AI in Operating Systems: An Expert Scheduler

Dale Tonogai

University of California, Berkeley

ABSTRACT

The allocation of resources in operating systems is currently based on ad hoc heuristics and algorithms. Some well-known heuristics have been developed in such areas as process scheduling, memory management and network routing. The possibility that better decisions could be made by an Intelligent Agent employing expert system techniques is investigated in this report. The ability to learn from experience and to deal with uncertainty are two characteristics of expert systems that would be essential aspects of such an Intelligent Agent.

As multiple processor systems become more widely available, applications involving multiple concurrent processes will increase in number and importance. This increased interdependency among processes poses interesting problems in the area of processor scheduling. How should the processes be scheduled to achieve some optimal level of performance? A scheduler based on an expert system may prove to be a viable alternative to those that have been proposed and (in some cases) implemented so far.

This report describes the implementation of a learning mechanism that attempts to handle the problem of processor scheduling in such a multiprocessor environment. In effect, the Intelligent Agent tries to "learn" its own set of heuristics for optimally scheduling a set of co-operating processes. By simulating a relatively simple multiprocessor system we examine the merits of such an approach. [1]

December 15, 1988

[1] This work was supported by the Defense Advanced Research Projects (DoD) ARPA Order No 4871, monitored by Naval Electronic Systems Command under Contract No N00039-84-C-0089 The views and conclusions in this document are those of the author, and should not be interpreted as representing official policies, either express or implied, of the sponsoring agency

1. INTRODUCTION

The scheduling of processes in a computer system is currently performed according to ad hoc algorithmic techniques. Often the algorithms are found to perform less than optimally and may require "fine-tuning" to achieve better performance. In addition, scheduling disciplines that were appropriate in stand-alone uniprocessor systems, such as round-robin, may not be as effective in dealing with today's multiple processor systems. In this category we include both shared-memory multiprocessor systems and distributed systems. The parallelism among processes in these systems may dictate the use of alternative scheduling policies. A discussion of some techniques for taking advantage of multiple processor architectures is presented in a subsequent section.

In this report, I argue that an Intelligent Agent, employing expert systems techniques, has the potential for effectively performing processor scheduling. This is not to say that such an Intelligent Agent should be incorporated into an operating system under existing technology. It is well-known that the poor performance characteristics of artificial intelligence systems make them unsuitable for use in applications where very fast, real-time decisions have to be made. Rather, it is advocated that a rule-based approach to processor scheduling has advantages over an algorithmic approach, and may actually be effectively used in its stead in future systems.

A particularly attractive feature of expert systems is the ability to learn. This provides a level of adaptability that does not exist in current scheduling disciplines. Detecting a bad state, or a situation that leads to a bad state, and how to avoid it is an example of the usefulness of such a learning capability. This report focuses mainly on the learning aspect of our Intelligent Agent.

There are many problems involved in trying to optimally schedule a set of processes. Section 2 examines some of these issues. Section 3 describes some artificial intelligence techniques applicable to processor scheduling. In Section 4 we develop a model for our Intelligent Agent, henceforth referred to as an Expert Scheduler. Section 5 outlines an implementation of the Expert Scheduler in a simulated computing environment. In particular, we describe in detail the learning mechanism employed. Section 6 provides results on the performance of the Expert Scheduler as well as comments about the limitations of our approach.

2. SCHEDULING ISSUES

During the course of its lifetime, a process will require a variety of system resources, and its needs are likely to change over time. Among the resources it may require are physical memory, I/O devices, disk files and the CPU itself. Scheduling processes in an optimal manner is a difficult problem due to the lack of complete knowledge about their resource requirements. In addition, there may be interdependencies among the processes which may complicate the scheduling decision. Thus, what may be optimal for a single process considered in isolation may prove to be less than optimal when viewed globally.

The definition of an optimal schedule depends on the performance metric being used and this, in turn, may depend on the applications being supported. A computing environment may be designed to support batch processing, real-time applications, an interactive time-sharing system or a combination of the above. In a batch processing system, job throughput and CPU utilization may be the performance measures of interest. However, in an interactive time-sharing system, the minimization of response time (or even of the variation in response time) may be the performance criterion. Other criteria commonly used (also to be minimized) are turnaround time and waiting time. The appropriateness of a scheduling policy will also depend on the performance measure(s) we are trying to optimize.

CPU scheduling can take two forms: short-term scheduling and long-term scheduling. The short-term scheduler deals with allocating the CPU to one of the processes currently ready to be executed. The long-term scheduler determines which processes should be allowed to enter the system. Load balancing in a distributed system falls under this category. When a new process is created, we would like to assign it to the machine which is the least loaded. If a process is not constrained to execute only on the machine to which it was originally assigned, we may also consider process migration. In some operating systems, due to the memory management mechanism, the long-term scheduler may also be required to select an entire process to be swapped in or out of memory. One final consideration in long-term scheduling may be to select a "good" mix of I/O-bound and CPU-bound processes. If we are not careful, we may, for instance, get a situation where the majority of processes are I/O-bound. As a result, the processor(s) may remain under-utilized as the I/O subsystem becomes a bottleneck and reduces the system's processing rate.

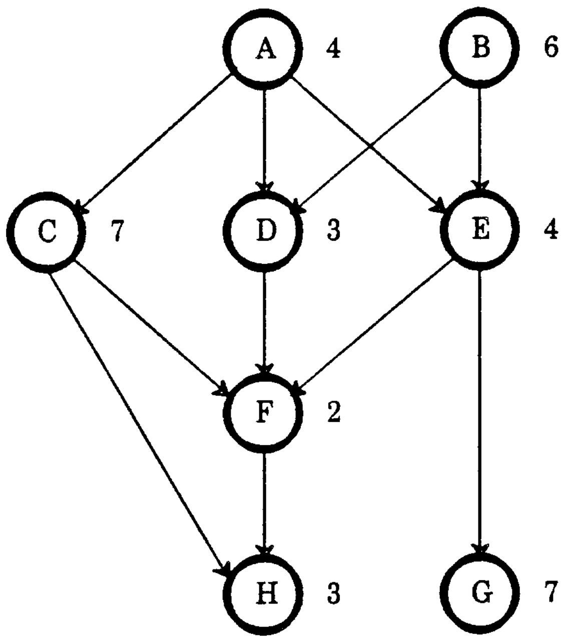
As mentioned above, memory requirements and machine load are two possible criteria on which the long-term scheduler may base its scheduling decision. Factors influencing the short-term scheduling policy include execution time requirements and process interdependencies. The exact execution time of a process is difficult to know a priori, but reasonable estimates can be made based on its past history or on information supplied by the user. The interactions between processes can be in the form of interprocess messages or, in shared memory systems, via locks and semaphores. In the former case, it may be advantageous to *coschedule* cooperating processes as is done in Medusa [OUST80]. In the latter case, it may be preferable not to remove a process from its processor if the lock on which it is waiting will soon be released by a process executing on another processor. We would also like to avoid descheduling a process while it holds a lock.

The problem we will be considering in the remainder of this report is that of short-term scheduling. Some current approaches that have been proposed for short-term scheduling are described in this and the following paragraphs. The shortest-job-first (SJF) scheduling discipline selects the process with the smallest execution time requirements. It is theoretically interesting because it is provably optimal in that it provides the minimum average turnaround time. This method has the disadvantage of requiring knowledge of at least an estimate of the expected processing time. A scheduling algorithm not requiring any knowledge of execution times is the round-robin discipline. This well-known algorithm cyclically schedules each process for a pre-determined quantum of time. If the process does not complete in its time quantum, it is suspended and assigned to the end of the ready queue, to be serviced again at a later time. This policy is much fairer than SJF in that long running processes cannot be starved by the continual arrival of short running processes.

A class of scheduling disciplines that attempts to favour short execution time processes but still provide some service to long running processes is that of the policies based on multi-level feedback queues. Each queue is managed in first-in first-out order and has associated with it a priority and a fixed time quantum: the higher the priority the smaller the time quantum. When a process becomes runnable, it is assigned to the highest priority queue. Each time a scheduling decision must be made, the processor selects a process from the highest priority queue that is non-empty. If the process does not complete in its time quantum, it is reassigned to the next lower priority queue. In this manner, CPU-bound processes gradually sink to queues with lower priority but with greater time quantums. In order to prevent starvation, a process can be moved to a higher priority queue if it has been waiting too long a time. Note that this feedback mechanism provides an elementary form of learning by attempting to migrate processes to the most appropriate queue.

The above algorithms do not support any mechanism to take advantage of multiple processor architectures, and, in fact, make no distinction between uniprocessor and multiprocessor systems. Current multiprocessor systems encourage the use of multiple concurrent processes to perform a single task. These processes often communicate via messages. Ousterhout [OUST82] has proposed a technique that attempts to reduce the context switching overhead encountered by naive scheduling policies. When a process, A, must wait for an event to occur (e.g., the receipt of a message), it is usually suspended by removing it from its processor. But if the event on which it is waiting occurs very soon afterward (e.g., a process, B, executing on another processor, sends a message to A), then we must perform two context switches on A. Whereas, if we had kept A loaded and idled A's processor for a short time (termed a *pause*), we could have eliminated the context switches with only a slight penalty in processor utilization. If the event does not occur within the *pause* time, then the process can be removed from the processor. However, the optimal value for the *pause* time may be difficult to determine. Ousterhout also describes three different algorithms that attempt to *coschedule* those processes that are known to be cooperating via message-passing. The intention is to improve the likelihood that a process that wishes to receive a message will receive it while it is assigned to a processor and hence reduce the number of context switches.

The problem of scheduling a set of concurrent processes constrained by precedence relationships (i.e., where a process cannot execute until some set of predecessor processes has completed) has been widely studied. The partial ordering among the processes is often represented by a directed acyclic graph known as a precedence graph, where the nodes represent processes and the arcs represent precedence constraints. This is in fact the process model that will be used to carry out our study. Optimal algorithms that minimize the completion time, given the precedence constraints, have been discovered for particular cases [GONZ77]. For example, Coffman and Graham [COFF72] have developed an algorithm which is provably optimal in the two-processor case in which the execution times are known. Although there is no known efficient algorithm for finding the optimal schedule in the general case, in a study conducted by Adam et al. [ADAM74] the heuristic known as *Highest Levels First with Estimated Times* (HLFET) was found to be near optimal in over 95% of the cases. Again, in this heuristic, estimates of process execution times are assumed known and are used to label the nodes of the precedence graph. Based on this labelling, the *level* of each process is computed as the sum of its own execution time plus the longest path to a terminating process (a process with no successors). Figure 1 shows an example of a precedence graph with the nodes labelled with the execution times and the *level* for each process. The



PROCESS	LEVEL
A	16
B	17
C	12
D	8
E	11
F	5
G	7
H	3

Figure 1: Precedence Graph

intention of the HLFET heuristic is to schedule as early as possible those processes which lie in the critical path of the precedence graph.

3. EXPERT SYSTEMS

Expert systems have been successfully designed for a variety of applications. Some notable examples include MYCIN, which is used in medical diagnosis, and Dendral, which has discovered previously unknown rules of chemistry [ANDR85]. What makes an expert system potentially useful in managing an operating system, and in particular, performing processor scheduling, is its ability to deal with uncertainty and to adapt to changing conditions. Adaptability and the ability to learn from experience are two features which are particularly attractive considering that the workload in a computer system can exhibit large variability. Uncertainty may manifest itself simply in not knowing the execution time requirements of a single process or in a more complex problem, such as trying to discern the instantaneous global state of a distributed system. There are many methods that have been proposed to deal with uncertainty: among them are *fuzzy logic* [ZADE83], *certainty factors* [SHOR75] and techniques based on Bayes' rule [GENE87].

Two other Artificial Intelligence techniques which may be appropriate in the design of an expert scheduler are non-monotonic and temporal reasoning. In traditional systems, monotonic reasoning methods are employed whereby beliefs are added to the knowledge base and new beliefs can be inferred from existing ones. Non-monotonic reasoning [MCDE80] permits beliefs to be removed from the knowledge base. In particular, the inference mechanism allows beliefs to be inferred based on incomplete knowledge; that is, by making assumptions and using default reasoning. If some new belief is later discovered that contradicts an existing one, the latter must be discarded. Such a technique seems appropriate to computer systems where the environment may change significantly enough over the course of time that a particular rule may no longer be valid. The other class of reasoning methods, known as temporal logics [MCDE82] [ALLA81], which are in fact a form of non-monotonic reasoning, also seem applicable to the problem of processor scheduling. These logics include the element of time in their representations. Concepts such as causality, persistence and the relative timing between events are the focus of these methods, thus making them well-suited for modelling systems undergoing continuous change.

Pasquale [PASQ87] has investigated the use of a knowledge-based system to manage a distributed computer system. Based on decentralized control, each node in the system would contain an instance of a so-called Expert Manager. Each Expert Manager attempts to infer the current global state of the distributed system based on partial and uncertain information received over the network. As information is received, hypotheses are generated to explain the observed evidence, and then experiments are possibly performed to validate/invalidate competing hypotheses. Hypotheses that are believed to be true beyond a certain threshold become facts and are added to the knowledge database. Pasquale's model also incorporates a learning feature which attempts to detect what conditions lead to bad states. The application of this approach to the problem of decentralized load balancing is given in [PASQ88].

4. THE EXPERT SCHEDULER

Traditional expert systems require a human expert to impart his expertise into a knowledge base. This function, also known as knowledge engineering, requires the human expert to express his knowledge in terms of heuristics in some machine acceptable format. This task is by no means trivial. However, in the application which we are considering, there is no human expert to whom we can turn. Although many useful heuristics have been derived for multiprocessor scheduling, the problem is still far too complex due to the myriad factors involved. Our Expert Scheduler is likely to require a greater capacity for learning in order to be successful. Indeed, the ability to learn will be a critical part of the Expert Scheduler.

In addition, the Expert Scheduler will have functional components that have very different real-time requirements. For example, scheduling decisions must be made at context-switching speeds. The operation of much of the learning subsystem, on the other hand, is not as time critical and can be relegated to "background" mode.

4.1. An Architecture for the Expert Scheduler

The Expert Scheduler envisioned (see Figure 2) would be composed of at least the following modules:

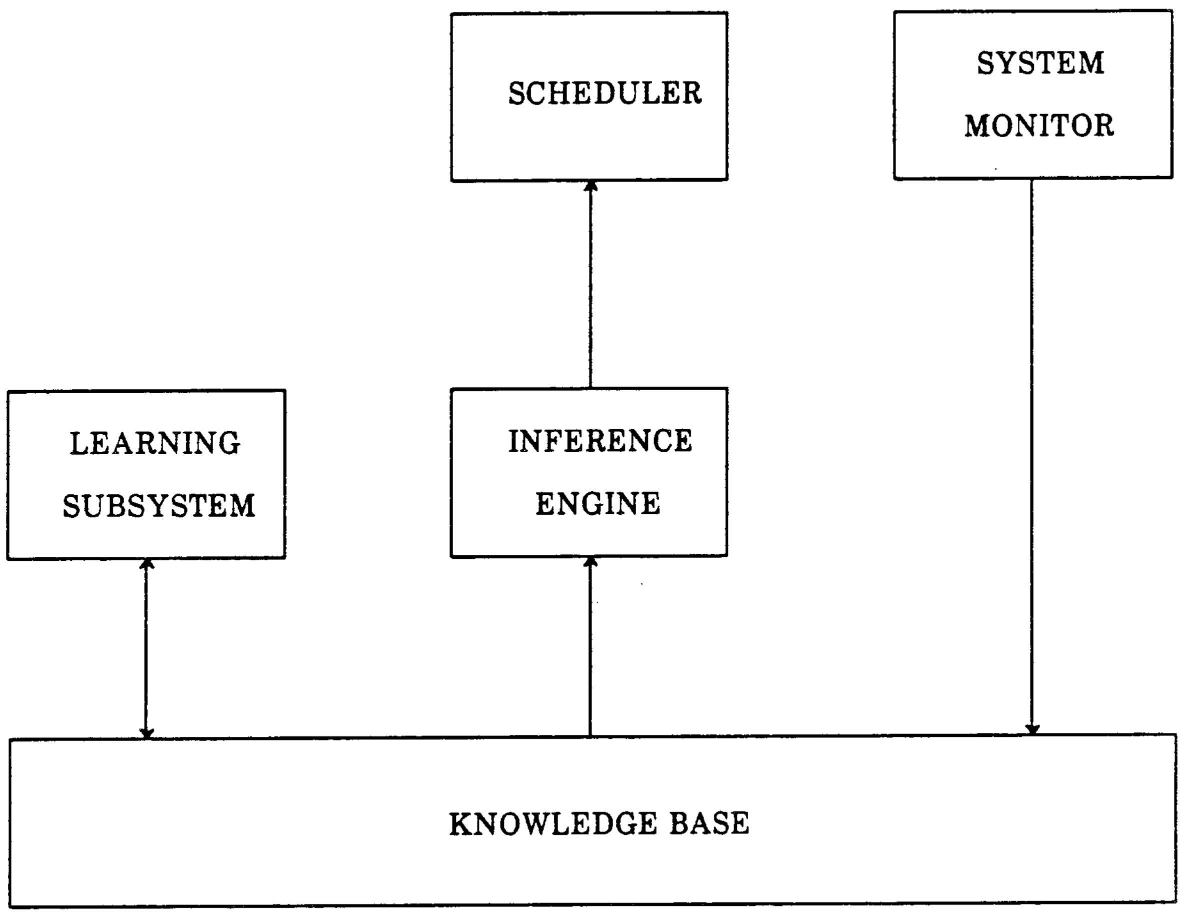


Figure 2: Expert Scheduler

1) Scheduler - The Scheduler would actually be responsible for assigning processes to a processor. It would incorporate an inference engine which uses the rules from the Knowledge Base to make its decisions.
2) System Monitor - This module monitors the activity in the system and records its observations in the Knowledge Base for use by the Learning Subsystem. Information such as process histories, queue lengths and other statistical information could be recorded.
3) Learning Subsystem - This part of the model consists of several components (see Figure 3), and its function is described further in the next section.
4) Knowledge Base - This database serves as a repository for information (i.e., rules, statistics, hypotheses) required by each of the above modules.

There are other components that could be incorporated into our model. For instance, a hypothesis generator, such as is used in Pasquale's Expert Manager, may prove to be useful. Such a mechanism could allow the Expert Scheduler to experiment, such as by changing the size of a time quantum, in order to determine its effect on the schedule.

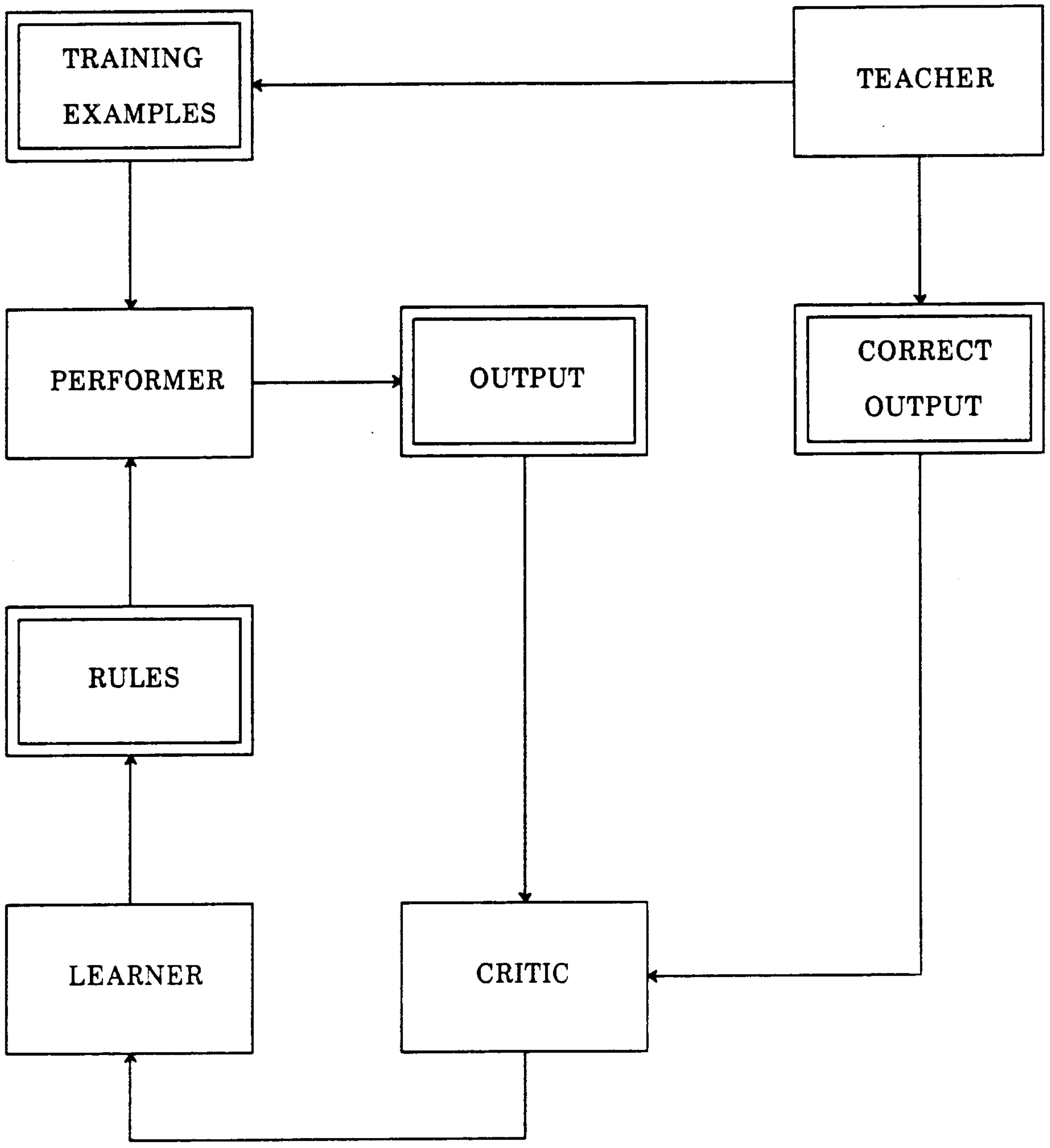


Figure 3: Learning Subsystem

4.2. The Learning Subsystem

The Learning Subsystem permits the Expert Scheduler to try to improve its performance. It is aware of the performance goals and attempts to generate heuristics that will achieve those goals. As new heuristics are learned, they must of course be passed to the Scheduler. The basic idea behind the learning mechanism is that, as successive scheduling problems are presented to the Learning Subsystem, heuristics will be generated that produce better schedules. The more scheduling problems examined, the better the heuristics will be. In essence, the Expert Scheduler will learn by experience. The components of the Learning Subsystem and their functions are as follows:

1) Learner - This component is responsible for generating scheduling rules based on training examples supplied by the Teacher and on feedback from the Critic.
2) Teacher - Using data collected by the System Monitor in the form of scheduling problems, the Teacher generates the training examples. The selection of "good" training examples is crucial to the success of the Learner.
3) Performer - The Performer actually uses the rules generated by the Learner on the set of training examples.
4) Critic - The Critic compares the scheduling decisions made by the Performer to the "correct" decision as specified by the Teacher. It is responsible for evaluating the performance of a scheduling rule.

5. IMPLEMENTING AN EXPERT SCHEDULER

Although Pasquale [PASQ88] was able to prove the feasibility of using an expert system approach to load balancing, given the nature of current AI techniques in terms of space and time requirements, an Expert Scheduler attempting to perform short-term scheduling in a real computer system may not be very feasible. Hence, this section describes an implementation of the Learning Subsystem presented in Section 4.2. Such an implementation would allow us to perform some experiments and to answer some basic questions about machine learning. Can it be applied to short-term processor scheduling? If so, how quickly does it take to converge on a good scheduling heuristic? What kinds of problems may arise in this sort of application?

5.1. The Computing Model

In order to keep the implementation relatively straightforward, many assumptions and simplifications have been made. For instance, no attempt has been made to model any I/O activity or context-switching overhead. The computing environment which will be simulated and in which our Learning Subsystem will operate will be based on a system of N identical processors in a tightly coupled arrangement. Processes that are waiting to be allocated to a processor are placed in a single ready queue accessible by any processor. Each processor executes its currently assigned process for one time quantum. If the process completes its execution time requirements in that time quantum, it is retired from the system; otherwise, it is returned to the ready queue. It is assumed that all the processors operate in synchrony. That is, all processors simultaneously complete their time quantums and make scheduling decisions. In this manner all runnable processes will be available for selection at the same time.

The scheduling problems that the Learning Subsystem will attempt to solve are of the type that can be represented by precedence graphs. The processes that enter the system are assumed to arrive with known CPU time requirements, expressed as an integral number of time quantums, as well as their precedence constraints. Although the exact execution time requirements are generally unknown, reasonable estimates can usually be made. We make the additional simplifying assumption that these estimates are exact. There are no other dependencies between the processes. In particular, we have not modelled any kind of interprocess communication. The performance measure that the Learning Subsystem will try to optimize is the completion time.

5.2. The Teacher

In general, learning mechanisms employed previously were based on supplying a learner with a set of training examples. Each training example consisted logically of a problem and the correct solution. Armed with a sufficient number of these training examples, the learner would attempt to derive a set of problem-solving heuristics. Then, when presented with a completely new instance of the problem, the learner would use its own heuristics to produce a hopefully correct solution. The fundamental difference between our application and previous work is the lack of a pre-defined set of training examples in our case. How then is our Expert Scheduler to learn?

We still wish to supply our Learner with training examples and it is the job of the Teacher to generate those training examples. The source of the training examples will consist of the scheduling problems that occur at run time, hereafter referred to as *training problems*. The generation of the training examples proceeds as follows.

The Teacher first finds an optimal or near-optimal solution to a *training problem*. Although this problem is NP-complete, there are a variety of AI techniques that the Teacher could employ to aid in this process. (This particular aspect of the Teacher will be discussed in greater detail in a subsequent section). In our implementation, we have taken a simplistic approach, and found a solution by using an exhaustive search. An exhaustive search is, of course, impractical in general because of the combinatorial explosion of the search space that can result. However, in our simulations we have reduced the size of the search to an acceptable level by limiting the size of the *training problem*. This is done by putting an upper bound on the number of processes in the precedence graph. The search algorithm used is a simple Breadth First Search, and included in the implementation is a hard-coded limit, such that the search, if it exceeds this limit, is abandoned.

There are potentially multiple schedules that are good solutions to a *training problem*. One of these solutions is selected at random and, from it, the Teacher generates the training examples. In our computing model, each training example consists of a set of runnable processes and an indication of which member(s) of the set could be assigned to a processor to achieve optimal performance. Since we express a solution as a series of *scheduling moments*, where a *scheduling moment* represents the subset of processes in the ready queue that were assigned to processors at a particular instant of time, it is a simple matter to create training examples from each of the *scheduling moments*. For example, suppose that in a 2-processor system we have a *scheduling moment* such that there are four processes, denoted $p1$, $p2$, $p3$ and $p4$, in the ready queue, and that $p1$ and $p2$ were assigned to the two processors. Then two training examples are generated, one to correspond to each processor, since each would have to make a scheduling decision in a real situation. The training examples generated would take on a form similar to the following:

	Runnable	Schedulable
training example 1.	$p1, p2, p3, p4$	$p1, p2$
training example 2.	$p2, p3, p4$	$p2$

In the original version of the Teacher, only $p1$ would have been included as schedulable in training example 1 above. This turned out to be a reasonably large difficiency in the Teacher to the extent that a rule that would select the processes in the opposite order, i.e., $p2$ followed by $p1$, payed an unfortunately high penalty.

In order to improve the "quality" of the training examples, not every *scheduling moment* results in the generation of training examples. Only those *scheduling moments* in which the number of runnable processes is greater than N, the number of processors, are used as sources for training examples. *Scheduling moments* in which the number of runnable processes is not greater than N are discarded. This is done for the following reason. In order to be useful or "informative" to the Learner, good training examples should give an indication not only of what processes to schedule, but also of what processes not to schedule. This allows the Learner to differentiate between what leads to a good schedule and what leads to a bad schedule. The training examples that would result from *scheduling decisions* in which the number of runnable processes is not greater than N would not be very informative to the Learner, since all the runnable processes are being scheduled.

As mentioned earlier, there is likely to be more than one good solution to a *training problem*, with every schedule producing the same completion time being the extreme case. In this extreme case, note that any heuristic would achieve the same result if applied to this kind of *training problem*. Hence, selecting one of these solutions at random could produce training examples which cause the Learner to converge on a less than optimal heuristic. For this reason, training examples are only generated from *training problems* in which the number of good solutions is some reasonably small subset of the total number of solutions.

5.3. The Learner

The learning mechanism that has been employed is based on an algorithm known as a Beam Search [FORS86]. It is based on the idea of starting with a rule that applies to a specific training example, and then generalizing it until it handles enough training examples to the satisfaction of the Critic. This is explained more fully in the following paragraphs.

In each training example, each runnable process has associated with it a set of attributes such as its CPU time requirements, the time since it entered the system, and so on. It is on the basis of these attributes that the Learner forms its scheduling heuristics. Hence a rule might take on the following form:

```
if ready(process) && longest_waiting(process) && shortest_CPU_time(process)
   then schedule(process)
```

In order to generalize a rule, we merely drop one of the attributes from the scheduling rule. Thus

```
if ready(process) && shortest_CPU_time(process)
   then schedule(process)

          AND

if ready(process) && longest_waiting(process)
   then schedule(process)
```

are both generalized versions of the first rule given above.

The algorithm initially selects *t* training examples and generates rules that apply specifically to those training examples. This set of rules we will refer to as the set R. The Beam Search then proceeds along the following steps:

1. The Performer uses each rule in R on the set of runnable processes in each training example. Depending on how the attributes are used, the scheduling rule may potentially select zero, one or more than one process. The application of a rule to a training example can result in four possible outcomes:

   O_1: The process(es) selected is correct.

   O_2: No process is selected at all.

   O_3: None of the processes selected is correct.

   O_4: Some of the processes selected are correct and some are incorrect.

2. The performance of each rule is evaluated by the Critic, which assigns to each rule a *rating*. The calculation of the *rating* is described in the next section.

3. The *p* rules in the set R with the best *ratings* are each generalized in all possible ways to form a new set R. Since it is possible that this new set may contain rules that have been previously tested, these duplicate rules must be discarded. If R is not empty, then go to step 1, otherwise continue on to step 4.

4. The *q* rules with the best overall *ratings* are saved as the result of the Beam Search and constitute the learned heuristics.

Note that the parameters *p*, *q* and *t* are all easily adjustable.

The execution of the algorithm is shown graphically in Figure 4, where *p* has been set to three. Each node represents a scheduling rule. The solid nodes at each level indicate the *p* best rules that should be further generalized. The learned heuristics would be selected from among all the rules represented by solid nodes.

Since the time to conduct the Beam Search grows exponentially with the number of process attributes, these were kept to a minimum. The attributes initially under consideration were:

- arrival time

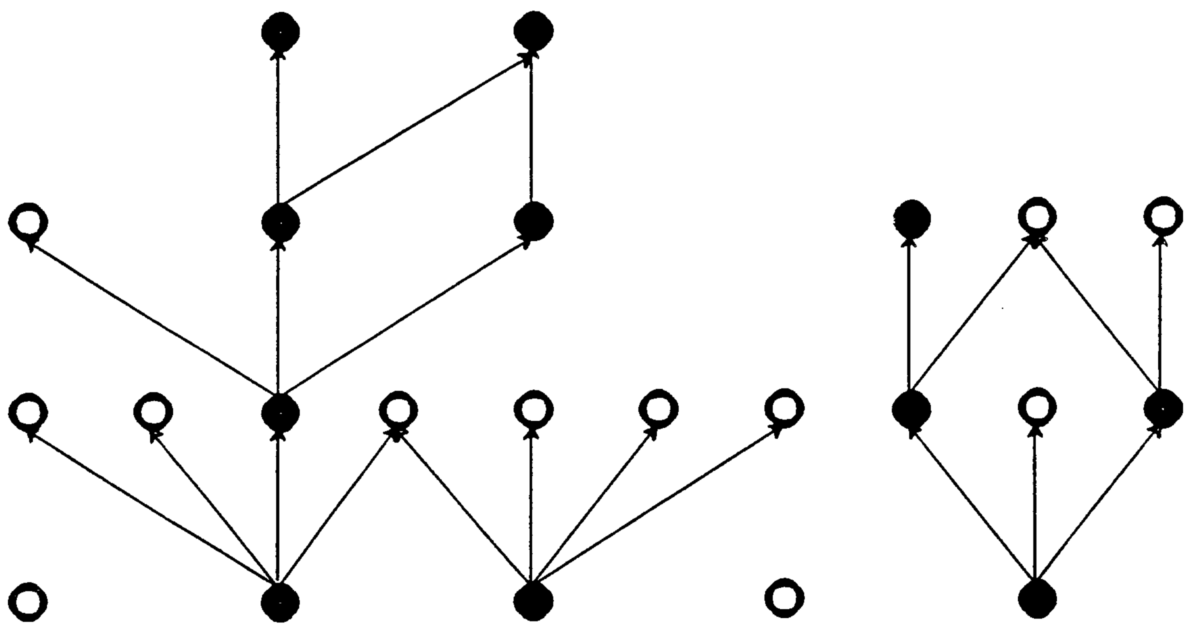


Figure 4: Beam Search

- amount of CPU time consumed (in integral number of time quantums)
- last time the process was assigned a processor
- estimate of the amount of CPU time remaining (which in our model, as pointed out earlier, is assumed to be known exactly)
- number of successor processes (i.e., number of processes that cannot execute until this one completes)
- *level* of the process as computed from the precedence graph

After some experimentation, it became obvious that some attributes would have no effect on a good scheduling heuristic. They were initially included just to test the correctness of the Learner. In order to minimize the execution time of the Learner, all the results that follow are based on using only the last four attributes listed above.

5.4. The Critic

The assignment of a *rating* by the Critic must be carefully considered, and is indeed a difficult problem. If the evaluation technique used by the Critic is inappropriate, then the Beam Search may not converge upon the best heuristic(s). This is a case of a more general class of problems common in AI theory, known as the *credit assignment* problem. As was noted in the previous section, there are four possible outcomes when a scheduling rule is applied to a training example. Based on the outcomes taken over all the training examples, the Critic computes the rule's *rating*, r, using the following formula:

$$r = \alpha_1 n_1 + \alpha_2 n_2 + \alpha_3 n_3 + \alpha_4 n_4,$$

where

n_1 = number of outcomes of type O_1
n_2 = number of outcomes of type O_2
n_3 = number of outcomes of type O_3
n_4 = number of outcomes of type O_4.

The α_i's represent constant weights associated with the four possible outcomes. An α_i is set to a positive value if its associated outcome is favourable, and a negative value if unfavourable.

The difficulty lies in assigning appropriate values to the α_i. Consider what each outcome type means:

O_1: The rule is performing well, and so α_1 should be assigned a relatively large positive value.

O_2: Since the rule has failed to select any process, this probably is an indication that the rule is too specific and needs to be generalized. Hence, α_2 should be assigned a negative value, although how large a negative value is open to question.

O_3: The rule is performing as poorly as it possibly can in this case, and α_3 should be assigned a relatively large negative value.

O_4: This is the case which is the most difficult to deal with. The rule may have been over-generalized at this point and, as a result, selected too many processes. Another possibility is that the attributes used in the rule did not serve to differentiate the processes in this particular case (i.e., the attributes took on the same value for many of the processes). If the rule has been over-generalized, then α_4 should be assigned a relatively large negative value since the rule is essentially no good. If it is a matter of the attribute(s) having the same value for each process, then either (a) the attribute(s) just may not have any bearing on a good rule, in which case the Critic should discard the rule (i.e., assign α_4 a large negative value), OR (b) we have just come across an unfortunate case in which the rule does not work very well, so α_4 should be assigned a small negative value.

It should also be noted that our training examples are not perfect. Because of the way they are generated, there is no guarantee that a training example will lead the Learner towards a good heuristic. This must also be taken into consideration when assigning values to the α_i. For instance, O_3 may be more the result of a poor training example than of a poor rule.

6. RESULTS

The results of our study are based on simulations of 2, 3 and 4 processor systems of the type described in Section 5.1. The simulation is actually broken up into three separate components. In the first component, *training problems* are input to the Teacher and a set of training examples are produced. In the second component, training examples are supplied to the Learner, which generates a scheduling heuristic. The final component accepts as input a scheduling rule and a scheduling problem, and computes the completion time by simulating an N-processor system.

All scheduling problems (and *training problems*) are in the form of trace files. Each entry in the trace file supplies the following information for one process:

- arrival time
- CPU time requirements (in integral number of time quantums)
- set of predecessor processes

To create the scheduling problems, a process generator (PG) program, that produced trace files in the requisite format, was written. The data required in the trace files is generated randomly according to the following parameters:

- maximum CPU time requirements per process, T_{max}
- minimum height of the precedence graph, H_{min}
- maximum height of the precedence graph, H_{max}
- maximum number of processes per level, L_{max}
- maximum number of predecessor processes per process, P_{max}

The height of the precedence graph, the number of processes in each level, the number of predecessor processes for each process, and the CPU time requirements are all computed using a uniform distribution. The actual set of predecessors for each process is selected at random from among the set of processes at lower levels.

6.1. Generation of Training Examples

As mentioned above, the *training problems* were generated using the process generator. However, the PG parameters had to be carefully selected. If the *training problems* were too "simple", i.e., had few or no interprocess dependencies, then the Teacher had difficulty producing training examples because all the schedules for the problem would often have identical completion times. For example, using the 2-processor *training problems* in the 3-processor simulations failed to produce any training examples at all. We are thus faced with the following tradeoff: keeping the problem size small at the expense of having to solve many *training problems* to get a sufficient number of training examples OR allowing larger problems at the expense of increased search times.

After some brief experimentation, two sets of *training problems* were generated for each of the 2, 3 and 4 processor cases. Table I gives the PG parameters used for Set A and Table II gives the parameters for Set B. L_{max}, the maximum number of processes per level, was set to one more than the number of processors in order to increase the likelihood that there would be more runnable processes than processors. Note that the difference between Set A and Set B is that P_{max} has been increased from 3 to 4. This was done to determine what kind of an effect, if any, different shapes of precedence graphs would have on the generation of training examples.

Set A			
PG Parameter	2-processors	3-processors	4-processors
$T_{\max}$	4	4	4
$H_{\min}$	3	3	3
$H_{\max}$	3	3	3
$L_{\max}$	3	4	5
$P_{\max}$	3	3	3

Table I

Set B			
PG Parameter	2-processors	3-processors	4-processors
$T_{\max}$	4	4	4
$H_{\min}$	3	3	3
$H_{\max}$	3	3	3
$L_{\max}$	3	4	5
$P_{\max}$	4	4	4

Table II

Figures 5 and 6 show the number of training examples generated as a function of the number of *training problems* for both sets. A sufficient number of *training problems* were supplied to the Teacher until a minimum of 100 training examples had been generated. It seemed to be generally true that, as the number of processors increased, it became more difficult to produce training examples. With more processors a particular scheduling decision is likely to have less impact on the overall schedule. The *training problems* in Set B also produced a higher yield of training examples. This was not unexpected, since the precedence graphs in Set B tended to be slightly more complex and hence, were more "interesting" to the Teacher. This, however, was achieved at the cost of longer search times. In fact, in one of the 4-processor *training problems* the search limit was exceeded.

6.2. Performance of Learned Heuristics

Once the training examples had been generated, they were input to the Learner. The number of training examples supplied to the Learner was varied, and the learned heuristic in each case was applied to a set of 20 test scheduling problems. The values assigned to the parameters of the Beam Search, p, q and t, were 4, 2 and 4, respectively. The test problems were generated using the PG parameters given in Table III. The effectiveness of the learned heuristic was measured by comparing its completion time against that produced by the "optimal" schedule (for the 2-processor, case the schedule produced by Coffman's algorithm was used as the optimal; for the 3 and 4-processor cases, the schedule produced by the HLFET heuristic was chosen as the optimal).

Figures 7-12 show the performance of the learned heuristics as a function of the number of training examples supplied to the Learner. The "percentage within optimal" is the average value of this percentage over all 20 test cases, with 1.0 being perfectly optimal. As a further point of comparison, the performance of

Test Scheduling Problems	
PG Parameter	Value
T_{max}	8
H_{min}	3
H_{max}	12
L_{max}	6
P_{max}	5

Table III

round-robin and random scheduling is also shown in the graphs. The Learner converges fairly quickly on an optimal scheduling heuristic, and, in fact, generally requires far less than 100 training examples. Note also the relatively large downward spike in Figure 12. This illustrates one of the deficiencies in the Beam Search algorithm: if the initial set of training examples is poorly selected, a poor heuristic will result. The chance of this occurring can be reduced, however, by increasing the parameter t.

6.3. Credit Assignment

The setting of appropriate values for the α_i was difficult to determine without some initial experimentation. Different values for the α_i could produce substantial differences in the quality of the learned heuristics. It was found, however, that the values of α_2 and α_4 had the greatest impact on the Learner. If $|\alpha_2|$ was small compared to $|\alpha_4|$, specific rules would be favoured over general rules. If $|\alpha_2|$ was large compared to $|\alpha_4|$, the opposite would be true. Since the learned heuristic was to be applied in all situations, a general rule seemed preferable, and so the latter was chosen. One other consideration, the fear that the particular settings used for the α_i might lead the Learner along a path that would not allow it to converge on a good general rule, turned out to be unfounded.

The actual values used for the α_i are given in the first row of Table IV.

	α_1	α_2	α_3	α_4
case A	8	-4	-8	-1
case B	8	-1	-8	-1
case C	8	-1	-8	-4
case D	8	-4	-8	-4

Table IV

To demonstrate the effect of changing the settings of the α_i, Figure 13 shows the performance of the learned heuristic, computed under each combination in Table IV, for Set B in the 3-processor case. Depending on the application, the determination of the optimal values for the α_i may require no more than generating learned heuristics using a few combinations similar to those listed in Table IV, testing the heuristic on a few sample problems, and selecting the final values for the α_i according to which case performs the best. If this is not sufficient, a more sophisticated algorithm, such as *hill-climbing*, could be used. It is unclear, however, how an appropriate starting point would be determined.

6.4. Limitations of the Expert Scheduler

The computing system simulated in this study was necessarily simplistic, and hence did not model many factors. The effects of interprocess communication, context switching overhead, and so on, were all ignored; nor was the possibility of satisfying multiple, and possibly competing, performance criteria considered. To investigate more thoroughly the benefits of an Expert Scheduler requires more elaborate models. This would complicate the implementation of the Expert Scheduler, but would not likely detract from the basic architecture.

The method used by the Teacher to generate training examples was adequate in our simplified environment. However, an exhaustive search of a "small" *training problem* may not be feasible in a more complex model. For instance, there may be no such thing as a "small" problem. This area requires more careful study, but a possible approach could be to use any existing learned heuristics to help guide the search. In order to learn any new heuristics, the Teacher would also have to have the capability to deviate from the existing heuristics, perhaps randomly, to find a potentially better solution. Other AI techniques to help prune the search tree could also be used here.

The heuristics generated by the Learner were also fairly simple. The same learned heuristic was applied whenever a scheduling decision had to be made. Considering the changing load conditions the Expert Scheduler would have to operate in, this would seem insufficient. To make the Expert Scheduler more effective, the Learner could generate multiple heuristics along with an indication of the circumstances under which each should be applied. In addition, "negative" heuristics of the type "do not schedule process A when in state B" may be useful. Along the same lines, the Teacher could produce "negative" training examples.

Although the Expert Scheduler has no initial knowledge of what a good scheduling heuristic is, it must know what attributes a process possesses, for this is what it bases its learned heuristics on. This is where a human expert is likely to be needed, to supply useful process characteristics to the Expert Scheduler. This task is not trivial. Irrelevant attributes will only slow down the functioning of the IA. Relevant attributes may be non-intuitive or not directly observable (e.g., *level* attribute used in our model).

In our particular computing model, we have made the assumption that each process's CPU time requirements are known exactly. In fact, the best that we can generally hope for is a good estimate. To overcome this deficiency, the simulation could be easily enhanced to accept CPU time requirements that are given as estimates, with some non-zero probability that they will turn out to be incorrect. We could, for example, model the time requirement as a Poisson distribution and use the mean as the estimate.

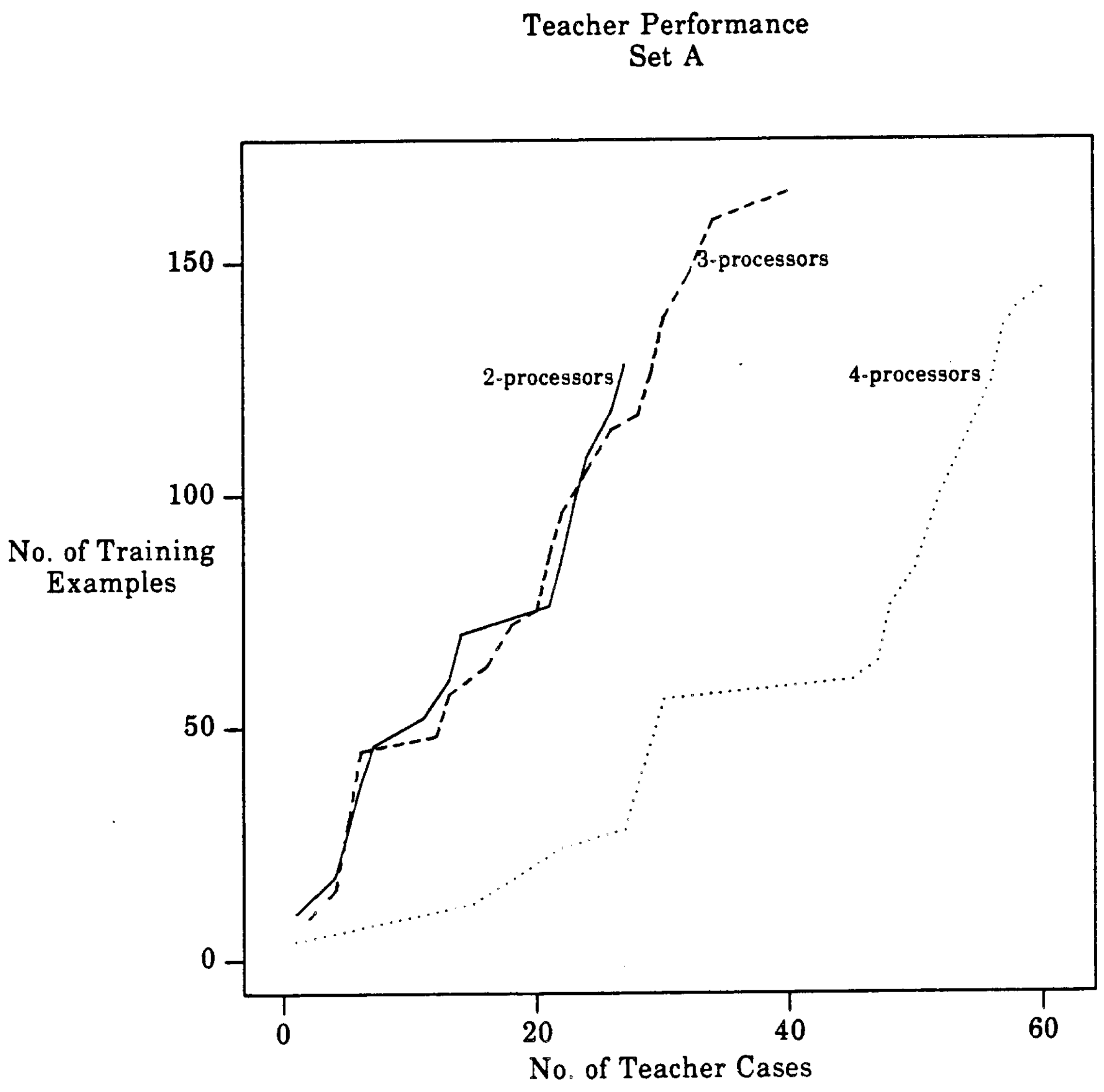

Teacher Performance
Set A
No. of Training Examples
No. of Teacher Cases
0
50
100
150
0
20
40
60
2-processors
3-processors
4-processors

Figure 5

Teacher Performance
Set B

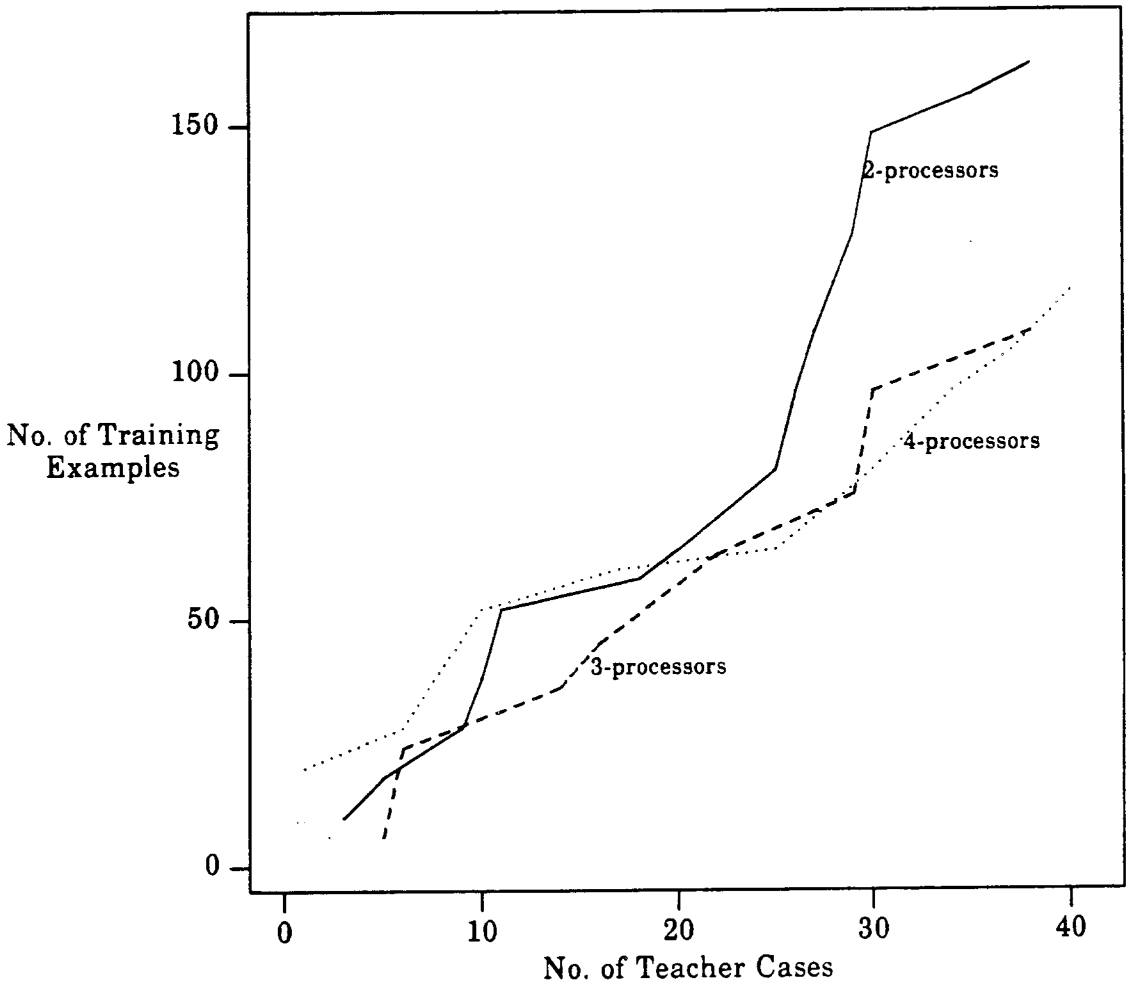


Figure 6

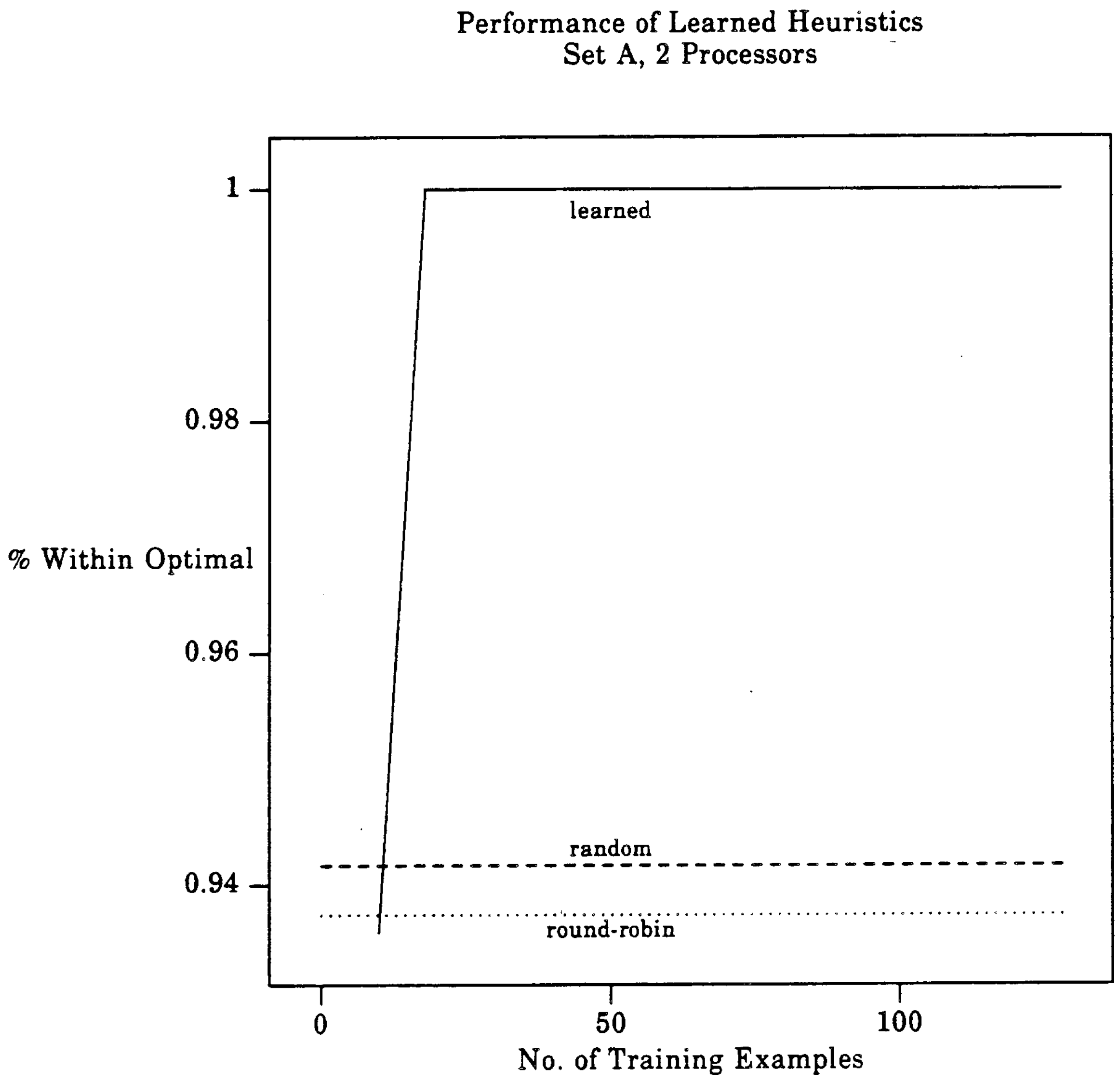

Performance of Learned Heuristics
Set A, 2 Processors
1
0.98
0.96
0.94
% Within Optimal
learned
random
round-robin
0
50
100
No. of Training Examples

Figure 7

Performance of Learned Heuristics
Set A, 3 Processors

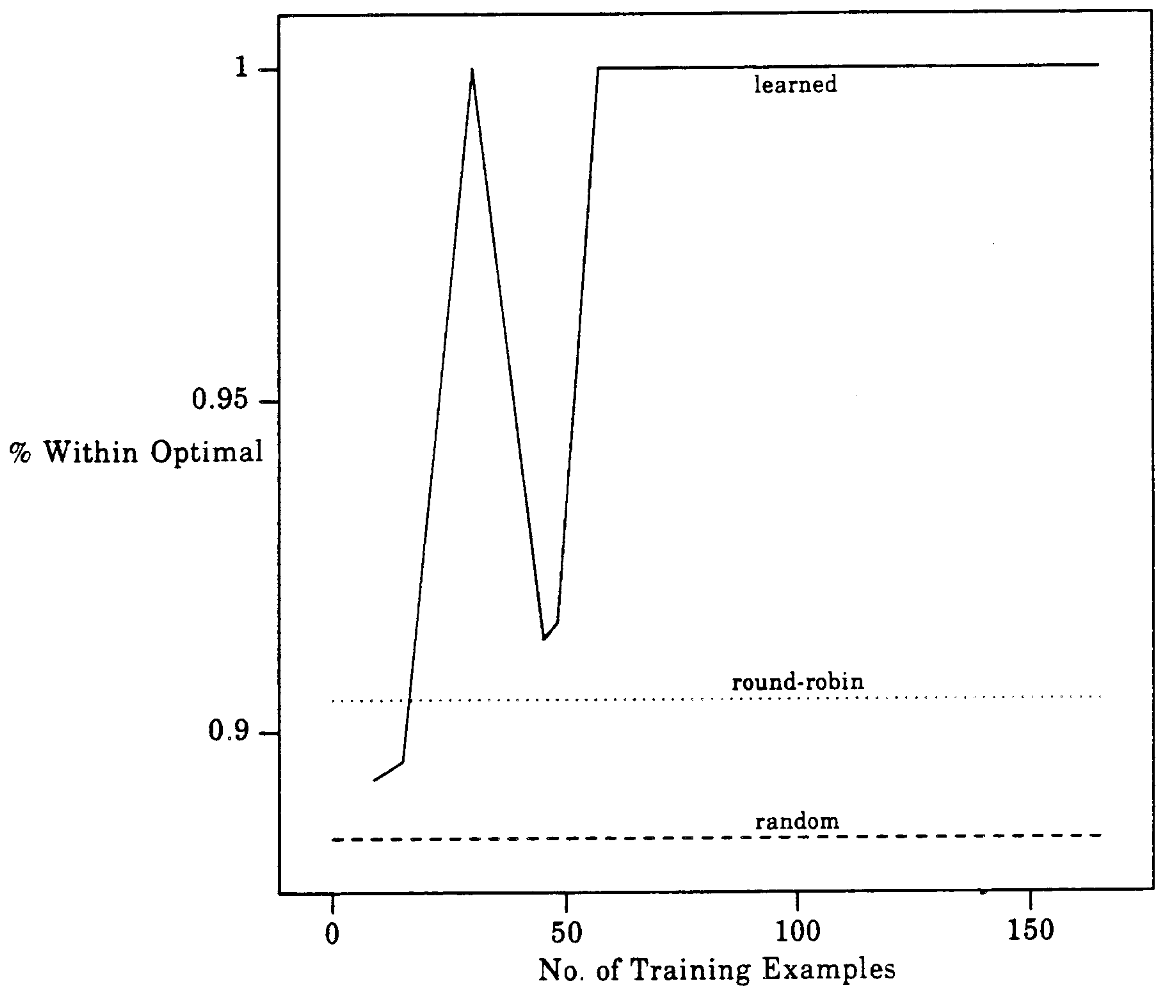


Figure 8

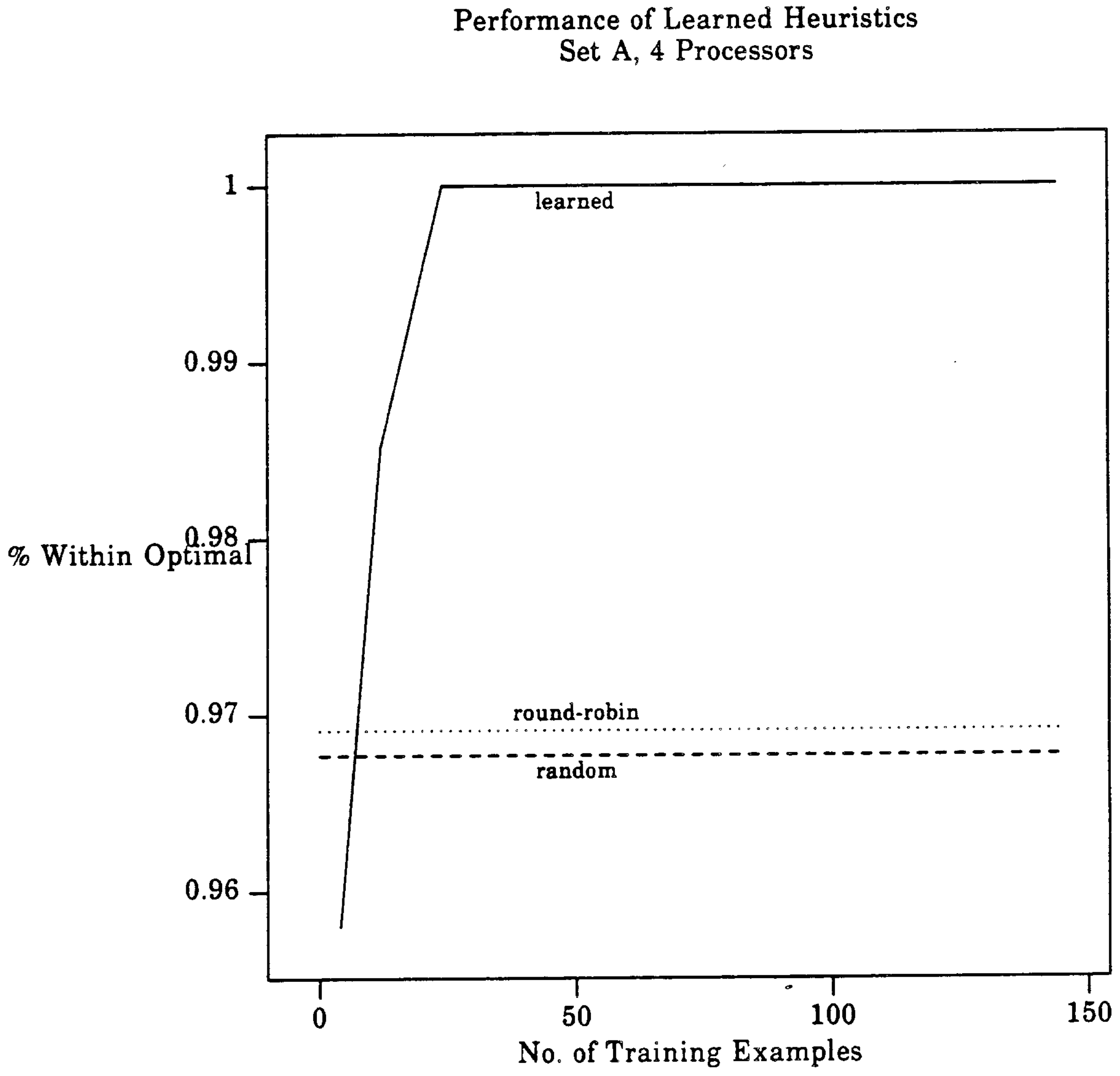

Performance of Learned Heuristics
Set A, 4 Processors
% Within Optimal
1
0.99
0.98
0.97
0.96
learned
round-robin
random
0
50
100
150
No. of Training Examples

Figure 9

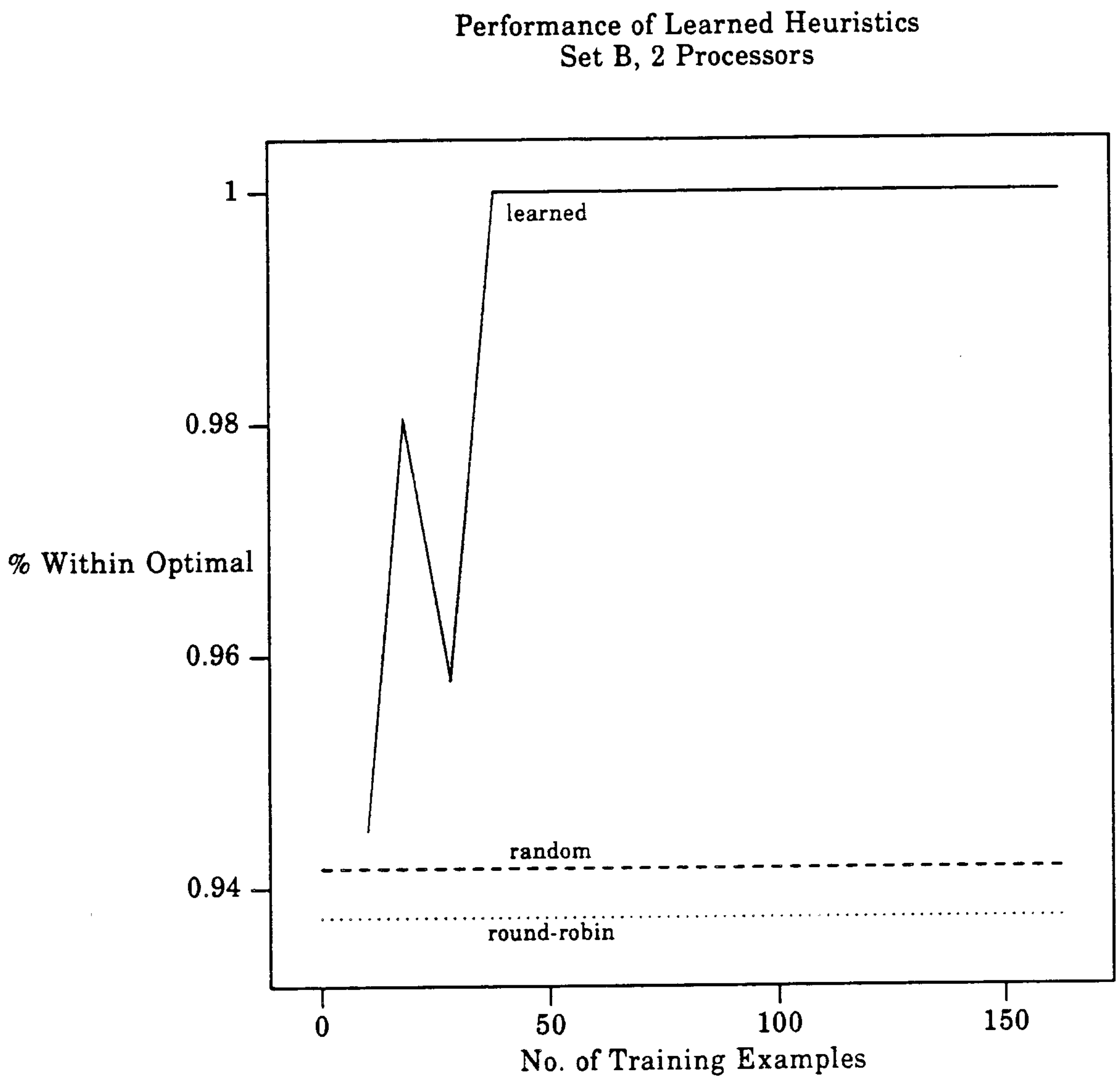

Performance of Learned Heuristics
Set B, 2 Processors
1
0.98
0.96
0.94
% Within Optimal
learned
random
round-robin
0
50
100
150
No. of Training Examples

Figure 10

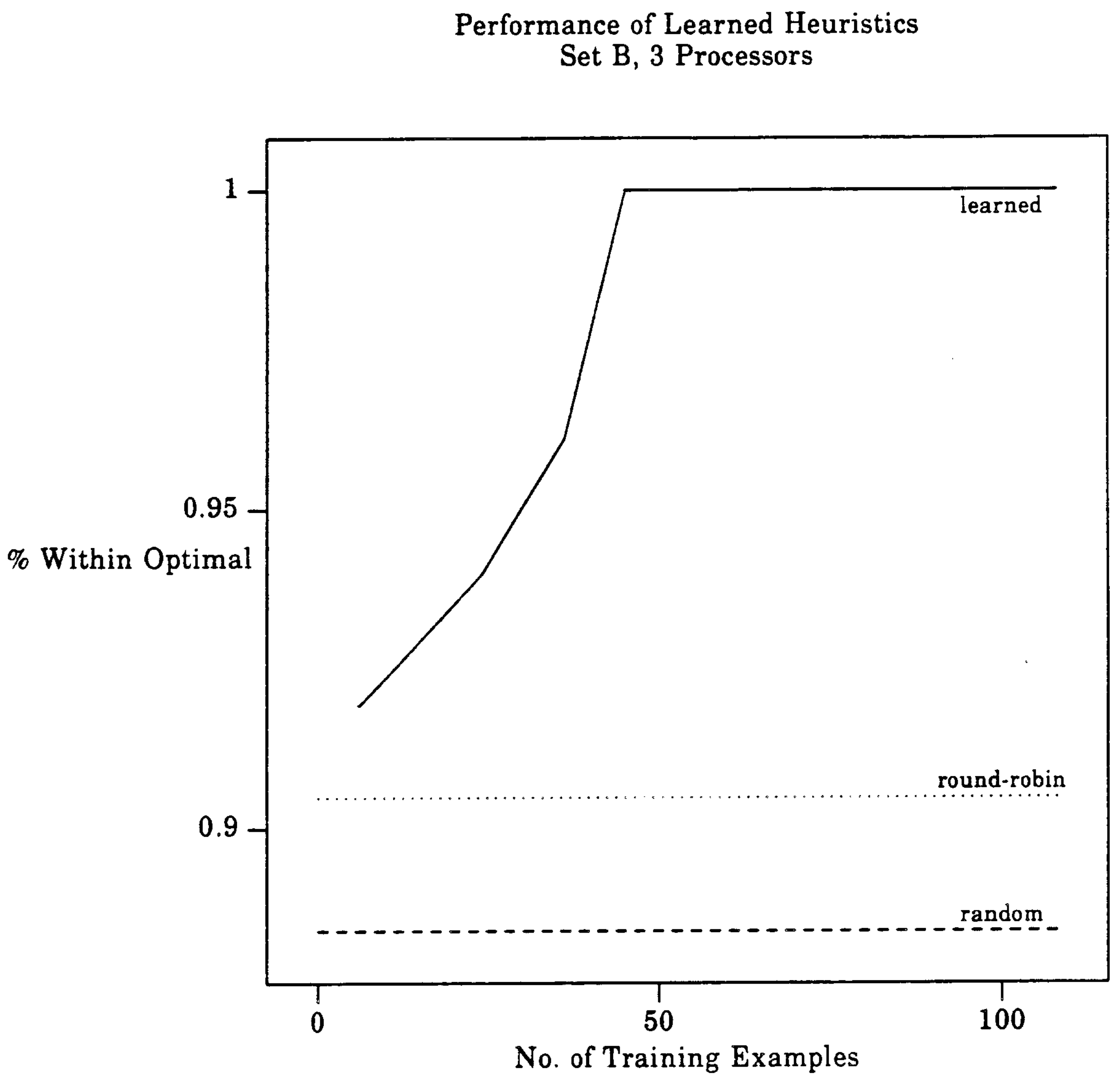

Performance of Learned Heuristics
Set B, 3 Processors
1
0.95
0.9
% Within Optimal
learned
round-robin
random
0
50
100
No. of Training Examples

Figure 11

Performance of Learned Heuristics
Set B, 4 Processors

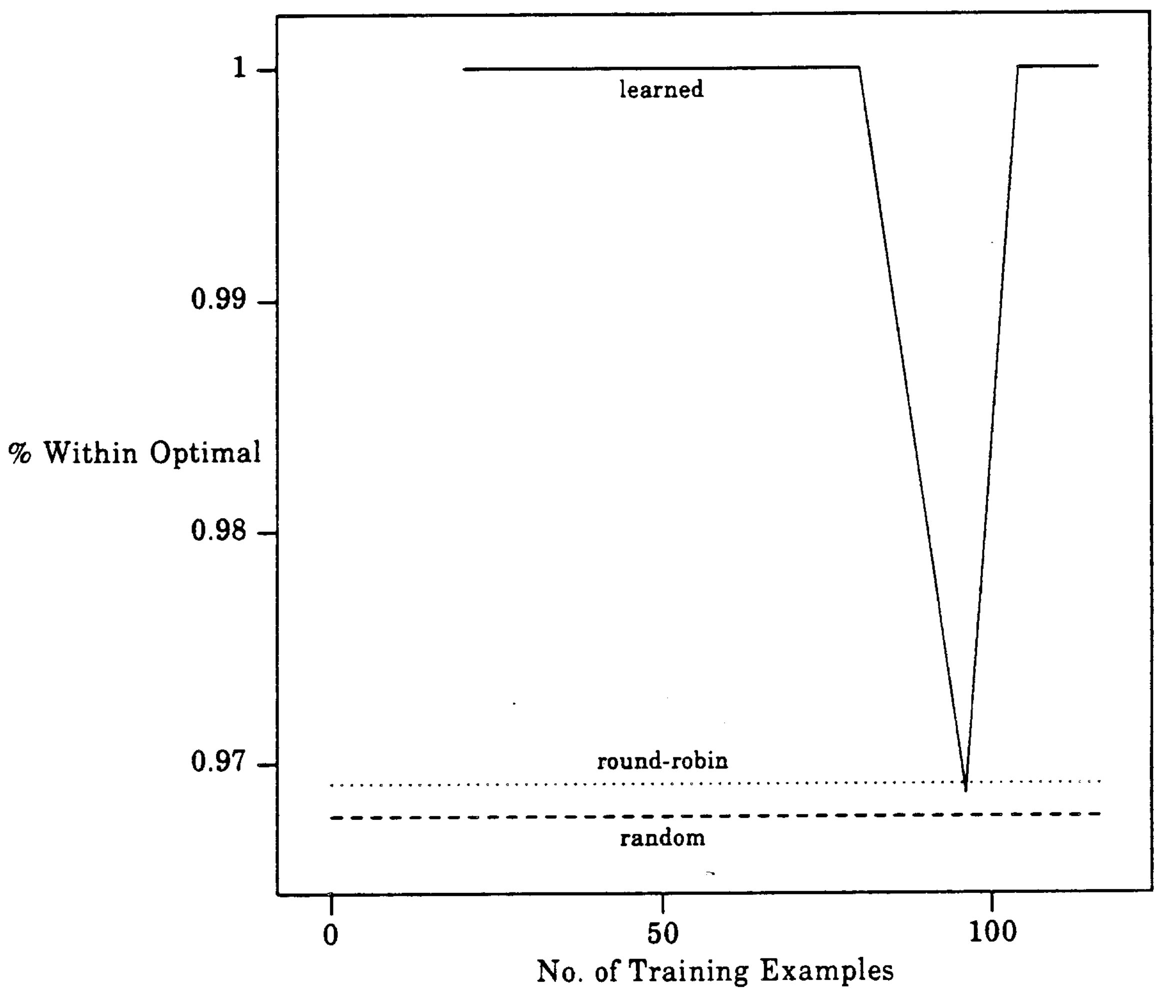


Figure 12

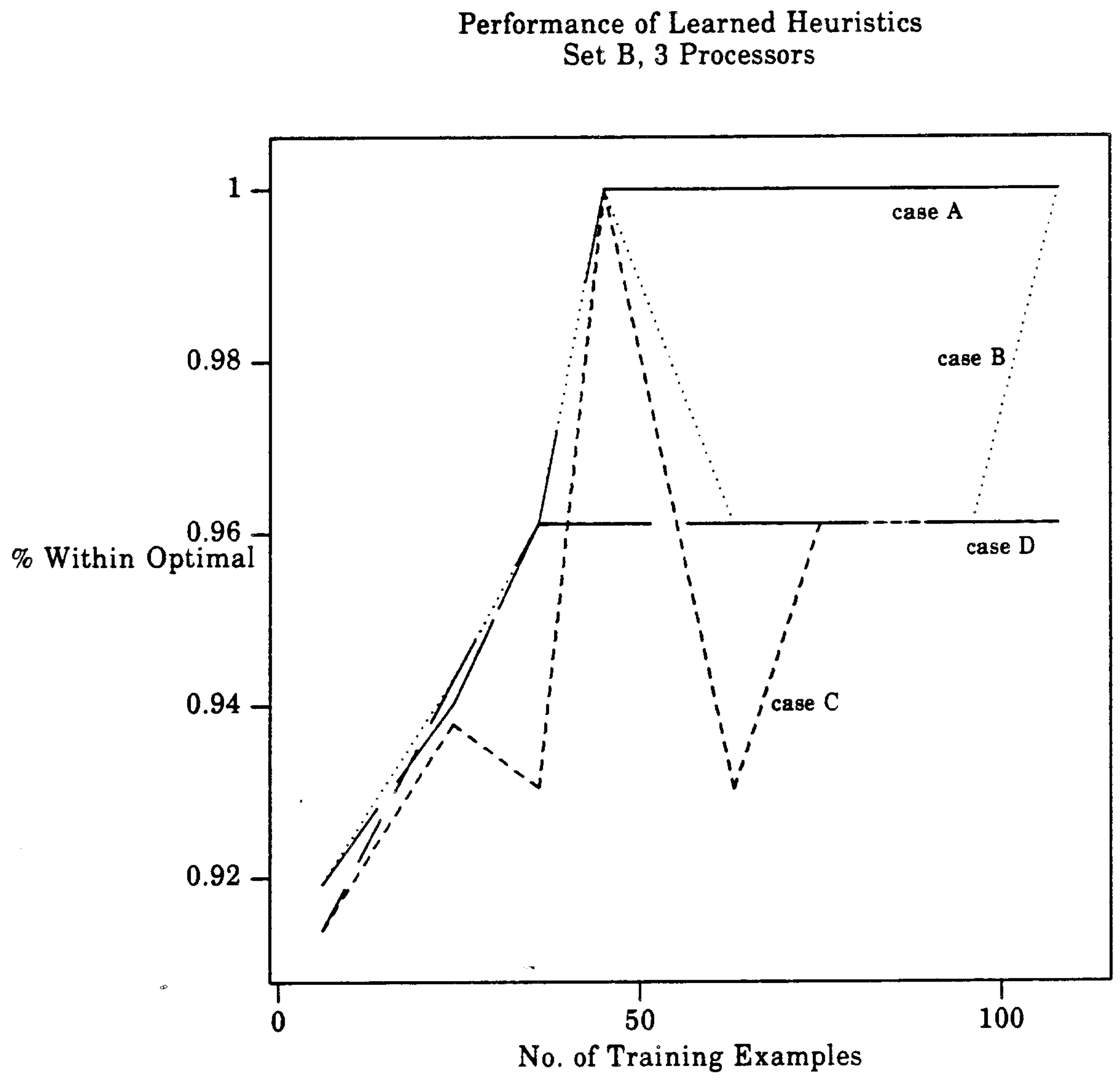

Performance of Learned Heuristics
Set B, 3 Processors
1
0.98
0.96
0.94
0.92
% Within Optimal
case A
case B
case C
case D
0
50
100
No. of Training Examples

Figure 13

7. CONCLUSIONS

Much work remains to be done in this area before any definite conclusions can be made. Machine learning has been successfully employed in other areas, but not in the field of operating systems. The results presented here do not rule out its use in processor scheduling. Certainly, the theoretical possibilities are attractive. Consider a system that could detect bottlenecks and make corrections for them automatically. Currently, this requires the use of performance monitoring tools followed by human analysis.

Making efficient usage of CPU resources has always been an important operating system design consideration. As applications attempt to take advantage of the parallelism possible in multiple processor environments, the problem of processor scheduling will only increase in complexity. This complexity may best be handled by an Intelligent Agent. Of course, we must depend on the development of a suitable technology that can support such a system.

ACKNOWLEDGMENTS

I would like to thank Professors Ferrari and Ashany for their guidance in the preparation of this paper.

REFERENCES

[ADAM74] Adam T. L., Chandy K. M. and Dickson J. R. "A Comparison of List Schedules for Parallel Processing Systems," Communications of the ACM, December 1974.

[ALLA81] Allan J. F. "Maintaining Knowledge about Temporal Intervals," Technical Report TR86, University of Rochester, Department of Computer Science, 1981.

[ANDR85] Andriole S. J. "Applications in Artificial Intelligence," Petrocelli Books, 1985.

[BUNT76] Bunt R. B. "Scheduling Techniques for Operating Systems," IEEE Computer, October 1976.

[COFF72] Coffman E. G. and Graham R. L. "Optimal Scheduling for Two-processor Systems," Acta Informatica, 1972.

[COFF73] Coffman E. G. and Denning P. J. "Operating Systems Theory," Prentice Hall, 1973.

[DIET83] Dietterich T. G. and Michalski R. S. "Discovering Patterns in Sequences of Objects," Proceedings of the International Machine Learning Workshop, June 1983.

[FORS84] Forsyth R. "Expert Systems: Principles and Case Studies," Chapman and Hall, 1984.

[FORS86] Forsyth R. and Rada R. "Machine Learning: Applications in expert systems and information retrieval," Ellis Horwood Ltd., 1986.

[GENE87] Genesereth M. R. and Nilsson N. J. "Logical Foundations of Artificial Intelligence," Morgan Kaufmann, 1987.

[GONZ77] Gonzalez M. J. "Deterministic Processor Scheduling," Computing Surveys, September 1977.

[HABE76] Habermann A. N. "Introduction to Operating System Design," Science Research Associates, 1976.

[LO87] Lo S. and Gligor V. D. "A Comparative Analysis of Multiprocessor Scheduling Algorithms," 7th International Conference on Distributed Computing Systems, 1987.

[MCDE80] McDermott D. and Doyle J. "Non-Monotonic Logic I," Artificial Intelligence, 1980.

[MCDE82] McDermott D. "A Temporal Logic for Reasoning about Processes and Plans," Cognitive Science, 1982.

[MICH83] Michalski R. S., Carbonell J. G. and Mitchell T. M. "Machine Learning: an Artificial Intelligence Approach," Volume 1, Morgan Kaufmann Publishers Inc., 1983.

[MICH86] Michalski R. S., Carbonell J. G. and Mitchell T. M. "Machine Learning: an Artificial Intelligence Approach," Volume 2, Morgan Kaufmann Publishers Inc., 1986.

[OUST80] Ousterhout J. K., Scelza D. A. and Sindhu P. S. "Medusa: An Experiment in Distributed Operating System Structure," Communications of the ACM, February 1980.

[OUST82] Ousterhout J. K. "Scheduling Techniques for Concurrent Systems," Proceedings of the 3rd International Conference on Distributed Computing Systems, 1982.

[PASQ87] Pasquale J. "Using Expert Systems to Manage Distributed Computer Systems," Report No. UCB/CSD 87/334, January 1987.

[PASQ88] Pasquale J. "Intelligent Decentralized Control in Large, Distributed Computer Systems," PhD dissertation, University of California, Berkeley, 1988.

[PETE83] Peterson J. L. and Silberschatz A. "Operating System Concepts," Addison Wesley, 1983.

[RUSS85] Russell S. "The Compleat Guide to MRS," Report No. KSL-85-12, Stanford University, June 1985. Addison Wesley, 1983

[SHOR75] Shortliffe E. H. and Buchanan B. G. "A Model of Inexact Reasoning in Medicine," Mathematical Biosciences, 1975.

[ZADE83] Zadeh L. A. "The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems," Fuzzy Sets and Systems, 1983.