A Grid-Based Game Tree Evaluation System

Pangfeng Liu[*] Shang-Kian Wang[*] Jan-Jan Wu[†] Yi-Min Zhung[*]

October 15, 2004

Abstract

Game tree search remains an interesting subject in artificial intelligence, and has been applied to many board games, including chess, Chinese chess, and GO. Given the exponential nature of the growth of tree size, a naive search of all the possible moves in the game tree (i.e. the min-max algorithm) is time consuming, and the search level, as well as the strength of the program, will be severely limited. Pruning the unnecessary part of the game tree (game tree pruning) is an important issue in increasing search efficiency.

In this paper, we propose a grid-based generic game tree search tool with alpha-beta pruning. The user of this tool can contribute program pieces as the plug-ins specific for a game (e.g., Chinese chess), and the system will automatically distribute the game tree search tasks to the processors on the grid. The user only needs to supplies game-specific information including the legal move generator, the evaluation function, and the end game determination. The control logics of alpha-beta pruning, workload distribution, and result integration are all automatically taken care of by the tool. Experimental results from an MPI implementation on a cluster of eight processors are reported in this paper, and we will report the results on the Taiwan UniGrid (consisting of eight clusters) once the operating environment is set up.

1 Introduction

Game tree search is an interesting topic in artificial intelligence. Using the enormous computing power of a modern computer, a computer program can enumerate all the possible scenarios after a given game situation, and choose the best move. Game tree search has been applied to many board games, including chess, Chinese chess, and go. The state-of-the-art computer chess program (e.g. Deep Blue) is now able to beat the top human chess grandmaster. The current best Chinese chess program is ranked about six dan, and is expected to achieve the strength of grandmaster soon. On the contrary, the progress of computer Go is relatively slow, due to the fact Go emphasizes more on strategic thinking than tactical evaluation. Now the current best program can only achieve the level of six kyu, which is much much weaker than most of the human Go player.

The basic strategy of game tree search is to enumerate all the possible moves from a given configuration, and choose the best one. Inevitably the cost of search goes as an exponential function of the search depth. Since the number of legal moves is usually very large in most games (e.g. chess), A brute force search of all the possible moves is time consuming, and the search level, as well as the strength of the program, will be severely limited. As a result, to avoid the part of the game that will not affect the final answer, or to "prune" the unnecessary part of the game tree (game tree pruning), is an important issue in search efficiency.

The current most often used game tree pruning technique is alpha-beta pruning. We divide the game tree into odd and even levels. During an odd level we choose a move that will maximize our gain in the game, no matter how our opponent reacts. During an even level our opponent chooses a move that will minimize our gain in the games, independent of how we respond. Suppose we have finished the evaluation of a move A, which will bring in G in gain. Now suppose we choose a different move B, and find out that the opponent has a counter-move that will make our gain less than G. Without evaluating any other counter-moves from our opponent, we declare that the move is inferior to the move A, and prune, i.e., skip the evaluation of the subtree

[*]Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, R.O.C., pangfeng@csie.ntu.edu.tw

[†]Institute of Information Science, Academia Sinica, Taipei 115, Taiwan, R.O.C., wuj,ice@iis.sinica.edu.tw

represented by B, in order to speed up the search.

We propose a generic game tree search system with alpha-beta pruning for grid infrastructure. The user of this tool can contribute program pieces as the plug-ins specific for a game (e.g., Chinese chess), and the system will distribute the game tree search tasks to processors in the grid. The system takes care of control logics of alpha-beta pruning, workload distribution, and result integration. The user supplies plug-ins including the legal move generator, the evaluation function, and the end game determination. We consider the end game of Chinese chess as our first application, and concentrate on continuous checking end game (sN). We implement our communication in MPI, which is compatible with Globus and most grid systems.

2 Game Tree Search

C. E. Shannon [5] introduced the concept of game trees along with a simple algorithm for searching them. A simple game tree for a two-player game is presented in Figure 1. A node in the tree represents a position in the game while a branch represents a move available at a particular position. Player 1 is on move at rectangular nodes and player 2 is on move at circle nodes. For example, at the *Root* node, player 1 is on move and the player has two moves available: *a* and *b*. Each leaf node is assigned a score that indicate how valuable that position is. A positive score indicates that player 1 is winning, while a negative score indicates that player 2 is winning. A score of 0 indicates a draw. The magnitude of the scores also conveys important information. The higher the score, the more favorable the position is for player 1. Similarly, the lower the score, the more favorable the position is for player 2.

The value of a game tree is the score of the leaf node that is reached when both sides exercise their best options. The problem we need to solve is to find the option at the root that leads to the game tree value. For example, in Figure 1, the ideally best option for player 1 is to move toward position N because it has the highest score (8) from his viewpoint. Assume that player 1 chooses move s to start to make progress toward position N. As far as player 2 is concerned, move e will play right into player 1's hand. To prevent this from happening, a careful player 2 will choose move d instead, and so player 1 has to follow up the move and choose move k, resulting in final score of -1.

If at the root node, player 1 chooses move b, then player 2 has three follow-up moves available, f, g and h. If player 2 chooses move f, player 1 will follow up and choose move o, resulting in final score of 4. Similarly, final scores of 6 and 7 will be returned if player 2 chooses move g and move h respectively. Since move f results in the lowest score (4), player 2 will choose move f. Now let's look at the root node again, since move a and move b result in a score of -1 and 4 respectively, the best score that player 1 can achieve when player 2 exercises his/her best options is 4.

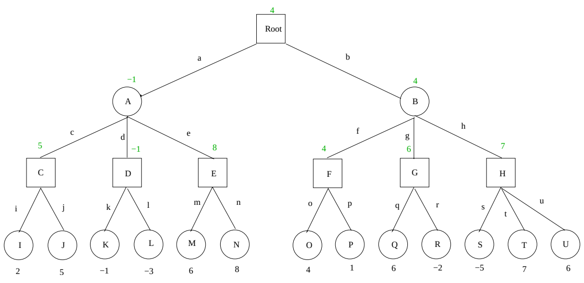

Figure 1: A simple game tree of two players.

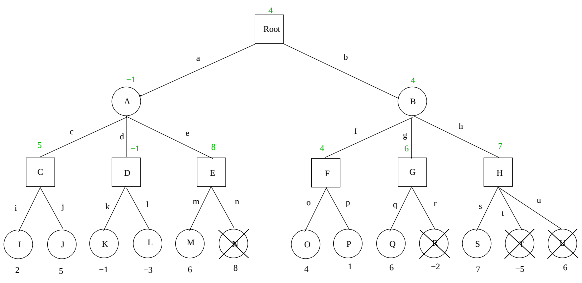

Figure 2: A simple game tree with pruned branches.

2.1 Min-Max Algorithm

C. E. Shannon [5] introduced a simple algorithm *min-max* for searching game trees. In the tree of Figure 1, at the nodes where player 1 is on move, player 1 will choose the move that maximizes the score. Similarly, the nodes where player 2 is on move, player 2 will choose the move that minimizes the score. Therefore, we can classify the tree nodes into two kinds of nodes: *maximizing* or *minimizing*.

The *min-max* algorithm traverses the entire tree in a depth-first fashion, and depending on whether a node is maximizing or minimizing, the algorithm keeps track of the largest or the smallest score, respectively. When a leaf node is reached, its score is determined by an evaluation function. Figure 3 depicts the min-max algorithm.

```
MinMax(node)
{
  if node is leaf then
     return Evaluate(node)
  if node.type == maximizing then
     score = - infty
  else score = + infty

  for (i = 1, node.number_of_branches)
  {
    value = MinMax(node.branch[i])
    if node.type == maximizing then
    {
       if value > score then
          score = value
    }
    else
    {
       if value < score then
          score = value
    }
  }
  return score
}
```

Figure 3: The Min-Max Algorithm

Since Min-max explores every node in the game tree, the algorithm is not practical for a game tree with many branches or depths. For example, a chess position has about 32 to 35 possible moves. A chess tree of depth n would contain 35^n nodes. Clearly it will not be practical to use the min-max algorithm for a chess tree with depth of more than 6. In complex board games such as chess, it is important to search as deeply as possible. Min-max does not allow for a very deep search, because the effective branching factor is extremely high.

2.2 Alpha-Beta Algorithm

The min-max algorithm can be improved in the following way, using Figure 1 as an example. We start at the root node, which initially has a score of $-\infty$. Node A is a minimizing node and hence starts with $+\infty$. The process of recursive calls continues until the leaf node I is reached and its score 2 is turned to node C. The initial score of $-\infty$ at node C is replaced by the new score 2, and then by the score 5 at node J in the next search. Node C returns its final score of 5 to node A and node A replaces its initial score $+\infty$ with this new score of 5. The recursive call continues for the second branch of node A, move d, which then returns a final score of -1. Since the new score -1 is smaller than the old one 5, node A's score is replaced by -1. Next, node A explores branch e. Node E, a maximizing node, has an initial score of $-\infty$. On exploring branch m, node E obtains a score of 6, which is where the improvement can be made. Since node E is a maximizing node, the score can only go higher than 6. However, it is also known that at node A, a minimizing node, the score is -1. Node A will not accept any value that is greater than -1. Therefore, the unexplored branches rooted at node E (in this case, branch n) do not need to be searched because they have no effect on the score at node A. Figure 2 shows the braches that can be pruned.

Knuth and Moore [3] proposed an efficient algorithm, *alpha-beta*, for sequential game tree search. The idea to cut-off unncessary branches is to keep two scores in the search. The first one is alpha (*lower bound*), which keeps track of the highest score obtained at a maximizing node higher up in the tree and is used to perform pruning at minimizing nodes. Any move made from the maximizing node with score less than or equal to alpha is of no improvement and can be pruned, because there is a strategy that is known to result in a score of alpha. The second score is beta (*upper bound*, which keeps track of the lowest score obtained at a minimizing node higher up in the tree and is used to perform pruning at maximizing nodes. Beta can be viewed as the worst-case scenario for the oppoent, because there is a way for the opponent to force a situation no worse than beta. If the search finds a move that returns a score of beta or greater, the rest of the legal moves do not have to be searched, because there is some choice the opponent will make to prevent that move from happening. The resulting algorithm, called alpha-beta algorithm, is shown in Figure 4.

```
AlphaBeta(node,alpha,beta)
{
  if node is leaf then
     return Evaluate(node)

  if node.type == maximizing then
     score = alpha
  else score = beta

  for (i=1, node.number_of_branches)
  {
     if node.type == maximizing then
     {
       value = AlphaBeta(node.branch[i],
         score,beta)
       if (value >= beta) then
          return beta
       if value > score then
          score = value
     }
     else
     {
       value = AlphaBeta(node.branch[i],
         alpha, score)
       if value <= alpha then
          return alpha
       if value < score then
          score = value
     }
  }
  return score
}
```

Figure 4: The Alpha-Beta Algorithm

3 A Game Tree Evaluation System and Its Parallel Implementation

3.1 Game Tree Evaluation System

In many board games, both players know where the pieces are, they alternate moves, and they are free to make any legal move. The object of the game is to checkmate the other player, to avoid being checkmated, or to achieve a draw if that's the best thing given the circumstances.

A board game program selects moves via use of a search function. A search function is a function that is passed information about the game, and tries to find the best move for side that the program is playing. An obvious sort of search function to use is a tree-searching function. For example, a game of chess can be considered as a large n-ary tree. The position that is on the board now is the root position or root node. Positions that can be reached in one move from the root position are reached by branches from the root position. These positions are called successor positions or successor nodes. Each of these successor positions has a series of branches emanating from it, each of which represents a legal move from that position. A heuristic function, traditionally called "Evaluate", is used to assign values to these positions. These values are usually educated guesses. Evaluate is a function that returns an exact value for the position, if possible, and an heuristic value if an exact value is not available. Using chess as an example, the function Evaluate can be defined as follows. The function returns a very large positive value if Black is checkmated, a very large negative value if White is checkmated, and a constant value, probably zero or something near zero, if the game is drawn now (for instance if the side to move is stalemated, or if there are bare kings). If the position doesn't represent the end of the game, an heuristic value is returned. The value returned by the heuristic function will always be positive if White has won or is winning, negative if Black has won or is winning, and around zero if the game is even or is a draw. The generation of legal moves from a board position and the definition of the function Evaluate may vary depending on the game. Our game tree evaluation system provides interfaces for the users to plug-in these functions.

Our game tree evaluation system is implemented as a set of C codes with MPI primitives for inter-processor communication. The main codes consist of three modules, `master`, `worker`, and `game`. The `master` and `worker` modules implement a master-worker model for parallel tree search, which will be described in more details in Section 3.2. The `game` module defines programming interfaces for the user plug-in functions, `generate_moves` and `evaluate`. These two functions are defined as follows.

```
void generate_moves(int *move_num,
  char *board_states[MAX_BRANCH])
{
  move_num = decide number of legal moves;
  for (board_state_index = 0, *move_num)
    decide the value of
      board_states[board_state_index];
}

int evaluate(char *state)
{
  compute the score at the given
    position (state).
}
```

3.2 Parallel Game Tree Search

There are several parallelization methods for game tree search reported in the literature [1, 2, 4]. Some were targeted for shared-memory machines and the others were designed with distributed-memory machines in minds. Of the shared-memory algorithms, the most recent and efficient one is *dynamic tree splitting* (DTS) [2]. DTS maintains a global list of active split-points (SP-LIST). An idle processor consults SP-LIST to find work to do. DTS was able to achieve spectacular speed-up on some shared-memory machines. However, since DTS was designed with shared memory in mind and used global lists in its implementation, it was not suitable for distributed-memory machines.

For distributed-memory machines, *principle variation splitting* (PVSplit) [1], has been a popular algorithm for searching game trees. In PVSplit, the first branch at a PV node must be searched before parallel search of the remaining branches may begin. Experiments with PVSplit on massively parallel systems have shown that speed-up is limited to a large extent by synchronization overhead.

In this section, we present our parallel implementation of game tree search in distributed environment. Since the min-max algorithm is not practical, we will focus on the alpha-beta algorithm in our parallel implementation. Parallelization of the alpha-beta algorithm is difficult. A parallel implementation involves several overheads: (1) communication overhead, (2) search overhead, and (3) delay caused by imbalance load. The sequential alpha-beta algorithm updates its two bounds, alpha and beta, as the search of the game tree progresses. When search in parallel, if a processor finds an improvement to alpha or beta, it needs to inform other processors working on other branches so that they can make use of the tighter bounds. Passing updated alpha and beta between processors requires communication overhead. Search overhead is the consequence of parallel alpha-beta algorithm. When parallel search is initiated at one node, the best score might not have been discovered yet. As a result, parallel search is conducted with a wider search window than in the sequential case. Furthermore, in parallel search, a processor might perform useless work when a better bound is discovered that has proven that the branch that processor is exploring can be pruned.

Our approach to balancing these overheads is based on a simple master-worker model. At the start, the master processor is given ownership of the Root node while the worker processors remain idle. The master processor first decides the split-point according to the number of worker processors. The tree is split at level L if the number of nodes at level $L+1$ is greater than or equal to the number of worker processors. The pool of nodes at level $L+1$ represent the search work to be done by the worker processors. An idle worker processor sends a message to the master requesting for work. If there are nodes available in the pool, the master chooses one node and sends the node id and current value of alpha and beta (the bounds) to the worker. If a worker finds an improvement to the bounds, then the new score is transmitted to the master. Next time when another worker requests for work, the master will despatch work with the updated bounds. A worker processor may also discover a pruning condition with the node it is given. In this case, the search is complete and the worker processor proceeds to request another work from the master or returns to idle state if there is no work available. Our master-worker parallel implementation has the following properties.

- Our implementation reduces communication overhead as much as possible. Whenever a new bound is discovered, it is not broadcasted among the processors, instead, the new value is only transmitted from a completing worker to the master, and then from the master to a worker that requests new work from the master. This point-to-point communication assumption is appropriate for a grid-based or a distributed environment because (1) broadcasting is expensive in distributed systems, and (2) in distributed systems, there might not be communication links between the worker processors, making broadcasting impossible. With our approach, a worker is informed of the newest bounds as soon as it requests work from the master. updating of bounds.

- In our master-worker implementation, a worker processor never sits idle when there is work available, thus reducing load imbalance in game tree search.

4 Experimental Results

We use a "pick-the-last-one-loses" game as an example of our parallel alpha-beta game tree system. The game board is triangular with pieces arranged as in Figure 5. The player can remove a segment of consecutive pieces in one move. The segment removed must be parallel to the three axis along the three directions how pieces are placed. The player that is forced to remove the last piece loses. For a given board configuration, our system searches for a move. If a winning is found, the program will choose it, otherwise the program will randomly choose one.

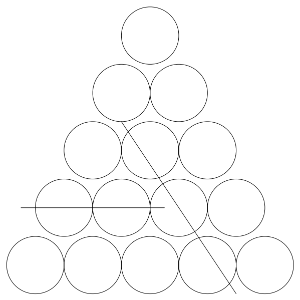

Figure 5: A pick-the-last-one-loses game.

We conduct our simulation in a cluster consisting of eight processors. In addition, the manage host is a SMP machine so there are nine available processors in total. All these processors are Pentium III 1GHz processors, and each processor has 512M byte of memory.

The board configurations are chosen as follow. We set the height of the triangular board to 5 and 6. We randomly place pieces in the board, and number of pieces ranging from 6 to 14. We use the number of nodes searched and time elapsed as our metrics of performance evaluation. For each number of pieces we measure these numbers, and take the average from 20 runs.

We compare the performance of the sequential version and MPI version program. We observed that when the height of the triangle is 6 and the number of pieces is smaller than 8, the sequential version is faster than the MPI version program. The reason is that there is not much workload to distributed in these small cases. However, when the number of pieces reaches 9, the execution time from the sequential version increases rapidly. When the number of pieces reaches 14 the sequential version is three times slower than the MPI version. We also observe the same trend when the height of the triangle is 5.

From Table 2 we observed that the number of nodes searched by the sequential version is always less than those searched by the MPI version program. The reason is that the MPI version searches nodes concurrently, and some processors may still be searching although the path has been found in some other processors node. We do not find a regular pattern in the difference between the number of nodes searched by the sequential version verses the MPI version program. It seems that the difference depends on the board configuration, and even when two board configurations differ by one piece, the number of tree nodes searched may be very different between the two implementations.

5 Conclusions

In this paper, we propose a grid-based generic game tree search tool with alpha-beta pruning. The user of this tool can contribute program pieces as the plug-ins specific for a game, and the system will automatically distribute the game tree search tasks to the processors on the grid. The user only needs to supplies game-specific information including the legal move generator, the evaluation function, and the end game determination. The control logics of alpha-beta pruning, workload distribution, and result integration are all automatically taken care of by the tool.

Our experimental results from an MPI implementation suggest that alpha-beta pruning is not easy to parallelize. When the number of pieces is 14 in a triangular board of height 6, we report a speedup of 3.13 on a cluster of 8 processors. This suggests that the important bounds in alpha-beta pruning should be exchanged between worker processors in a more efficient way.

The future work include an implementation on the Taiwan UniGrid, which consists of eight clusters. The implementation should be straight forward since the GLOBUS toolkit support MPI communication library, based on which our system is implemented. Also we will investigate other games, including Chinese chess end game, to demonstrate the versatility of our system.

	Number of pieces on the board								
	6	7	8	9	10	11	12	13	14
MPI (height=5)	0.05	0.05	0.10	0.20	0.40	2.95	13.85		
SEQ (height=5)	0.00	0.00	0.05	0.20	1.00	9.50	42.20		
MPI (height=6)	0.05	0.05	0.10	0.20	0.50	2.95	29.35	183.10	1762.80
SEQ (height=6)	0.00	0.00	0.05	0.25	1.60	8.35	106.05	631.50	5524.45

Table 1: The total execution time in seconds.

	Number of pieces on the board						
	6	7	8	9	10	11	12
MPI (height=5)	964	2254	20539	109940	396740	3957260	19796435
SEQ (height=5)	510	1716	11611	66934	292256	2907487	12759256
MPI (height=6)	908	2558	16730	91395	497349	3377081	33766472
SEQ (height=6)	468	1888	6752	57436	414297	2095231	26623136

Table 2: The number of game tree nodes searched.

References

[1] R. Feldmann. Distributed Game Tree Search. *ICCA Journal*, 12(2):65–73, 1989.

[2] R. Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, Paderborn, Germany, 1993.

[3] D. E. Knuth and R. W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[4] T. A. Marsland and F. Popowich. Parallel Game-Tree Search. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-7(4):442–452, 1985.

[5] C. E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(7):256–275, 1950.