Notes on Scheduling

Ernest Davis

Problem 1:

Every task $I = 1 \dots N$ has a length $L[I]$ and a deadline $D[I]$. There is a single processor that can execute one task at a time. Is it possible to satisfy all the deadlines?

Solution: There is a greedy algorithm that finds a solution if one exists.

Let $S[I] = D[I] - L[I]$ be the latest possible starting time for I.
Execute the tasks in increasing order of $S[I]$. If there is any solution satisfying the constraints, this is one.

Proof: Suppose that you run this algorithm and it doesn't work; task K runs late, so it was started later than $S[K]$. However, all the tasks that were executed before K had starting times earlier than $S[K]$, so none of them can be started later than $S[K]$. Let U be the set of all tasks with start time earlier than $S[K]$; then K was started at time $\sum_{I \in U} L[I] > S[K]$. So all of U must be started before $S[K]$ (by definition) and there is no way to complete all of U before $S[K]$. So if this algorithm doesn't satisfy the schedule, the schedule can't be satisfied.

Problem 2

Every task $I = 1 \dots N$ has length 1, and value $V[I]$. There is an overall deadline D, an integer. Find a schedule that maximizes the sum of the tasks executed before D.

The solution is a trivial greedy algorithm: Execute the D most valuable tasks.

Problem 3

Every task has length 1, value $V[I]$ and integer deadline $D[I]$. You will be paid $V[I]$ if you complete the task I before $D[I]$ and 0 for I if you do not. Find the optimal schedule.

Solution: Sort the tasks in decreasing order of V. Let D be the maximum value of $D[I]$ and construct a matrix $T[0...D]$ where $T[M]$ is the task scheduled for time M. Then,

```
for (I=1 to N)
  if (there are any open slots in T less than D[I]-1) {
     Q = the latest open slot in T less than D[I]-1;
     T[Q] = I; }
end for
```

It's intuitively obvious that this is right, because each task T occupies only one slot, and therefore if T weren't there, there would only be room for one other task, that currently isn't being scheduled, and that task must be of lower value. Moreover, by scheduling T as late as possible, you make sure that it interferes with as few lower-value tasks as possible.

To make this rigorous, use the following loop invariant: "The schedule has the greatest possible value V for all schedules of tasks $1..I$; and no schedules with value V has more open slots less than W for any W between 1 and D." Assume that this holds for $I-1$, and we are now on iteration I. Let S be the schedule over $1...I-1$ to which we are now trying to add I.

If there is an empty slot Q less than $D[I]-1$, and we put I there, then clearly the new schedule has the greatest possible value for schedules over $1..I$. Moreover, for $W > Q$, we have reduced the number of open slots less than W by 1, but any placement of I must do that, and for $W < Q$, we have left the number of open slots less than W the same. So the statement about the open slots remains ture.

Suppose there is no empty slot less than $D[I]-1$. Let Z be the index of the smallest empty slot; thus $Z \geq D[I]$ and all the slots less than Z are filled with tasks more valuable than I, with a starting time less than Z (otherwise, they would have been put in Z or later). Suppose that S' is a schedule over $1 \dots I$ that includes I. Since I must be scheduled before $D[I]$, there must some task J that is scheduled for $1...Z$ in S' which is not scheduled in S. But since $V[J] > V[I]$, removing J and adding I cannot be a net gain in going from S to S'. So S' is less valuable than S. (The tasks scheduled for times greater than Z in S don't matter, since these can obviously be assumed to be the same in S' as in S.

Problem 4

Every task I has a length $L[I]$ and a value $V[I]$. There is an overall deadline D. Choose the tasks that maximize the total value.

This problem is also known as the knapsack problem (think of the lengths as weights, and the deadline as the maximal amount you can carry). It is a hard problem in general. In fact it is an NP-hard problem, meaning that there is very good reason to believe that is no polynomial time algorithm that gives the optimal solution.

There are two general approaches to hard optimization problems.

(1) Construct a polynomial-time algorithm that gives a good solution, though not the optimal solution. These are often greedy algorithms. Sometimes you can prove that they give an answer that is "not too much worse" than the optimal answer; in that case, they are known as approximation algorithms.

(2) Construct an algorithm that finds the optimal solution, that is exponential in the worst case, but runs as efficiently as possible. Such an algorithm must be designed so that all the options are considered, but it may be possible to reject some dead-ends at an early stage.

A common design here is "branch and bound". You build up a solution — e.g. a schedule — piece by piece. When you have a complete schedule, you record its value. Then you go back to the last choice point, and try other option. If you can see that an option can't possibly lead to a better solution than the one you already have, then there is no point in pursuing it further.

In running the branch and bound option, it is often a good idea to start by constructing the solution in (1) and, more generally, to consider the options in the order suggested by the idea in (1). That way, you have a reasonably good solution at the start to use as your bound.

Consider the following example of the knapsack problem

Task	A	B	C	D	E	F
Length	100	70	50	40	10	2
Value	150	140	90	70	30	8

Deadline:100.

A simple greedy algorithm is this: List the tasks in decreasing order of value, and add every task if it fits. Applying that algorithm, one would schedule task A with value 150 and length 100. Now the time available is full, so nothing more can be added. The solution has value 150.

A cleverer greedy algorithm would be to enumerate the tasks in decreasing order of value per length: F (8/2=4), E (30/10=3), B, C, D, A, In that case we add F, then add E then add B, then skip C, D, and A, which don't fit. The solution has value 8+30+140=178.

To do a systematic search, we will begin by listing the tasks in decreasing order of value per length, as in our second greedy algorithm. We will then build up the set of tasks in order, considering first the option of including the task, then the option of not including it. Whenever the length of an task is greater than the room remaining, it automatically gets included in the set Out.

If the total value of the tasks not in Out is less than the best solution we have found, then we can prune this branch; we will certainly not beat the best solution we've found without changing something. (One could formulate stronger pruning criteria, but not easily.)

The search proceeds as follows

```
In {} Out {}   Length=0.  Value: 0.                        Best solution: 0
   Include F. A no longer fits.
   In {F} Out {A}  Length=2.  Value: 8                     Best solution: 8


      Include B. C and D no longer fit.
      In {F,B}  Out {A,C,D} Length=72.  Value=148.         Best solution: 148

        Include E.
        In {F,B,E}  Out {A,C,D}. Length=82. Value=178.     Best solution: 178

                 All tasks decided. Backtrack,

        Exclude E
          In {F,B} Out: {C,D,E,A}  Length=72. Value=148.

      Exclude B. In{F}  Out{A,B}
         Include C.
         In {F,C} Out {A,B}.  Length=52.  Value=98.

           Include{D}. E no longer fits.
           In {F,C,D}. Out. {A,B,E} Length=92.

           All tasks decided. Backtrack.

Exclude F.
   Include B. A,C,D no longer fit.
   In {B} Out{F,A,C,D}. Total value of tasks not in out is 170.  Prune.

   Exclude B.
      Include C. A no longer fits.
      In {C} Out{F,B,A}. Length=50. Value=90

        Include D.
        In {C,D}. Out {F,B,A}. Length=90. Value=160.
```

```
          Include E.
          In {C,D,E}.  Out {F,B,A}. Length=100. Value=190.  Best solution:190.
             All tasks decided. Backtrack

[Quickly determined that there are no other option]

Answer: {C,D,E}. Value=190.
```

Note that you are in effect carrying out a depth-first search through a tree whose nodes are the In sets:

```
{}--->{F}--->{F,B}--->{F,B,E}
   |      |
   |      |->{F,C}--->{F,C,D}
   |
   |->{B}
   |
   |->{C}--->{C,D}--->{C,D,E}
```

This is known as an *implicit* or *virtual tree* (or *state space*) because you do not start given the tree as a whole, you construct the states as you need them. There is no need to construct this tree as a data structure; that just wastes space.

Problem 5

Each task I has a length $T[I]$. Tasks are organized in a partial order, represented as a DAG; if there is an arc $U \rightarrow V$, then task U must be completed at the time that task V is started. There are two processors. Find the schedule with the earlier possible overall completion time.

Again, the problem is NP-hard.

Note, first, that *any* sensible strategy follows the following general outline: whenever a processor becomes free, you assign it some task that is ready to run; that is, its predecessors have all completed. In this problem, nothing is ever gained by waiting. The only question is, when several tasks are ready to run, which do you choose? (You may have seen a similar problem in operating systems but there it's different for two reasons: (a) That's a dynamic, eternal setting, where tasks keep getting added to the system. (b) The objective here is simply to have the minimal overall completion time; the objective in the OS setting involves a more complicated mix of goals, including various kinds of fairness.)

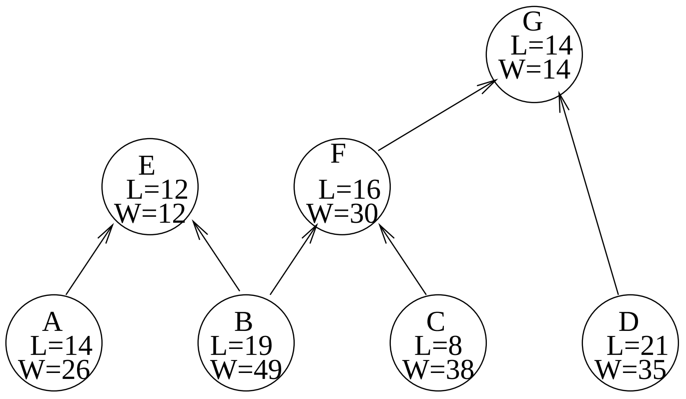
We can write this general strategy in pseudo-code as follows: We use an array $F[1..2]$ where $F[P]$ is the last finishing time for the tasks thus far assigned to P. $S[I]$ is the start time for I and $A[I]$ is the processor assigned to I.

```
F = [0,0];
repeat until (all tasks are complete) {
   if (F[1] < F[2]) then P=1 else P=2;
   find a task I that is ready to run; that is, all its prerequisites have
       completed;
   A[I] = P;
   S[I] = F[P];
   F[P] = F[P] + L[I];
}
```

It is not immediately obvious what is a good strategy for choosing the task I, but the following seems plausible. In the dependency DAG, add a new vertex "DONE" and add an arc from every vertex with no outarcs to "DONE". Now label every arc $U \rightarrow V$ with $L[U]$; this is the minimal delay between $S[U]$ and $S[V]$. Now compute $W[I]$ as the cost of the *longest* path from I to DONE. (This is easily done in linear time using DFS.) Then, if I is not scheduled earlier than time T, then the set of tasks on the longest path must be done in sequence, and so the whole set certainly cannot be finished earlier than $T + W[I]$. Therefore it seems plausible at each stage to choose the task with the largest value of $W[I]$; necessarily, that task is ready to run, since any preconditions must have larger values of $W[I]$.

The value of $W[I]$ can also be used as a lower bound on the best possible extension of the current partial schedule, for pruning in branch-and-bound, as follows. Let I be any task that has been scheduled; then the overall completion time must be at least $\max_I S[I] + W[I]$. On the other hand, if I is not scheduled and ready at time T, then the overall completion time must be at least $T + W[I]$ Therefore, if we have already found a solution that is better than the maximum of these two expressions over all I, then we need not further explore extensions of the current schedule.

For example, suppose that we have the system of tasks shown below. The values of L and W are shown in each node.



The algorithm proceeds as follows (note that the first solution found corresponds to the simple greedy algorithm).

```
P1=[].  P2=[]

   T=0. Ready: A,B,C,D.
   Assign B to P1 starting at T=0
   P1=[B] ends at 19.  P2=[]

      T=0. Ready: A,C,D.
      Assign C to P2 starting at T=0
      P1=[B] ends at 19.  P2=[C] ends at 8.

         T=8.  Ready: A,D.
         Assign D to P2 starting at T=8
         P1=[B] ends at 19.  P2=[C,D] ends at 29.
   T=19. Ready: A,F
   Assign F to P1 starting at T=19
   P1=[B,F] ends at 35. P2=[C,D] ends at 29.

       T=29.  Ready: A.
       Assign A to P2 starting at T=29.
       P1=[B,F] ends at 35. P2=[C,D,A] ends at 43.

          T=35. Ready: G.
          Assign G to P1 starting at T=35.
          P1=[B,F,G] ends at 49. P2=[C,D,A] ends at 43.

             T=43. Ready: E.
             Assign E to P2 ending at T=55.
             P1=[B,F,G] ends at 49. P2=[C,D,A,E] ends at 55.

                                                Best solution: 55.

    Backtrack to last choice point at T=19.
    P1=[B] ends at 19.  P2=[C,D] ends at 29.
    Assign A to P1 starting at T=19.
    P1=[B,A] ends at 35. P2=[C,D] ends at 29.

       T=29. Ready: F.
       But S[F]+W[F] = 59, so this can't lead to a better solution.

Backtrack to last choice point at T=8.
P1=[B] ends at 19.  P2=[C] ends at 8. Ready: A,D
Assign A to P2 starting at T=8.
P1=[B] ending at 19. P2=[C,A] ending at T=22.

     T=19. Ready: D,F.
     Assign D to P1 at T=19.
     P1=[B,D] ending at T=40.  P2=[C,A] ending at T=22.

        T=22. Ready: E,F.
        Assign F to P2 starting at 22.
        P1=[B,D] ending at T=40. P2=[C,A,F] ending at T=38.

           T=38. Ready: E.
           Assign E to P2, starting at T=38.
           P1=[B,D] ending at T=40.  P2=[C,A,F,E] ending at T=50.

              T=40. Ready: G.
              Assign G to P1, starting at T=40.
              P1=[B,D,G] ending at T=54. P2=[C,A,F,E] ending at T=50.

                                           New best solution: 54.

Note that S[D]+W[D]=54.
Therefore, any better solution must assign D earlier than T=19.
```

```
        Last choice point before T=19 is T=0 (second choice).

    Backtrack to the second T=0.
    P1=[B] ends at 19.  P2=[]. A,C,D are ready.
    Assign D to P2 ending at 21.
    P1=[B] ends at 19, P2=[D] ends at 21.

       T=19.  Ready: A,C.
       But T+W[C]=57, so this can't lead to a better solution.


    Backtrack to the second T=0.
    P1=[B] ends at 19.  P2=[]. A,C,D are ready.
    Assign A to P2 ending at T=14.
    P1=[B] ends at 19.  P2=[A] ends at 14.

       T=14. C,D are ready.
       Assign C to P2 starting at 14.
       P1=[B] ends at 19.  P2=[A,C] ends at 22.

           T=19. Ready: D.
           But T+W[D]=54, so this can't lead to a better solution.

  Backtrack to the first T=0

and so on.
```

‘

The overall search tree is as follows:

```
[/]--->B/ --->B/C--->B/CD--->BF/CD--->BF/CDA--->BFG/CDA--->BFG/CDAE (55)
    |      |      |       |
    |      |      |       |->BA/CD
    |      |      |
    |      |      |->B/CA--->BD/CA--->BD/CAF--->BD/CAFE--->BDG/CAFE (54)
    |      |
    |      |->B/D
    |      |
    |      |->B/A--->B/AC  [Point X: note below]
    |
    |->C/ --->C/B--->CD/B--->CD/BF (prune on T=29+W[A]=55)
    |      |  [Point Y: note below]
    |      |
    |      |->C/D (prune on T=8+W[B]=57)
    |      |
    |      |->C/A (prune on T=8+W{B]=57)
    |
    |->D/ --->D/B (prune on T=19+W[C]=57)
    |       |
    |       |->D/C (prune on T=8+W[B]=57)
    |       |
    |       |->D/A (prune on T=8+W[B]=57)
    |
    |->A/ --->A/B--->AC/B (prune on T=19+W[D]=54)
           |
           |->A/C (prune on T=14+W[B]=63)
           |
           |->A/D (prune on T=14+W[B]=63)
```

Notes on the above. First, one can actually stop the search at Point X, using the following reasoning: If B is not scheduled to start at time 0, then it must be scheduled after some other task. The shortest task is C with length 8, so B cannot be scheduled before time 8. However 8+W[B]=57, so this can't lead to a better solution.

Second, at point Y, where you have the state P1=[C], P2=[B], the search algorithm should recognize that this is actually no different from the state P1=[B], P2=[C], which has already been explored, since the two processors are identical. Therefore there is no point in following this branch further.

The general problem is that there is no end to the clever pruning strategies of this kind that can be thought up, and it is difficult to know which are worth implementing. Of the two above, the second, which is part of the general category of recognizing repeated states under symmetric transformations, certainly is worth implementing; the first is much more doubtful.