Combinatorial Scheduling Theory

Ronald L. Graham

Things had not been going at all well in the assembly section of the Acme Bicycle Company. For the past six months, the section had consistently failed to meet its quota and heads were beginning to roll. As newly appointed foreman of the assembly section, you have been brought in to remedy this sad state of affairs. You realize that this is your big chance to catch the eye of upper management, so the first day on the job you roll up your sleeves and begin finding out everything you can about what goes on in the section.

The first thing you learn is that the overall process of assembling a bicycle is usually broken up into a number of specific smaller jobs:

FP —Frame preparation which includes installation of the front fork and fenders.

FW—Mounting and aligning front wheel.

BW—Mounting and aligning back wheel.

DE —Attaching the derailleur to the frame.

GC —Installing the gear cluster.

CW—Attaching the chain wheel to the crank.

CR —Attaching the crank and chain wheel to the frame.

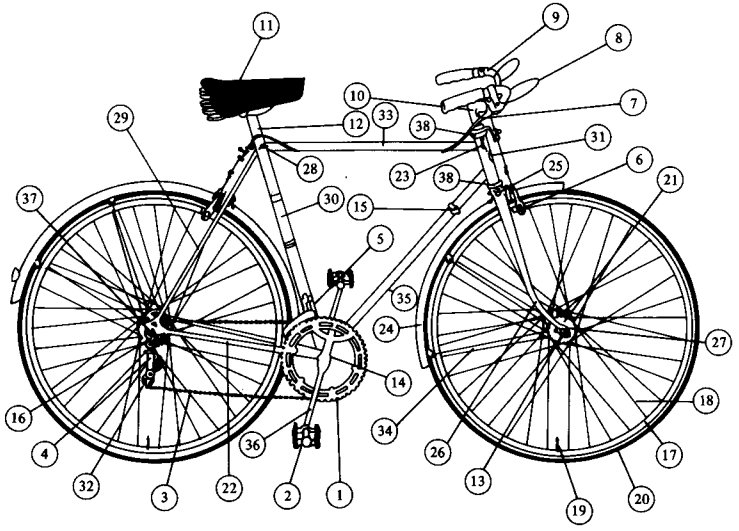
RP —Mounting right pedal and toe clip.

LP —Mounting left pedal and toe clip.

FA —Final attachments which includes mounting and adjusting handlebars, seat, brakes, etc.

You also learn that your recently departed predecessor had collected reams of data on how long each of these various jobs takes a trained assembler to

The Acme Bicycle



Key to Bicycle Parts:

1. Chainwheel
2. Pedal
3. Chain
4. Rear derailleur
5. Front derailleur
6. Caliper brake
7. Brake lever
8. Brake cable
9. Handlebars
10. Handlebar stem
11. Seat (saddle)
12. Seat post
13. Quick-release skewer (for instant wheel removal)
14. Bottom bracket
15. Gear-shift lever for rear derailleur
16. Freewheel gear cluster
17. Rim
18. Spoke
19. Valve
20. Tire
21. Hub (high-flange type)
22. Chainstay
23. Lug
24. Fender
25. Fork crown
26. Fork
27. Wheel dropout
28. Seat cluster lug
29. Seat stay
30. Seat tube
31. Steering head
32. Tension roller, rear derailleur
33. Top tube
34. Fender brace
35. Down tube
36. Cotterless crank
37. Rear drop out
38. Headset (top and bottom)

perform, which he had conveniently summarized in the following table:

Job:	FP	FW	BW	DE	GC	CW	CR	RP	LP	FA
Time:	7	7	7	2	3	2	2	8	8	18

Because of space and equipment constraints in the shop, the 20 assemblers in the section are usually paired up into 10 teams of 2 people each, with each team assembling one bicycle at a time. You make a quick calculation: One bicycle requires altogether 64 minutes of total assembly time, so a team of two *should* manage this in 32 minutes. This means that in an eight-hour day, each team could assemble 15 bicycles and with all 10 teams doing this, your quota of 150 bicycles per day can be met. You can already taste your next promotion.

Your enthusiasm dwindles considerably, however, when you realize that bicycles can't be put together in a random order. Certain jobs must be done before certain others. For example, it is extremely awkward to mount the front fork to the frame of a bicycle if you have first already attached the handlebars to the fork! Similarly, the crank must be mounted on the frame before the pedals can be attached. After lengthy discussions with several of the experienced assemblers, you prepare the following chart showing which jobs must precede which others during assembly.

This job	must be preceded by	These jobs
FA		FP, FW, BW, GC, DE
BW		GC, DE
GC, CW		DE
LP, RP		CR, CW, GC
CR		CW

In addition to these mechanical constraints on the work schedule, there are also two rules (known locally as "busy" rules) which management requires to be observed during working hours:

Rule 1: No assembler can be idle if there is some job he or she can be doing.

Rule 2: Once an assembler starts a job, he must continue working on the job until it is completed.

The customary order of assembling bicycles at Acme Bicycle has always been the one shown in the schedule in Figure 1. The schedule shows the activity of each assembler of the team beginning at time zero and progressing to the time of completed assembly, called the *finishing time*, some 34 minutes later. Although this schedule obeys all the required order-of-assembly constraints given above, it allows each team to complete only slightly over 14 bicycles per day. Thus the total output of the section is just over 140 bicycles per day, well under the quota of 150.

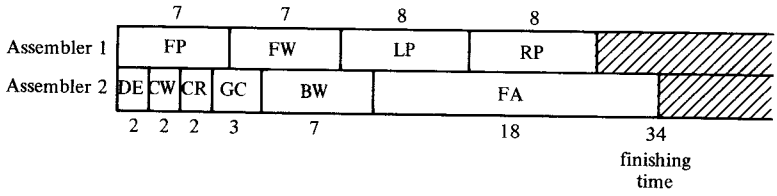


Figure 1. Standard assembly schedule.

After wasting numerous pieces of paper trying out various alternative schedules with no success, you decide, in haste, to furnish all the assemblers with rented all-electric tools. This decreases the times of *each* of the jobs by exactly one minute, so the total time required for all the individual jobs is only 54 minutes. With a little luck, you hope it will be possible to push the output per team to nearly 18 bicycles per day. However, at the end of the first week using the rented tools, you notice that production has now gone *down* to less than 14 bicycles per day. This seems hard to understand so you start jotting down a few possible schedules using the new reduced job times. Surprisingly, you find the best you can do is the one shown in Figure 2. All schedules obeying Rules 1 and 2 take at least 35 minutes for each assembled bicycle!

So you return the rented electrical tools and in desperation decide on a brute force approach: you hire 10 extra assemblers and decree that from now on, each of the 10 teams will consist of *three* assemblers working together to put the miserable bicycles together. You realize that you have increased labor costs by 50%, but you are determined to meet the quota or else.

It only takes you two days this time to decide that something is seriously wrong. Production has now dropped off to less than 13 bicycles per day for each 3-man team! Reluctantly, you again outline various schedules the teams might use. A typical one is shown in Figure 3. Curiously enough, you discover that every possible schedule for a 3-man team obeying Rules 1 and 2 requires 37 minutes for each assembled bicycle. You spend the next few days wandering around the halls and muttering to yourself "Where did I go wrong?" Your termination notice arrives at the end of the week.

This parable actually has quite serious implications for many real scheduling situations. Indeed, some of the earliest motivation for studying these types of scheduling anomalies arose from work on the design of com-

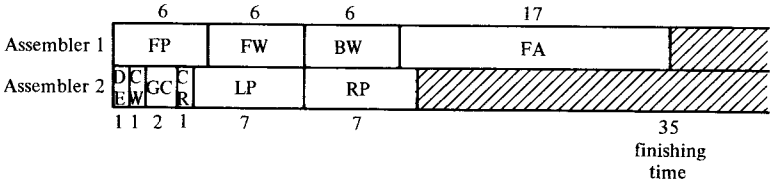


Figure 2. Best schedule with reduced job times.

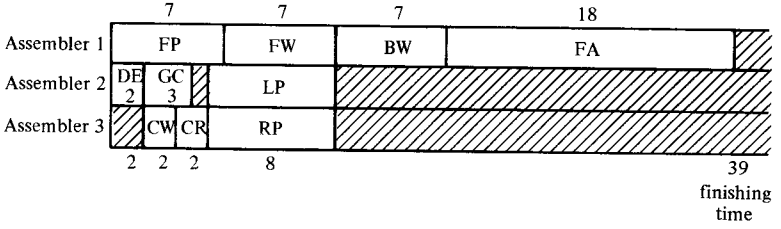


Figure 3. Best schedule for a 3-man team.

puter programs for antiballistic missile defense systems. There it was discovered (fortunately, by simulation) that decreasing job times and increasing computer resources could result in *more* time being used to finish the whole job. (An interesting discussion of this well-known "too-many-cooks-spoil-the-broth" effect, sometimes called "Brooks's Law", can be found in the book "The Mythical Man-Month" by Frederick Brooks.) Furthermore, it was found that adding more computing power to the system was no guarantee that the assigned jobs would all be completed on time. In a missile defense system, this was clearly a cause for some concern.

One might well ask just where it was that our hypothetical foreman at Acme Bicycle did go wrong. It will turn out that he was a victim of Rules 1 and 2 (and a little bad luck). The short-sighted greediness they demand resulted, as if often does, in an overall loss in performance of the system as a whole. In each case, assemblers were forced (by Rule 1) to start working on jobs which they couldn't interrupt (by Rule 2) when a more urgent job eventually came up.

Generally speaking, scheduling problems arise in almost all areas of human activity. Typically such problems can range from the allocation of manpower and resources in a highly complex project (such as the Viking Mars mission which required coordination of the simultaneous activities of more than 20,000 people or the instantaneous control of a modern electronic telephone network) to something as (relatively) simple as the preparation of a 7-course French meal. During the past few years, a number of scheduling models have been rather thoroughly studied in order to understand just why the unpredictable behavior like that in our bicycle example occurred, how bad such effects can ever be and how they can be avoided or minimized. What we will do in this article is to describe what has been recently learned about some of these problems and to point to the exciting new directions that researchers are beginning to pursue. Frequently we will examine a particular problem from several different viewpoints, showing how a variety of approaches can furnish us with a powerful arsenal of tools.

A very important aspect of this subject, both from the point of view of understanding specific scheduling procedures as well as for discovering exactly what is true and what is not true, is the use of examples. Indeed, much of the article will be devoted to various examples which often illustrate very vividly the unexpected subtleties that can occur in this field. From this discussion we hope that the reader will gain insight not only into scheduling

theory itself, but also into the kind of productive interaction which can (and often does) occur between mathematics (in this case, combinatorics), computer science (in this case, the theory of algorithms), and problems from the real world.

A Mathematical Model

In order to discuss our scheduling problems in a somewhat more precise way, we need to isolate the essential concepts occurring in the bicycle example. We do this by describing an abstract model of the scheduling situation. The model will consist of a system of m identical processors (before, the assemblers) denoted by $P_1, P_2, \ldots, P_m$, and a collection of tasks A, B, $C, \ldots, T, \ldots$ (before, the various jobs FP, BW, ...) which are to be performed by the processors according to certain rules. Any processor is equally capable of performing any of the tasks although at no time can a processor be performing more than one task. Each task T has associated with it a positive number $t(T)$ known as its processing time (before, these were the job times). Once a processor starts performing (or executing) a task T, it is required to continue execution until the completion of T, altogether using $t(T)$ units of time (this was Rule 2 before).

Between various pairs of tasks, say A and B, there may exist precedence constraints (before, these were the "order-of-assembly" constraints), which we indicate by writing $A \rightarrow B$ or by saying that A is a predecessor of B and that B is a successor of A. What this signifies is that in any schedule, task A must always be completed before task B can be started. Of course, we must assume that there are no cycles in the precedence constraints (for example, $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow A$) since this would make it obviously impossible to finish (or even start) these tasks. If at any time a processor can find no available task to execute, it becomes idle. However, a processor is not allowed to be idle if there is some task it could be performing (this was Rule 1 before).

The basic scheduling problem is to determine how the finishing time depends on the processing times and precedence constraints of the tasks, the number of processors, and the strategies used for constructing the schedules. In particular, we would like a method to determine schedules that have the earliest possible finishing time.

Performance Guarantees

The approach we will focus on for measuring the performance of our scheduling procedures will be that of "worst-case analysis." What we try to determine in this approach is the *worst possible* behavior that can occur for any system of tasks satisfying the particular constraints under consideration.

As a simple example of this approach, we might want to know just how much the finishing time can increase because of a *decrease* in the processing times of the individual tasks. The answer is given by the following result (due to the author) which dates back to 1966 and is one of the earliest results of this type in the literature.

Let f denote the finishing time of a schedule for m processors using times $t(A), t(B), \ldots$ and let f' denote the finishing time of a schedule using processing times $t'(A), t'(B), \ldots$. If $t'(T) \leq t(T)$ for all tasks T, then

$$\frac{f'}{f} \leq 2 - \frac{1}{m}.$$

For example, if there are two processors, $m = 2$; so $f'/f \leq 3/2$. This means that there can be an increase of at most 50% in finishing time if processing times are decreased.

This bound is a performance guarantee. It asserts that no matter how complicated and exotic a set of tasks with its precedence constraints and processing times happens to be, and no matter how cleverly or stupidly the schedules are chosen, as long as $t'(T) \leq t(T)$ for all tasks T (and this allows for the possibility that $t'(T) = t(T)$ for all T) then it is always true that the ratio of finishing times f'/f is never greater than $2 - (1/m)$. Furthermore, it turns out that this bound of $2 - (1/m)$ cannot be improved upon. What we mean by this is that we can always find examples for which $f'/f = 2 - (1/m)$. We give such an example for $m = 6$. In this example the times $t(T)$ are as follows:

Task T:	A	B	C	D	E	F	G	H	I	J	K	L	M
Time $t(T)$:	7	4	5	6	5	3	7	6	5	5	4	3	12

There are no precedence constraints and $t(T) = t'(T)$ for all tasks T. In Figures 4 and 5 we show the worst and best possible schedules. The corresponding finishing times are $f' = 22$ and $f = 12$, respectively. The ratio f'/f is $22/12$ which is exactly equal to $2 - (1/m)$ when $m = 6$.

In the box on p. 192-193 we give a brief sketch showing how a result of this type can be proved. With such a result, it is possible to know in ad-

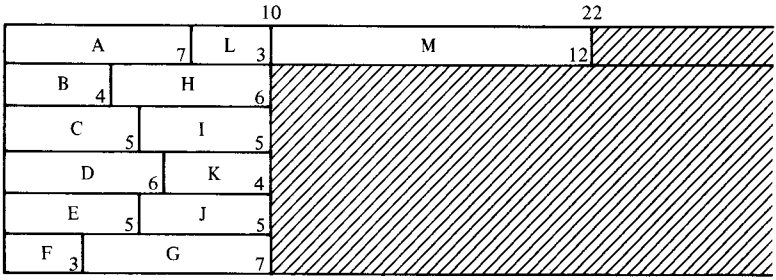


Figure 4. The worst possible schedule.

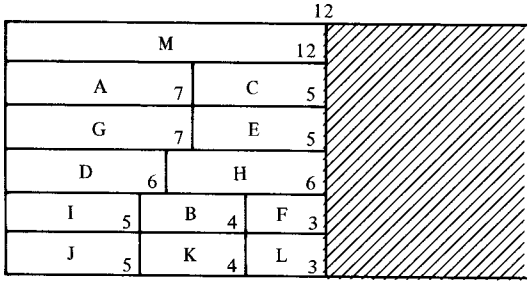


Figure 5. The best possible schedule.

vance the possible variation in f and so to build in a suitable safety factor (of 2, for example) to allow for the potential increase in finishing time. This could be extremely valuable in situations such as the previously mentioned ABM defense system application.

Critical Path Scheduling

One of the most common methods in use for constructing schedules involves what is known as "critical path" scheduling. This forms the basis for the construction of so-called PERT networks which have found widespread use in project management. The basic idea is that any time a processor has finished a task, it should try to choose the "most urgent" task as the next task to start working on. By "most urgent" we mean that task which heads the chain of unexecuted tasks which has the greatest sum of processing

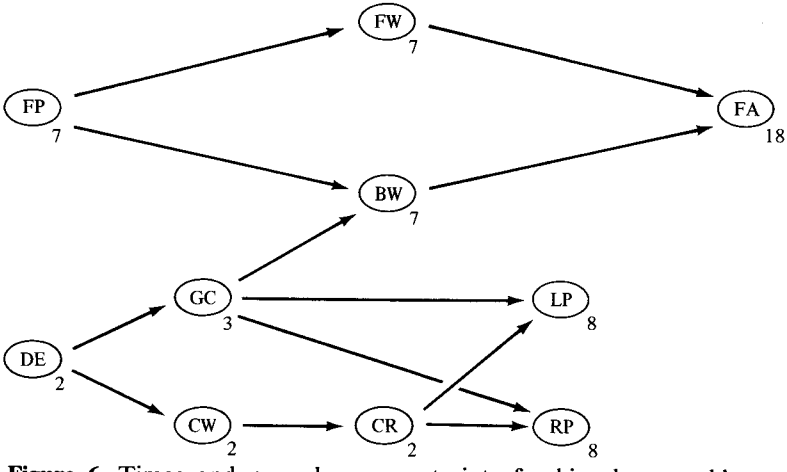


Figure 6. Times and precedence constraints for bicycle assembly at Acme Bicycle.

times. This "longest" chain is termed a "critical path" since its tasks are the most likely to be the bottlenecks in the overall completion of the whole set of tasks. In critical path (CP) scheduling, tasks which head the current critical paths are always chosen as the next tasks to execute.

As an example let us look again at our bicycle assembly example. We can combine the information on task times and precedence constraints in Figure 6. At the start the critical paths are FP → FW → FA and FP → BW → FA, both with length 32. This, in a CP schedule, FP is started first. Once FP is started, the new critical path becomes DE → GC → BW → FA which has length 30. Hence, in a CP schedule, the other processor (assembler) will start on DE. If we continue this scheduling strategy, we will finally end up with the schedule shown in Figure 7. Note that this

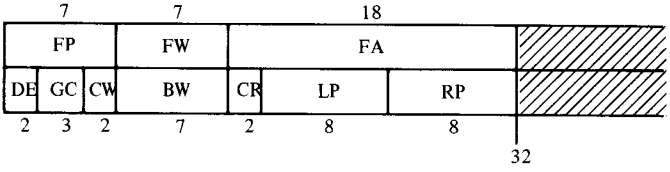


Figure 7. A critical path schedule for bicycle assembly.

schedule has a finishing time of 32 which is clearly the best we could hope for. If only our hypothetical foreman had known about CP scheduling.

Of course, this example is really too small to show the power of CP scheduling. The extensive literature on this technique attests to its wide usefulness in a variety of applications. Nevertheless, from our point of view of worst-case analysis, it should be noted that CP scheduling can perform very poorly on some examples. In fact, not only is there no guarantee that CP scheduling will be close to optimal, but it can happen that this procedure will result in the worst schedule possible!

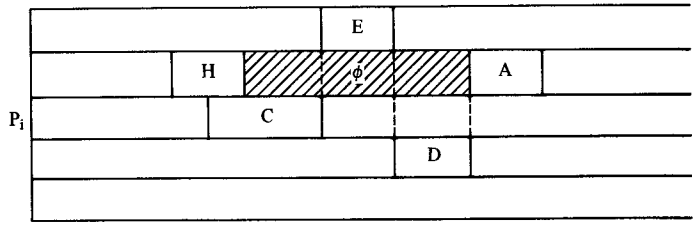
Here is an example with four processors. The tasks, processing times and precedence constraints are shown in Figure 8, as are a CP schedule with finishing time $f_{CP} = 23$ and an optimal schedule with finishing time $f_{OPT} = 14$. Note that the ratio of finishing times f_{CP}/f_{OPT} is equal to $^{23}/_{14}$, only slightly less than $2 - (^1/_4)$ which (according to our performance guarantee result) is the *worst* possible value of the ratio for any two finishing times for four processors.

Scheduling Independent Tasks

Critical path scheduling gives a rational and, in practice, useful method for constructing schedules. We did find, however, that it is possible for a CP schedule to be very bad. In fact, there is no guarantee that CP scheduling won't produce the worst possible schedule, giving a finishing time ratio (compared to least possible finishing time) of $2 - (^1/_m)$. There are special situations, though, where CP scheduling will never perform this poorly. The

Performance Guarantee: A Proof

In order to give the mathematically-minded reader a feeling for how the bound of $2 - (1/m)$ on the ratio f'/f is obtained, we outline a proof. Consider a schedule which uses the execution times $t'(T) \leq t(T)$ and has finishing time f':



In particular, let us look at an interval of time during which some processor, say P_i, is idle; such intervals are denoted above by the symbol $\varnothing$. From the figure we see that P_i became idle after task H was completed and did not start executing tasks again until task A was begun. The only reason that A wasn't started immediately after the completion of H was that some predecessor of A, say D, was not yet completed at that time. Now either D was already being executed when H was completed or D hadn't yet been started. If it hadn't yet been started, then again this must be caused by one of D's predecessors, say E, not being finished then. By continuing this argument, we conclude that there must be a chain of tasks $\ldots C \rightarrow E \rightarrow D \rightarrow A \ldots$ which are being executed at all times that P_i is idle.

The same ideas show in fact that there must be some chain $\mathscr{C}$ of tasks with the property that some task of $\mathscr{C}$ is being executed whenever *any* processor is idle. Let $t'(\mathscr{C})$ denote the sum of the execution times of all the tasks in the chain $\mathscr{C}$ and let $t'(\varnothing)$ denote the sum of all the idle times in the schedule. It follows from what we have just said that

$$t'(\varnothing) \leq (m-1)\, t'(\mathscr{C}) \tag{1}$$

since the only time a processor can be idle is when some task in $\mathscr{C}$ is being executed, and then we must have at most $m - 1$ processors actually being idle.

Of course

$$f \geq t(\mathscr{C}) \tag{2}$$

since the tasks of $\mathscr{C}$ form a chain and consequently must be executed one at a time in any schedule. Thus, if t^* denotes the sum of all the decreased execution times and $\bar{t}$ denotes the sum of all the original execution times, we see that

$$f' = \frac{1}{m}(t^* + t'(\varnothing)) \qquad \text{(since between times 0 and } f' \text{ each processor is either executing a task or is idle)}$$

$$\leq \frac{1}{m}(t^* + (m-1)t'(\mathscr{C})) \qquad \text{(by (1))}$$

$$\leq \frac{1}{m}(t^* + (m-1)t(\mathscr{C})) \qquad \text{(since } t'(T) \leq t(T) \text{ for all } T)$$

$$\leq \frac{1}{m}(t^* + (m-1)f) \qquad \text{(by (2))}$$

$$\leq \frac{1}{m}(\bar{t} + (m-1)f) \qquad \text{(since } t^* \leq \bar{t})$$

$$\leq \frac{1}{m}(mf + (m-1)f) \qquad \text{(since } \bar{t} \text{ cannot exceed } mf)$$

$$= (2 - \frac{1}{m})f.$$

In other words,

$$\frac{f'}{f} \leq 2 - \frac{1}{m}$$

which is the desired bound.

most common of these, which we now take up, is the situation where there are no precedence constraints among the tasks.

Imagine that you are the supervisor of a typing pool in a large company. Part of your job each morning is to assign the various papers submitted the night before to particular typists in the pool. Because of differences in typewriters, internal page number references, etc., each paper must be typed by a single typist. How should you assign papers so as to minimize the time required to complete all the papers?

As we saw in the example in Figure 8, a bad CP schedule can take almost twice as much time as a good schedule. But if CP scheduling is used for a situation similar to the typing pool, then the following inequality always holds:

$$\frac{f_{CP}}{f_{OPT}} \leq \frac{4}{3} - \frac{1}{3m}.$$

Thus, the CP schedule for independent tasks (no precedence constraints) always finishes within 33 $^1/_3$% of the optimum finishing time. Furthermore, as in the case of the $2 - (^1/_m)$ bound, it is not hard to find examples which actu-

Task:	A	B	C	D	E	G	F	H	I	J	K	L
Time:	1	1	1	1	10	10	10	3	3	3	3	10

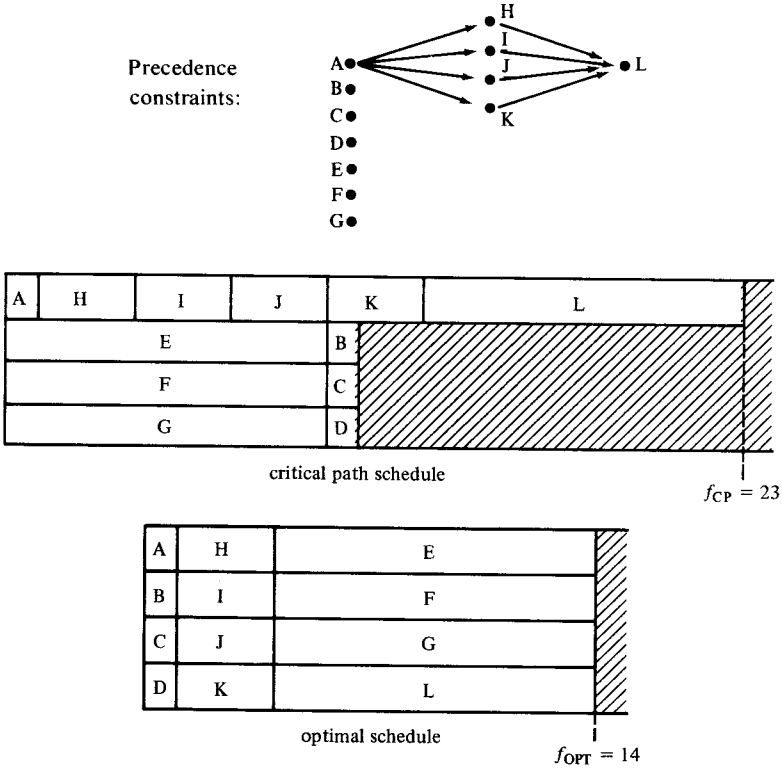


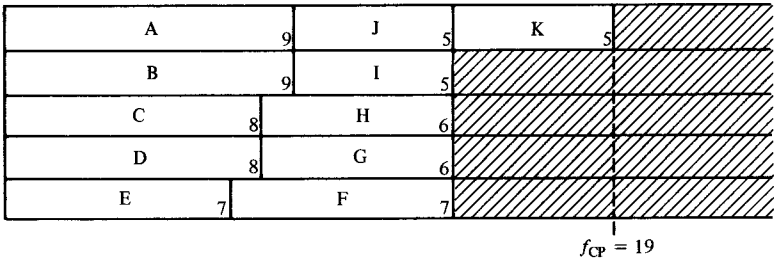
Figure 8. A critical path schedule does not always yield the best results. In this example, the critical path schedule for four processors is nearly twice as long as the optimal schedule.

ally achieve the bound $(4/3) - (1/3m)$ showing that it is not possible in general to improve the bound. In Figure 9 we give such an example for $m = 5$. A quick calculation shows that the ratio f_{CP}/f_{OPT} is equal to $19/15 = (4/3) - (1/15)$ which is just the bound $(4/3) - (1/3m)$ with $m = 5$.

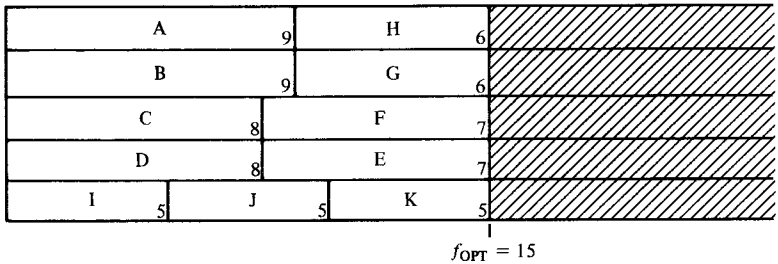
NP-complete Problems

At this point one might well ask, "Why should I settle for a schedule which is not best possible? Why not use a method that will always generate the schedule with the least possible finishing time?" A laudable ambition, to be sure. After all, any particular set of tasks you might be faced with is a *finite*

Task:	A	B	C	D	E	F	G	H	I	J	K
Time:	9	9	8	8	7	7	6	6	5	5	5



critical path (CP) schedule



optimal schedule

Figure 9. A critical path schedule and an optimal schedule for five processors. Critical path scheduling without precedence constraints, such as in a typing pool, produces results no worse than 4/3 of the optimal schedule.

set so you could always just examine *all* possible schedules and choose the best one.

The trouble with this brute force approach is that the number of possible schedules grows so explosively that there is no hope of looking at even a small fraction of them when the number of tasks is large. If we start with n tasks, then the number of different schedules with 2 processors is 2 multiplied by itself n times, or 2^n. Even for relatively small numbers n, 2^n is a very large number. For example, when $n = 70$, even if we could check 1,000,000 schedules each second, it would require more than 300,000 centuries to check all 2^{70} possible schedules. What we really need is a procedure (or, as computer scientists call it, an algorithm) which will not blow up so drastically as the number of tasks increases.

Unfortunately, this goal is very likely to remain beyond our reach for all time. This gloomy prospect is the result of the fundamental work of Stephen

Cook of the University of Toronto, who in 1972 introduced the concept of "NP-complete" problems. This class of problems is now known to contain literally hundreds of different problems notorious for their computational intractability. NP-complete problems, which occur in areas such as computer science, mathematics, operations research and economics, have two important properties: First, if any particular NP-complete problem had an efficient solution procedure then all of them would. Second, all methods currently known for finding general solutions for any of the NP-complete problems can always blow up exponentially in a manner similar to the behavior of 2^n. Mathematicians strongly suspect (but have not yet proved) that our inability to discover an efficient solution procedure is inherent in the nature of NP-complete problems: they believe that no such procedure can exist. We illustrate in the box on p. 197 one of the more well-known examples of an NP-complete problem, the so-called Traveling Salesman Problem.

As the reader may by now suspect, scheduling problems are, in general, NP-complete. In fact, even without precedence constraints and using only two processors, scheduling is still an NP-complete problem. This is the basis for the pessimistic outlook we took at the beginning of the section. One of its effects has been to redirect much of the earlier effort of trying to find good methods for determining *exact* solutions to the problems to the more fruitful direction of determining good *approximate* solutions easily. What we will do next is to look at the scheduling problem for two-processor systems from this viewpoint.

Getting Closer to the Best Schedule

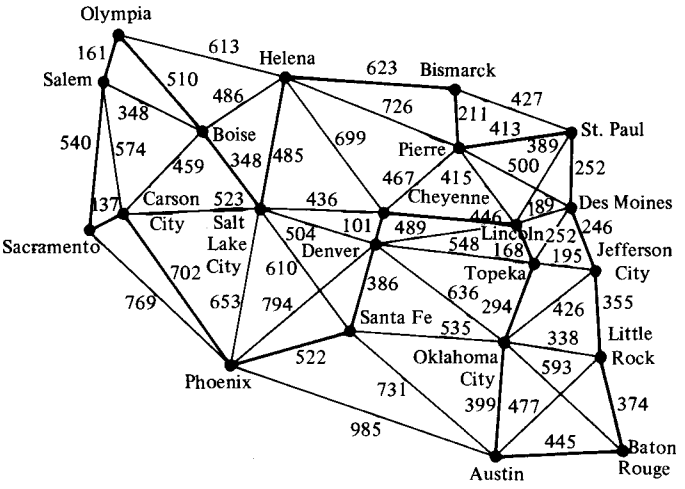
Critical path scheduling for tasks having no precedence constraints is, as we have seen, always guaranteed to finish within a factor of $^4/_3$ of the shortest possible finishing time. But to determine a CP schedule we must arrange the processing times of the n tasks in decreasing order. This sorting takes time: for n tasks it can be done in an amount of time which grows like $n \log_2 n$, which is only slightly faster than linear growth in n.

Suppose now that we are willing to do more work, but we want an answer which we know would be very close to the best possible. One approach for doing this is to choose, for some integer k, the $2k$ longest tasks and construct the best possible two-processor schedule for them; then schedule the remaining tasks arbitrarily. If we denote the finishing time of this schedule by f_k then

$$\frac{f_k}{f_{\text{OPT}}} \leq 1 + \frac{1}{2k+2}.$$

For a set of n tasks, the whole procedure can be done in at most $n \log_2 n + 2^{2k}$ operations, where the term $n \log_2 n$ comes from choosing and

The Traveling Salesman Problem



The problem of finding the shortest route which visits each of a given collection of "cities," finally returning to the city from which it began, is traditionally called the Traveling Salesman Problem. Such problems occur in a variety of contexts, e.g., collecting the money from coin telephones, periodic servicing of a dispersed set of vending machines, security guard inspections of locations in a factory, delivery routes for a product to different stores in a city, etc.. In the figure below we show the state capitals of the states west of the Mississippi with approximate road mileages between some of them. The route shown in bold has a total length of 8119 miles. However, the shortest route has length only 8117 miles. Can you find it?

sorting the $2k$ longest tasks and the term 2^{2k} comes from examining all possible schedules on two processors. Since k is a fixed number, the growth of this function as n increases is still moderate.

For example, choosing $k = 3$, we can guarantee that $f_3/f_{\text{OPT}} \leq {}^9/_8$ with an amount of work proportional to at most $n \log_2 n + 64$. More generally, any desired degree of accuracy can be guaranteed provided we are willing to pay for it. Unfortunately the price we may have to pay can increase very rapidly. For instance, to guarantee a value of f_k within 2% of the optimum could take time proportional to $n \log_2 n + 2^{49}$, which is sufficient to use up more than a few computer budgets.

This behavior should not be too surprising. After all, if an exponential amount of time seems to be required to find an optimal solution, we might

well expect the cost of approximate solutions to behave similarly as their guaranteed accuracy increased. What is surprising is that this exponential increase in cost can be avoided. Oscar Ibarra of the University of Minnesota and Chul Kim of the University of Maryland have very recently developed an algorithm for producing schedules with finishing times f_k which are guaranteed to satisfy

$$\frac{f_k}{f_{\text{OPT}}} \leq 1 + \frac{1}{k}$$

and which requires at most only kn^2 steps. The function kn^2 is an example of a "polynomial" in k and n. When k and n become large, the value of kn^2 is much smaller than the exponential function 2^k. The ideas underlying this procedure involve a clever combination of dynamic programming and "rounding" and are beyond the scope of this article. However, this type of approximation may well be able to guarantee very nearly optimal results using a reasonable amount of computer time.

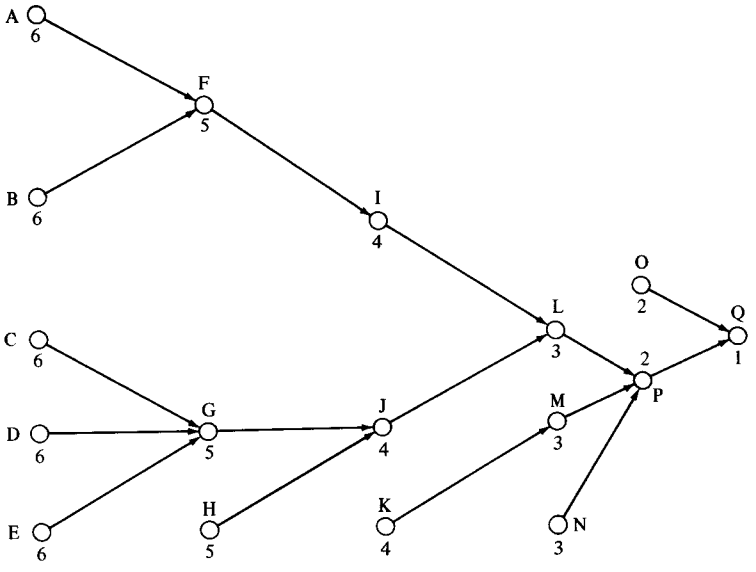
Finding the Best Schedule in Special Cases

Even though the goal of finding efficient techniques for constructing optimal schedules appears, in general, to be permanently out of reach, there are several interesting special cases of the scheduling problems for which this is actually possible. In this section we would like to discuss two of these cases and briefly describe the philosophy behind the various approaches in order to help understand why they work.

Much of the complexity in our scheduling problems comes from the complicated structure the precedence constraints can have and the complex number-theoretic properties the processing times can have. In our first special case, we restrict both of these factors: we assume that all processing times are equal to 1, and that the precedence constraints are tree-like. What this means is that every task T has at most one successor, i.e., there is at most one arrow leaving T. We show an example of tree-like precedence constraints in Figure 10.

It turns out with these particular restrictions critical path schedules are always optimal. This result, proved in 1961 by T.C. Hu of the University of California, was one of the first results in the field. To construct critical path schedules in this case we simply assign to each task T the length $L(T)$ of the longest chain headed by T. This number $L(T)$ is also known as the *level* of T. (We have assigned levels to tasks in Figure 10.) Any time a processor is available, we always choose a task which has the highest level possible. Figure 10 also shows a CP schedule on 3 processors for the tasks displayed in that Figure.

The second case we examine will allow the complexity of arbitrary (not necessarily tree-like) precedence constraints; but we will still require all processing times to be 1 and, in addition, we only allow *two* processors.



(a)

A	D	G	J	L	P	Q	
B	E	H	K	M			
C	F	I	N	O			

$f_{CP} = 7$

(b)

Figure 10. For tasks of equal length connected with tree-like precedence constraints (a), CP schedules are optimal (b).

There are now known three essentially different methods for producing optimal schedules in this situation, each somewhat complex; the box on p. 201-203 is devoted to these methods.

From these examples we hope the reader can get a feeling for the variety of techniques which can be successfully applied to special scheduling problems. One might hope that extensions of these ideas would lead to similarly successful algorithms for related questions, e.g., for the problem of three processors with all execution times equal to 1. At present, however, this tantalizing question remains completely unanswered. It should be noted that the "slightly" generalized problem having two processors with processing times of either 1 or 2 has recently been shown to be NP-complete. As we have previously mentioned, this provides strong theoretical evidence that in

this case no efficient algorithms can exist which are guaranteed to produce optimal schedules.

Bin Packing

One of the most interesting types of scheduling problems is one which turns the usual question around. Instead of fixing the number of processors and trying to finish as soon as possible, we can ask how few processors can be used and still complete all the tasks by a given deadline d. (For the time being, we shall restrict ourselves to the case where there are no precedence constraints between the tasks.) This new scheduling problem commonly goes under the name of *bin packing*. In the standard statement of this problem we are given a set of items $I_1, I_2, \ldots$ with item I_k having weight w_k. The object is to pack all the items in a minimum number of identical containers, called *bins*, so that no bin contains items having a total weight greater than some fixed number W which is called the *capacity* of the bin. (In the scheduling analogy, the bins are processors: we try to pack jobs into as few processors as possible, subject to the condition that no processor can be assigned jobs whose total time exceeds the processor's capacity, namely, the deadline by which all jobs must be completed.)

The bin packing problem occurs in numerous practical situations and in a variety of guises. For example: a plumber may be trying to minimize the number of standard-length pipes needed from which some desired list of pipe lengths can be cut; a paper producer must furnish customers with various quantities of paper rolls of assorted widths which he forms from rolls of a standard width by "slicing," and he would like to minimize the number of standard rolls he must use; or a television network would like to schedule its advertisers' commercials of varying lengths into a minimum number of standard length station breaks. Another easily pictured example is that of a person confronting a standard postage stamp machine with an armful of assorted letters in one hand and a handful of quarters in the other.

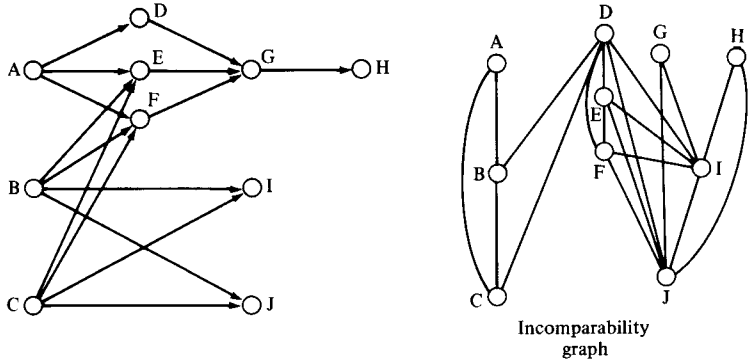
In general, bin packing problems can be extremely difficult. At present, the only known methods for producing optimal packings, i.e., those using the minimum number of bins, involve examining essentially all possible packings and then choosing the best one. For very small problems this may be feasible but as soon as the problem size becomes moderately large, these techniques are hopelessly inadequate. See, for example, the problem in Figure 11. The approach commonly taken to circumvent these difficulties is to design efficient heuristic procedures which may not always produce optimal packings but which are guaranteed to find packings which are reasonably close to optimal. We have already seen examples of this philosophy in some of the earlier scheduling problems. Some of the deepest results in the theory

Two-Processor Algorithms

There is now a variety of methods for determining an optimal schedule for two processors dealing with tasks of identical length. Here are three of them:

The FKN algorithm (after M. Fujii, T. Kasami, and K. Ninomiya (1969)).

The idea behind this method is that we can execute two tasks A and B during the same time interval only if they are *incomparable*, i.e., if neither task must be finished before the other is started. First create a diagram in which we connect two tasks if and only if they are incomparable. (This is called the incomparability graph of the set of tasks.)



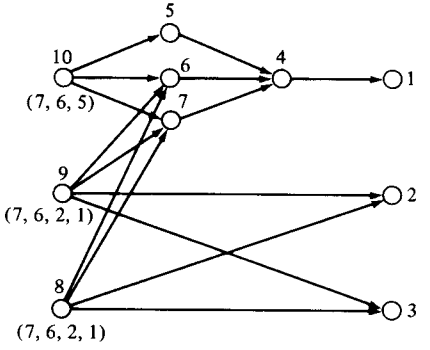
Then find the largest number of lines between tasks in the diagram so that no two go to a common task. (In our example, the lines between A–B, C–D, E–F, G–I, and H–J form such a collection.) These pairs designate the tasks to execute simultaneously; the other tasks we execute one at a time. (Once in a while we may have to do a mild amount of interchanging in order to actually produce a valid schedule. For example, if we have the set of four tasks P, Q, R, and S with precedence constraints $P \rightarrow Q$ and $R \rightarrow S$, then the FKN algorithm can choose the pairs P–S and R–Q to execute simultaneously, which is impossible. However, we *can* exchange R and S and execute P–R and S–Q with no trouble.) Since general methods are known for finding the largest number of disjoint lines in a graph, requiring at most $n^{5/2}$ steps for graphs with n points, the FKN method does indeed satisfy our requirements of efficiency and optimality.

The CG algorithm (after Edward Coffman of the University of California and Ronald Graham (1972)).

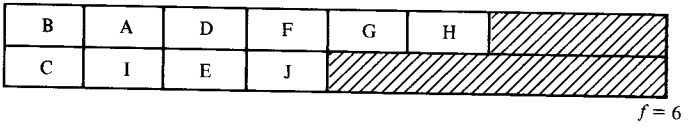
The approach here is much in the spirit of the level algorithm discussed on p. 198. The idea is that we are going to assign numbers to the tasks which will depend on the numbers assigned to *all* the successors of a

task, and not just on a single successor as was the case for the level algorithm. In order to apply this algorithm we must first remove all extraneous precedence constraints. In other words, if $A \rightarrow B$ and $B \rightarrow C$, then we do not include $A \rightarrow C$ since this is implied automatically. (Removing these extraneous edges is called forming the "transitive reduction" of the set of precedence constraints; it can be done in at most $n^{2.81}$ operations for a set of n tasks.) The CG algorithm proceeds by assigning the number 1 to some task having no successor and thereafter, for each task which has *all* its successors numbered, forming the *decreasing* sequence of its successors' numbers and choosing that task which has the smallest sequence in so-called "dictionary" order. (For example, (5,3,2) is earlier in "dictionary" order than (6, 1), (5,4,2) and (5,3,2,1).) After all tasks have been numbered, using the numbers from 1 to n, the schedule is then formed by trying to execute tasks in order of *decreasing* numbers beginning with the task having number n.

In the figure below we give a CG numbering with some of the successor number sequences also shown.



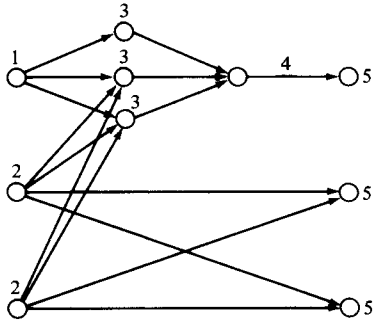
Note that after D, E, F, G, H, I, and J have been numbered, then among the three candidates for label 8, B and C with successor sequences (7,6,2,1) are chosen over A which has the larger successor sequence (7,6,5). This results in A getting the number 10 and hence, being executed first. Basically, what the CG algorithm does is to give tasks which either head long chains *or* have many successors larger numbers and consequently it tends to place them earlier in the schedule. Note that if we just use simple CP scheduling, executing tasks only on the basis of the longest chain they head, there is no reason why B and C couldn't be executed first, resulting in the schedule shown below.



The GJ algorithm (after Michael R. Garey and David S. Johnson of Bell Laboratories (1975)).

In order to apply this algorithm we must put *in* all arrows implied by the precedence constraints. (This is called forming the "transitive closure" of the set of precedence constraints and like the transitive reduction, can also be done for a set of n tasks in at most $n^{2.81}$ operations.) Suppose that we would like to try to finish all tasks by some deadline d. The GJ algorithm computes a set of deadlines (which are always $\leq d$) which would have to be met if in fact all tasks could be executed by time d. This is done by first assigning all tasks with no successors a deadline of d. Thereafter, for any task T which has all its successors assigned deadlines, we do the following: For each deadline d' assigned to a successor of T, determine the number $N_{d'}$ of successors of T having modified deadline d' or less. Set the deadline of T to be the smallest value achieved by any of the quantities $d' - [1/2 N_{d'}]$ (where $[x]$ denotes the smallest integer which is greater than or equal to x).

It is easy to see that if T has $N_{d'}$ successors which all have deadlines which are $\leq d'$, then T must be finished by time $d' - [1/2 N_{d'}]$ if all deadlines are to be met. After all deadlines have been assigned, the GJ schedule is formed by choosing the tasks in order of earliest deadlines first. The theorem which Garey and Johnson prove is that if there is any schedule which finishes by time d then this method will produce one. It follows from this that a GJ schedule has the minimum possible finishing time (and it doesn't matter what d we started with in forming it!). In the figure below we assign deadlines to the tasks using the value $d = 5$.



As is apparent, the earliest deadline (1) belongs to task A, which as we saw, must be executed first in any optimal schedule.

Finally, here is the optimal schedule produced by each of these three algorithms:

A	C	E	G	H	
B	D	F	I	J	

f = 5

1415926535	5820974944	8979323846	5923078164	2643383279
8214808651	4811174502	3282306647	8410270193	0938446095
4428810975	4564856692	6659334461	3460348610	2847564823
7245870066	7892590360	0631558817	0113305305	4881520920
3305727036	0744623799	5759591953	6274956735	0921861173
9833673362	6094370277	4406566430	0539217176	8602139494
0005681271	1468440901	4526356082	2249534301	7785771342
4201995611	5187072113	2129021960	4999999837	8640344181
5024459455	7101000313	3469083026	7838752886	4252230825
5982534904	8903894223	2875546873	2858849455	1159562863
0628620899	5028841971	8628034825	6939937510	3421170679
8521105559	5058223172	6446229489	5359408128	5493038196
4543266482	3786783165	1339360726	2712019091	0249141273
4882046652	9628292540	1384146951	9171536436	9415116094
1885752724	8193261179	8912279381	3105118548	8301194912
2931767523	6395224737	8467481846	1907021798	7669405132
4654958537	7577896091	1050792279	7363717872	6892589235
2978049951	5981362977	0597317328	4771309960	1609631859
5875332083	3344685035	8142061717	2619311881	7669147303
9550031194	8823537875	6252505467	9375195778	4157424218

Figure 11. Can the 100 weights shown be packed into 10 bins of capacity 150,000,000,000? Even with the use of all the computing power in the world currently available, there seems to be no hope of knowing the answer to this question. This example, while not particularly realistic (at present), illustrates the enormous difficulties in solving even relatively small bin packing problems.

of algorithms have arisen in connection with this approach to bin packing problems. We now turn to several of these.

To begin, let us arbitrarily arrange the weights of the items we are to pack into a list $L = (w_1, w_2, \ldots)$; no confusion will arise by identifying an item with its weight. One obvious way to pack the weights of L, called the *first-fit* packing of L, is to pack the weights in the order in which they occur in L, filling each bin as much as possible before proceeding to subsequent bins. More precisely, when it is w_k's turn to be packed, it is placed into the bin B_i having the *least* index i in which w_k can validly be packed, i.e., so that the new total of the weights in B_i still does not exceed W. (Of course, we assume at the outset that no w_k exceeds W since otherwise no packing at all is possible.)

We let $\mathrm{FF}(L)$ denote the number of bins required when the first-fit packing algorithm is applied to the list L and, similarly, we let $\mathrm{OPT}(L)$ denote the number of bins required in an optimal packing of L. The natural question of performance guarantee is this: How much larger than $\mathrm{OPT}(L)$ can $\mathrm{FF}(L)$ ever be? The answer, due to Jeffrey Ullman of Princeton University, is quite interesting. Ullman showed in 1973 that for any list of weights L,

$$\mathrm{FF}(L) \leq \frac{17}{10}\mathrm{OPT}(L) + 2.$$

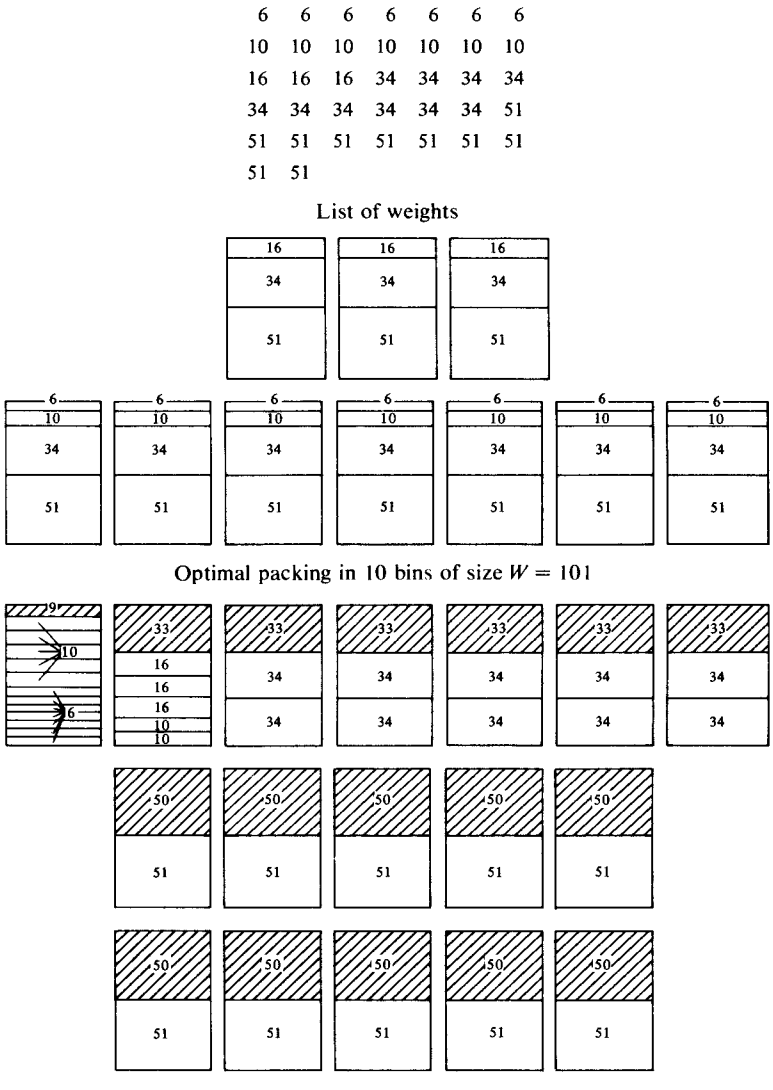


Figure 12. The given weights can be packed, optimally, in 10 bins of capacity $W = 101$. But the first-fit algorithm—packing the weights in order of their appearance on the list—requires 17 bins of this same size. The ratio $^{17}/_{10}$ is the worst that can ever happen with the first-fit algorithm.

He also showed that the unexpected fraction $^{17}/_{10}$ cannot be replaced by any smaller number. In Figure 12 we give an example of a list L which has $\mathrm{FF}(L) = 17$ and $\mathrm{OPT}(L) = 10$, so that $\mathrm{FF}(L) = {}^{17}/_{10}\mathrm{OPT}(L)$. It is suspected that, in fact, Ullman's inequality always holds with the term +2 re-

moved. However, it is not known if in this case equality can ever hold when OPT(L) becomes very large.

The appearance of the rather mysterious value $^{17}/_{10}$ turns out ultimately to depend on properties of so-called "Egyptian" fractions, that is, fractions which have numerator 1. Such fractions were treated extensively in one of the earliest known mathematical manuscripts, the famous Rhind papyrus of Aahmes (c. 1690 B.C.); at that time it was common to express fractional quantities as sums of "Egyptian" or unit fractions. For example, $^{25}/_{28}$ would be written as $(^1/_2) + (^1/_4) + (^1/_7)$.

The reason FF(L) was so large in the example given in Figure 12 was because all the large weights were left until last to be packed. As in the scheduling of independent tasks, it would make more sense to place large weights near the beginning of the list and let the smaller weights be used at the end for filling up gaps. More precisely, let us arrange the weights of L into a new *decreasing* list in which larger weights always precede smaller weights (exactly as in our earlier formation of CP schedules with no precedence constraints) and then apply the first-fit packing algorithm to this new list. This packing is called the *first-fit decreasing* packing of L; the number of bins it requires we denote by FFD (L).

The behavior of first-fit decreasing packings turns out to be surprisingly good. Corresponding to the $^{17}/_{10}$ bound for FF(L), we have in this case the following inequality: For any list of weights L,

$$\text{FFD}(L) \leq \frac{11}{9}\,\text{OPT}(L) + 4 \cdot$$

Thus, for lists of weights requiring a large number of bins (where the +4 becomes relatively insignificant), the first-fit decreasing packing is guaranteed never to use more than approximately 22% more bins than does the theoretically optimal packing, compared to a 70% waste which could occur in a first-fit packing which uses a particularly unlucky choice for its list. The class of examples given in Figure 13 shows that the coefficient $^{11}/_9$ cannot be replaced by any smaller number.

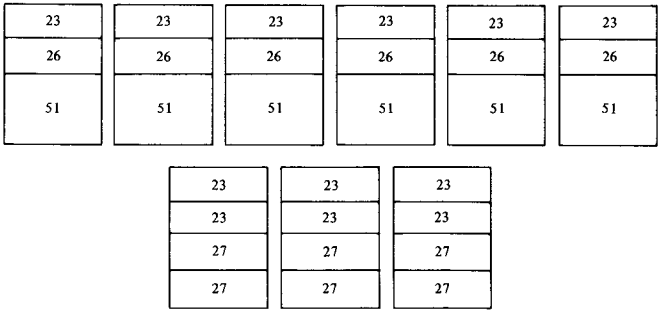
The deceptively simple appearance of this bound for FFD(L) gives little evidence of the substantial difficulties one encounters in trying to prove it. The only proof known at present, due to David Johnson at Bell Laboratories, runs over 75 pages in length. In contrast to the situation for $^{17}/_{10}$, no one currently understands the appearance of the fraction $^{11}/_9$.

One reason for the complex behavior of various bin packing algorithms is the existence of certain counterintuitive anomalies. For example, suppose the list of weights we are required to pack is

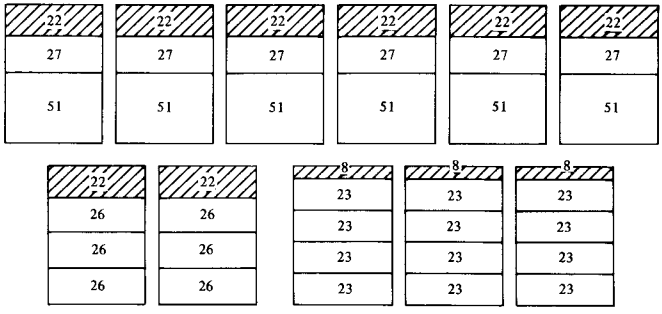
442	252	127	106	37	10	10
252	252	127	106	37	10	9
252	252	127	85	12	10	9
252	127	106	84	12	10	
252	127	106	46	12	10	

51	27	26	23	23
51	27	26	23	23
51	27	26	23	23
51	27	26	23	23
51	27	26	23	23
51	27	26	23	23

List of weights



Optimal packing in bins of capacity W = 100



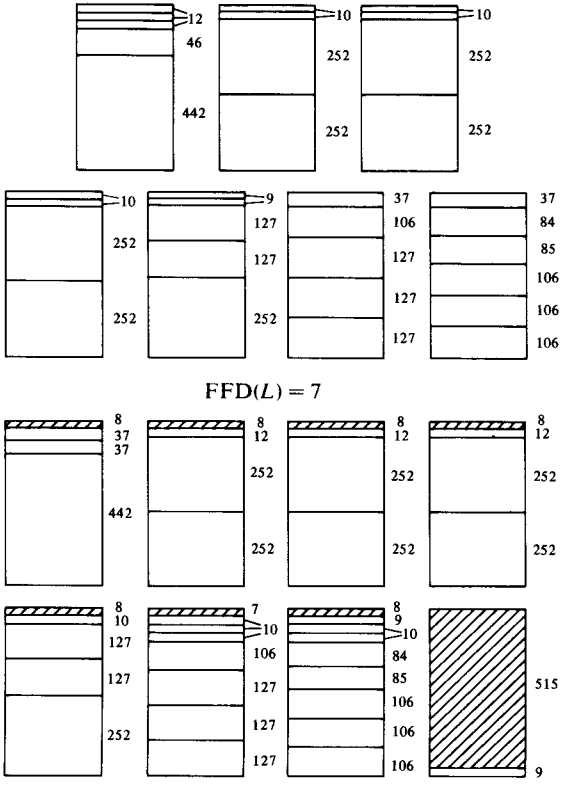
First-fit decreasing algorithm requires 11 bins.

Figure 13. First-fit packing of the listed weights arranged in decreasing order requires 11 bins of capacity W = 100 instead of the optimal 9. By repeating each weight n times, we can create as large an example as desired which has this $^{11}/_{9}$ ratio.

and suppose the bin capacity W is 524. In Figure 14 we show a first-fit decreasing packing of L; seven bins are required. However, if the weight 47 is removed from the list, a first-fit decreasing packing of the diminished list L' now requires *eight* bins (see Figure 14). Not surprisingly, this kind of behavior can cause serious difficulties in a straightforward approach to bin packing problems.

442 252 127 106 37 10 10
252 252 127 106 37 10 9
252 252 127 85 12 10 9
252 127 106 84 12 10
252 127 106 46 12 10

List of weights



FFD(L) = 7

FFD (L ↓) = 8 Where L ↓ = L −{47}.

Figure 14. Decreasing first-fit packing of the listed weights requires 7 bins of size 524, but if one weight is removed from this list, then this same algorithm requires 8 bins!

Pattern Arrangements

One natural extension of the bin packing problem just discussed is the so-called two-dimensional bin packing problem. In this we are given a list of plane regions of different sizes and shapes which we are to pack without overlapping into as few as possible "standard" regions. Common examples

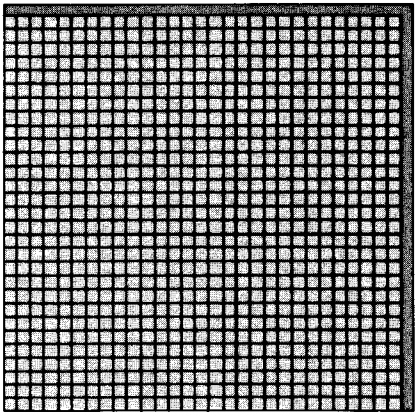


Figure 15. The obvious packing of $(100{,}000)^2$ unit squares into a large square whose side is 100,000.1 leaves slightly over 20,000 square units uncovered. But no unit square can fit in the uncovered region.

are the placement of sewing patterns onto pieces of material, or the arrangement of patterns for shoe components on irregular pieces of leather. It might be thought that the main source of difficulty here is due primarily to the irregularities in the shapes of the pieces involved. However, even for pieces with very regular shapes, it is usually far from obvious what the best packings are. Indeed, such questions will probably never be completely answered.

As an example, let us consider the following simple geometrical packing problem: How many nonoverlapping unit squares can be placed inside a large square of side s? Of course, if s is equal to some integer N then it is easy to see that the right answer is N^2. But what if s is *not* an integer, e.g., $s = N + {}^1/_{10}$. What should we do in this case? Certainly one option is to fill in an N by N subsquare with N^2 unit squares in the obvious way (see Figure 15) and surrender the uncovered area (nearly s/5 square units) as unavoidable waste. But is this really the best which can be done? Quite surprisingly the answer is *no*. It has been shown very recently by Paul Erdös of the Hungarian Academy of Science, Hugh Montgomery of the University of Michigan, and the author, that as s becomes large, there actually exist packings for any s by s square which leave an uncovered area of at most $s^{(3-\sqrt{3})/2} \approx s^{0.634\ldots}$ square units; this is significantly less than the s/5 units of uncovered area left by the obvious packing when s becomes very large. For example, when $s = 100{,}000.1$ the conventional packing (as in Figure 15) fits in $100{,}000^2 = 10^{10}$ unit squares, leaving slightly over 20,000 $(= 100{,}000/5)$ square units uncovered. However, since $100{,}000.01^{0.634} = 1479.11$, it is possible by clever packing to fit in more than 6000 extra squares (see Figure 16). The figure $s^{0.634}$ is probably not the best possible ultimate bound for large values of s; it seems to be extremely difficult to decide what the

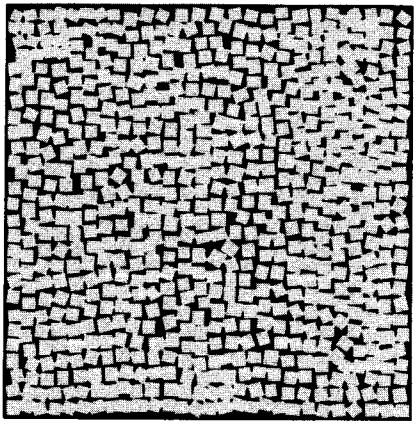


Figure 16. By placing the squares into a different arrangement, it is possible to fit more than $(100{,}000)^2 + 6000$ unit squares into a large square whose side is 100,000.1. The area left uncovered in the new arrangement is less than in the standard arrangement. (The illustration only suggests how this rearrangement may be done. The actual pattern cannot be drawn with as few squares as appear above.)

correct order of growth for the unavoidably uncovered area really is, although $\sqrt{s} = s^{0.5}$ looks like a likely candidate.

The particular scheduling model we have examined in this article, while quite simple and basic, still possesses enough structure to exhibit the unpredictable behavior so typical of more complex (and more realistic) situations. What happens is that processors start relatively unimportant tasks (since unnecessary idleness is not permitted) and having started, cannot stop until these tasks are completed, thereby delaying tasks which had since become more urgent. The fact that such behavior is not uncommon in real world scheduling situations testifies to the reasonableness of the model's assumptions.

The problem of deciding the proper order in which the tasks should be chosen so as to minimize the overall finishing time is extremely difficult. In most cases of realistic complexity, we must be satisfied with obtaining solutions which (we hope) are reasonably close to the optimum. Fortunately, it is often possible to find efficient procedures for closely approximating optimal solutions; this is frequently the most useful approach for tackling scheduling problems; in fact, for NP-complete problems it is the only general method that offers any hope at present.

Naturally, many extensions of our basic model are possible. They can include, for example, interruption of tasks before completion, introduction of various resource requirements for the tasks, random arrival of tasks, nonidentical processors and different performance measures, to name a few. By subjecting these extended models to the type of analysis we have described, researchers today are rapidly gaining insight into the very difficult problems of scheduling.

Suggestions for Further Reading

General

Graham, Ronald L. and Garey, Michael R. The limits to computation. 1978 *Yearbook of Science and the Future*. Encyclopaedia Britannica, 1977, pp. 170–185.

Knuth, Donald E. Mathematics and computer science: coping with finiteness. *Science* **194** (December 17, 1976) 1235–1242.

Kolata, Gina Bari. Analysis of algorithms: coping with hard problems. *Science* **186** (1974) 520–521.

Steen, Lynn Arthur. Computational unsolvability. *Science News*, **109** (1976) 298–301.

Technical

Garey, Michael R., Graham, Ronald L., and Johnson, D.S. Performance guarantees for scheduling algorithms. *Operations Research* 26(1978) 3–21.

Garey, Michael R. and Johnson, David S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1978.

Graham, Ronald L. Bounds on the performance of scheduling algorithms. In Coffman, E. G., *Computer and Jobshop Scheduling Theory*. Wiley, New York, 1976.