Integrating Machine Learning in Parallel Heuristic Search

R. Craig Varnell
Stephen F. Austin State University
varnell@cs.sfasu.edu

Diane J. Cook
University of Texas at Arlington
cook@cse.uta.edu

Abstract

Many artificial intelligence applications rely on searching through large, complex spaces. Iterative Deepening-A* (IDA*) is a procedure capable of finding a least cost path to a goal; however, the execution time on a single processor is often too long for most applications.

Parallel processing can be used to reduce the complexity and run time of a search problem by dividing the work load over multiple processors. A number of techniques have been introduced that improve the performance of heuristic search. Each approach has a number of parameters that are often selected by the user before the problem is executed. It is common to find that the selection of parameters for one approach will conflict with parameters for another approach and poor performance will result.

This research uses a knowledge base constructed by the C4.5 machine learning system to continuously select approaches and the associated parameters throughout the execution of a problem. Therefore, as the search moves deeper into the tree structure, the approach will be modified to adapt to the tree structure. A series of experiments has been performed with the fifteen puzzle problem domain. A significant improvement in performance has resulted from using machine learning to select an appropriate approach for a problem.

Introduction

Many artificial intelligence applications depend on a search algorithm that is required to locate a goal of some form in a very large space. A number of heuristic-based techniques have been incorporated into the brute-force search algorithms that reduce the time to locate a goal. Serial processing of such algorithms results in long execution times for problems that commonly need real-time responses. This section will examine both serial and parallel approaches to heuristic search.

Search Spaces and Tree Structures

The typical search problem consists of three basic components:

- An initial state
- A goal description
- A set of operators

The *initial state* represents the starting point of the search problem. A search problem is often represented as a tree structure, and the initial state is the root of the tree. The *goal state* represents what is to be accomplished or what the final product should be. For each state, the operators are applied producing a new state. The role of the search algorithm is to transform the initial state into a goal state by applying the correct sequence of operators. From any given configuration, or state, in the search space a set of operators is applied to the state producing a new set of states. This creates a parent/child relationship in that each of the new states is generated from its parent state. One way that a search problem is commonly represented is as a graph structure. The states of the search space are the vertices of the graph, and the operators applied to each state serve as the edges of the graph.

The graph structure does not represent the search tree in the most concise form since there are no cycles of duplicate nodes in a search problem and tree algorithms are simpler than graph algorithms. For this reason, the search problem is also commonly represented as a tree with the initial configuration serving as the root of the tree, children in the tree that are generated as the result of applying one or more operators to the parent state, and a goal located somewhere in the tree structure.

The A* Algorithm

The A* algorithm serves as the foundation for most single-agent heuristic search algorithms. A heuristic estimating function, $h(x)$, is used to return a heuristic estimate of the distance that node x is from a goal node. The value $g(x)$ is used to represent the cost associated with moving from the initial state to node x. Combining these two values into a simple formula, $f(x) = g(x) + h(x)$, creates an estimate of the solution path cost containing node x. The value of $f(x)$ represents the cost incurred from the initial state plus an estimate of the cost to reach a goal state. When

$h(x)$ becomes zero, the goal has been located and the algorithm terminates locating an optimal solution.

The problem associated with A* is that memory will quickly be exhausted, just as the case for breadth-first search. New nodes are placed on the stack and for every iteration of A* the stack grows by $b - 1$ nodes.

Iterative-Deepening A*

The depth-first iterative deepening algorithm ensures a solution is located by incrementing the cutoff value from one iteration to the next and performing a pruning operation when the depth obtained during the search exceeds the cutoff value. A similar approach is described by Korf (Korf 1985) using the A* cutoff criteria. The initial cutoff value is set to the heuristic value of the initial state and a pruning operation is performed when the total cost of a node, $f(x) = g(x) + h(x)$, exceeds the cutoff value. Instead of increasing the cutoff value by one each iteration, the cutoff is increased to the minimum f value that exceeds the previous threshold. This ensures that an optimal solution is not bypassed by increasing the cutoff value too much and not performing unnecessary work by increasing the cutoff by too little. Iterations of IDA* are performed until a solution is located that represents an optimal path from the initial state to the goal.

The complexity of IDA* is the same as A* in that the same number of nodes are generated in locating a goal. The advantage over A* is that a reduced amount of memory is needed to perform a problem. A significant amount of wasted work is performed with the IDA* algorithm; however, the greatest number of nodes are generated during the final iteration and therefore the previous iterations represent a small percentage of the total execution time for locating a goal node.

Parallel Heuristic Search

Advances in parallel and distributed computing offer potential improvements in performance to such compute-intensive tasks. As a result, a number of approaches to parallel AI have been developed that make use of parallel processing hardware to improve various search algorithms. While existing applications in 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, architectures, and applications. The use of a wrong approach can produce parallel execution time greater than that of a serial version of the algorithm; whereas another approach can produce near-linear speedup.

We addressed this problem in previous research by designing the Eureka parallel search architecture(Varnell 1997) which utilizes a machine learning system to automate the process of selecting an approach and associated parameters for solving a problem. Eureka solves a problem in two phases: first, a shallow search is performed on a single processor where the characteristics of a search tree are gathered. Second, the statistics are

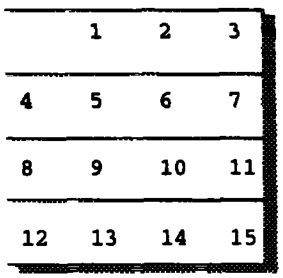

Figure 1: Fifteen Puzzle Problem

used as input into the C4.5 machine learning system and the approaches and associated parameters are automatically determined. The knowledge base is developed using problems across various domains and problem structures. This paper shows additional research performed on the Eureka architecture where the shallow search is no longer required for measuring the characteristics of the search tree. This step is performed in the parallel search phase.

Fifteen Puzzle Problem

The fifteen puzzle is constructed with a set of fifteen movable tiles in a four-by-four frame with one blank spot. The tiles which are horizontally or vertically adjacent to the blank spot may be slid into the blank spot. The object of the puzzle is to find a sequence of tile movements that will transform the initial tile arrangement into a specified goal tile arrangement. An example of the fifteen puzzle problem is shown in figure 1. A set of 100 instances of the fifteen puzzle problem were developed by Korf (Powley & Korf 1991), ranging from extremely easy to extremely difficult.

Parallel Search Approaches

A number of researchers have explored methods for improving the efficiency of search using parallel hardware. In this section, we will summarize these existing methods used for task distribution, balancing work among processors, and for changing the left-to-right order of the search tree.

Task Distribution

When multiple processors are used to solve a problem, the work must be divided among the processors using an efficient distribution scheme. Examples of task distribution include parallel window search (Powley & Korf 1991) and distributed tree search(Kumar & Rao 1990). A hybrid approach combines the features of parallel window search with distributed tree search dividing processors into *clusters* (Cook & Nerur 1993). Each cluster is given a unique cost threshold and the search space is divided between processors within each cluster.

Load Balancing

When a problem is broken into disjoint subtasks, the workload will likely vary among processors. Load balancing is an essential component of parallel search as one processor will become idle before others (Kumar, Ananth, & Rao 1991; Kumar & Rao 1989). The decisions to be made include *how* to perform load balancing and the *frequency* of performing load balancing(Kumar, Ananth, & Rao 1991). A number of approaches have been introduced for performing load balancing (Huang & Davis 1989; Kumar, Ananth, & Rao 1991; Kumar & Rao 1989; Mahapatra & Dutt 1995; Saletore 1990) each of which performs differently based on the problem structure. As to the frequency of load balancing, the actual load balancing operation can be performed when a processor becomes idle or on a scheduled basis.

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. With tree ordering the order of creating new children is changed in an attempt to quickly locate the goal. Some examples of tree ordering include local ordering (Powley & Korf 1991) and transformation ordering (Cook, Hall, & Thomas 1993).

The Eureka Search Architecture

We consider two approaches in designing a system that will choose the best technique for a particular problem. The first is to manually build a set of rules which can be very time consuming and not produce accurate results for all problems. For a problem such as parallel heuristic search, choosing a technique by comparing the execution time of one approach to another leads to minimal success. In most cases, performance is determined by a problem characteristic such as goal location or branching factor which are frequently not obvious when comparing only problem execution times.

A second approach, which we find to be the more effective, is to develop a rule base by using a machine learning system. For our research, we use the C4.5 machine learning system (Quinlan 1993) which creates a decision tree and rule base from the learning examples. A decision tree induction method for machine learning is chosen since it is possible to produce a set of human-readable rules.

Phases of Operation

The Eureka architecture is based on two phases of operation. The first is an information gathering operation. A set of problems are run and statistical measurements are obtained that characterize one tree structure from another. This data is used to train the machine learning system along with a classification of the approach that performed best. The second phase is the production operation where the approach used for solving a new problem is determined by the machine learning system.

Problems Addressed

The problem with this was three fold. First, the serial shallow tree expansion used time that could be better spent on the parallel search. The serial job would perform complete iterations of IDA* until a predetermined number of nodes were generated. Since search trees will vary there can be a wide variation in the time required to perform the shallow search. The second problem involves the procedure for determining the structure of a search tree. The tree characteristics are obtained from this shallow search in the form of statistical measurements which accurately reflect the nature of the search space at a low level in the tree but not necessarily for the deeper levels. Therefore, a better technique is needed for gathering the tree measurements. Third, is the method for establishing approaches and associated parameters. Once the serial job completes, the statistics are input to the knowledge base created with the C4.5 machine learning system. The approaches used for the parallel job are determined from the shallow search and remain the same for the duration of the parallel job.

To address these problems, the Eureka architecture was modified to use the parallel search iterations to dynamically reconfigure the strategy choices. The initial expansions on a serial processor were therefore not required. Statistical measurements are obtained from the parallel search job throughout the execution of a problem. Also the tree statistics are gathered from each iteration of IDA* and the approaches and parameters are reestablished based on the new information received on the tree structure. From one iteration to the next, the number of nodes expanded increases by moving deeper in the search space and, as a result, the measurable statistics more accurately represent the tree structure.

The actual steps for developing a rule base are as follows:

1. Timings are captured from a variety of problem domains and search techniques. Multiple domains are used since certain factors, such as the branching factor, tend to be problem specific. Each problem is tagged with the strategy which performs best. For example, if we are examining strategies to load balancing, a number of different methods exist for performing load balancing. A problem is classified with the strategy that performed best for load balancing and these will serve as the *classes* for the machine learning system. Statistics are also gathered that can correctly classify one problem structure from another. These will serve as the *problem attributes* in the machine learning system.
2. The attributes are combined with the corresponding classes and input as training examples to a machine learning system. A knowledge base consisting of a set of if/then rules is produced.
3. A new problem makes use of the knowledge base to pick a parallel search strategy for a given parallel architecture. The measurable statistics are constantly

gathered on the search problem and provided as input to the learning system. The knowledge based generated during the training sessions are accessed and a method that is appropriate for the problem structure is used.

4. As the new problem is executing, statistics are being gathered that characterize the problem and, as the search moves deeper into the search space, these statistics better represent the true structure. Therefore, the strategy taken for a particular problem will continue to change throughout the execution of the problem. The statistics are supplied to the machine learning system which decides the best approach to take.

Measurable Characteristics

In the previous section, we indicated that statistics were gathered in order to characterize one problem from another. This section describes these factors that characterize a search tree.

1. *Branching factor*

   The branching factor reflects the number of children for each node averaged across the entire search space. The branching factor contributes to the complexity of the IDA* algorithm.

2. *Heuristic Branching Factor*

   The heuristic branching factor is defined as the ratio of the number of nodes with a given value of f with the next smaller value of f (Powley & Korf 1991). This gives an indication as to the number of node expansions which must be expanded below a leaf node of a previous iteration of IDA* in order to complete the current iteration.

3. *Tree Imbalance*

   The tree imbalance is a statistical measurement between zero and one indicating the degree of balancing in a search space. This is measured at the second level of the search space off of the root node. The degree of imbalance of a search tree can affect the performance of many parallel search techniques by reducing the number of nodes and thus reducing the execution time to locate a goal.

4. *Heuristic Error*

   Heuristic error is defined as the difference, on average, between the estimated distance to a goal node and the true distance to the closest goal node. It is impossible to know the exact location of a goal in a search space until one is generated. The heuristic, being an estimate of the distance to a goal, is sometimes extremely accurate and, at other times, extremely inaccurate. Therefore, we are estimating the heuristic error by substituting the goal node with the node having the lowest heuristic for a threshold of IDA*. The node with the lowest heuristic does not always represent a node on the path to a goal; however we have found it to be common for the lowest heuristic to be a node on the path to the goal. This can be explained by the fact that there are usually several nodes in a search space with the same heuristic. The heuristic error is able to show the strength of the heuristic function for a particular problem domain. A large heuristic error represents an uninformed heuristic function, whereas a small heuristic error represents a strong heuristic function.

5. *Solution cost*

   In the IDA* algorithm a cost is associated with the generation of a child node and this cost accumulates as the search advances deeper into the search space. We define the solution cost as the cumulative cost of moves from the initial state to a goal state. Since the solution to a search problem is never known until a goal is located, the solution cost for a non-successful iteration of IDA* can only estimate a solution cost. This can be performed a number of ways. We made the assumption that a goal node could possibly be located in the path of a node in the search space in which the cost estimate (h) is the least for the iteration.

Each parameter in isolation is incapable of characterizing a search space or describing the best approach to a particular problem; however, when combined together they can differentiate a problem to a degree by its unique characteristics. Each of these factors is utilized as an antecedent for the rule base.

Eureka Architecture Results

In this section we will describe the results obtained from the Eureka architecture. To create the test cases, we ran each problem instance multiple times, once using each parallel search strategy in isolation. The search strategy producing the best speedup is considered to be the "correct" classification of the corresponding search tree. The resulting strategy is then provided as a learning case to the C4.5 machine learning system. The tests were performed on 64 processors of an nCUBE2 parallel computer system

The C4.5 results are produced by performing a tenfold cross validation run of the 98 problems of the fifteen puzzle data. A cross validation scheme repeatedly splits the training instances into two groups: one of which is used by C4.5 to develop the decision tree and rule set and the other which is used for testing the resulting decision tree and rules. For each strategy covered in this section we compare the individual approaches to the C4.5 recommended approach. For example, in analyzing the optimal number of clusters to be assigned to a problem, we show the average speedup for one, two, and four clusters. This is followed by the speedup obtained when C4.5 decides the number of clusters to be used for each problem.

For this research we examined the area subtask distribution and specifically, the selection of the number of

Table 1: Clustering Results

Approach	15Puzzle	Sig. Test
1Cluster	52.02	0.41
2Cluster	57.04	0.39
4Cluster	56.83	0.00
Eureka selection	**65.37**	

clusters. The strategy taken and resulting performance increase will be covered in this section.

Clustering

We tested the clustering algorithm using 1, 2, and 4 clusters on 64 processors of an nCUBE-2 parallel computer. Test results for the clustering algorithm are in table 1.

In this table, the values in the column labeled *Speedup* are produced with the equation Speedup=Serial time / Parallel time and is a measurement commonly used to compare the execution time of a job run with multiple processors (Parallel time) compared to the same job run on a single processor (Serial time). The first three rows in this table contains the average speedup of the 98 problems with each problem run with a fixed number of clusters. For the fourth row, indicated by *Eureka*, the number of clusters was dynamically selected based on the problem's structure. The average speedup obtained using this approach produced better results.

Future Work and Conclusions

We are currently in the process of implementing new problems within our system. Natural language processing is one area that can impact from this research. We are using the WordNet 1.6 (Miller 1995) for text parsing and inferencing. A second goal of future research is to apply the automation selection and use of machine learning to other strategy selection problems. Embedding a machine learning system into other applications allows for the program to assume a dynamic problem solving approach where the decisions are based on the problem structure.

The process of choosing a technique for solving a problem is integral to all scientific fields. The factors influencing the selection of one approach over another become more complicated than simply choosing the correct data structure for a problem.

Our research demonstrates that a machine learning system, such as C4.5, can be incorporated into a system where multiple strategies exist for solving the problem and produce an improvement in performance.

References

Cook, D., and Nerur, S. 1993. Maximizing the speedup of parallel search using HyPS*. In *Proceedings of the Third International Workshop on Parallel Processing for Artificial Intelligence - IJCAI-93.*

Cook, D.; Hall, L.; and Thomas, W. 1993. Parallel search using transformation-ordering iterative deepening-A*. *International Journal of Intelligent Mechanisms.*

Huang, S., and Davis, L. 1989. Parallel iterative A** search: An admissible distributed search algorithm. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 23–29.

Korf, R. 1985. Depth-first iterative deepening - an optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Kumar, V., and Rao, V. 1989. Load balancing on the hypercube architecture. In *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, volume 1.

Kumar, V., and Rao, V. 1990. Scalable parallel formulations of depth-first search. In Kumar, V.; Gopalakrishnan, P.; and Kanal, L., eds., *Parallel Algorithms for Machine Intelligence and Vision.* Springer-Verlag. 1–41.

Kumar, V.; Ananth, G.; and Rao, V. 1991. Scalable load balancing techniques for parallel computers. Technical Report 91-55, Computer Science Department.

Mahapatra, N., and Dutt, S. 1995. New anticipatory load balancing strategies for parallel A* algorithms. In *Proceedings of the DIMACS Series on Discrete Mathematics and Theoretical Computer Science.*

Miller, G. 1995. Wordnet: A lexical database. *Communications of the ACM* 38(11):39–41.

Powley, C., and Korf, R. 1991. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13(5).

Quinlan, J. 1993. *C4.5: Programs For Machine Learning.* Morgan Kaufmann.

Saletore, V. 1990. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. In *Proc. Fifth Distributed Memory Computing Conference.*

Varnell, R. C. 1997. *An Architecture for Improving the Performance of Parallel Search.* Ph.D. Dissertation, The University of Texas at Arlington.