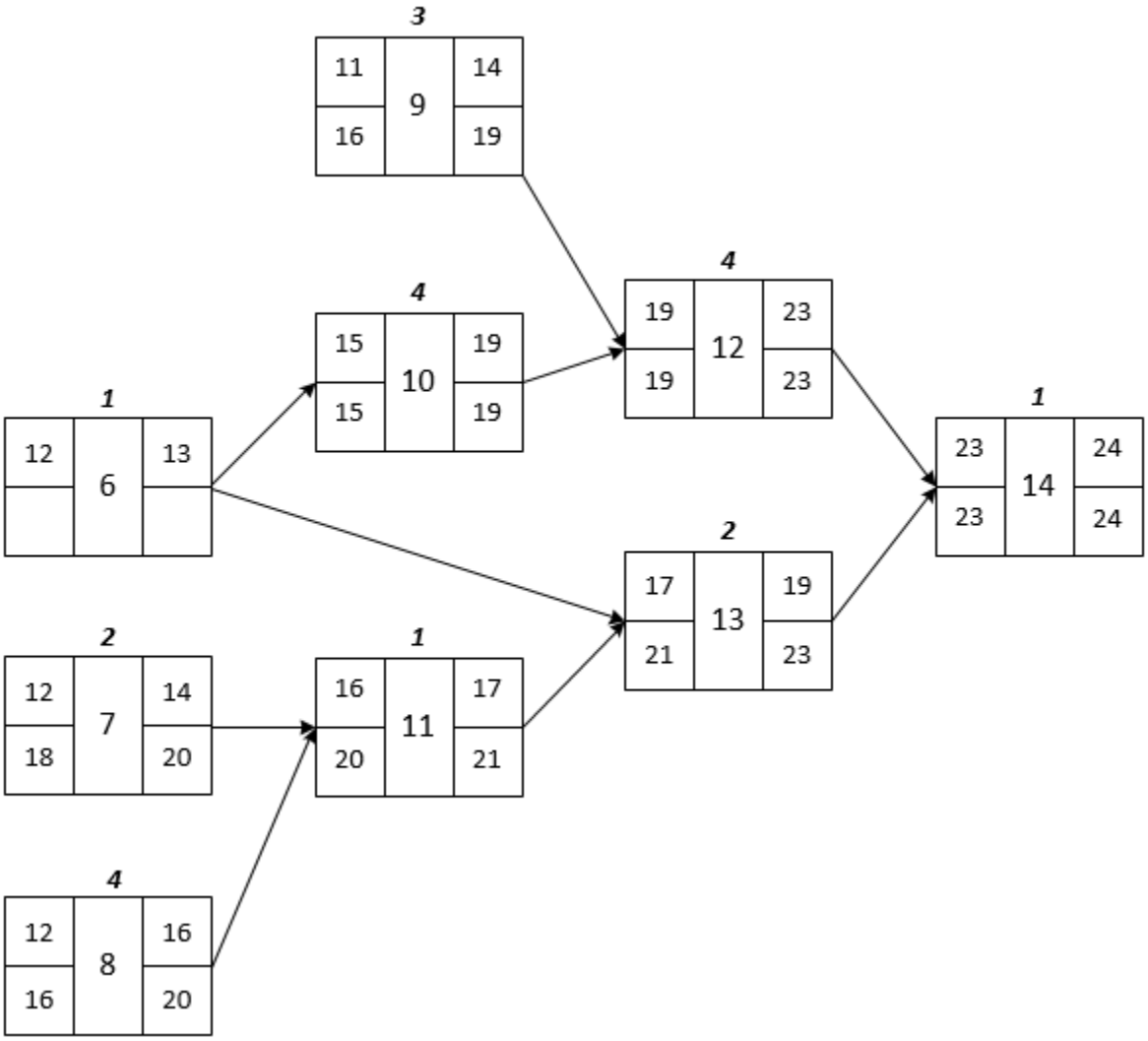
For tasks other than the last task on the critical path, the latest finish time is the minimum start time for all (immediately) subsequent tasks. That is, letting S_i denote the set of tasks that are subsequent to task *i,* the latest finish time for task *i* is given by

$$LF_i = \min_{j \in S_i} LS_j$$

For example, consider task 13. Since the only task subsequent to task 13 is task 14, it follows that $S_{13} = \{14\}$. Hence, $LF_{13} = LS_{14} = 23$. The latest start time is then $LS_{13} = LF_{13} - T_{13} = 23 - 2 = 21$.



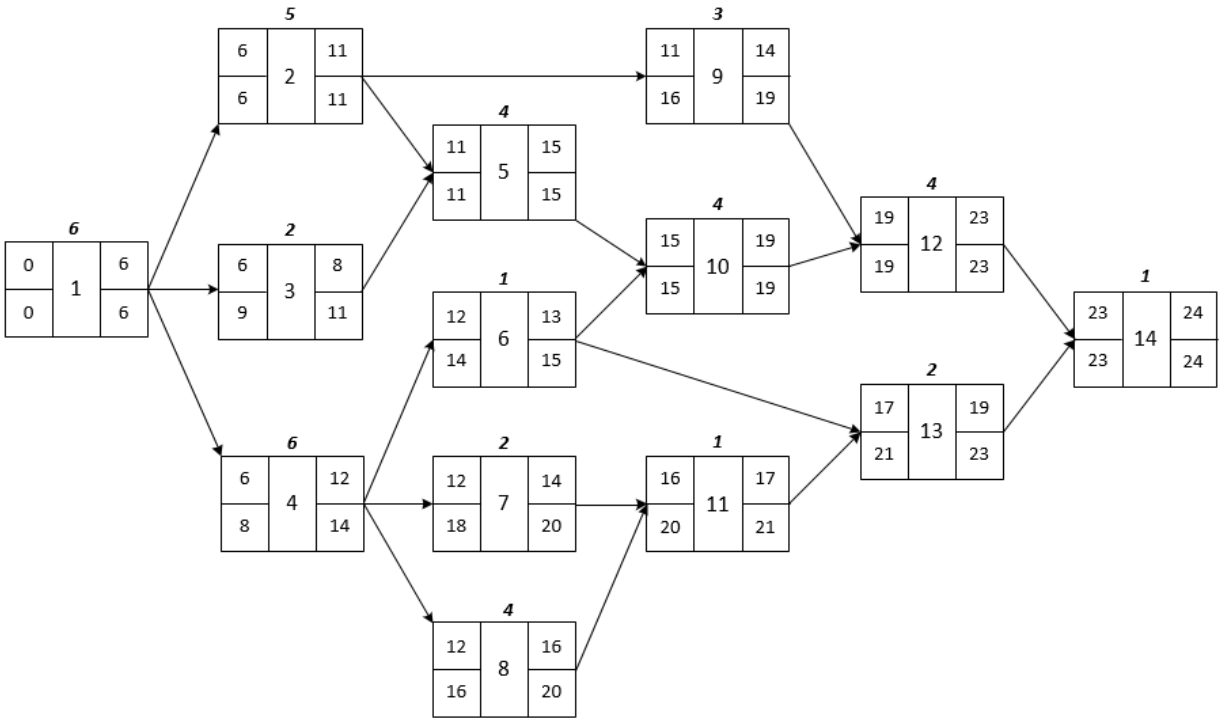
The calculation continue in exactly the same way until (working backwards) task 6 (which is different from tasks 7 through 13 in that it has more than one subsequent task.



Specifically, task 6 must be completed before tasks 10 and 13. So, at first glance, you might think that it's latest end time can be either task 10's latest start time (i.e., 15) or task 13's latest start time (i.e., 21). However, if task 10 doesn't end until 21 then task 10 will be delayed. Hence, task 6's latest end time must be the minimum of the latest start time of all successor tasks (i.e., 15).

Formally, $S_6 = \{10, 13\}$ and $LF_6 = \min \{LS_{10}, LS_{13}\} = \min \{15, 21\} = 15$. The latest start time is then $LS_6 = LF_6 - T_6 = 15 - 1 = 14$.

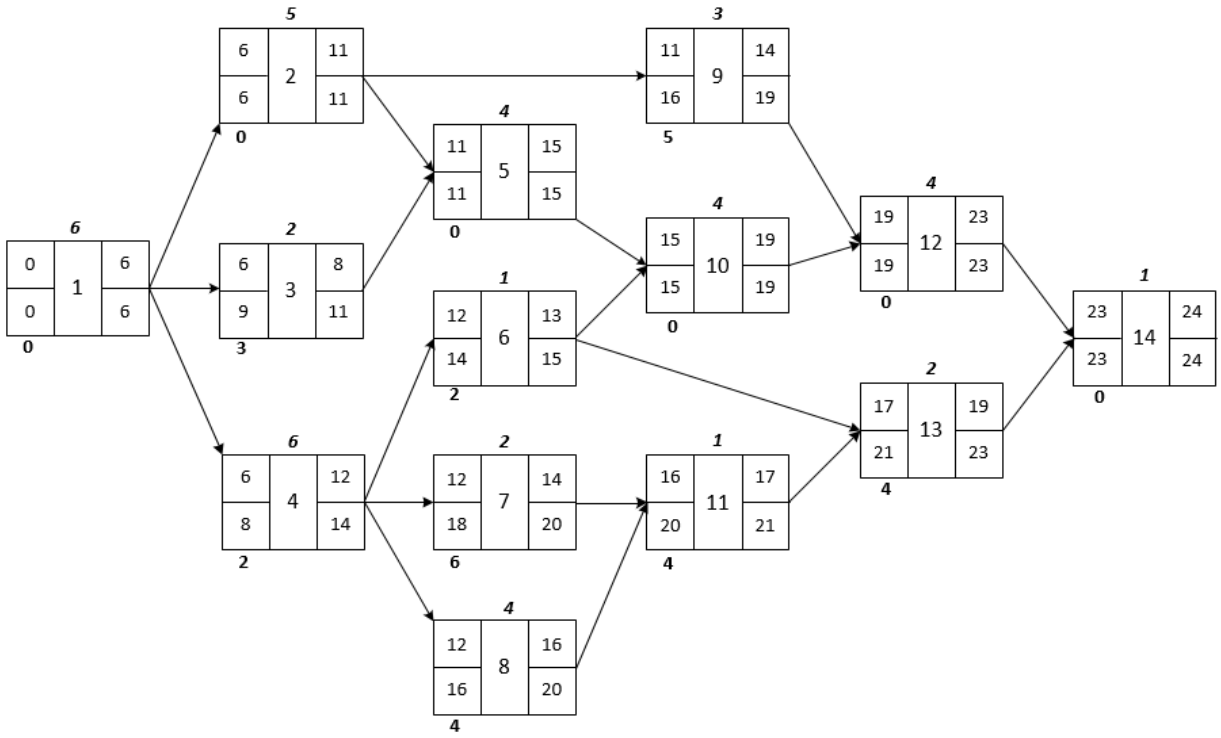
Continuing in this fashion (paying particular attention to tasks 4, 2 and 1) yields the following.



The **slack time** for task *i*, denoted by S_i, is the difference its latest finish time and its earliest finish **time**. That is

$$S_i = LF_i - EF_i$$

So, not surprisingly, the slack time for task 14 is $S_{14} = LF_{14} - EF_{14} = 24 - 24 = 0$. Indeed, this will be true of all of the tasks that are on the critical path (since that's precisely what it means to be critical). Specifically, the slack for tasks 1, 3, 5, 10, 12, and 14 are all 0, meaning they are the vertices in the critical path. On the other hand, all of the other tasks can slip somewhat, without delaying the overall project. For example, the latest finish time for task 13 is 23 while the earliest finish time is 19. Hence, task 13 can slip 4 days without delaying the overall project. All of the slack times are shown below.


6
0 1 6
0 6
0
5
6 2 11
6 11
0
2
6 3 8
9 11
3
4
11 5 15
11 15
0
6
6 4 12
8 14
2
1
12 6 13
14 15
2
2
12 7 14
18 20
6
4
12 8 16
16 20
4
3
11 9 14
16 19
5
4
15 10 19
15 19
0
1
16 11 17
20 21
4
4
19 12 23
19 23
0
2
17 13 19
21 23
4
1
23 14 24
23 24
0