Optimization of Parallel Search Using Machine Learning and Uncertainty Reasoning

Piotr J. Gmytrasiewicz, Chiu-Che Tseng, and Diane J. Cook

University of Texas at Arlington
Box 19015, Arlington, TX 76019
{piotr,tseng,cook}@cse.uta.edu
Office: (817) 272-3785
Fax: (817) 272-3784

Abstract

Artificial intelligence techniques often rely on heuristic search through large spaces. Because the computational effort required to search these spaces limits the scalability of the techniques, a number of parallel and distributed approaches to search have been introduced. However, theoretical and experimental results have shown that the effectiveness of parallel search algorithms can vary greatly from one search problem to another.

In this paper we investigate the use of machine learning techniques to automatically choose the parallel search techniques that maximize the resulting speedup. The approach described here is implemented in the EUREKA system, an architecture that includes diverse approaches to parallel search. When a new search task is input to the system, EUREKA gathers information about the search space and automatically selects the appropriate search strategy. We compare the effectiveness of a decision tree learner and a Bayesian network to model the influence of problem features on speedup and select strategies that will yield the best performance. We present preliminary results on search problems drawn from the Fifteen Puzzle domain.

1 Introduction

Because of the dependence AI techniques demonstrate upon heuristic search algorithms, researchers continually seek more efficient methods of searching through the large spaces created by these algorithms. Advances in parallel and distributed computing offer potentially large increases in performance to such compute-intensive tasks. In response, a number of approaches to parallel AI have been developed that make use of MIMD and SIMD hardware to improve various aspects of search algorithms [Kumar and Rao, 1990; Mahapatra and Dutt, 1995; Mahanti and Daniels, 1993; Powley and Korf, 1991]. While existing approaches to parallel search have many contributions to offer, comparing these approaches and determining the best use of each contribution is difficult because of the diverse search algorithms, machines, and applications reported in the literature.

In response to this problem, we have developed the EUREKA parallel search engine that combines many of these approaches to parallel heuristic search. EUREKA is a parallel IDA* search architecture that merges multiple approaches to task distribution, load balancing, and tree ordering, and can be run on a MIMD parallel processor, a distributed network of workstations, or a single machine with multithreading.

Our goal is to create a system that automatically selects an optimal parallel search strategy for a given problem space and hardware architecture. Unfortunately, many pertinent features of the problem space and architecture are unknown and can only be estimated. In this paper, we compare the results of using a decision tree and a belief network learning algorithm to automatically select parallel search strategies based on features of the particular search problem and hardware platform. Comparisons are based on search problems selected from the Fifteen Puzzle domain.

2 Related Work

Applying machine learning to customize and optimize software applications has generated interest among researchers [Norvig and Cohn, 1997], and implementations are starting to emerge. As an example, Minton [Minton, 1996] uses learning algorithms to automatically synthesize problem-specific versions of constraint-satisfaction algorithms. Research in other areas of computer science has yielded similar ideas of customizable environments applied to computer networks [Samrat Bhattacharjee and Zegura, 1997; Steenkiste *et al.*, 1997] and to interactive human-computer interfaces [Frank *et al.*, 1995; Lieberman, 1998].

Some research has focused on run-time tuning of software to performance of parallel systems. Much of this work focuses on load balancing algorithms, in which the current load of processors is periodically monitored in order to maintain an even work load. For example, Zhou's Centex algorithm collects load information every UP time units to determine when jobs should be migrated, and where they should be placed [Zhou, 1988]. Xu and Hwang [Xu and Hwang, 1993] not only collect run-time information to transfer tasks, but also modify the load threshold and update time based on current performance. However, even these researchers note that the sensitivity of load balancing algorithms to the parameter values suggests "that some form of adaptive load balancing may be able to provide good performance when system load changes widely" [Zhou, 1988]. More recently, Taylor et al.'s Hamlet system used learned rules to control thread creation in general multithreaded software applications [Taylor *et al.*, 1998]. The work described here is unique in allowing both problem-specific and architecture-specific features to influence the choice of strategies and in applying adaptive software techniques to parallel search.

3 Parallel Search Approaches

A number of researchers have explored methods for improving the efficiency of search using parallel hardware. We will review existing methods for task distribution, for balancing work between processors, and for changing the left-to-right order of the search tree.

3.1 Task Distribution

A search algorithm implemented on a parallel system requires a balanced division of work between contributing processors to reduce idle time and minimize wasted effort. One method of dividing up the work in IDA* search is with a parallel window search (PWS), introduced by Powley and Korf [Powley and Korf, 1991]. Using PWS, each processor is given a copy of the entire search tree and a unique cost threshold. The processors search the same tree to different thresholds simultaneously. When an optimal solution is desired, processors that find a goal node must remain idle until all processors with lower cost thresholds have completed their current iteration.

One advantage of parallel window search is that the redundant search inherent in IDA* is performed concurrently instead of serially. A second advantage of parallel window search is the improved time in finding a non-optimal solution. Processors that are searching beyond the optimal threshold may find a solution down the first branch they explore, and can return that solution long before other processors finish their iteration. This may result in superlinear speedup because the serial algorithm does not

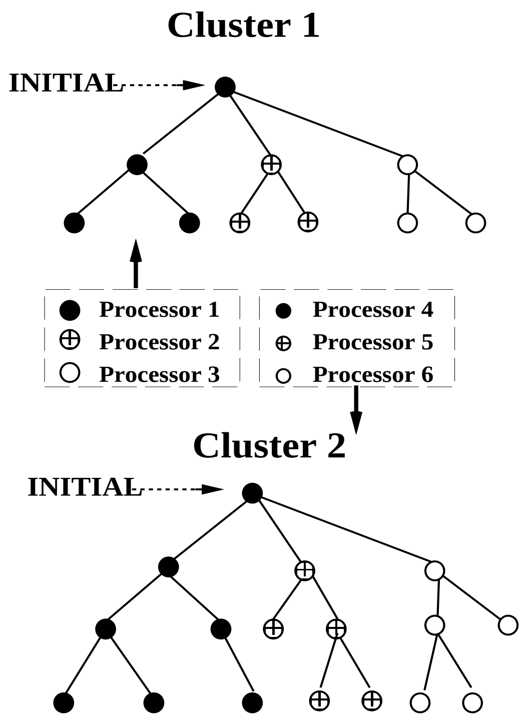

Figure 1: Space searched by two clusters, each with 3 processors

look beyond the current threshold. On the other hand, parallel window search can face a decline in efficiency when the number of processors is significantly greater than the number of iterations required to find a solution, in which case many processors are performing wasted search beyond the optimal solution threshold (depth overshoot).

An alternative parallel search approach relies on distributing the tree among the processors [Kumar and Rao, 1990]. With this approach, the root node of the search space is given to the first processor and other processors are assigned subtrees of that root node as they request work. As an alternative, the distributed tree search algorithm (DTS) employs breadth-first expansion until the host processor can distribute unique subtrees to each processor. After all processors have finished a single iteration, they begin a new search pass through the same set of subtrees using a larger threshold.

One advantage of this distribution scheme is that no depth overshoot is present. It is possible, however, for DTS to perform wasted work at the goal depth (horizontal overshoot). Another disadvantage of this approach is the fact that processors are often idle waiting for other processor to finish a current iteration. The efficiency of this approach can be improved by periodically balancing the load between processors.

A compromise between these approaches is to divide the set of processors into *clusters* [Cook and Varnell, 1997]. Each cluster is given a unique cost threshold, and the search space is divided between processors within each cluster, as shown in Figure 1.

3.2 Load Balancing

When a problem is broken into disjoint subtasks the workload will likely vary among processors. Because one processor may run out of work before others, load balancing is used to activate the idle processor. The first phase of load balancing involves selecting a processor from which to request work. One example is the nearest neighbor approach [Mahapatra and Dutt, 1995]. Alternative approaches include selecting random processors or allowing a master processor to keep track of the load in the other processors and to send the ID of a heavily loaded processor to one that is idle. During the second phase of load balancing, the requested processor decides which work, if any, to give. The choice of load balancing technique depends on such factors as the amount of imbalance in the size of the search subspaces, the number of processors, and the communication latency between processors.

3.3 Tree Ordering

Problem solutions can exist anywhere in the search space. Using IDA* search, the children are expanded in a depth-first manner from left to right, bounded in depth by the cost threshold. If the solution lies on the right side of the tree, a far greater number of nodes must be expanded than if the solution lies on the left side of the tree. If information can be found to re-order the operators in the tree from one search iteration to the next, the performance of IDA* can be greatly improved.

Powley and Korf suggest two methods of ordering the search space [Powley and Korf, 1991]. First, children of each node can be ordered and expanded by increasing heuristic distance to a goal node. Alternatively, the search algorithm can expand the tree a few levels and sort the *frontier set* (the set of nodes at that level in the tree) by increasing h value. Search begins each iteration from the frontier set and this frontier set is updated each iteration. In both of these cases, although the nodes may have the same f value, nodes with smaller h values generally reflect a more accurate estimated distance and are preferred.

Instead of ordering individual nodes, Cook and Hall [Cook *et al.*, 1993] order the set of operators to guide the next IDA* iteration to the *most promising node.* The most promising node is the node from the cut-off set (a child node not expanded in the previous iteration) with the smallest h value. As an example, Figure 2 shows a search tree expanded using one iteration of IDA*. The path to the most promising leaf node (indicated with a star) is 1 3 3 2 0. The new operator ordering is computed using the

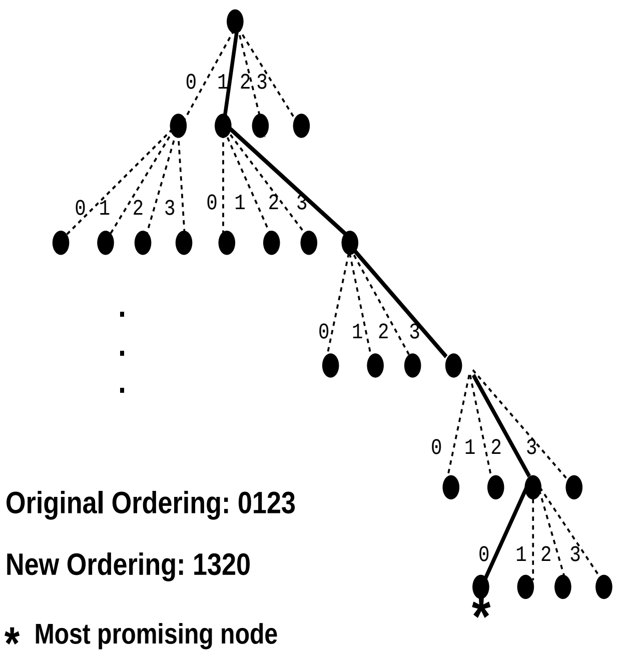

Figure 2: Operator ordering example

order of operators as they appear in this path after removing duplicates. Operators not appearing in the path are added to the end of operator list, retaining their original relative ordering. Thus the ordering of operators for the example in Figure 2 changes from 0, 1, 2, 3 (always try operator 0 first, operator 1 next, operator 2 next, and operator 3 last) to 1, 3, 2, 0.

4 The EUREKA System

The EUREKA system merges together many of the strategies discussed in the previous section. Parameters can be set that control the task distribution strategy, the load balancing strategies, and the ordering techniques. In particular, the strategies that can be selected include:

- Distribution strategy [Kumar and Rao, distributed tree search]
- Number of clusters [1..#processors]
- Fraction of Open list to donate upon neighbor request [0..1]
- Anticipatory load balancing trigger [0. . .SizeOf_OpenList]
- Load balancing [Off, Neighbor, Random]
- Ordering [None, Local, Operator Ordering]

To automate the selection of parallel search strategies, EUREKA can make use of gathered information that describes the search space and the underlying hardware. To characterize the search space, the system searches a sampling of the space (roughly 200,000 nodes, depending on average node expansion time) and calculates the features listed below. The initial sampling of the space only requires a few second and does not significantly affect the total search run time.

Branching Factor (*b*)**:** The average branching factor of the search tree.

Heuristic Error (*herror*)**:** The difference, on average, between the estimated distance to a goal node and the true distance to the closest goal node.

Heuristic Root Estimate (*hroot*)**:** The estimated distance from the root to the nearest goal node.

Imbalance (*imb*)**:** The degree to which nodes are unevenly distributed among subtrees in the space.

Goal Location (*loc*)**:** The left-to-right position of the first discovered optimal solution node.

Heuristic Branching Factor (*hbf*)**:** The ratio of nodes expanded between the current and previous IDA* iterations.

In addition to problem space features, information describing the hardware is also used including the number of processors and average communication latency. Initially, we used C4.5 [?] to induce a decision tree from pre-classified training examples. Training examples represent runs of sample problem spaces with varying search strategies, and the correct "classification" of each training example represents the search strategy yielding the greatest speedup.

For each new problem, EUREKA performs a shallow search through the space to collect features describing the new problem space and architecture. If a goal is not found during the shallow search, the features of the tree are calculated at this point and used to index appropriate rules from the C4.5 database. EUREKA then initiates a parallel search from the root of the space, employing the selected strategies.

The selected features each demonstrate a significant influence on the optimal search strategy. Although feature values can change dramatically from one problem to the next, each feature remains fairly stable between levels of the same tree. As a result, computing the values of these features at a low level in the tree provides a good indication of the structure of the entire tree.

Two sources of uncertainty arise with this approach to the control of parallel search strategy selection, and these factors can prevent traditional machine learning techniques from performing well. First, the search space features are not known a priori. Instead, they are estimated given a small sample of the overall search space. Similarly, communication latency is estimated based on average usage of the machine. These features may not accurately reflect the true nature of the problem or architecture.

Stat	Small	Medium	Large
Avg Coef Var	1.32	8.82	9.95
Avg Speedup	7.23	52.40	54.09

Table 1: Average Speedup Standard Deviation

The second problem is that while traditional machine learning systems require a deterministic classification of each training example, there does not always exist a clear strategy winner for each training case. On some problem instances one strategy dramatically outperforms the others. On other problem instances two or more strategy selections perform almost equally well. This problem is exacerbated by the fact that there is some noise inherent in the collected run times. To demonstrate the amount of error that can be present in the timings we select twelve instances of the fifteen puzzle problem (four small, four medium, and four large instances), and time five runs of each instance with identical strategy parameters on an nCUBE 2. We compute the standard deviation of the speedups for five runs of the same problem instance, and then divide the result by the sample mean to ensure the result is not affected by the magnitude of the speedup values. This coefficient of variation averaged over all problem instances in the category is listed in Table 1 along with the average speedup for the instances in the problem category. As shown, the amount of error present in the timings can be quite large, and when two strategies perform almost equally well, the winner for any given run can be almost arbitrary.

In response to this problem, a second C4.5 decision tree was constructed and tested using only a portion of the problem instances; namely, the 33 problem instances that exhibited the largest variation in run time between strategy choices (the cases in which there was a clear "winner"). This approach yields better performance but is not effective for all problem instances.

Because these sources of uncertainty exist in this application, in our second approach we choose to model the problem with a belief network. In the next section we describe the motivation behind this choice and the particular model used for this work.

5 Belief Network Model of Parallel Search

Belief networks [Pearl, 1988; Russell and Norvig, 1994] are directed acyclic graphs that compactly represent probabilistic dependencies among the variables of interest. They are well suited to reason with features describing parallel search for two main reasons. First, many features and factors describing parallel search are causally related. For example, the search tree imbalance directly influences the load distribution among the processors, and the horizontal overshoot directly influences the search overhead. These dependencies are represented in Figure 3 as directed links between the corresponding nodes. Second, as mentioned, many of the features are a priori unknown, and remain uncertain during the task distribution process, but are nevertheless stochastically determined by the values of other features. For example, the average processor idle time remains uncertain, but is related to the number of processors, which is known, and to latency and load distribution, which are uncertain.

EUREKA not only reasons about features, but also makes decisions about which parallel search strategy is expected to be the best, given the characteristics of the search problem at hand. To facilitate this decision-making we use an extension of Belief networks, called influence diagrams [Shachter, 1986]. Apart from nodes that represent uncertain variables of the problem domain, called the chance nodes, influence diagrams also have decision nodes and a utility node. The decision nodes represent the decisions that a decision-maker can execute to influence the domain. The utility node represents the preferences of the decision-maker; it is used to assign a number (utility) to states of the domain that represents the degree to which the decision-maker's preferences are satisfied by a given state of the domain. By representing the way alternative decisions influence the state of the domain, and the quantitative representation of preference over the states, the influence diagram can be used to arrive at optimal decisions.

In EUREKA, the decisions that can be executed are the distribution strategy, number of clusters, fraction of open list donated to neighbors, anticipatory load trigger, type of load balancing, and the ordering used, as we described above. These choices influence the domain in various causal ways, interacting with the known and uncertain features along the way, as depicted in Figure 3. In this set of experiments we discretized all numeric values into two ranges for one experiment (NeticaBinary) and into more than two ranges for the second experiment (NeticaMultiple). For the parallel search problem, EUREKA's utility node in Figure 3 directly depends on a single node of the domain – speedup.

Apart from the graphical representation, as in Figure 3, the parameters needed to fully specify the network are numerical representations of the probabilistic dependence relations. These are encoded as conditional probability tables, or CPTs. For example, a CPT for the node "load distribution" specifies the probabilities of this node's values for all possible values of its parent nodes, imbalance (labelled "imb") and "clusters".

A feature of Belief networks is that the conditional probabilities contained in the CPTs can be learned from a sufficiently large number of training cases. For the network we have created as well as for the C4.5 algorithm, we used three-fold cross validation to train the system on two-thirds of the data and test the results on one-third of the data, averaging the results over three random partitions. Netica currently only learns the conditional probability relationship at each node – the structure of the network was hand constructed. All of the training cases were obtained from experimental runs of parallel search for the Fifteen Puzzle on an nCUBE machine, as we describe below.

Thus, by using the influence diagram in Figure 3, with CPTs learned based on a rich sample of test runs, EUREKA can choose the selectable search parameters to have values that result in maximum expected speedup of the performed search. The results of search speedups obtained in our experiments are described below.

6 Experimental Results

We tested the ability of the influence diagram to provide a basis of making parallel search strategy decisions by comparing decisions based on predicted speedup values from the influence diagram with decisions based by the C4.5 learning system. In this experiment, we used the Netica belief network system to model the network and learn the CPTs. For our preliminary results, we have modeled all nodes in the diagram as chance nodes, but we will refine this model in later experiments.

To create test cases, we ran each problem instance multiple times on 32 processors of an nCUBE, once using each parallel search strategy in isolation. The measured features of the search space, the underlying architecture, and the resulting speedup are stored in a file and fed as input to the belief network. The belief network learns conditional probability tables reflective of the data in the training set.

In this paper we show the results of automatically selecting one parallel search parameter, the number of clusters. We compared the results of Netica-selected strategies on test data to C4.5-selected strategies and to each strategy used exclusively for 100 Fifteen Puzzle problem instances. Continuous-valued variables were discretized into fixed-width ranges before input to Netica. In one experiment two values for each continuous variable were allowed, and in another experiment multiple values were created, the number depending on the variance of values observed for the variable. Speedup results for various strategy decisions averaged over all problem instances are shown in Table 2 below.

From the results of this experiment, the belief network did outperform all of the fixed strategies as well as C4.5 using all 100 problem instances. C4.5Fil yielded the best results, but was only trained and tested on cases with clear winners. As the numbers

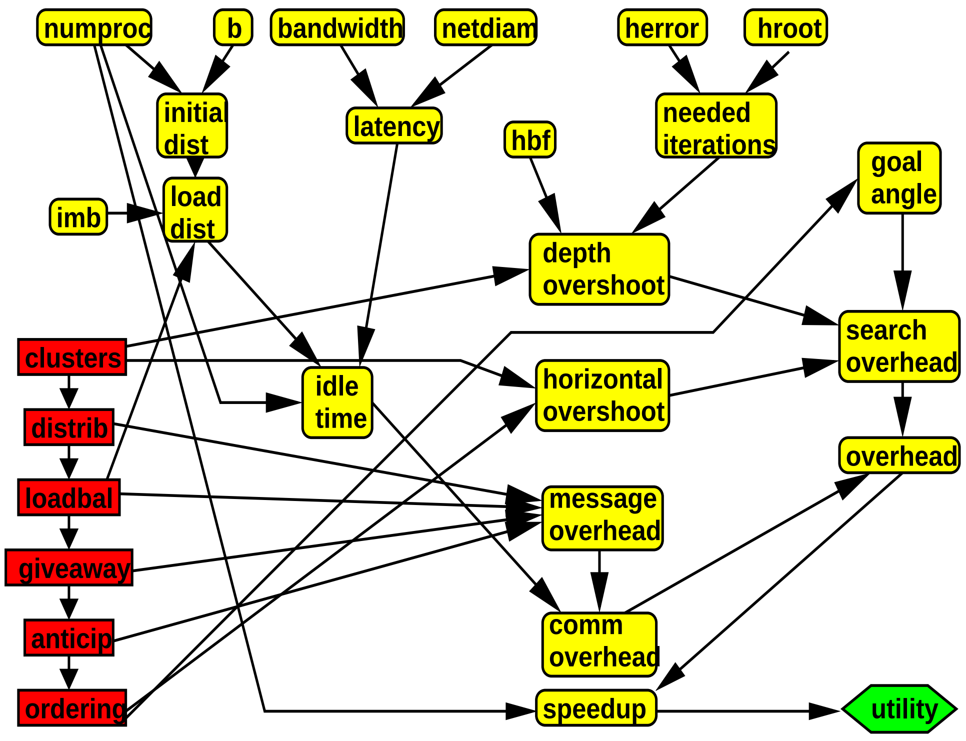

Figure 3: A model of the factors that influence speedup of parallel search algorithms.

Approach	Speedup
1 Cluster	55.30
2 Clusters	60.98
4 Clusters	58.79
C4.5	**57.14**
C4.5Fil	**86.76**
NeticaBinary	**64.93**
NeticaMultiple	**68.66**

Table 2: Clustering Speedup Results

indicate, superlinear speedup was often achieved. This occurs primarily when a goal node is found on the right side of the tree. The serial algorithm employing IDA* search would exhaustively search the left subtree before examining any nodes in the right subtrees, but processors in the parallel algorithm may start the search in the right subtrees. Because search is finished as soon as a goal node is found, the parallel algorithm does not perfectly mimic the serial algorithm and thus often yields greater than linear speedup. All of the machine learning approaches could benefit from additional training examples, as this problem represents a very large hypothesis space.

7 Conclusions and Future Work

This project reports on work performed to combine the benefits of parallel search approaches in the EUREKA system. Experimentation reveals that strategies developed over the last few years offer distinct benefits to improving the performance of AI applications, and strategies need to be chosen based on the characteristics of a particular problem. Results indicate that machine learning techniques can be used to perform automatic selection of parallel search strategies. Because uncertainty exists in the characterization of a search problem, the underlying architecture, and the classification of training examples, a mechanism such as a belief network may also be useful in making optimal decisions.

The results shown in this paper are preliminary. We are continuing to investigate methods of refining the decisions make by EUREKA. In addition, the EUREKA system will benefit from incorporation of additional search strategies, problem domain test cases and architectures. We hope to demonstrate that uncertainty reasoning techniques can be used to effectively control strategy selection in parallel search techniques and in many other areas of AI.

8 Acknowledgements

This work was supported by National Science Foundation grants IRI-9502260 and DMI-9413923.

References

[Cook and Varnell, 1997] D J Cook and R C Varnell. Maximizing the benefits of parallel search using machine learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 559–564. AAAI Press, 1997.

[Cook *et al.*, 1993] Diane J Cook, Larry Hall, and Willard Thomas. Parallel search using transformation-ordering iterative-deepening A*. *International Journal of Intelligent Systems*, 8(8):855–873, 1993.

[Frank *et al.*, 1995] Martin Frank, Piyawadee Sukavirija, and James D Foley. Inference bear: designing interactive interfaces through before and after snapshots. In *Proceedings of the ACM Symposium on Designing Interactive Systems*, pages 167–175. Association for Computing Machinery, 1995.

[Kumar and Rao, 1990] V Kumar and V N Rao. Scalable parallel formulations of depth-first search. In Kumar, Kanal, and Gopalakrishan, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 1–41. Springer–Verlag, 1990.

[Lieberman, 1998] Henry Lieberman. Integrating user interface agents with conventional applications. In *Proceedings of the ACM Conference on Intelligent User Interfaces*. Association for Computing Machinery, 1998.

[Mahanti and Daniels, 1993] A Mahanti and C Daniels. SIMD parallel heuristic search. *Artificial Intelligence*, 60(2):243–281, 1993.

[Mahapatra and Dutt, 1995] Nihar R. Mahapatra and Shantanu Dutt. New anticipatory load balancing strategies for parallel A* algorithms. In *Proceedings of the DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 197–232. American Mathematical Society, 1995.

[Minton, 1996] Steven Minton. Automatically configuring constraint satisfaction programs: a case study. *Constraints*, 1(1):7–43, 1996.

[Norvig and Cohn, 1997] Peter Norvig and D Cohn. Adaptive software. *PC AI Magazine*, 1997.

[Pearl, 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufman, 1988.

[Powley and Korf, 1991] Curt Powley and Richard E Korf. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477, 1991.

[Russell and Norvig, 1994] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1994.

[Samrat Bhattacharjee and Zegura, 1997] Ken Calvert Samrat Bhattacharjee and Ellen W Zegura. An architecture for active networking. In *High Performance Networking*, 1997.

[Shachter, 1986] R. D. Shachter. Evaluating influence diagrams. *Operations Research*, 34:871–882, 1986.

[Steenkiste *et al.*, 1997] Peter Steenkiste, Allan Fisher, and Hui Zhang. Darwin: resource management for application-aware networks. Technical Report CMU-CS-97-195, Carnegie Mellon University, 1997.

[Taylor *et al.*, 1998] Scott Taylor, David Levine, Krishna Kavi, and D. J. Cook. A comparison of multithreading implementations*. In *Yale Multithreaded Programming Workshop*, 1998.

[Xu and Hwang, 1993] Jian Xu and Kai Hwang. Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Parallel and Distributed Computing*, 18(1):1–13, 1993.

[Zhou, 1988] Songnian Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.