Acta Informatica 37, 727–742 (2001)

Acta Informatica

© Springer-Verlag 2001

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

Rob van Stee*, **Han La Poutré**

Centre for Mathematics and Computer Science (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands (e-mail: {rvs,hlp}@cwi.nl)

Received: 31 March 1999 / 5 April 2001

Abstract. We consider the problem of running a background job on a selected machine of a collection of machines. Each of these machines may become temporarily unavailable (busy) without warning. If the currently selected machine becomes unavailable, the scheduler is allowed to restart the job on a different machine. The behaviour of machines is characterized by a Markov chain, which can be compared to [6]. The objective is to minimize the expected completion time of the job. For several types of Markov chains, we present elegant and optimal policies.

1 Introduction

In networks of workstations, a considerable amount of capacity is unused, since the primary users are only using the workstations 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 the 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,

* Research supported by The Netherlands Organization for Scientific Research (NWO), project number SION 612-30-002.

or to restart 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, where the completion time should be minimized and the job can be executed on one machine at a time.

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 continuously, 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 move the job and restart it from scratch on a different machine. The goal is to minimize the expected completion time. This was mentioned as an open problem in [8].

Scheduling n jobs on m machines under availability constraints has been studied in a number of papers (see the surveys [7, 8]). Most attention has gone to deterministic availability constraints, both for non-preemptive and preemptive problems. Also the non-resumable case has been studied, where a job must restart if its machine goes down: it cannot wait for the machine to become available again. In [3], stochastic scheduling is studied: here the task *durations* are stochastic and the availability periods are deterministic, which is the opposite of 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 (significantly 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 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 availability of the workstations is captured by a Markov chain. This has similarities with the modeling in [6], where the paging problem was addressed in a similar way, i. e., by modeling the behaviour of a program by a Markov chain. Every behaviour of workstations can indeed be modeled by a Markov 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 "unavailable", with the expected unavailability time as its cost. Making more elaborate Markov chains based on (on-line) system statistics and additional information, enables finer grained

descriptions and improved scheduling strategies, yielding lower completion times.

We will consider Markov chains that, if the idle state is deleted, become acyclic. In practical situations, a Markov chain is obtained and approximated from statistical information, and approximating "infinite cycling behaviour" by just one or a couple of states will fall within the statistical and practical accuracies. We also refer to [6] for some general comments on Markov chains.

In the paper, we present elegant and optimal scheduling strategies. The actual job size does not need to be known (but it does not help to know it either). The computational complexity of our strategies is $O(n \cdot e)$, where n is the number of nodes in the Markov chain and e is the number of edges. 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 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 not more than 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: we show in [9] 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. (And if there is extreme occupation, no scheduler can hope to perform well.)

The Markov chain of a machine, together with the possibility of a restart, induces a Markov decision process on the job J, as follows. We define J to be in *time state* t if it has been worked on for t time steps since its latest restart, not counting the time that the current machine was busy. For every time state $0, \dots, d-1$ of J and every possible interrupting job, we need to decide what to do in case of an interruption. Do we restart J or wait? J is completed when it reaches time state d.

3 The basic case

In the basic case, all machines behave according to the Markov chain shown on the left in Fig. 1. A more compact way of picturing this is shown on the

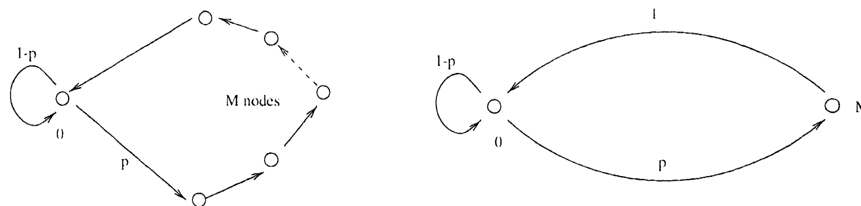

Fig. 1. Markov chain of one machine in two forms

right in the same figure, where the M-node costs M units of time. Note that this is just a simplifying picture and cannot be used as the basis of calculations. The induced Markov chain on J is shown in Fig. 2. Costs are in bold type.

In Markov decision theory, this is known as a first-passage problem [5]. 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 $O(d^3)$ 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.

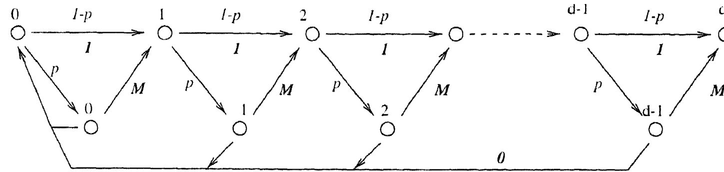

Fig. 2. Markov chain of our job

Clearly, in the top row of this Markov chain (representing the idle node), it is always optimal to continue. Only when the process moves to one of the nodes in the lower row, meaning that the current machine is busy because of a higher priority job, we need to make a choice.

3.1 The optimal policy

It is known [5] 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 time state. Also, it is deterministic.

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

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

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

It follows from (1) that a restart in time state t is optimal if and only if

$$f(0) \leq M + f(t+1), \tag{2}$$

and in that case restarting is optimal in all the previous time states as well, since $f(t)$ is monotonically decreasing. Thus $a_t = a_{t-1} = \ldots = a_0 = 1$.

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 $a(k)$:

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

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

We define $C(k)$ as the expected cost to reach k for the first time, using strategy $a(k)$.

Theorem 1 *The optimal policy for the basic case is given by* $a(k^*) = (1, \overset{k^*}{\ldots}, 1, 0, \overset{d-1-k^*}{\ldots}, 0)$, *where* k^* *is determined by*

$$k^* = \left\lceil \frac{\log(1 + (M-1)p)}{-\log(1-p)} \right\rceil . \tag{3}$$

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

Proof. It follows from (2) that restarting is optimal as long as $f(0) - f(t+1) < M$. Since $f(0) - f(t+1)$ is the expected total optimal cost minus the expected optimal cost starting in $t+1$, it is equal to $C(k^*)$, where k^* is the optimal choice of k. Thus, we need to calculate $C(k)$ for general k and we need to find the smallest k for which $C(k) > M$.

We first calculate $C(k)$. Define R_k as the event that a restart occurs before reaching time 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 to reach k 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

$$C(k) = (C_R(k) + C(k))(1 - (1-p)^k) + k(1-p)^k \tag{4}$$

or

$$C(k) = C_R(k)\frac{1-(1-p)^k}{(1-p)^k} + k. \tag{5}$$

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

$$C_R(k) = \sum_{t<k} t \cdot \mathbb{P}(\text{restart after } t \text{ steps})/(1 - (1-p)^k).$$

Using $\mathbb{P}(\text{restart after } t \text{ steps}) = (1-p)^t$, we can derive

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

If $C(k) > M$, it is no longer advantageous to restart J in time state k. Since $C(k^*) \leq M$ implies $(1-p)^{-k^*} - 1 \leq M\frac{p}{1-p}$, we have

$$k^* = \left\lfloor \frac{\log(1 + \frac{Mp}{1-p})}{-\log(1-p)} \right\rfloor = \left\lceil \frac{\log(1 + (M-1)p)}{-\log(1-p)} \right\rceil .$$

After time state k^*, the job is not restarted anymore. The expected completion time from then on is at most $(d-k^*)(1+(M-1)p)$, since $d-k^*$ more units of work need to be done on J, which are each expected to take $(1-p)\cdot 1+pM$ time. Therefore, the total costs are at most $M+(d-k^*)(1+(M-1)p)$.

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. Note that on the other hand, the behaviour of the machines is now more precisely modeled.

4 Two or more jobs

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

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

A policy for this problem has the form

$$a = \begin{pmatrix} a_0^1 \, a_1^1 \ldots a_{d-1}^1 \\ a_0^2 \, a_1^2 \ldots a_{d-1}^2 \end{pmatrix},$$

where for all t and i, $a_t^i \in \{0,1\}$. $a_t^i = 1$ means that J will be restarted if interruption M_i occurs in time state t, and $a_t^i = 0$ means J will continue. In this section, we give only the results; the calculations are related to Sect. 3 but more elaborate, and can be found in [9].

We find that we can divide the time states of J in three consecutive *phases* (time state intervals). In phase 1, J gets restarted when M_1 or M_2 interrupts it. In phase 2, it is restarted just when M_2 (the largest of M_1 and M_2) interrupts it, and in phase 3 it is not restarted at all.

The probability that J gets interrupted in phase 1 is $p_1+p_2 =: q_1$. Analogously to section 3, we find for the optimal length l_1^* of phase 1

$$l_1^* = \left\lceil \frac{\log(1+(M_1-1)q_1)}{-\log(1-q_1)} \right\rceil, \tag{6}$$

Define C_i as the expected cost to reach phase $i+1$ for the first time, starting in time state 0. E. g. C_1 is the cost to reach time state l_1^* from time state 0.

We have $C_1 = \frac{1-p}{p} \cdot ((1-p)^{-l_1^*} - 1)$. Using this value, we can calculate the optimal length l_2^* of phase 2. We find

$$l_2^* = \left\lfloor \log \left(\frac{C_1 + \frac{1-q_2}{q_2} T_2}{M_2 + \frac{1-q_2}{q_2} T_2} \right) \Big/ \log(1-q_2) \right\rfloor , \tag{7}$$

where $q_2 = p_2$ and T_2 is the expected time to go from one time state to the next in phase 2 when there is no restart. We now have the following theorem.

Theorem 2 *The optimal policy for two interrupting jobs is*

$$\begin{pmatrix} 1 \overset{l_1^*}{\ldots} 1 \; 0 \overset{l_2^*}{\ldots} 0 \; 0 \overset{d-1-l_1^*-l_2^*}{\ldots} 0 \\ 1 \ldots 1 \; 1 \ldots 1 \; 0 \quad \ldots \quad 0 \end{pmatrix},$$

where l_1^* and l_2^* are determined by (6) and (7).

If there are $r > 2$ interrupting jobs $M_1, \ldots, M_r$, then l_1^* does not change. For $i = 2, \ldots, r$, we find

$$l_i^* = \left\lfloor \log \left(\frac{C_{i-1} + \frac{1-q_i}{q_i} T_i}{M_i + \frac{1-q_i}{q_i} T_i} \right) \Big/ \log(1-q_i) \right\rfloor , \tag{8}$$

where $q_i = p_i + \ldots + p_r$, C_{i-1} is the expected cost to reach the end of phase $i-1$ starting from 0, and T_i is the expected time to go from one time state to the next in phase i. See [9] for details on how to calculate C_{i-1}; T_i can be calculated directly from the definition of phase i. We have the following theorem.

Theorem 3 *For r interrupting jobs $M_1, \ldots, M_r$, the optimal policy for each M_i is given by*

$$(1, \overset{l_1^*+\ldots+l_i^*}{\ldots}, 1, 0, \ldots, 0),$$

where the l_i's are given by (6) and (8).

5 General Markov chains

Finally, we consider the situation where n different jobs, connected by a Markov chain M, can interrupt the scheduler's job J. We assume that M, including the 'idle' node, denoted by 0, is irreducible (all states communicate), and that $M \backslash \{0\}$ is acyclic.

Node i of M represents a job of size M_i ($i = 1, \ldots, n$). For each node i, the costs associated with a restart are 0 and the cost of continuing (waiting) is M_i. Define $OUT(i)$ as the set of nodes j where edge (i, j) exists. The probability that a machine moves from node i to node j is denoted by p_{ij}. Recall that this is independent of the scheduler's decisions.

5.2 Nodes should be unblocked exactly once

Lemma 1 *From one time state to the next, when using the optimal policy, the cost of a node in the Markov chain can never increase more than the cost of a restart in that same node.*

Proof. Take a time state t. We use an induction. Note that as t grows, the cost of restarting grows, because more work is lost by restarting. To simplify the wording, we will now color non-blocking nodes green and blocking nodes red.

Consider a green node i and suppose that the cost of all its successors has not increased more than that of a restart since time state $t-1$. In that case, no successor turned red.

We divide $OUT(i)$ in two sets, $OK_t(i)$ and $BAD_t(i)$, where the "bad" nodes are nodes where J is restarted in time state t. We need to show that the cost of i can not increase faster than the cost of a restart, which is $R^{0,0}_{t+1,0}$. Write

$$R^{t,i}_{t+1,0} = M_i + \sum_{k \in OUT(i)} p_{ik} x_k(t+1),$$

where

$$x_k(t+1) = \begin{cases} R^{t,k}_{t+1,0} & k \in OK_t(i) \\ R^{0,0}_{t+1,0} & k \in BAD_t(i) \end{cases}.$$

Write the increase in cost starting from i as $\delta = R^{t,i}_{t+1,0} - R^{t-1,i}_{t,0}$, the increase in the costs of a successor k as $\delta_k = x_k(t+1) - x_k(t)$ for all $k \in OUT(i)$ and finally the increase in the cost of a restart (while i is green, non-blocking) as $\delta_{YES} = R^{0,0}_{t+1,0} - R^{0,0}_{t,0}$.

We need to show $\delta \le \delta_{YES}$. For $k \in OUT(i)$ we have three cases:

- k remained green, then $\delta_k = R^{t,k}_{t+1,0} - R^{t-1,k}_{t,0} < \delta_{YES}$ because of the induction hypothesis.
- k remained red. Then $\delta_k = R^{0,0}_{t+1,0} - R^{0,0}_{t,0} = \delta_{YES}$.
- k turned green: $\delta_k = R^{t,k}_{t+1,0} - R^{0,0}_{t,0} < R^{0,0}_{t+1,0} - R^{0,0}_{t,0} = \delta_{YES}$, otherwise k could not be green now.

In other words, $\delta_k \le \delta_{YES}$ for all k, so $\delta = \sum_{k \in OUT(i)} p_{ik}\delta_k \le \delta_{YES}$.

Corollary 1 *If a node of M is non-blocking, it must remain non-blocking until the job is completed, unless the job is restarted.*

Proof. The previous lemma implies that if a node of M is non-blocking in time state t, it will not be blocking in $t+1$, for any t.

5.1 Definitions and notations

In the following, we always use the word *node* to refer to the state of the current machine, and keep using *time state* to refer to the state of J. When describing policies, we continue to use subscripts to indicate time states, and we introduce superscripts do indicate nodes (states of the machine). A policy a for this problem consists of n policies a^i, one for each node i, and we write $a^i = (a_0^i, a_1^i, \dots, a_{d-1}^i)$, where the subscript denotes the time state of J. $a_t^i = 1$ means that J will be restarted if i is visited in time state t, and $a_t^i = 0$ means J will continue. The optimal policy is deterministic and stationary.

A node i of the Markov chain is said to be *blocking* in time state t if $a_t^i = 1$, and it is *reachable* in time state t if there exists a path in the Markov chain from 0 to i without blocking nodes. We say that an algorithm *unblocks* node i in time state t when $a_{t-1}^i = 1$ and $a_t^i = 0$.

We introduce two important costs:

- $B_t(i)$ is the total cost of reaching time state t, starting in time state 0 in the idle node, if i is blocking before time state t.
- $U_t(i)$ is the total cost of reaching time state $t+1$, starting in time state 0 in the idle node, if i is unblocked in time state t.

Say a and b are time states and x and y are nodes in the chain. Assume $b \neq 0$. We will write

- $D_{b,y}^{a,x}$ is the cost of going directly (without restart) from $\{a, x\}$ to $\{b, y\}$ (which we call the goal),
- $p_{b,y}^{a,x}$ is the probability of this happening.
- $R_{b,y}^{a,x}$ is the *total* cost of this move, including possible restarts and assuming no restarts take place in $\{a, x\}$.

Note that the optimal decision in $\{a, x\}$ does not depend on the number of times this node and state have been visited.

We write $D_{b,y}^{a,x}(\neg i)$ and $p_{b,y}^{a,x}(\neg i)$ to indicate costs and probabilities when it is assumed that node i is not visited in the meanwhile. If $a = t$, but $b \leq t+1$, then it is assumed that $t+1$ is not reached before the goal. We use the notations $D_{0,0}^{a,x}$ and $p_{0,0}^{a,x}$ to indicate the cost and probability of a restart when starting in $\{a, x\}$. (As an example, $p_{0,0}^{t,0}(\neg i)$ is the probability of a restart, starting in time state t and node 0, without visiting node i or time state $t+1$.)

Finally, the *cost of node* x in time state t is given by $R_{t+1,0}^{t,x}$.

5.3 Thresholds

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

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

$$f(t,i) = \min\{f(0,0), M_i + \sum_{k \in OUT(i)} p_{ik} f(t,k)\}, \quad (9)$$

similar to the earlier cases. We do not have a simple interpretation for $f(0,0) - f(t,k)$ anymore. Instead, we use the following lemma.

Lemma 2 *The optimal policy in time state t and node i can be determined by minimizing the cost from there until time state $t+1$.*

Proof. 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 beyond t_1.

Let $f_{t+1}(t,i)$ denote the optimal cost of reaching $t+1$ for the first time, starting in (t,i). Then $f_{t+1}(t,i) = \min\{M_i + \sum_{k \in OUT(i)} p_{ik} f_{t+1}(t,k), f_{t+1}(0,0)\}$. Since the decision in time state t and node i is the same each time this pair is visited, we can in fact replace this equality by

$$f_{t+1}(t,i) = \min\{B_{t+1}(i), R^{t,i}_{t+1,0}\} \quad (10)$$

if we calculate these costs for the optimal policy (see Fig. 3).

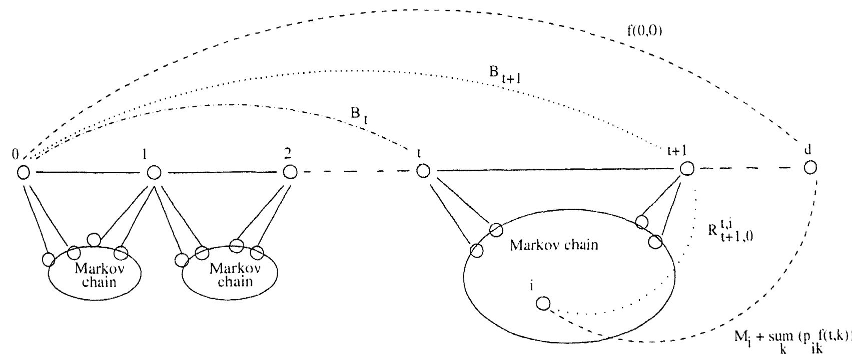

Fig. 3. Markov chain of J in the general case

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

- $R^{t,i}_{t+1,0}$ is equal to the cost of node i, which is M_i, plus the expected cost if there is no restart, plus finally the expected cost if there is one. We have

$$R^{t,i}_{t+1,0} = M_i + p^{t,i}_{t+1,0} D^{t,i}_{t+1,0} + p^{t,i}_{0,0}(D^{t,i}_{0,0} + U_t(i)), \tag{11}$$

where $p^{t,i}_{t+1,0} + p^{t,i}_{0,0} = 1$.
- Similarly, we can derive the following connection between $B_{t+1}(i)$ and $B_t(i)$:

$$B_{t+1}(i) = B_t(i) + p^{t,0}_{t+1,0}(\neg i) D^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i}(D^{t,0}_{t,i} + B_{t+1}(i)) + p^{t,0}_{0,0}(\neg i)(D^{t,0}_{0,0}(\neg i) + B_{t+1}(i))$$

Thus,

$$B_{t+1}(i) = \left(B_t(i) + p^{t,0}_{t+1,0}(\neg i) D^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i} D^{t,0}_{t,i} + p^{t,0}_{0,0}(\neg i) D^{t,0}_{0,0}(\neg i)\right) / p^{t,0}_{t+1,0}(\neg i). \tag{12}$$

Note that $p^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i} + p^{t,0}_{0,0}(\neg i) = 1$.
- For $U_t(i)$ we find similarly

$$U_t(i) = \left(B_t(i) + p^{t,0}_{t+1,0}(\neg i) D^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i}(D^{t,0}_{t,i} + R^{t,i}_{t+1,0}) + p^{t,0}_{0,0}(\neg i) D^{t,0}_{0,0}(\neg i)\right) / (p^{t,0}_{t+1,0}(\neg i) + p^{t,0}_{t,i}).$$

Using (12), we can write this as

$$U_t(i) = \alpha B_{t+1}(i) + (1-\alpha) R^{t,i}_{t+1,0} \quad \text{for some } \alpha \in [0,1]. \tag{13}$$

According to (10), the optimal policy in each node is to unblock it if this is cheaper than keeping it blocking; in other words, if $R^{t,i}_{t+1,0} < B_{t+1}(i)$. Note that if $t = 0$, restarting is always cheaper.

We are therefore especially interested in those values of t, where $R^{t,i}_{t+1,0} = B_{t+1}(i)$, because this is a time state where i should be unblocked. Using (13), this implies $R^{t,i}_{t+1,0} = U_t(i) = B_{t+1}(i)$. Using these equalities in (11), we find

$$B_{t+1}(i) = \frac{M_i + p^{t,i}_{t+1,0} D^{t,i}_{t+1,0} + p^{t,i}_{0,0} D^{t,i}_{0,0}}{p^{t,i}_{t+1,0}}. \tag{14}$$

Finally, we have from (12) that

$$B_t(i) = p^{t,0}_{t+1,0}(\neg i)\{B_{t+1}(i) - D^{t,0}_{t+1,0}(\neg i)\} - p^{t,0}_{0,0}(\neg i) D^{t,0}_{0,0}(\neg i) - p^{t,0}_{t,i} D^{t,0}_{t,i}.$$

Combining this with (14), we can see that $B_{t+1}(i) > R^{t,i}_{t+1,0}$ is equivalent to $B_t(i) > TH_t(i)$, where

$$TH_t(i) = -p^{t,0}_{0,0}(\neg i)D^{t,0}_{0,0}(\neg i) - p^{t,0}_{t,i}D^{t,0}_{t,i} + p^{t,i}_{t+1,0}(\neg i)\left(\frac{M_i + p^{t,i}_{t+1,0}D^{t,i}_{t+1,0} + p^{t,i}_{0,0}D^{t,i}_{0,0}}{p^{t,i}_{t+1,0}} - D^{t,0}_{t+1,0}(\neg i)\right) \quad (15)$$

$TH_t(i)$ is called the *threshold* of node i in time state t. Thresholds determine when nodes should be unblocked.

Lemma 3 *From time state t, the optimal policy is that the subset of blocking nodes remains constant until the first $t_0 \geq t$ for which there is a blocked node i for which $B_{t_0}(i) \geq TH_{t_0}(i)$ and $B_{t_0-1}(i) < TH_{t_0-1}(i)$. In time state t_0, i is unblocked.*

Proof. The threshold $TH_t(i)$ depends only on which nodes are blocking in time state t and which are not. This is because it consists of costs and probabilities of moves in the Markov chain in time state t, without including costs after restarts. Therefore, as long as the subset of blocking nodes does not change, $TH_t(i)$ is a constant. Furthermore, $B_t(i)$ is strictly increasing in t, since it is not cheaper to reach t than it is to reach $t-1$.

This implies that, as long as the subset of non-blocking nodes does not change over time, for every node i there is one specific time state t_i, where for $t < t_i$ we have $R^{t,i}_{t+1,0} > B_{t+1}(i)$ and for $t \geq t_i$ we have $R^{t,i}_{t+1,0} \leq B_{t+1}(i)$. This time state is determined by the threshold for each node i. A node for which the threshold is first reached should be unblocked at that time. Before then, the optimal subset of blocking nodes does not change, because blocking nodes must never become non-blocking by Corollary 1.

We show in [9] how to calculate all relevant thresholds, using that the Markov chain is acyclic.

5.4 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 time state in which it is unblocked. We will 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 unblocked next. This way, some nodes will be unblocked before they can even be reached from 0, but 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 i_x gets unblocked. We do this until all nodes

are non-blocking (or the job finishes). Within each interval $[t_{x-1}, t_x - 1]$ the thresholds are constant. We put $i_0 = t_0 = 0$.

Each step x consists of a number of calculations. There are always a number of relevant nodes, for which some calculations are necessary, in the correct order. 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 *non-blocking* nodes from which i_{x-1} can be reached. For every node i in this set, recalculate $D^{t,i}_{t+1,0}$, $D^{t,i}_{0,0}$ and $p^{t,i}_{t+1,0}$. Do this in reverse order (walking backwards through the graph), beginning with the nodes that have no successor in A_x.
- Define B_x as the set of *blocking* nodes from which 0 or a non-blocking 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.
- Determine the node $i_x \in B_x$ which first reaches its threshold. Calculate t_x and unblock i_x in time state t_x.

A few notes on this algorithm:

1. For blocking nodes $i \notin B_x$ we always have that the threshold is not reached yet, since it is always cheaper to restart immediately than to wait until the next (blocking) node and then to restart.
2. For nodes $i \in B_x \cap B_{x-1}$ from which i_{x-1} can not be reached, the threshold now is the same as in step $x - 1$.
3. It is possible for two or more nodes to have the minimal threshold value. In this case, unblock the latest one in the acyclic ordering.

Theorem 4 *This algorithm is optimal.*

Proof. This follows immediately from Lemma 3, the structure of the algorithm and the notes above.

Theorem 5 *This algorithm runs in $O(n \cdot e)$ time, where e is the number of edges in the Markov chain.*

Proof. Consider one step in the algorithm. The calculations for A_x take $O(\text{number of outgoing edges from } A_x)$, which is certainly $O(e)$.

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

The entire process therefore is $O(n \cdot e)$.

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.

5.5 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 unblocked, this can be represented internally by tagging each node with the time state in which it is unblocked. Every time the machine switches to a different node, the scheduler can check its tag and compare it to the current time 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.

Finally, more than one scheduler can use this strategy: if all schedulers have different priorities, they can disregard jobs (and schedulers) with lower priorities. They can then construct their own Markov chain, their own model of the availability of the workstations, and run jobs (one scheduler must run only one job at a time). As long as there are not too many background jobs being run, the important assumption that it does not take too long to restart will still hold.

6 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 [4] (by considering each part of the job as a separate job). Also, note that our solutions do not depend on d, neither in their computational complexity, nor as input parameter. This means that they only need to be computed once, for all possible occurring jobs. Then the nodes can be tagged with the time state in which they are unblocked, 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.

Recall however, that we can approximate a cyclic Markov chain by our acyclic ones (after deleting $\{0\}$), see Sect. 1.

Acknowledgements. 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, F. T. Leighton: Making commitments in the face of uncertainty: How to pick a winner almost every time. In: Proceedings of the 28th Annual ACM Symposium on Theory of Computing, 1996
2. A. Borodin, R. El-Yaniv: Online Computation and Competitive Analysis. Cambridge University Press 1998
3. P. Chrétienne, E. G. Coffman, J. K. Lenstra, Z. Liu: Scheduling Theory and its Applications. New York: Wiley 1995
4. E. G. Coffman, Jr., Leopold Flatto, Paul E. Wright: A stochastic checkpoint optimization problem. SIAM J. Comput. **22**(3), 650–659 (1993)
5. C. Derman: Finite State Markovian decision processes. New York: Academic Press 1970
6. A. Karlin, S. Phillips, P. Raghavan: Markov paging. SIAM J. Comput. **30**, 906–922 (2000)
7. C.-Y. Lee, L. Lei, M. Pinedo: Current trends in deterministic scheduling. Ann. Oper. Res. **70**, 1–42 (1997)
8. E. Sanlaville, G. Schmidt: Machine scheduling with availability constraints. Acta Inf. **35**(9), 795–811 (1998)
9. R. van Stee, J.A. La Poutré: Running a job on a collection of dynamic machines, with on-line restarts. Technical Report SEN-R9841, CWI, Amsterdam, December 1998