

BS/MS Project (CS 4513)  
DISTRIBUTED SEARCH OF GAME TREES AND  
THE GAME OF GO

Thomas Liu

May 5, 2009

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Basic Rule Processing</b>	<b>4</b>
2.1	Board Representation . . . . .	4
2.2	Move Generation . . . . .	5
<b>3</b>	<b>Basic Search Trees</b>	<b>5</b>
<b>4</b>	<b>Parallel Alpha-Beta Search</b>	<b>5</b>
4.1	Introduction . . . . .	5
4.2	Parallelization of Alpha-Beta Search . . . . .	7
4.3	Node Classification . . . . .	8
4.4	PVSplit . . . . .	9
4.5	Young Brother's Wait Concept . . . . .	10
4.6	Dynamic Tree Splitting . . . . .	12
4.6.1	Finding a Split Point . . . . .	12
4.7	Benchmarks . . . . .	14
4.8	The Fallacy of $\alpha\beta$ Search in the Game of Go . . . . .	14
<b>5</b>	<b>Monte-Carlo Go</b>	<b>15</b>
5.1	Upper Confidence bounds applied to Trees . . . . .	16
5.2	Parallelization of Monte-Carlo Go . . . . .	17
5.2.1	Leaf Parallelization . . . . .	17
5.2.2	Root Parallelization . . . . .	18
5.2.3	Multiple-runs Parallelization . . . . .	18
5.2.4	Tree Parallelization . . . . .	19
5.3	Benchmarks . . . . .	19
<b>6</b>	<b>Conclusion</b>	<b>20</b>
<b>7</b>	<b>Bibliography</b>	<b>21</b>

# 1 Introduction

The nature of my project is to explore the searching of game trees in particular in the game of Go. Throughout my paper I draw from several techniques used in Chess programming. Chess, while a very different game, poses some of the same challenges when it comes to searching game trees. Unfortunately, the scale of a Go game differs greatly from that of a Chess game and thus some of the techniques used in Chess cannot efficiently be applied to Go. As thus many other techniques and algorithms have been explored.

## 2 Basic Rule Processing

### 2.1 Board Representation

A simple beginning to this venture is looking at ways to represent the Go board. Many chess engines use piece centric representations of the board, using bits to represent where the pieces are on the board. In contrast to this is a board centric representation, where an abstract version of the board is kept in memory, denoting each space as occupied with a certain piece or empty. [1]

The piece centric representation is not suitable for Go. The status of pieces on the board depends entirely on the pieces in the surrounding vicinity, so in order to evaluate board positions from a piece centric representation one would first have to convert it into a board centric representation.

Thus the board centric representation I use is a simple integer array of  $n \times m$ , where each spot denotes -1 for opponent's color, 1 for empty, and 0 for a friendly piece. This makes for simple calculation of liberties as simple addition of cardinal spaces gives the liberties of a piece.

### 2.2 Move Generation

Move generation is a difficult topic. It is difficult for a computer to think the way that a human does and thus our solution tends to be to have the computer search many game trees looking for an ideal move, using the ability of computers to do a large number of computations very quickly.

A start to move generation is defining valid moves. What constitutes a valid move and how do we check for it? A valid move is a move at any vacant intersection on the board as long as it is not:

- suicide: if the stone being placed or the group it connects to would have zero liberties, it is considered suicide.
- ko: if the intersection has been determined to be the spot of the current (if any) ko fight.

## 3 Basic Search Trees

Each node of the search tree represents the board after a given move has been played and each child of a node is a possible board position that could be obtained by playing

another piece.

The basic idea is to use a depth first search to find a node where the player has won the situation based on the win conditions listed above.

## 4 Parallel Alpha-Beta Search

### 4.1 Introduction

The  $\alpha\beta$  algorithm has been a very popular way of pruning game search trees for many years and remains the most popular way of pruning search trees in games like chess and Go. The algorithm eliminates nodes that have been proven irrelevant from the search. Two bounds are maintained,  $\alpha$  and  $\beta$ , which represent the lower and upper bounds of the minimax value (value of the board evaluated at each node) of the search tree (essentially the search window). Once the algorithm proves that the score of a node is outside the search window, further effort at the node is irrelevant and that node no longer needs to be searched. [2]

One of the many enhancements to the  $\alpha\beta$  algorithm is the use of a transposition table to improve search efficiency. A transposition table is a cache of results from previously searched sub-trees. [3]

A PBV is a preliminary backed-up value. PBVs are often used because it can be very difficult to evaluate intermediate nodes in a search tree. Using PBVs, we explore down to the end of the tree using a depth-first search and evaluate the lowest-level node using our evaluation function. Next we bring the values up to the next higher node according to the following rules:

- If it's your move, bring up the largest value, possibly replacing a smaller value.
- If it's your opponent's move, bring up the smallest value, replacing any larger values.

From here, we can look for  $\alpha$  and  $\beta$  cutoffs.

- $\alpha$  cutoffs occur when:
  - It is *your opponent's turn* to make a move.
  - You have computed a PBV for the parent of the current node.
  - The node's parent has a *higher* PBV than this node.
  - The current node has other children that you have yet to consider.
- $\beta$  cutoffs occur when:
  - It is *your turn* to make a move.
  - You have computed a PBV for the parent of the current node.
  - The node's parent has a *lower* PBV than this node.
  - The current node has other children that you have yet to consider.

If a node has been determined to be an  $\alpha$  or  $\beta$  cutoff, we no longer need to consider the children of that node. [4]

## 4.2 Parallelization of Alpha-Beta Search

Parallelization of the  $\alpha\beta$  search includes distributing the processing of sub trees to different processors or threads. By doing so we can achieve a significant speed up but there is associated overhead with the distribution of the tasks. The overhead can be split into three types [5]:

- communication overhead
- synchronization overhead
- search overhead

The  $\alpha\beta$  algorithm updates bounds for  $\alpha$  and  $\beta$  as the search progresses. Thus, there will be wasted computation unless all processors searching a tree can share  $\alpha$  and  $\beta$  values. These values need to be communicated to all agents in the network either through a broadcast or by updating a shared transposition table. This communication causes the aforementioned communication overhead.

Synchronization overhead occurs when one or more processors are waiting (doing nothing) for an event to occur. For instance, a processor could be waiting on the results of another processor, and this down time is synchronization overhead.

When an  $\alpha\beta$  search is done in parallel, it is possible for some processors to examine nodes that would have been avoided by a non-distributed sequential version of the search. This occurs when the best possible score for a node has yet to be determined and sub trees of this node are assigned to different agents in the network. If one agent discovers a cutoff implying that the rest of the sub trees have no need to be searched, any other agents searching the now irrelevant sub trees have simply wasted computation. This wasted processing is search overhead.

This overhead can make the gain of parallelization close to useless and thus many ways to reduce overhead have been proposed.

## 4.3 Node Classification

Knuth and Moore first classified nodes into types that we can be expected to find within a minimal  $\alpha\beta$  tree when describing the  $\alpha\beta$  algorithm [7]. These node types were later further analyzed by Judea Pearl [9]. These node types are for a perfectly ordered tree, in which left most nodes provide the best solution to the tree. The node types and their properties are classified as follows:

- PV (Type 1) Nodes, also known as principal variation nodes, are nodes that are inside the  $\alpha\beta$  search window. These nodes have all moves searched and the value can be brought up to the root. In other words, these nodes will have PBVs brought up to or from them.
  - The root of the tree and left most nodes are always PV nodes.
  - All siblings of a PV node are Cut nodes.
- Cut (Type 2) nodes, also known as fail-high nodes, have had a beta cutoff occur at them. A minimum of one move at a cut node needs to be searched.

- The successor of a Cut node is an All node.
- The ancestor of a Cut node is either a PV node or an All node.
- All (Type 3) nodes, also known as fail-low nodes, are nodes where no move's score has exceeded alpha.
  - The successors of an All node are Cut nodes.
  - The ancestor of an All node is a Cut node.

These classifications are important because the algorithms for distributing the  $\alpha\beta$  tree look at these different types of nodes.

Since a perfectly ordered search tree would require an oracle, the  $\alpha\beta$  algorithm tries to work with trees sorted to be as minimal as possible. Since a heuristic to evaluate a Go board is difficult to obtain and out of the scope of this project, I have suggested and used a simple move ordering. Moves are sorted in order of importance, where more important moves are:

1. A move that captures an enemy stone.
2. A move that increases the liberties of a friendly group.
3. A move that decreases the liberties of an enemy group.
4. A move that does neither (2) or (3).
5. A move that places a friendly group in atari (only one liberty remains and a subsequent enemy move will cause a capture).
6. Any other moves.

Next we investigate methods of distributing searching of the tree to multiple processors.

## 4.4 PVSplit

One of the ways to cut down on overhead is called PVSplit, or Principal Variation Splitting. The concept behind PVSplit [8] is that the first branch at a PV node has to be searched before parallel search of the remaining branches should begin.

All of the processors explore the first branch at each PV node until they reach the PV node that is one level above the leaf nodes of the search tree. Here, one processor searches the first branch (left most leaf) and the other processors wait for it finish. The first processor computes the value of the branch and brings the value up to the PV node. Once this is done, all processors join the search and each traverse down a branch, determining its value. If a processor discovers an improvement to the score at the node, all of the processors are informed of this change, effectively updating the  $\alpha$  and  $\beta$  bounds for the search. If there are no unassigned branches, a processor with no work remains idle until the other processors finish.

Once all of the branches have been searched, the processors move to the parent of the node, continuing this process until all branches of the root node have been searched.

Marsland and Popowich's reasoning behind their idea is pretty straightforward. We already know that all branches of a PV node will need to be searched, so parallel searching is a good idea at PV nodes. Since they require that the first branch be evaluated before all of the processors begin searching, parallel searching does not begin until a bound has already been determined [8].

Unfortunately the PVSplit method suffers from significant synchronization overhead. First of all, while the first processor searches the left most branch the remaining processors need to wait. In addition, consider what happens when parallel search of a node nears the finish. Most machines in the search will be idle waiting for some processors that may have received more "difficult" branches. Difficult branches occur when there are more nodes in the branch than normal.

## 4.5 Young Brother's Wait Concept

The Young Brother's Wait Concept (YBWC) was first explored by Feldmann, Monien, Mysliwicz and Vornberger in 1989 [9] and then later improved by Feldmann in 1993 [11].

In YBWC the rule is simple: the eldest brother (first branch to be expanded at a node) must be examined before parallel search of younger brothers should begin. This is similar to PVSplit, but parallel search in YBWC is possible at all nodes, not just PV nodes.

In addition, YBWC introduces the concept of node ownership. When a processor owns a node, it is responsible for returning the node's evaluation to the parent node. This is likely to involve communication if the owner of the parent and the child are different processors. An additional detail is that in YBWC once a processor is assigned ownership of a node the ownership cannot be transferred.

A little more terminology used with YBWC as well as later with Dynamic Tree Splitting.

- split point - a node whose value of the oldest (first) branch is known.
- highest split point (of a tree) - a split point which has the least depth.

YBWC works as follows: Let processor  $P_1$  own the root of the tree and begin to search using  $\alpha\beta$  search.  $P_1$  maintains split point information.

While the game tree has not been fully searched, any idle processor  $P_i$  does the following:

1. Look for processes that have unexplored split points.
2. Get a branch from a highest split point and take ownership of this sub tree.
3. Begin to search using  $\alpha\beta$  searching and maintain split point information.
4. Return information to the owner of the parent node of the split point once the sub tree is completely searched.

YBWC has an obvious improvement in synchronization overhead since any idle processors can pick up entire sub trees and start to process them, unlike PVSplit which has idle processors wait around for other branches of the current node to be processed.

Unfortunately there is extra communication overhead as all processors need to be communicating about split points for processing. Some optimization also needs to be done to prevent splitting of small trees. This needs to be done by calculating the cost of splitting a node [12].

## 4.6 Dynamic Tree Splitting

Dynamic Tree Splitting [13, 14] uses a peer to peer approach in which the idea of ownership of a node is discarded. Instead, many processors may work together on a node, and the processor that finishes last is responsible for returning the evaluation to the parent.

One processor starts by searching the root node while the rest of the processors remain idle.

While the tree is being searched an idle processor will consult a global list of active split points to find work to do. If a split point with work is found (there is a branch to be explored) then the idle processor joins other processors at that split point and works in parallel exploring branches.

If an idle processor cannot find work in the split point list, the idle processor broadcasts a HELP message to all processors. When a busy processor receives a HELP message it copies the state of the sub tree it is exploring to a shared area. Then the idle processor can examine the shared area to find a suitable split point. If a split point is found it is copied into the global split point list and the idle processor will begin to work on it. If no suitable split point can be found in the shared area the idle processor waits a bit and then broadcasts the HELP message again.

When a processor finishes searching a branch and there is no longer any more work at the split point it was helping to search the processor enters an idle state and tries to find work at another node. If the finishing processor is the last processor at the split point it is responsible for reporting the evaluation of the node to the parent node, where it will try to find more work to do.

### 4.6.1 Finding a Split Point

Finding a split point in DTS is more complicated than in YBWC and PVSplit, leading to more synchronization overhead ideally in return for much less search overhead. Hyatt classifies nodes a little differently, as follows:

- D-PV - a node that has the same  $\alpha$  and  $\beta$  values as the root.
- D-Cut - A minimizing node with the same beta as the root or a maximizing node with the same alpha as the root
- D-All - Any node that does not fit the criteria for D-PV or D-Cut

D-PV and D-Cut are in fact equivalent to Knuth's PV and Cut nodes. Hyatt defined a few more pieces of node logic. First, if a D-Cut node has had more than three branches examined at it and an  $\alpha$  or  $\beta$  cut off has not appeared the node becomes a D-All node. Dynamic Tree Splitting does not allow more than two D-All nodes to be consecutive in the search tree. After the second D-All node, the nodes are put in an alternating sequence of D-Cut and D-All [13]. In addition Hyatt defines a confidence factor for D-Cut and D-All

nodes. When up to three branches have been searched at a D-Cut node the confidence level that it is actually a D-Cut node is lowered (any more and it becomes a D-All node). The more moves that have been searched at a D-All node the higher the confidence level is that the node is a D-All node.

With these definitions we can look at the suitability of any given node in Dynamic Tree Splitting to become a split point based on four factors:

1. The node must be of type D-PV or D-All.
2. Nodes higher in the tree (closer to the root) represent more work and thus should be processed in parallel.
3. If the node is a D-PV node the first branch must have already been searched (similar to YBWC).
4. If the node is a D-All node it should have a high confidence level.

This process of finding split points is much more complicated than how YBWC or PVSplit does it but is an attempt to eliminate search overhead. Unfortunately Hyatt designed Dynamic Tree Splitting for shared memory systems. This means that there would be significant communication overhead when the work is not on a shared memory architecture.

## 4.7 Benchmarks

Hyatt tested Dynamic Tree Splitting as well as PVSplit and recorded the rate of speed up that was achieved [14].

Processors	1	2	4	8	16
Speed	100%	180%	300%	410%	460%

*Performance of PVSplit on chess trees.*

Processors	1	2	4	8	16
Speed	100%	200%	370%	660%	1110%

*Performance of Dynamic Tree Splitting on chess trees.*

It seems clear that PVSplit quickly diminishes in usefulness as we increase the number of processors, suggesting that there is significant communication overhead. Dynamic Tree Splitting nets larger returns on larger numbers of processors.

## 4.8 The Fallacy of $\alpha\beta$ Search in the Game of Go

While  $\alpha\beta$  search has been very successful in searching Chess trees, it has failed to provide efficient Go strategies for a number of reasons. First of all the size and branching factor of a tree in Go are significantly larger than Chess. The Chess board has  $8 \times 8$  spaces, while the Go board has  $19 \times 19$ . The number of potential moves at any given time is a few hundred as opposed to the few dozen in Chess.

Secondly no one has yet discovered a good evaluation function approximating that minimax value of a position on a Go board. As  $\alpha\beta$  search hinges on the ability to evaluate

board positions for a value, without a good evaluation function it is difficult to perform the search. One of the things that makes it very difficult to gauge whether one move is better than another is that all of the stones on the board have the same weight, unlike Chess in which some pieces are obviously superior (the queen is more powerful than the bishop).

Even so,  $\alpha\beta$  search could be used for localized go problems on a smaller scale, to evaluate local fights.

## 5 Monte-Carlo Go

Computers remained very weak at playing the game of Go until breakthroughs were made and new algorithms including the Monte-Carlo algorithm for tree search were invented. The Monte-Carlo Tree Search (MCTS) [15, 16] is a best-first search method that, unlike  $\alpha\beta$  search, does not require an evaluation function capable of evaluating different positions on a Go board. The only evaluation function needed is one to judge the winner at the end of a game, which is a simple task even in the game of Go.

The essence of MCTS is exploration of *random* game possibilities followed by a statistical analysis of the explored games. MCTS uses results of previous board explorations to grow a game tree, and progressively becomes possible of more accurately guessing the best move in a given board position.

The steps that MCTS takes are as follows:

1. Traverse the tree from the root node and find a leaf node  $L$  that has not been added to the tree yet.
2. Add  $L$  to the tree.
3. Play moves in a self-play mode until the end of the game is reached.
4. Evaluate who the winner of the game is (black: +1, tie: 0, white: -1) and propagate the results through the tree. In other words, calculate the average of the results in each of the nodes traversed.

Essentially we do a basic search to a given, generally shallow depth and then play random games at each leaf node, accumulating statistics to determine how well the move will do. MCTS simulates games for as long as possible, and once time runs out it chooses the leaf node that had the highest chance of success.

### 5.1 Upper Confidence bounds applied to Trees

Upper Confidence Bounds applied to Trees (UCT) is worth mentioning since it is the algorithm currently used by the strongest Go programs in the world to improve upon Monte-Carlo Go [16, 17].

While working out simulations of moves in a Go game we would like to focus on more promising moves. How do we decide when a move is not promising? This introduces a tradeoff between *exploration* and *exploitation*. Exploration is sampling each potential move enough times to have an accurate estimate of value and exploitation is spending the

most samples on the moves that have been the most successful so far. If a move seems promising then it would make more sense to spend more time confirming that it in fact is a good move.

To this respect, UCT is an algorithm for choosing which moves to sample. UCT chooses the move with the greatest sum of *value* and *uncertainty*. This means that UCT will always look for moves which show promise or moves that have yet to be explored. As the algorithm works it will shift towards sampling of moves that show promise. UCT consists of exploring the move that maximizes  $c_i = \mu_i + C \times \sqrt{\frac{\log t}{s}}$  where  $\mu_i$  is the mean result of games starting with the move  $c_i$ ,  $t$  is the number of games played in the current node, and  $s$  is the number of games that start with move  $c_i$ .  $C$  is a constant that can be changed to adjust the level of exploration vs exploitation. High values favor exploration and low values favor exploitation. More detail about UCT can be found in Gelly, Wang, Munos and Teytaud's work on one of the current strongest Go programs in the world, MoGo [17].

## 5.2 Parallelization of Monte-Carlo Go

Now we move onto parallelization of Monte-Carlo Go and UCT. Even with an improved algorithm it is still difficult to quickly analyze a large sub set of moves. In order for computer Go to be effective in a tournament with a time limit parallelization of the algorithms is very necessary. Recently MoGo defeated a Korean professional 8-dan player in a 9-stone handicap game (MoGo was given the handicap). This version of MoGo was running on a super computer with 800 cores! Surely this shows that even the most recent algorithms cannot hope to hold out in an even contest with professionals.

Once again, we discuss parallelization techniques and the various types of overheads associated with them.

### 5.2.1 Leaf Parallelization

This is one of the simplest ways to parallelize Monte-Carlo Tree searches introduced by Cazenave and Jouandeau as at the leaves parallelization [19]. The idea is simple. One processor begins to expand the search tree and at every leaf node (every move we want to analyze for helpfulness) we let each processor simulate a give number of random games. Each of the procesors is given a different seed for their simulations and each of them simulates different games to the end. When the simulations are finished the result of the simulated games is collected by one of the processors and reported to the parent.

Leaf parallelization is easy to implement and does not require any use of mutual exclusion. Unfortunately, it suffers from synchronization overhead. The time required to simulate a game is highly variable, and playing  $n$  games with  $n$  processors will take a longer time no average than the average time it takes one thread to play one game, since the program has to wait for the longest simulated game [20]. There is also some communication overhead since communication occurs between processors at every leaf that is explored.

### 5.2.2 Root Parallelization

Root parallelization was also called single-run parallelization by Cazenave and Jouandeau. Instead of distributing simulations like in leaf parallelization, root parallelization distributes creation of the game tree. One processor is dubbed the master and waits at the root node for responses. From the root of the tree each processor is given a random seed to create a different Monte-Carlo sub tree. Each processor then explores and simulates the tree without communication with the other processors [19].

Once a time limit has been reached or a random game limit has been reached, each processor reports back to the master the number of games and the number of wins for all of the moves at the root node of the tree. The master process simply adds the number of games and wins of moves retrieved from the slave processes. From this, we can decide on a best move to make.

The great thing about this is that communication overhead is minimal. Synchronization overhead is also minimized since all processors do work until an agreed stopping point. However, there can be significant search overhead as several processes might explore similar games that all lead to failure.

### 5.2.3 Multiple-runs Parallelization

Also introduced by Cazenave and Jouandeau, multiple-runs parallelization is similar to single-run parallelization except that the processes exploring sub trees do communication with each other. This consists of updating at shared Monte-Carlo tree every certain amount of fixed time [19].

A master process begins by sending the current board to all of the slave processes, who form their own search trees based on a random seed. After a set amount of computation time the slaves all report their current results back to the master, who adds everything together and then sends the information back to the slaves. The slaves then create new Monte-Carlo trees based on the information and continue the process until a time has elapsed. This results in the slaves being able to update their search trees every now and then so that the slaves can decide when simulations at a leaf no longer need to be run. This results in significantly less search overhead at the cost of communication overhead.

### 5.2.4 Tree Parallelization

Tree parallelization is a new way of parallelizing Monte-Carlo search trees suggested by Chaslot, Winands and Herik [20]. Their idea is to use one single shared tree from which processors can play simultaneous games. Every processor can write to the tree, so mutexes are used to lock certain parts of the tree to prevent corruption of data.

The main part of tree parallelization lies in a concept called *virtual loss*. If several processes start from the root at the same time it is likely that they will traverse the tree in a similar fashion, since simulated games might start at leaf nodes which are in the same neighborhood as each other on the board. This leads to significant search overhead.

A search tree tends to have millions of nodes; therefore it is inefficient to search the same small section of the tree many times. Thus, when a processor visits a node in the tree, a virtual loss is assigned to it: the value of the node is decreased. The next processor will only select a node to explore if the value of the node is more than the values of the

sibling nodes. The virtual loss is removed when the processor that visited the node has propagated the result of the finished simulated game.

As a result of this processors will not visit a node where a processor is currently simulating games and will instead seek new nodes to explore. All in all, nodes that clearly have a better score will be explored by all threads, while unevaluated nodes will be explored by only one processor.

Chaslot, Winands and Herik argue that this keeps a balance between exploration and exploitation in a parallelized Monte-Carlo tree search [20].

### 5.3 Benchmarks

Chaslot, Winands and Herik performed thorough benchmark testing to see how the parallelization methods would work against a Go program named GNU Go. They tested the above methods of parallelization using threads on a multiprocessor machine. Their results showed that in a multiprocessor machine with shared memory root parallelization produced the best results. In fact, tree, multiple-run and leaf parallelization performed very similarly but root parallelization created a Go player that was significantly more effective [20].

One interesting statistic is that when they used four threads, they observed that the results seemed to imply that it was more efficient to run four independent searches for length of one second than to run one large search for four seconds.

## 6 Conclusion

I have discussed several methods of parallelizing two popular, proven to be effective methods of searching game trees.  $\alpha\beta$  tree search has been the method of searching that has produced the strongest computer Chess players in history, but the sheer scale of the game of Go has proven it to be inefficient.

Monte-Carlo Go has been a recent breakthrough that has let computer Go players leap a level ahead in strength. For the most part, strong computer Go players are run on largely distributed, shared memory systems, since the exponential nature of the problem is best analyzed when broken into several pieces.

An 800 core super computer was able to win against a very strong professional player while taking a nine stone handicap, but computer Go remains relatively weak and only about as strong as a strong amateur player.

Future research for me will include looking into other ways of approaching the problem of computer Go, since game tree searching is not the only way. Machine learning and training based on professional games has been slowly gaining popularity in recent years. However, it is hard to say if the game of Go will ever be "solved", because it is very difficult to evaluate whether one move is better than another.

## 7 Bibliography

- 1 Hyatt, Robert. "Chess program board representations."
- 2 Kishimoto, Akihiro, and Jonathan Schaeffer. "Distributed Game-Tree Search Using Transposition Table Driven Work Scheduling." Print.
- 3 D. Slate and L. Atkin. CHESS 4.5 - The Northwestern University Chess Program. In P.Frey, editor, Chess Skill in Man and Machine, pages 82-118. Springer-Verlag, 1977.
- 4 Matuszek, David. "Alpha-Beta Search." (2008).
- 5 Schaeffer, J. (1989). Distributed Game-Tree Searching. Journal of Parallel and Distributed Computing, Vol. 6, No. 2, pp. 90 114.
- 6 Marsland, T.A. and Campbell, M.S. (1982). Parallel Search of Strongly Ordered Game Trees.
- 7 Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. Artificial Intelligence, Vol. 6, No. 4, pp. 293 326.
- 8 Marsland, T.A. and Popowich, F. (1985). Parallel Game-Tree Search. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-7, No. 4, pp. 442 452.
- 9 Judea Pearl (1982) The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and its Optimality.
- 10 Feldmann, R., Monien, B., Mysliwicz, P. and Vornberger, O. (1989). Distributed Game Tree Search.
- 11 Feldmann, R. (1993). Game Tree Search on Massively Parallel Systems. Ph.D. Thesis, University of Paderborn, Paderborn, Germany.
- 12 Manohararajah, V. (2007). Parallel Game Tree Search.
- 13 Hyatt, R.M. (1988). A High-Performance Parallel Algorithm to Search Depth-First Game Trees. Ph.D. Thesis, University of Alabama, Birmingham.
- 14 Hyatt, R.M. (1997). The Dynamic Tree-Splitting Parallel Search Algorithm.
- 15 R. Coulom. (2007). Efficient selectivity and backup operators in Monte-Carlo tree search.
- 16 L. Kocsis and C. Szepesvari.(2006). Bandit Based Monte-Carlo Planning.
- 17 Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go.
- 18 Drake, P., Pouliot, A., Schreiner, N., and Vanberg, B. (2007). The Proximity Heuristic and an Opening Book in Monte Carlo Go.
- 19 T. Cazenave and N. Jouandeau. (2007). On the parallelization of UCT.
- 20 Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik, Parallel Monte-Carlo Tree Search