Iteration	*Error*
1	*0.00101410035182936232807644 0339*
2	*0.00000000737625099203510885 1841*
3	*0.00000000000000000018313060847 8*

Table 3.1: Errors in Calculating π

"tail." However, most of the probability is concentrated in the interval $[0, t_1]$. *Hence, we might choose to set the run-time limit at* t_1, *rather than at the worst-case execution time.*

If a primary version completes successfully, we do not need its corresponding alternative for that period. This time can thus be reclaimed. Such reclamation can result in time becoming available for other primaries to be executed, which were not part of the original schedule. The algorithm for doing so is quite a simple modification of what we have seen above, and is left to the reader as an exercise.

3.3 Uniprocessor Scheduling of IRIS Tasks

Thus far in this chapter, we have assumed that to obtain acceptable output, a task had to be run to completion. Put another way, if the task is not run to completion, we get zero reward from it (i.e., it may as well not have been run). However, there is a large number of tasks for which this is not true. These are *iterative* algorithms. The longer they run, the higher is the quality of their output (up to some maximum runtime).

Example 32 *Figure 3.32 contains an algorithm to compute the value of* π. *The greater the number of times that step 2 is executed, the more accurate is* P *as an approximation of* π *(subject, of course, to limitations due to finite numerical precision).*

The error from the actual value of π *as a function of the iteration number is provided in Table 3.1. The error is greatest for the first iteration; it diminishes rapidly after that.*

Search algorithms for finding the minimum of some complicated function are also another example of such tasks. The longer we search the parameter space, the greater is the chance that we will obtain the optimum value, or something close to it.

1. *Set* $A = \sqrt{2}$, $B = \sqrt[4]{2}$, $P = 2 + \sqrt{2}$.

Repeat Step 2 as long as necessary.

2. *Compute*

$$\begin{aligned} A &:= \frac{\sqrt{A} + 1/\sqrt{A}}{2} \\ P &:= P\left(\frac{A+1}{B+1}\right) \\ B &:= \frac{B\sqrt{A} + 1/\sqrt{A}}{B+1} \end{aligned}$$

B *is an approximation of* π.

Figure 3.32: Algorithm for Calculating π

Example 33 *Chessplaying algorithms evaluate the goodness of moves by looking ahead several moves. The more time they have, the further they can look, and the more accurate will be the evaluation.*

Tasks of this type are known as *Increased Reward with Increased Service* (IRIS) tasks. The reward function associated with an IRIS task increases with the amount of service given to it.

Typically, the reward function is of the form

$$R(x) = \begin{cases} 0 & \text{if } x < m \\ r(x) & \text{if } m \le x \le o + m \\ r(o+m) & \text{if } x > o + m \end{cases} \tag{3.82}$$

where $r(x)$ is monotonically nondecreasing in x. The reward is 0 up to some time m: if the task is not executed up to that point, it produces no useful output. Tasks with this reward function can be regarded as having a *mandatory* and an *optional* component. The mandatory portion (with execution time m) must be completed by the deadline if the task is critical; the optional portion can be done if time permits. The optional portion

requires a total of o time to complete. In each case, the execution of a task must be stopped by its deadline, d.

The scheduling task can be described as the following optimization problem:

> *Schedule the tasks so that the reward is maximized, subject to the requirement that the mandatory portions of all the tasks are completed.*

It can be shown that this optimization problem is NP-complete when there is no restriction on the release times, deadlines, and reward functions. However, for some special cases, we do have scheduling algorithms. We now turn to studying these. In what follows, m_i and o_i denote the execution time of the mandatory and optional parts, respectively of T_i.

3.3.1 Identical, Linear Reward Functions

For task T_i, the reward function is given by

$$R_i(x) = \begin{cases} 0 & \text{if } x \leq m_i \\ x - m_i & \text{if } m_i \leq x \leq o_i + m_i \\ o_i & \text{if } x > m_i + o_i \end{cases} \tag{3.83}$$

That is, the reward from executing a unit of optional work is one unit. A schedule is said to be optimal if the reward is maximized subject to all tasks completing at least their mandatory portions by the task deadline.

Theorem 13 *The EDF algorithm is optimal if the mandatory parts of all tasks are 0.*

Proof: *If the mandatory portions are zero, then we can execute as little of any task as we please. It is easy to see that reward is maximized if the processor is kept busy for as much time as possible. But this is exactly what the EDF algorithm does: if the processor is idle at some time t, that is because* (a) *all the previously released tasks have either completed or their deadlines have expired by time t, and* (b) *no other tasks have been released.* **Q.E.D.**

We can use this result to obtain an optimal scheduling algorithm for the case when the mandatory portions are not all zero. The tasks $T_1, \cdots, T_n$

have mandatory portions $M_1, \cdots, M_n$, and optional portions $O_1, \cdots, O_n$. Define

$$\begin{aligned} \mathbf{M} &= \{M_1, \cdots, M_n\} \\ \mathbf{O} &= \{O_1, \cdots, O_n\} \\ \mathbf{T} &= \{T_1, \cdots, T_n\} \end{aligned}$$

The optimal algorithm, IRIS1, is shown in Figure 3.33. Although it looks a little forbidding, the basic idea behind it is quite simple. First, since we receive one unit of reward for each unit of the optional portion completed (for any task), the highest reward, subject to the constraint that all mandatory portions are completed, is obtained when the processor carries out as much execution as possible.

We begin by running the EDF algorithm for the total run time of each task. Call the resulting schedule S_t. S_t maximizes the total processor busy time. If S_t is a feasible schedule, we are clearly done: we have given each task as much time as it needs to finish executing both its mandatory and optional portions, and still met each task deadline. Suppose we do not obtain a feasible schedule. That is, some task cannot be given its full execution time and still meet its deadline. In that case, we run the EDF algorithm on the mandatory portions of each task, to yield schedule S_m. If this results in an infeasible schedule, then we must stop since we can't even execute the mandatory portions of each task. Suppose that S_m is feasible. Then, we adjust S_t to ensure that each task receives at least its mandatory portion of service.

Example 34 *Consider the set of four tasks with parameters shown in the following table.*

Task Number	m_i	o_i	r_i	D_i
1	*1*	*4*	*0*	*10*
2	*1*	*2*	*1*	*12*
3	*3*	*3*	*1*	*15*
4	*6*	*2*	*2*	*19*

In Step 1 of IRIS1, we run the EDF algorithm with task execution times 5, 3, 6, and 8, respectively (for tasks 1 to 4), to produce S_{t_0} in Figure 3.34. It is impossible to meet the deadline of task 4. Hence, we go to step 2.

Running the EDF algorithm on the task set **M** *produces the feasible schedule S_m shown in Figure 3.34. All the deadlines are met, so we can*

1. Run the EDF algorithm on the task set $\mathbf{T}$ to generate a schedule, S_t.
If this is feasible,
An optimal schedule has been found: STOP.
Else,
go to step 2.
end if

2. Run the EDF algorithm on the task set $\mathbf{M}$, to generate a schedule S_m.
If this set is not feasible,
$\mathbf{T}$ cannot be feasibly scheduled: STOP.
Else,
Define a_i as the i'th instant in S_m when either the scheduled task changes, or the processor becomes idle, $i = 1, 2, \cdots$.
Let k be the total number of these instants.
Define a_0 as when the first task begins executing in S_m.
Define $\tau(j)$ as the task that executes in S_m in $[a_j, a_{j+1}]$,
Define $L_t(j)$ and $L_m(j)$ as the total execution time given to task $\tau(j)$ in $S_t(j)$ and $S_m(j)$ respectively, after time a_j.
Go to step 3.
end if

3. $j = k - 1$
do while $(0 \leq j \leq k - 1)$
if $(L_m(j) > L_t(j))$ then
Modify S_t by
(a) assigning $L_m(j) - L_t(j)$ of processor time in $[a_j, a_{j+1}]$ to $\tau(j)$, and
(b) reducing the processor time assigned to other tasks in $[a_j, a_{j+1}]$ by $L_m(j) - L_t(j)$.
Update $L_t(1), \cdots, L_t(j)$ appropriately.
end if
$j = j - 1$
end do
end

Figure 3.33: Algorithm IRIS1

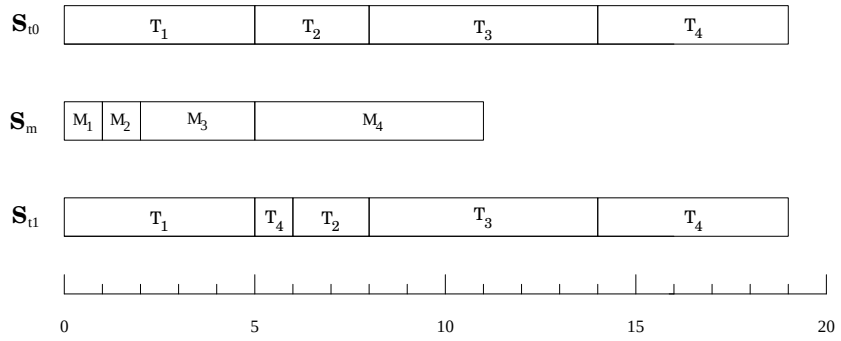

Figure 3.34: Schedules produced by IRIS1 for Example 34

proceed to Step 3. We have $a_0 = 0, a_1 = 1, a_2 = 2, a_3 = 5, a_4 = 11$. *Also,* $k = 4$.

Now, we move to Step 3 of the algorithm. Let us start with a_3. *We have task* T_4 *scheduled in* S_m *over the interval* $[a_3, a_4]$, *and given 6 units of time. In schedule* S_{t0}, T_4 *is given only 5 units of time. Hence, we modify* S_{t0} *by adding* $6-5 = 1$ *unit of time to* T_4 *in the interval* $[a_3, a_4)$, *and taking away 1 unit from the task originally scheduled at* a_3 *in* S_{t0}, *namely* T_2. *This results in task* T_4 *being scheduled for a total of 6 units beyond* a_3. *The resultant schedule is* S_{t1}. *Let us now move to the interval* $[a_2, a_3)$. T_3 *is scheduled beyond that time in* S_m, *for a total of 3 units. In* S_{t1}, T_3 *has been scheduled for a total of 6 units beyond* a_2. *So, no modifications are needed:* T_3 *has enough time to meet its mandatory portion. Next, we consider* $[a_1, a_2)$. T_2 *is scheduled for that interval in* S_m. *Let us consider the time given to* T_2 *beyond* a_1 *in* S_{t1}. *It is 2 units, which is greater than the mandatory requirement. So, no modifications are needed. Finally, consider* $[a_0, a_1)$. T_1 *is scheduled there in schedule* S_m *for one unit. In* S_{t1}, T_1 *is scheduled for 5 units, which exceeds the time given in* S_m. *So, no modifications are needed, and the optimal schedule is* S_{t1}.

Theorem 14 *For IRIS tasks with reward functions of the type being considered in this section, Algorithm IRIS1 is optimal.*

Proof: *We leave a formal proof to the reader. Here, we will merely sketch the ideas behind the proof. If a feasible schedule is generated in Step 1, then each task has been run to completion, and we are done. If not, then*

from Theorem 13, we know that the EDF algorithm is optimal when none of the tasks has any mandatory portion: in that case, the schedule we would obtain would be S_t in step 1. But, the transformations that we do in step 3 does not change the total time for which the processor runs: it only ensures that all mandatory portions are completed. This completes the proof sketch. **Q.E.D.**

3.3.2 Nonidentical, Linear Reward Functions

The reward function for task T_i is given by

$$R_i(x) = \begin{cases} 0 & \text{if } x \leq m_i \\ w_i(x - m_i) & \text{if } m_i \leq x \leq m_i + o_i \\ w_i o_i & \text{if } x > m_i + o_i \end{cases} \tag{3.84}$$

Each task has a weight, w_i, associated with it. Assume that the tasks are numbered in non-increasing order of weights, i.e., $w_1 \geq w_2 \geq \cdots \geq w_n$. The procedure for optimally scheduling such tasks is obvious: always run the available task with the greatest weight, subject to the need to execute the mandatory portions of all tasks by their respective deadlines. This is done by algorithm, IRIS2, which is shown in Figure 3.35.

The idea behind this algorithm is the following. As with IRIS1, we check to see if we can feasibly schedule all the mandatory portions. If not, we stop right away. If we succeed, we proceed by running IRIS1 with mandatory task set equal to the mandatory portions of the tasks and the set of optional portions equal only to optional portion of task T_1. That is, the optional portions of the other tasks are considered not to exist. IRIS1 is executed. It provides as much time as possible to T_1, consistent with the need to meet the mandatory portions of all the tasks.

We now take this schedule, and label as mandatory the part of the optional portion of T_1 that was scheduled by IRIS1. Next, we run the IRIS1 algorithm with this revised mandatory portion and the optional portion of T_2, and continue in this way for the remaining tasks. .

Theorem 15 *If the reward functions are as defined in this section, algorithm IRIS2 is optimal.*

Proof: *Once again, we will leave the formal proof as an exercise and merely provide a brief sketch. We know from Theorem 14 that O'_n is the maximum*

1. Set $\mathbf{M}'$ to be the set of mandatory portions of all the tasks, and $\mathbf{O}' = \emptyset$. Run the EDF algorithm.

2. If S_m is not feasible,
    the task set $\mathbf{T}$ is not schedulable: STOP.
else
    $i = 1$
    do while $(1 \leq i \leq n)$
        Set $\mathbf{O}' = \mathbf{O}' \cup \{O_i\}$, and use IRIS1 to find an optimal schedule
        Define O_i' to be the part of O_i scheduled by IRIS1.
        Set $\mathbf{M}' = \mathbf{M}' \cup \{O_i'\}$
        $i = i + 1$
    end do
end if
end

Figure 3.35: Algorithm IRIS2

amount of service that can be given to O_n (which is the task with the greatest weight) if all the mandatory tasks are to meet their deadlines. Similarly, we have that O'_{n-1} is the maximum amount of service that can be given to O_{n-1}, subject to the constraint that all mandatory tasks must meet their deadlines and that as much of O_n as possible should be executed. In general, for $i < n$, we have that O'_i is the maximum amount of service that can be given to O_i, subject to the constraint that all mandatory tasks must meet their deadlines and that as much of $O_{i+1}, \cdots, O_n$ as possible should be executed. The result follows from this observation. **Q.E.D.**

3.3.3 0/1 Reward Functions

We assume here that for any task, i, the reward function is given by

$$R_i(x) = \begin{cases} 0 & \text{if } x < m_i + o_i \\ 1 & \text{if } x \geq m_i + o_i \end{cases} \tag{3.85}$$

That is, we get no reward for executing the optional portion partially. If we run the optional portion to completion, we obtain one unit of reward, else nothing.

The optimal strategy would therefore be to complete as many optional portions as possible, subject to the constraint that the deadlines of all the mandatory portions must be met. Unfortunately, when the execution times are arbitrary, the problem of obtaining an optimal schedule can be shown to be NP-complete.

Finding an efficient optimal scheduling algorithm under the 0/1 case is therefore a hopeless task. We must therefore make do with heuristics. One rather obvious heuristic is shown in Figure 3.36. The algorithm is based on the following reasoning. Since we get the same reward for completing the optional portion of any task, it is best to run the tasks with the shorter optional portions. So, we assign weights according to the inverse of the duration of the optional portions, and run IRIS2. If an optional part is not run to completion in the resultant schedule, we remove its optional portion from consideration and rerun IRIS2. We continue in this manner until each optional portion has been either scheduled to completion or dropped altogether.

1. Run the EDF algorithm on the set **M** of mandatory tasks.
If **M** is not EDF-schedulable, then
    Task set **T** cannot be feasibly scheduled: STOP.
else
    Go to Step 2.
end if

2. **O** is the set of optional portions.
Assign $w_i = 1/o_i$ for $i = 1, \cdots, n$.
Renumber the tasks so that their weights are in a non-ascending sequence, i.e., $w_1 \geq w_2 \geq \cdots \geq w_n$.

3. Run algorithm IRIS2 on a task set composed of
    the mandatory set **M** and optional set **O** to obtain schedule S_o.

4. If all the optional tasks in **O** are executed to completion in S_o,
    Return S_o and STOP.
else
    Let i_{min} be the smallest index i such that
        o_i is not run to completion in S_o.
    Redefine $\mathbf{O} = \mathbf{O} - \{o_{i_{min}}\}$.
    Go to Step 3.
end if

end

Figure 3.36: Algorithm IRIS3: A Simple Heuristic for the 0/1 Case

3.3.4 Identical Concave Reward Functions; No Mandatory Portions

In this section, we consider tasks with identical release times, and whose mandatory portions are zero. We assume that the reward function of T_i is given by

$$R_i(x) = \begin{cases} f(x) & \text{if } 0 \leq x < o_i \\ f(o_i) & \text{if } x \geq o_i \end{cases} \tag{3.86}$$

where the function f is one-to-one and concave. Recall that a function $f(x)$ is concave iff for all x_1, x_2 and $0 \leq \alpha \leq 1$,

$$f(\alpha x_1 + [1-\alpha]x_2) \geq \alpha f(x_1) + (1-\alpha)f(x_2) \tag{3.87}$$

Geometrically, this condition can be expressed by saying that, for any two points on a concave curve, the straight line joining them must never be above the curve. An example of a concave function is $1 - e^{-x}$.

We will also assume that the functions $f(x)$ are differentiable, and define $g(x) = df(x)/dx$. We will assume that the inverse function g^{-1} of g exists for all $i = 1, \cdots, n$. This will happen if the functions g be monotonically decreasing: we assume that to be the case. The tasks are numbered in non-decreasing order of their absolute deadlines, i.e., $D_1 \leq D_2 \leq \cdots \leq D_n$. For notational convenience, define $d_0 = 0$.

Since f is a concave function, we have nonincreasing marginal returns, and so the optimum is obtained by balancing the execution times as much as possible. If all the deadlines were equal, i.e., if $D_1 = \cdots = D_n = \delta$, then the algorithm would be trivial: just allocate to each task a total of δ/n of execution time before its deadline. If the deadlines are not all equal, the algorithm is a little more complicated. We will leave to the reader the problem of writing out the algorithm, IRIS4, formally. Here is an informal description.

The basic idea behind this algorithm is to equalize, as much as possible, the execution times of the tasks. The algorithm starts at the latest deadline and works backwards. In the interval $[D_{n-1}, D_n]$, only task T_n can be executed: it is allocated up to $a_n = \max\{D_n - D_{n-1}, e_i\}$ units of time in that interval. Next, move to the interval $[D_{n-2}, D_{n-1}]$. Over this interval, tasks T_{n-1} and T_n can be executed. In this interval, we try to allocate time to T_{n-1} and T_n so that in the interval $[D_{n-2}, D_n]$, the execution time these tasks receive is equalized as much as possible (subject to the obvious

constraints). We then go on to the interval $[D_{n-3}, D_{n-2}]$, over which tasks T_{n-2}, T_{n-1}, T_n are available, and so on until the beginning.

Example 35 *We have a five-task aperiodic system with the following deadlines:* $D_1 = 2,\ D_2 = 6,\ D_3 = 8,\ D_4 = 10,\ D_5 = 20$, *and each task having execution time of 8.*

Let us begin with the interval $(10, 20]$. *Only task* T_5 *can be scheduled in that interval, and we can give to it its entire execution time of 8. So, the allocation of execution times so far is:*

T_1	T_2	T_3	T_4	T_5
0	*0*	*0*	*0*	*8*

Next, we move to the interval $(8, 10]$. *Tasks* T_4, T_5 *can be scheduled in that interval, but we have already given full execution time to* T_5*, so we don't consider that task here. We devote this entire interval to* T_4. *The execution time allocations are now:*

T_1	T_2	T_3	T_4	T_5
0	*0*	*0*	*2*	*8*

Now, consider $(6, 8]$. T_3, T_4, T_5 *are eligible to run in that interval. As before, we don't have to consider* T_5. *We give 2 units to* T_3 *so that it is equalized with* T_4. *(This is the best possible balancing of the execution times). The execution time allocation are now:*

T_1	T_2	T_3	T_4	T_5
0	*0*	*2*	*2*	*8*

Move on to $(2, 6]$. T_2, T_3, T_4, T_5 *are eligible to run in this interval. Add 2 units to* T_2 *so that* T_2, T_3, T_4 *are each allocated 2 units. This leaves 2 units which we can allocate equally to each of these tasks over that interval. The execution time allocation is now:*

T_1	T_2	T_3	T_4	T_5
0	*2.66*	*2.66*	*2.66*	*8*

Finally, consider $(0, 2]$. *Here, we must clearly allocate 2 units to* T_1*, and the final allocation is:*

T_1	T_2	T_3	T_4	T_5
2	*2.66*	*2.66*	*2.66*	*8*

It is easy to check that the execution times have been balanced as much as possible, under deadline and execution time constraints. The schedule is shown in Figure 3.37.

Theorem 16 *Algorithm IRIS4 is optimal under the conditions listed in this section.*