Running a job on a collection of dynamic machines, with on-line restarts

R. van Stee, H. La Poutré

Software Engineering (SEN)

SEN-R9841 December 1998

Report SEN-R9841
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Running a Job on a Collection of Dynamic Machines, with On-Line Restarts

Rob van Stee[*]

Han La Poutré
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

We consider the problem of running a job on a selected machine of a collection of machines. Each of these machines may become temporarily unavailable (busy) without warning, in which case the scheduler is allowed to restart the job on a different machine. The behaviour of machines is characterized by a Markov chain, and objective is to minimize completion time of the job. For several types of Markov chains, we present optimal policies.

1991 Computing Reviews Classification System: F.2.2, G.3
1991 Mathematics Subject Classification: 68M20, 90B35, 90C40
Keywords and Phrases: scheduling, Markov chains, computer systems, background jobs, restarts
Note: Work carried out under theme SEN4 "Evolutionary Computation and Applied Algorithmics"

1. Introduction

In networks of workstations, a considerable amount of capacity is unused, since the primary users are only using them part of the time. Such machines could therefore be used for large(r) jobs that can be executed in the background or with low priority. This means that such a job gets the "free time" of the machine, i.e., the time that no higher-priority job is using it. However, this does not mean that a larger job does not have any objectives in completion time.

When a workstation is used (for a higher-priority job), there is information available on the type of job that is executed. This is e.g. available from the process manager (process statuses). With this information, it could be decided what to do: e.g., to just wait until the workstation is available again, or to start the larger job on another machine. In this way, the completion time of this job could be minimized.

The above situation is not only true for workstations, but for e.g. supercomputers or other scarce high-performance computers that are available in smaller quantities as well. We therefore study the problem of executing a large job as a background job,

[*]Supported by SION/NWO, project number 612-30-002

where the completion time should be minimized and the job can be executed on one machine at a time. The latter also often follows from system management requirements.

To be precise, we study the problem of scheduling a job J on a machine out of a collection of machines that are not available continously, without having full knowledge about when they are available. At the start, the scheduler must pick one machine to run the job on. If the machine becomes temporarily unavailable the scheduler is allowed to restart the job from scratch on a different machine. The goal is to minimize the expected completion time.

As mentioned before, the job can only be assigned to one machine at a time. Practical reasons are e.g. intensive I/O, system management guidelines, heavy use of external data or resources, fairness between users, and availability for other large jobs. Because of this, several such jobs can be run. Also, moving a job to another machine means a restart; this can be due to extensive production of (local) data, usage or production of (local) code, setting up (local) connections to other resources, etcetera. Sometimes in practice, such a job can divided into parts, using checkpoints; in that case such parts can be taken as the job we consider in our model.

In [1], a method is discussed for a different but related situation, viz., where a job J must be run in a specific time interval. The assumption made on the availability of the workstations was that at least one of the workstations would be available for a certain amount of time (significanty larger than the time required to run the job) during the interval in which the job was to be run. Using this assumption, a method was shown which had an $1 - O(1/m)$ probability of choosing a "good" workstation, so that J is completed on time, where m is the number of workstations. However, with this approach it is not possible to determine or minimize the expected running time of J. As it turns out, in order to give bounds for the completion time, it is necessary to use a different approach.

We study the case where the (typical) behaviour of the workstations (with respect to availability) is captured by a Markov chain. This has similarities with the modeling in [5], where the paging problem was addressed in a similar way, i. e., by modeling the behaviour of a program by a Markov chain. One of the reasons not to use the adversary approach [2] is that it is easy to force an online algorithm to take a factor of m longer for this problem. Therefore, the method of distinguishing between online algorithms by examining their competitive ratio [2] appears not to be appropriate.

The Markov chain is a model of the workstation behaviour. Every behaviour can be modeled by such a chain, depending on the grain of description. E.g, the most simple chain can be obtained by having, besides a state for "available" (idle), one state for "unvailable", with the expected unavailibility time as its cost. Making more elaborate Markov chains based on (on-line) system statistics and additional information, enables finer grained description and improved scheduling strategies, yielding lower completion times. We also refer to [5] for some general comments on Markov chains.

We present an optimal scheduling strategy for running large jobs on partly available machines. The actual job size J does not need to be known (but it does not help to know it either). The computational complexity of our strategy is $O(n^3)$, where n is the number of nodes in the Markov chain. So, this only needs to be computed once for all future large jobs. The strategy only depends locally on the machine the job is running on. This is in contrast with [1], where global decisions are needed.

In the paper, we begin by looking at a simple Markov chain, where only one user-job size can occur. We then examine more complex Markov chains, where jobs of different sizes can occur. Finally, we look at the case where the interrupting jobs themselves form a Markov chain (i. e. more is known about the sequences in which jobs are often started), thus enabling a fine-grained description of machine behaviour.

2. The Model

We have a job J which takes d units of time to complete. (Although d does not need to be known in advance, throughout the paper, we use d as if it were known.) At any time, we can allocate exactly one machine from a collection of machines to run J. If the machine becomes temporarily unavailable, the scheduler is allowed to restart the job from scratch on a different machine. The goal is to minimize the expected completion time of J.

The behaviour of every machine is characterized by a Markov chain. One state of this chain, called the idle state, represents the situation that the machine is available for executing a (new) large job. Any other state represents a local job or a job session, that makes the machine unavailable for the scheduler. Such local jobs or job sessions can have different sizes. Only the expectation of the size of each such job or job session needs to be known, since we minimize the expected completion time of J. However, for reasons of simplicity, we henceforth consider a state to correspond to just one job with a fixed size. The conversion to job sessions and expected size of those is not made explicit any more, but this is trivial since only expected (completion) times are considered.

The machines are identical, in the sense that they are modeled by the same Markov chain. All machines behave independently of each other and of the decisions made by the scheduler. The scheduler may use the information of the Markov chain. We assume that if the scheduler wants to restart J, there is always a machine available. This is realistic, since we will show that in a network of some non-trivial size, the expected time for the first machine to become available, starting in a randomly chosen time step, is very small as long as the Markov chain does not yield extreme occupation in this network.

We will consider Markov chains that, if the idle state is deleted, become acyclic. Note that cycles in a given (theoretical) Markov chain can be approximated by replacing cycles by paths, e. g., cycle *abc* could be replaced by the path *abcabcabcabc* or longer versions. By including some of these (shorter) paths and representing other paths or path tails by a new state, the (theoretical) chain can be approximated to an arbitrary precision. For practical situations, however, this is not important, since a Markov chain is obtained and approximated from statistical information, and since approximating "infinite cycling behaviour" by just one or a couple of states will fall within the statistical and practical accuracies.

3. The Basic Case

3.1 Problem definition

All machines behave according to the Markov chain shown in Figure 1 (left). A more

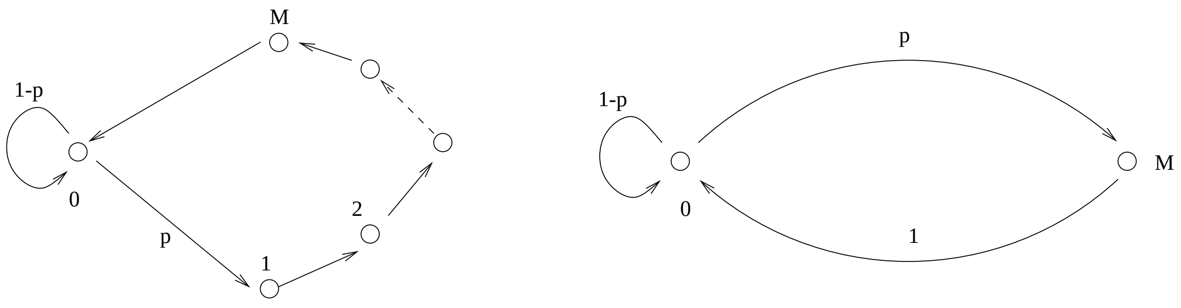

Figure 1: Markov chain of one machine in two forms

compact way of picturing this is shown on the right, where the M-node costs M units of time. This chain, together with the possibility of a restart, induces a Markov decision process on our job. We define J to be in state i if it has been worked on for i time steps since its latest restart, not counting the time that the current machine was busy (when a higher priority job was running on it). We then have the situation shown in Figure 2. Costs are in bold type.

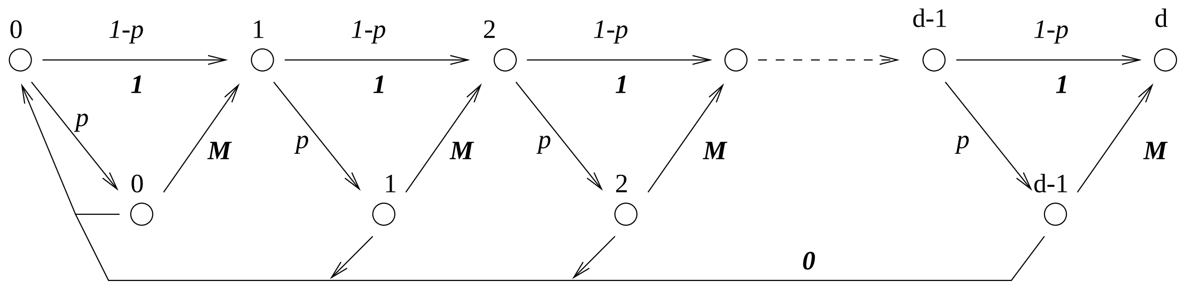

Figure 2: Markov chain of our job

In Markov decision theory, this is known as a first-passage problem [4]. Such problems can be solved using a linear program, but this requires introducing $2d$ variables, one for each node in the Markov chain. Solving a linear program with $2d$ variables can be done in $O(8d^3)$ time. This is clearly impractical, as this is far more than the running time of the (large) job itself. Furthermore, such a linear program would have to be solved for every occurring job size d. We show an optimal policy with time complexity $O(1)$, that is independent of, and does not need to know, d.

Clearly, in the top row of this Markov chain (representing the idle state), it is always optimal to continue. Only when the process moves to one of the nodes in the second row, meaning that the current machine is taken by a higher priority job, do we need to make a choice. For every state $0, \dots, d-1$, we need to decide what to do in case of such an interruption. Do we restart J, or pay M? When we reach state d, the job is finished.

3.2 The structure of the optimal policy

It is known [4] that for any first-passage problem there is an optimal policy that is stationary: it does not depend on the total time that the job has been running, or the number of times it has been in the current state. Also, it is deterministic. In [4], linear programming is used to obtain the optimal policy. Here it is possible to use a more efficient approach.

A policy will be denoted by a vector $a = (a_0, \dots, a_{d-1})$, where $a_i = 1$ means the scheduler will restart J if it gets interrupted in state i, and $a_i = 0$ means he will not restart. Define $f(i)$ to be the expected minimal costs (running time) to complete J, starting in state i. These costs satisfy $f(d) = 0$ and

$$\begin{aligned} f(i) &= (1-p)(1+f(i+1)) \\ &+ p \cdot \min\{f(0), M + f(i+1)\} \qquad i = 0, \dots, d-1. \end{aligned} \tag{3.1}$$

This holds because the probability of going directly to the next state is the probability of remaining in the idle node, $1-p$, and the optimal costs in that case are $1+f(i+1)$. When the machine becomes busy, the minimal costs are the minimum of the two choices there: restarting costs $f(0)$, and waiting costs $M + f(i+1)$.

It follows from (3.1) that a restart in state i is optimal if and only if

$$f(0) \le M + f(i+1), \tag{3.2}$$

and in that case restarting is optimal in all the previous states as well, since $f(i)$ is monotonically decreasing: $a_i = a_{i-1} = \cdots = a_0 = 1$.

It follows that an optimal policy is a *threshold policy*: interruptions cause restarts only up to a certain point. Therefore an optimal policy is of the form $a(k)$:

$$a(k) = (1, \overset{k}{\dots}, 1, 0, \overset{d-k}{\dots}, 0) \qquad k \in \{0, \dots, d\}.$$

Here k indicates the number of steps for which an interruption causes a restart, e. g. $k = 2$ means $a_0 = 1, a_1 = 1, a_2 = \cdots = a_{d-1} = 0$. We have $k \ge 1$ since in state 0, restarting is always cheapest.

3.3 When is the threshold reached?

It follows from (3.2) that restarting is optimal as long as $f(0) - f(i+1) < M$. Since $f(0) - f(i+1)$ is the expected total optimal cost minus the expected optimal cost starting in $i+1$, in other words, the expected optimal cost to reach $i+1$ for the first time, starting in 0, we need to calculate $C(k)$: the expected cost to reach k for the first time using strategy $a(k)$, and find the smallest k for which this is greater than the threshold M.

Define R_k as the event that a restart occurs before reaching state k, then $\mathbb{P}(R_k) = 1-(1-p)^k$. We write $C_R(k)$ for the cost until a restart, given that this occurs before k is reached. After a restart the costs are again $C(k)$. Using that the expectation of a random variable $\mathbb{E}X = \mathbb{E}(X|Y)\mathbb{P}(Y) + \mathbb{E}(X|\neg Y)\mathbb{P}(\neg Y)$ we can see that

$$C(k) = (C_R(k) + C(k))(1-(1-p)^k) + k(1-p)^k \tag{3.3}$$

or

$$C(k) = C_R(k)\frac{1-(1-p)^k}{(1-p)^k} + k. \tag{3.4}$$

Since $\mathbb{E}(X|Y) = \sum_{x\in Y} x\mathbb{P}(X=x)/\mathbb{P}(Y)$, we have that

$$C_R(k) = \frac{\sum_{i<k} i \cdot \mathbb{P}(\text{restart after } i \text{ steps})}{(1-(1-p)^k)}. \tag{3.5}$$

Using

$$\sum_{i=0}^{k-1} ip(1-p)^i = \frac{1-p}{p}(1-(1-p)^k) - k(1-p)^k$$

in (3.5), and combining this with (3.4), we get

$$C(k) = \frac{1-p}{p} \cdot \frac{1-(1-p)^k}{(1-p)^k} = \frac{1-p}{p} \cdot \frac{\mathbb{P}(R_k)}{\mathbb{P}(\neg R_k)}.$$

Note that $\mathbb{P}(R_k)/\mathbb{P}(\neg R_k) = 1/\mathbb{P}(\neg R_k) - 1$ is the expected number of failed runs (runs that ended in a restart), and $\frac{1-p}{p} = \frac{1}{p} - 1$ is the expected number of time steps *before* a job gets interrupted. We find

$$C(k) = \mathbb{E}(\text{length of a failed run}) \cdot \mathbb{E}(\#\text{failed runs}).$$

If $C(k) > M$, it is no longer advantageous to restart J. This happens after state

$$k^* = \lfloor \frac{\ln(1+\frac{Mp}{1-p})}{-\ln(1-p)} \rfloor = \lfloor \frac{\ln(1+(M-1)p)}{-\ln(1-p)} + 1 \rfloor. \tag{3.6}$$

Summarizing, we have the following theorem.

Theorem 1 *The optimal policy for the basic case is given by*

$$a(k^*) = (1, \overset{k^*}{\ldots}, 1, 0, \overset{d-k^*}{\ldots}, 0),$$

where k^* is determined by (3.6). The expected completion time is at most $M + (d-k)(1+(M-1)p)$.

Proof. Since k^* is the largest k so that $C(k) \leq M$, $C(k^*) = \Theta(M)$ and $C(k^*) \leq M$ (by at most a factor $(1-p)$). After this state, the job is not restarted, so the expected completion time is about $M + (d-k)(1+(M-1)p)$, since $d-k$ more units of work need te be done on J, which are each expected to take $(1-p) \cdot 1 + pM$ time.

If we compare this to [1], where the job was completed with probability $1 - O(1/m)$ if at least one machine was available for $\alpha d \log m$ time, we see that we now have a bound that does not depend on m. On the other hand, the behaviour of the machines is now more precisely modeled.

3.4 The cost of restarting

In the above calculations, it is assumed that whenever the scheduler wants to restart, an idle machine is immediately available. Of course, this does not always have to be the case, but it is not difficult to see that we can always expect some machine to be available quickly. The time until this happens is called the waiting time.

First we need the stationary distribution of the Markov chain on a single machine. This is fairly straightforward, and it turns out that the stationary probabilities are $Mp/(Mp+1)$ for node M and $1/(Mp+1)$ for the idle node.

The probability that all machines are busy when one is needed is

$$\left(\frac{Mp}{Mp+1}\right)^{m-1}.$$

On each *busy* machine the time until it is again available is distributed homogeneously on the values $1, 2, \ldots, M$. The expectation of the minimum of m homogeneously distributed variables is $M/(m+1)$. Therefore, the expected waiting time is

$$\left(\frac{Mp}{Mp+1}\right)^{m-1} \frac{M}{m+1}.$$

Until now we used (3.1) and compared $f(0)$ to $M + f(i+1)$ to determine the optimal policy in state i. Now we should compare $f(0)$ plus the waiting time to $f(i+1)$ plus M, so we need to check if

$$f(0) - f(i+1) < M' = M - \left(\frac{Mp}{Mp+1}\right)^{m-1} \frac{M}{m+1}$$

in stead of (3.2). The calculations do not change, so the only result is that in (3.6), M must be replaced by M'. But the waiting time is much smaller than M and therefore negligible. The situation in state 0 does not change either: although restarts are now no longer free, they still cost far less than M. So we still have that in the starting state, restarting is always optimal. Similar results hold if there are interrupting jobs of different sizes, but not for general Markov chains, because of the different structure of the algorithm in that case.

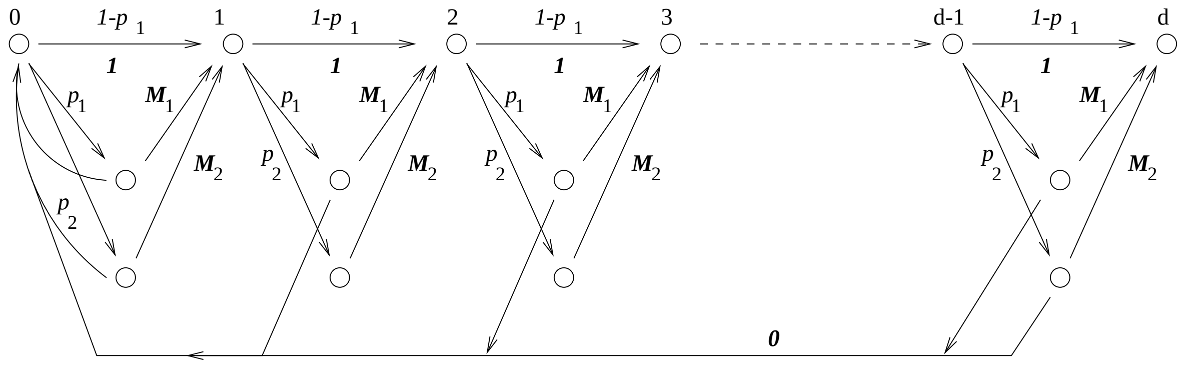

Figure 3: Markov chain of our job

4. Two Jobs

4.1 The optimal policy

Suppose there are two jobs that can interrupt J, of sizes M_1 and M_2, where $M_1 < M_2$. The probabilities of these interruptions are p_1 and p_2, respectively. We assume that these interruptions do not occur simultaneously. Then for the completion costs f we have

$$\begin{aligned} f(i) &= (1-p_1-p_2)(1+f(i+1)) \\ &\quad +p_1 \cdot \min\{f(0), M_1+f(i+1)\} \\ &\quad +p_2 \cdot \min\{f(0), M_2+f(i+1)\}. \end{aligned}$$

A policy for this problem has the form

$$a = \left(\begin{array}{cccc} a_0^1 & a_1^1 & \dots & a_{d-1}^1 \\ a_0^2 & a_1^2 & \dots & a_{d-1}^2 \end{array} \right),$$

where for all i and j, $a_i^j \in \{0,1\}$, and again

$$a_i^j = 1 \text{ is optimal } \Leftrightarrow f(0) - f(i+1) \leq M_j. \tag{4.1}$$

Because $f(i)$ is strictly decreasing, there is a largest l_1 such that $f(0)-f(l_1+1) < M_1$. For states $i > l_1$, a restart is no longer optimal when M_1 interrupts J. Since $M_1 < M_2$, there can be states $i > l_1$ where it is still optimal to restart in case of M_2. This holds until state, say, $l_1 + l_2$. After that, J must never be restarted.

This implies that we can divide the states of J in three phases (state intervals). In the first phase, when J is interrupted, it gets restarted. In the second phase, it is only restarted when M_2 interrupts it, and in the third phase it is not restarted at all.

4.2 Calculations

Because in the first phase J is always restarted when interrupted, we can determine the optimal length of this phase using the method of the previous section: it ends at the point where a restart becomes too expensive, that is, more expensive than M_1. This follows directly from (4.1), and these costs do not depend on the second or third phases, so l_1 can be determined independently. When we know the optimal l_1^*, we can derive l_2.

The event that a restart occurs in phase i is denoted by R_i. We denote the total expected costs from state 0 until the end of phase i by $C_i(l_i)$ is reached for the first time. In general, we can only calculate costs of reaching a certain state for the first time, since it is always possible that a restart occurs after that state. $C_i(l_i)$ depends on the l_j where $j \leq i$, but when we are going to calculate it, all l_j with $j < i$ will be known, so l_i is the only unknown.

We will also need to look at the expected cost from the beginning of a phase until a restart within that phase, which we will denote by $C_{R_i}(l_i)$. This cost depends only on the length l_i of phase i. Finally, we denote the expected time to go from one state to the next in phase i, given that there is no restart, by S_i. We have $1 = S_1 < S_2 < S_3$.

First we need to calculate for which states $f(0) - f(i+1) < M_1$, or for which l_1 the expected cost of reaching state l_1 for the first time (while restarting whenever J is interrupted) become larger than M_1. As noted above, this does not depend on l_2.

The probability that J gets interrupted in the first phase is $p_1 + p_2 =: q_1$. Analogously to section 3, we find

$$l_1^* = \lfloor \frac{\ln(1 + (M_1 - 1)q_1)}{-\ln(1 - q_1)} + 1 \rfloor. \tag{4.2}$$

The cost until l_1^* is reached is $N_1 := C_1(l_1^*) = \Theta(M_1)$. We need N_1 to calculate l_2, since $C_2(l_2)$ is equal to N_1 plus the expected cost in case of a restart, plus the expected cost if there is no restart:

$$C_2(l_2) = N_1 + \{C_{R_2}(l_2) + C_2(l_2)\}\mathbb{P}(R_2) + l_2 S_2 \mathbb{P}(\neg R_2).$$

This equation is similar to (3.3), except here there is a contribution of N_1 for the first phase, and the cost of taking one step is now greater than 1.

It follows that

$$\mathbb{P}(\neg R_2) C_2(l_2) = N_1 + C_{R_2}(l_2)\mathbb{P}(R_2) + \mathbb{P}(\neg R_2) l_2 S_2.$$

As was shown previously,

$$\begin{aligned} C_{R_2}(l_2)\mathbb{P}(R_2) &= \sum_{i=0}^{l_2 - 1} i \cdot S_2 q_2 (1 - q_2)^i \\ &= (\frac{1 - q_2}{q_2}\mathbb{P}(R_2) - l_2 \mathbb{P}(\neg R_2)) S_2. \end{aligned}$$

Therefore

$$C_2(l_2) = \frac{N_1}{\mathbb{P}(\neg R_2)} + \frac{1 - q_2}{q_2} \cdot \frac{\mathbb{P}(R_2)}{\mathbb{P}(\neg R_2)} \cdot S_2. \tag{4.3}$$

Looking at this equation, we see that the expected cost of reaching $l_1^* + l_2$ for the first time is equal to (cost in phase 1)·(#times phase 1 is traversed) + $\mathbb{E}$(length of a run in phase 2)·$\mathbb{E}$(#failed runs in phase 2)·(step size in phase 2).

This cost must be at most M_2. It follows that

$$l_2^* = \left\lfloor \ln\left(\frac{N_1 + \frac{1-q_2}{q_2}S_2}{M_2 + \frac{1-q_2}{q_2}S_2}\right) \Big/ \ln(1-q_2) \right\rfloor . \tag{4.4}$$

We can now combine everything into the following theorem.

Theorem 2 *The optimal policy for this type of Markov chain is of the form*

$$\begin{pmatrix} 1 & \overset{l_1^*}{\dots} & 1 & 0 & \overset{l_2^*}{\dots} & 0 & 0 & \overset{d-l_1^*-l_2^*}{\dots} & 0 \\ 1 & \dots & 1 & 1 & \dots & 1 & 0 & \dots & 0 \end{pmatrix},$$

where l_1^* and l_2^* are determined by (4.2) and (4.4).

5. r Interrupting Jobs

The calculations are completely analogous to those used in the previous section. First we find

$$l_1^* = \lfloor \frac{\ln(1 + (M_1 - 1)q_1)}{-\ln(1-q_1)} + 1 \rfloor . \tag{5.1}$$

Define $C_j(l_j)$ as the expected cost to reach the $j+1$st phase for the first time, starting in state 0, where l_j is the length of phase j. Define furthermore $N_j = C_j(l_j^*)$. Thus we find

$$C_j(l_j) = N_{j-1} + (C_{R_j}(l_j) + C_j(l_j))\mathbb{P}(R_j) + l_j S_j \mathbb{P}(\neg R_j).$$

For every phase, this gives us a formula of the form (4.3). Thus for $j = 1, \dots, r$ we find

$$l_j^* = \left\lfloor \ln\left(\frac{N_{j-1} + \frac{1-q_j}{q_j}S_j}{M_j + \frac{1-q_j}{q_j}S_j}\right) \Big/ \ln(1-q_j) \right\rfloor \tag{5.2}$$

This leads to the following theorem.

Theorem 3 *The optimal policy for each M_j is given by*

$$a^j = (1, \overset{l_1^*+\dots+l_j^*}{\dots}, 1, 0, \dots, 0),$$

and the l_j's are given by (5.1) and (5.2).

6. General Markov Chains

Finally, we consider the situation where n different jobs can interrupt the scheduler's job J. The interrupting jobs are connected via a Markov chain M. In this chain, node j represents a job of size M_j ($j = 1, \dots, n$), and node 0 represents the 'idle' state in which J can be run. In other words, for each node $j \neq 0$, the costs associated with a restart are 0 and the cost of continuing is M_j. The probability that the system moves from node i to node j is denoted by s_{ij}.

In the same way as in the previous sections, this induces a Markov decision process on J. Naturally, we assume that M is irreducible (all states communicate). Moreover, for simplicity we assume that $M \backslash \{0\}$ is acyclic.

A policy a for this problem consists of n policies a^j, one for each node j. We write $a^j = (a^j_0, a^j_1, \dots, a^j_d)$, where the subscript denotes the state of J. $a^j_s = 1$ means that J will be restarted if j is visited in state s, and $a^j_s = 0$ means j is *allowed* and J will continue. The optimal policy is denoted by $a_* = (a^1_*, \dots, a^n_*)$. Again, it is deterministic and stationary.

6.1 Definitions

We will now define some notions that we will need later.

- A node j of the Markov chain is called *allowed* in state s if $a^j_s = 0$, and it is called *reachable* in state s if there exists a path in the Markov chain from 0 to j where $a^{j'}_s = 0$ for all nodes j' on this path.
- $\overline{C_j}(t+1)$ is the cost of reaching state $t+1$, starting in state t at node j and assuming that j is allowed at state t. This consists of the cost of j itself, the cost of successfully reaching $t+1$ times the probability of this happening, and the expected cost when a restart occurs times its probability. (We use $\overline{C_{\dots}}$ to denote the *total* cost of an event, including restarts; later we will define costs where a restart is not allowed.)
- $\overline{C_{NO}}(t)$ is the total cost of reaching state t, starting in state 0 in the idle node, assuming that j is not allowed before state t. We will determine the strategy *iteratively*, state by state. That way, when we are considering state $t+1$, we already know the value of $\overline{C_{NO}}(t)$.
- $\overline{C_{YES}}(t+1)$ is the total cost of reaching state $t+1$, starting in state 0 in the idle node, assuming that j is allowed from state t onwards. To reach $t+1$ we first need to get to t, this costs $\overline{C_{NO}}(t)$. After that there are three possibilities:
  - we visit j (costs after this are $\overline{C_j}(t+1)$),
  - we reach $t+1$ without visiting j, or
  - we visit some forbidden node before reaching j or $t+1$, thus forcing a restart which again costs $\overline{C_{YES}}(t+1)$.

6.2 When should a node be allowed?

We begin by looking at individual nodes, and show locally optimal strategies. Later we will describe the global policy.

If we write $f(t,j)$ for the optimal completion costs, starting in state t and node j, we have that

$$f(t,j) = \min\{f(0,0), M_j + \sum_{k \in OUT(j)} f(t,k)\}, \tag{6.1}$$

similar to the earlier cases. We do not have a simple interpretation for $f(0,0) - f(t,k)$, which we did have earlier. But we do know that an optimal policy will minimize the cost to reach t for all t. (If it costs the policy more to reach t_1, it will cost more to reach any point after t_1.)

Therefore, to determine the optimal policy in a certain state t, we do not need to look more than one state ahead, to $t+1$. By minimizing the cost until $t+1$, we thus find the optimal policy. Let $f_{t+1}(t,j)$ denote the optimal cost of reaching $t+1$ for the first time, then $f_{t+1}(t,j) = \min\{f_{t+1}(0,0), M_j + \sum_{k \in OUT(j)} f_{t+1}(t,k)\}$.

Since the decision in state t and node j is the same each time this pair is visited, we can in fact replace this equality by

$$f_{t+1}(t,j) = \min\{\overline{C_{NO}}(t+1), \overline{C_j}(t+1)\} \tag{6.2}$$

if we calculate these costs for the optimal policy. Note that, although the minima in these last two equations are equal, the respective parts are not always equal. In the next section, it will be shown that once again, every node is forbidden until a certain state is reached (which can be different for each node).

6.3 Some important costs

We will now formulate equations for the three important costs from 6.1. All costs naturally depend on the chosen strategy, but we will not denote this explicitly in every equation.

We will adopt a uniform notation for all costs, C, and probabilities, p. Subscripts indicate the starting point from which the probability or cost is calculated, the first argument indicates the point that is to be reached (the goal) and the second argument, if present, indicates points that should be avoided. If the goal is not $t+1$, then it is assumed that $t+1$ is not reached before the goal. If the goal is not 0, we usually assume no restart occurs before the goal is reached. The three exceptions were mentioned in the previous subsection. The starting points of $\overline{C_{NO}}(t)$ and $\overline{C_{YES}}(t)$ are 0.

As an example, $p_t(0, \neg j)$ is the probability of a restart without visiting j or $t+1$, and starting in state t.

We first derive an equation for $C_j(t+1)$. This cost is equal to the cost of node j, which is M_j, plus the expected cost if there is no restart, plus finally the expected cost if there is one. Using our standard notation, we have

$$\begin{aligned}\overline{C_j}(t+1) &= M_j + p_j(t+1)C_j(t+1,\neg 0) \\ &\quad + p_j(0)(C_j(0) + \overline{C_{YES}}(t+1)),\end{aligned} \tag{6.3}$$

where $p_j(t+1) + p_j(0) = 1$. Similarly, we can derive the following connection between $\overline{C_{NO}}(t+1)$ and $\overline{C_{NO}}(t)$:

$$\begin{aligned}\overline{C_{NO}}(t+1) &= \overline{C_{NO}}(t) + p_t(t+1,\neg j)C_t(t+1,\neg j) \\ &\quad + p_t(j)\left\{C_t(j) + \overline{C_{NO}}(t+1)\right\} \\ &\quad + p_t(0,\neg j)\left\{C_t(0,\neg j) + \overline{C_{NO}}(t+1)\right\} \\ &= \{\overline{C_{NO}}(t) + p_t(t+1,\neg j)C_t(t+1,\neg j) + p_t(j)C_t(j) \\ &\quad + p_t(0,\neg j)C_t(0,\neg j)\}/p_t(t+1,\neg j).\end{aligned} \tag{6.4}$$

Note that $p_t(t+1,\neg j) + p_t(j) + p_t(0,\neg j) = 1$.

A similar equation holds for $\overline{C_{YES}}(t+1)$:

$$\begin{aligned}&\overline{C_{YES}}(t+1) \\ &= \overline{C_{NO}}(t) + p_t(t+1,\neg j)C_t(t+1,\neg j) + p_t(j)(C_t(j) + \overline{C_j}(t+1)) \\ &\quad + p_t(0,\neg j)(C_t(0,\neg j) + \overline{C_{YES}}(t+1)) \\ &= \{\overline{C_{NO}}(t) + p_t(t+1,\neg j)C_t(t+1,\neg j) + p_t(j)(C_t(j) + \overline{C_j}(t+1)) \\ &\quad + p_t(0,\neg j)C_t(0,\neg j)\}\,/(p_t(t+1,\neg j) + p_t(j)).\end{aligned}$$

Using (6.4), we can write this as

$$\begin{aligned}\overline{C_{YES}}(t+1) &= \frac{p_t(t+1,\neg j)}{p_t(t+1,\neg j) + p_t(j)}\overline{C_{NO}}(t+1) + \\ &\quad \frac{p_t(j)}{p_t(t+1,\neg j) + p_t(j)}\overline{C_j}(t+1) \\ &= \alpha\overline{C_{NO}}(t+1) + (1-\alpha)\overline{C_j}(t+1) \qquad (\alpha \in [0,1])\end{aligned} \tag{6.5}$$

Note that $p_t(j) = 0$ implies $\overline{C_{YES}}(t+1) = \overline{C_{NO}}(t+1)$.

According to (6.2), the optimal policy in each node is to allow it if this is cheaper than forbidding it; in other words, if $\overline{C_j}(t+1) < \overline{C_{NO}}(t+1)$. Note that if $t = 0$, restarting is always cheaper, even if we do not assume it is free (see 3.4).

We are therefore especially interested in those values of t, where $\overline{C_j}(t+1) = \overline{C_{NO}}(t+1)$ (we will soon see that there can be only one), because they are the border between states where j is allowed and states where it is not.

We can now see that this implies $\overline{C_j}(t+1) = \overline{C_{YES}}(t+1) = \overline{C_{NO}}(t+1)$. Using these equalities in (6.3), we find

$$\overline{C_{NO}}(t+1) = \frac{M_j + p_j(t+1)C_j(t+1,\neg 0) + p_j(0)C_j(0)}{p_j(t+1)}. \tag{6.6}$$

Finally, we have from (6.4) that

$$\begin{aligned}\overline{C_{NO}}(t) &= p_t(t+1,\neg j)\{\overline{C_{NO}}(t+1) - C_t(t+1,\neg j)\} \\ &\quad -p_t(0,\neg j)C_t(0,\neg j) - p_t(j)C_t(j) \qquad (6.7)\end{aligned}$$

Combining this with the previous equation, we can see that $\overline{C_{NO}}(t+1) > \overline{C_j}(t+1)$ is equivalent to

$$\begin{aligned}&\overline{C_{NO}}(t) > -p_t(0,\neg j)C_t(0,\neg j) - p_t(j)C_t(j) \\ &+p_t(t+1,\neg j)\left\{\frac{M_j + p_j(t+1)C_j(t+1,\neg 0) + p_j(0)C_j(0)}{p_j(t+1)} - C_t(t+1,\neg j)\right\} \qquad (6.8)\end{aligned}$$

The right side of this equation will be called the *threshold*. Note that it depends only on which nodes are allowed in state t and which are not. This is because it consists of costs to get from A to B, travelling through the part of the Markov chain which is allowed in state t, and probabilities of taking certain paths in the currently allowed part of the chain. As long as the subset of allowed nodes does not change, the threshold is a constant. Furthermore, $\overline{C_{NO}}(t)$ is strictly increasing in t (and we know what its value is for t, see subsection 6.1), since it is not cheaper to reach t than it is to reach $t-1$.

This implies that for any subset of allowed nodes (which does not change over time), there is one specific state t_j, where for $t < t_j$ we have $\overline{C_j}(t+1) > \overline{C_{NO}}(t+1)$ and for $t > t_j$ we have $\overline{C_j}(t+1) < \overline{C_{NO}}(t+1)$. This state is determined by the threshold: j is forbidden until its threshold is reached. We have not yet excluded the case where in some later state, when other nodes have been allowed and thus the subset of allowed nodes has changed, j must be forbidden again. We will do this in subsection 6.6.

6.4 Calculating thresholds

We have seen that thresholds play an important part in a policy for this problem. We will now show how to calculate thresholds. According to (6.8), a threshold depends on no less than five different costs and five probabilities. But we can see immediately that $p_j(0) = 1 - p_j(t+1)$ and $p_t(0,\neg j) = 1 - p_t(t+1,\neg j) - p_t(j)$.

Consider a node j which has a set of outgoing edges $OUT(j)$. For each state t we divide $OUT(j)$ in two sets, $OK_t(j)$ and $BAD_t(j)$, where the "bad" edges lead to nodes where J is restarted in state t. The associated end nodes are also called bad.

We call 0 a good node; for the following calculations (equations (6.9)–(6.11)) we put $p_0(t+1) = 1, C_0(t+1) = 0$ and $M_0 = 0$ whenever they appear in a right member. The success probability $p_j(t+1)$ is the total probability of going to a good node, weighted by the success probabilities of those nodes:

$$p_j(t+1) = \sum_{(j,k)\in OK_t(j)} s_{jk} \cdot p_k(t+1). \qquad (6.9)$$

To calculate $C_j(t+1,\neg 0)$, we must assume that J is not restarted, and therefore that one of the "good" outgoing edges is chosen when leaving j. If we normalize the transition probabilities on these "good" edges by putting $s'_{jk} = s_{jk}/\sum_{(j,j')\in OK_t(j)} s_{jj'}$, we get

$$C_j(t+1,\neg 0) = \sum_{(j,k)\in OK_t(j)} s'_{jk}(M_k + C_k(t+1,\neg 0)). \tag{6.10}$$

For $C_j(0)$, a similar equation holds. If $p_j(t+1) = 1$, we can put $C_j(0) = 0$. Otherwise, since we assume that a restart actually occurs, we must disallow the edge $(j,0)$ if it exists and in general any edge (j,k) for which $p_k(t+1) = 1$. We then normalize the transition probabilities on the set of remaining edges, called $REM_t(j)$, by putting $s'_{jk} = s_{jk}/\sum_{(j,k)\in REM_t(j)} s_{jk}$. Note that $REM_t(j)$ may contain bad nodes. If the system moves to a bad node, no more costs are incurred, because the job is then immediately restarted. In a good node k however, an extra cost of expected size $M_k + C_k(0)$ is incurred. Therefore

$$C_j(0) = \sum_{(j,k)\in OK_t(j)\cap REM_t(j)} s'_{jk}(M_k + C_k(0)). \tag{6.11}$$

Similar relations hold for $p_t(t+1,\neg j)$, $C_t(t+1,\neg j)$ and $C_t(0,\neg j)$, except that here it needs to be taken into account that j must not be visited. Finally, $C_t(j)$ and $p_t(j)$ can also be calculated using standard techniques. These last five values only change when a node gets allowed that is reachable from 0.

6.5 The algorithm

We are now finally ready to describe a method to determine the global strategy (for the whole Markov chain). We will eventually tag each node with the state in which it is allowed first.

What we need to know, when starting in the idle node, is first of all which of the possible interrupting jobs that can be reached in a single step need to be allowed first. But to find that out, we need to know what the optimal costs are after those jobs, i. e. in the rest of the Markov chain. We will therefore begin at the "end" of the Markov chain (which is well defined, since the chain is acyclic), and work our way back to nodes that can be reached from 0, each time calculating which node should be allowed next. This way, some nodes will be allowed before they can even be reached from 0, but we will later show that this has no adverse effect on the costs of the algorithm.

The method is divided into steps. In each step x we calculate the next time step t_x on which a node j_x gets allowed. We do this until all nodes are allowed or the job finishes. Within each interval $[t_{x-1}, t_x - 1]$ the thresholds are constant. We put $j_0 = t_0 = 0$.

Each step x consists of a number of calculations. There are always a number of relevant nodes, for which some calculations are necessary, but we need to do this in the correct order since some calculations depend on other results. We will define two sets of nodes: A_x and B_x. Each step consists of the following substeps.

- Define A_x as the set of *allowed* nodes from which j_{x-1} can be reached. For every node j in this set, recalculate $C_j(t+1, \neg 0)$, $C_j(0)$ and $p_j(t+1)$. Do this in reverse order (walking backwards through the graph): first for nodes that have no successor in A_x, and then for each node as soon as all the data from the successors are known.
- Define B_x as the set of *forbidden* nodes from which 0 or an already allowed node can be reached in a single step. Calculate the thresholds of this set in any order, using the new data from the set A_x when necessary.
- Allow the node $j_x \in B_x$ which first reaches its threshold. Calculate t_x.

A few notes on this algorithm:

- For forbidden nodes $j \notin B_x$ we always have that the threshold is infinite.
- For nodes $j \in B_x \cap B_{x-1}$ from which j_{x-1} can not be reached, the threshold now is the same as in step $x-1$.
- It is possible for two or more nodes to have the minimal $CV^j_{t_{x-1}}$. In this case, allow the latest one in the natural ordering.

6.6 Optimality

Thresholds are crossed once We show that if a node is allowed, it will remain allowed until the job is completed, unless the job is restarted. To simplify the wording, we will now color allowed nodes green and forbidden nodes red.

We will prove the following stronger statement.

Lemma 1 *From one state to the next, the cost of a node can never increase more than the cost of a restart.*

This implies that if a node is green in state t, it will not be red in state $t+1$, for any t.

Proof. Take a state t. We use an induction. Consider a green node j and suppose that the cost of all its successors has not increased more than that of a restart. In that case, no successor turned red.

We need to show that the cost of j can not increase faster than the cost of a restart. Consider the following equation:

$$\overline{C_j}(t) = M_j + \sum_{i \in OUT(j)} s_{ji} D_i(t),$$

where

$$D_i(t) = \begin{cases} C_i(t) & i \in OK_t(j) \\ \overline{C_{YES}}(t) & i \in BAD_t(j) \end{cases}.$$

Write the increase in cost starting from j as $\delta = \overline{C_j}(t+1) - \overline{C_j}(t)$, the increase in the costs of a successor i as $\delta_i = D_i(t+1) - D_i(t)$ for all $i \in OUT(j)$ and finally the increase in the cost of a restart (while j is green, allowed) as $\delta_{YES} = \overline{C_{YES}}(t+1) - \overline{C_{YES}}(t)$. Now suppose the cost of j has increased faster than that of restarting, so that $\delta > \delta_{YES}$, then $\sum_{i \in OUT(j)} s_{ji}\delta_i > \delta_{YES}$ and therefore, for some i, δ_i must be greater than δ_{YES}. We show a contradiction.

According to the induction hypothesis, no successor of j turned red in this state. Therefore we have three cases:

- i remained green, then $\delta_i = \overline{C_i}(t+1) - \overline{C_i}(t) < \delta_{YES}$ because of the induction hypothesis
- i remained red. Then $\delta_i = \overline{C_{YES}}(t+1) - \overline{C_{YES}}(t) = \delta_{YES}$.
- i turned green: $\delta_i = \overline{C_i}(t+1) - \overline{C_{YES}}(t) < \overline{C_{YES}}(t+1) - \overline{C_{YES}}(t) = \delta_{YES}$, otherwise i could not be green now.

In other words, $\delta_i \leq \delta_{YES}$ for all i, and therefore $\delta \leq \delta_{YES}$, a contradiction. The lemma is proved.

The strategy is optimal The strategy we have described is deterministic and stationary. However, it is not immediately clear that it is indeed the best such strategy, because this algorithm may allow a node which cannot yet be reached. This may seem unnecessary and even could cause (reachable) predecessors to be allowed later than when such a node would still be forbidden. However, we can show that this is never the case.

Suppose that for a strategy S and a certain state t, we color all the nodes red, green and blue. Red nodes are currently forbidden, green nodes are allowed (by our algorithm) and blue nodes are undetermined. We could color unreachable nodes blue, since the decision in such a node is not important yet: we can choose freely.

We will show that by changing the color of a blue node to green or red, the cost will not increase.

For end nodes, that have 0 as only successor, the costs are always fixed: the cost is the size of the job in that node. Since the costs of reaching state t (from 0) increase monotonically with t, after some point t_0 a restart in these nodes is more expensive than continuing, while before t_0, continuing is more expensive than restarting. Therefore, if S has colored such a node blue, then coloring it green or red does not increase the cost. Also, it is immediately clear that once such a node turns green, it remains green in state $t+1$ and beyond.

We will now walk backwards through the Markov chain, using induction. (We can do this because the graph is acyclic.) The induction hypothesis is that all successors of the current node are either green or red. This implies that the cost following this node is fixed, at least, the expectation of the cost is fixed (restarts may occur). Suppose our algorithm, which bases its decision solely on that cost, colors the node green.

That means that from there it is cheaper to continue than it is to restart. If another algorithm wants the color to be blue, then the costs can never be lower than if we always continue (green); therefore, we might as well color it green. The same argument holds if the node is red.

This shows that no algorithm can have lower costs than our algorithm, in other words, that it is optimal.

6.7 Efficiency

We consider the time complexity of this algorithm. Consider one step in the algorithm.

Calculating $C_j(t+1, \neg 0)$ for a node j in A_x takes $O(OK_t(j))$ steps according to (6.10). This also holds for $p_j(t+1)$ and $C_j(0)$. Therefore the calculations for A_x take O(number of outgoing edges from A_x), which is certainly $O(n^2)$.

For B_x, some costs and probabilities starting in t need to be recalculated if j_{x-1} is reachable. This requires walking backwards through the graph starting in j_{x-1}, and doing O(number of outgoing edges) calculations in each node. Since each edge is used only once, and the Markov chain is acyclic, the total costs of this are $O(n^2)$ as well.

The entire process therefore is $O(n^3)$. A first-passage problem like this can also be solved using a linear program; however, in this case for each node in the Markov decision process a variable needs to be introduced. Solving a linear programming problem with nd variables can be done in $O(d^3n^3)$ time. This is clearly far worse, especially if the Markov chain of the machines is fairly small relative to the size of the job, so that $d >> n$. Moreover, this linear program needs to be solved for every occurring job size d. Since the time complexity is the third power of the job size itself, this is impractical.

6.8 On the use of this strategy

All the calculations for this strategy can be done before the job starts, and in fact this is necessary. When for every node it is known when it should be allowed, this can be represented internally by tagging each node with the state in which it is first allowed. Every time the machine switches states, the scheduler can check the tag and compare it to the current state to see whether the job needs to be restarted. Since the strategy does not depend on the length of the job, this tagging only needs to happen once and then many jobs could be run. Also, for jobs with checkpoints [3] these tags could be used. A checkpoint is a point on which all data of a job is saved, so that if it is restarted in the future, the work before this checkpoint does not have to be done again. In this case the tags must be compared to the current state minus the last checkpoint.

7. Conclusion

We have shown optimal policies for scheduling on Markovian machines, for several types of Markov chains, and an efficient way of calculating them. These policies can be readily extended to run many jobs simultaneously on one network, or to run jobs with checkpoints. Also, note that our solutions do not depend on d, neither in their computational complexity, nor as input parameter. This means that it only needs to be computed once, for all possible occurring jobs. Then the nodes can be tagged with the state in which they are allowed first, and any job can be run on the network.

An open question is whether it is possible to do this for a Markov chain that has cycles, since in our method and proofs we use heavily that it is possible to walk backwards through the graph. Perhaps in this case one would have to resort to using a linear programming formulation, using dn variables. This can be solved in $O(d^3n^3)$ time, which is much more than d itself. As noted, this is substantially worse than in the acyclic case, and requires new computations for every possible job size d. Hence, this would be an impractical approach.

Furthermore, one could look at Markov chains that have more than one idle state, each with different interrupting jobs connected to it. This could be used to model different parts of the day, if it is known that some jobs are e. g. especially often run in the afternoon. This can be modeled by having very small transition probabilities between two idle states, so that the Markov chain is divided into nearly unconnected subchains. In this case it is much harder to formulate a policy, since now it is no longer certain that when restarting in some state you will later return to the same state and subchain. Therefore it is no longer sufficient to consider the costs until a state t. However, it seems that in practice we can restart our algorithm whenever the process switches idle states to get a reasonable policy. For instance, if we want to run jobs that take half an hour and the idle state is switched twice a day, we can determine policies for both idle nodes and switch between them if necessary.

8. Acknowledgments

We would like to thank Sindo Nuñez Queija, Bert Zwart and Ger Koole for their support on the basic case.

References

1. B. Awerbuch, Y. Azar, A. Fiat and T. Leighton. Making Commitments in the Face of Uncertainty: How to Pick a Winner Almost Every Time. *28th Annual ACM Symposium on Theory of Computing*, 519–530, 1996.
2. A. Borodin, R. El-Yaniv. *Online Computation and Competitive Analysis*, Cambridge University Press, 1998.
3. A. G. Coffman, J. Leopold Flatto and P. E. Wright. A Stochastic Checkpoint Optimization Problem. *SIAM J. Comput. 22*, 650–659, 1993.
4. C. Derman, *Finite State Markovian decision processes*, Academic Press, New York, 1970.
5. A. Karlin, S. Phillips and P. Raghavan. Markov Paging. *33rd Annual IEEE Symposium on Foundations of Computer Science*, 208–217, 1992.