Chapter 2

Search Methods

This chapter describes basic search methods for turn-based games. We focus on the $\alpha\beta$ framework and Monte-Carlo Tree Search. These methods form the basis for the chapters that will follow. First, Section 2.1 defines several notions and concepts which we use throughout this thesis. Section 2.2 explains the minimax algorithm. The popular pruning mechanism $\alpha\beta$ is discussed in Section 2.3. The success of $\alpha\beta$ is strongly dependent on the move ordering (Marsland, 1986). The most prominent move-ordering techniques are explained in Section 2.4. Section 2.5 describes iterative deepening. Transposition tables are discussed in Section 2.6. Finally, Section 2.7 describes Monte-Carlo Tree Search and its enhancements.

2.1 Searching in Games

Russell and Norvig (2003) define a *game* as a search problem with the following components:

- The initial position; it includes the board configuration and an indication whose move it is.
- A set of operators, which define the legal moves that a player can make.
- A terminal test, that determines whether the game is over.
- An evaluation function, returning a value for the outcome of a game.

A *game tree* represents the state space of a game. A *node* in the tree represents a position in the game and an *edge* represents a legal move. A sequence of edges in the game tree forms a *path* if each edge has a node in common with the preceding and succeeding edge. The *root* of the tree is a representation of the initial position. A *terminal position* is a position where the game has ended (a win, a draw, or a loss). A *terminal node* represents a terminal position. A node is *expanded* when all successors are generated according to the game rules. Nodes on the path between the root and a node N are *ancestors* of N. This implies that N is a *descendant* of node M, if M is an ancestor. The nearest ancestor and descendants of a node are

called *parent* and *children*, respectively. Nodes having the same parent are *siblings*. A node is *interior* if it has at least one child.

A game tree is created by expanding all interior nodes. The game tree for the initial position is an explicit representation of all possible games which may be played (Pearl, 1984). The *game-theoretic value* is the value of the root when all participants play optimally. A *minimal game tree* is the minimal part of the game tree required to determine the game-theoretic value. Because for most games the (minimal) game tree is extremely large, determining the game-theoretic value is computationally not feasible.

The *search tree* is that part of the game tree which is analyzed. This tree has the same root but does not contain all nodes of the game tree. Nodes are generated during the *search process*. Nodes that do not have children (yet) are *leaf nodes*. A *subtree* of a tree is a node with all its descendants. The depth of a tree is the largest depth of all its leaves, counted in *plies*. A ply corresponds to a turn of a player (Samuel, 1959). The *search depth* of a node is the number of plies which still need to be searched at this node.

2.2 Minimax

For two-player games, the players are called MAX and MIN. The MAX player tries to maximize the evaluation function while the MIN player tries to minimize it.

The *minimax* algorithm (Von Neumann and Morgenstern, 1944) is designed to find the optimal move for the MAX player, whose turn it is. Every move in the initial position is expanded in a recursive depth-first way until the end of the game is reached, creating a search tree. The evaluation function is used to assign game-theoretic values to each outcome. One step at a time, these values are backed up to the root, where MAX always chooses the highest value and MIN always chooses the lowest value. When all moves have been investigated at the root, the move with highest value is played.

An example of a minimax tree is depicted in Figure 2.1. In this case, a game of Tic-Tac-Toe is played. It is the X player's turn at the root, which takes the role of the MAX player. The moves are explored to the end of the game where the evaluation function is used to assign a value to a terminal position. The values for loss, draw, and win are -1, 0, and 1, respectively. The numbers along the edges indicate in which order nodes are explored.

For non-trivial games, it is usually not possible within a limited amount of time to traverse the complete game tree until terminal positions are reached. Therefore, the game tree is searched until a fixed depth, where a *heuristic* evaluation function indicates the desirability of the position at the leaf node.

Instead of having two separate MAX and MIN methods, it is possible to use a single method, saving a significant number of lines of code. At every ply, the values of the children are negated. This method is called *negamax* (Knuth and Moore, 1975). The pseudo code for negamax can be found in Algorithm 2.1. The variable *turn* is used for distinguishing MAX and MIN nodes, and can have value 1 for a MAX node, and -1 for a MIN node.

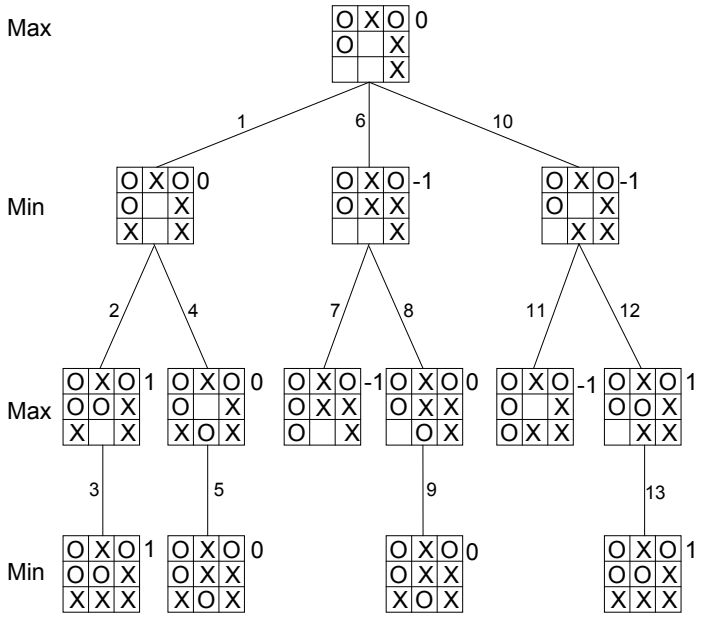

Figure 2.1: An example minimax tree.

```
 1: negamax(position, depth, turn)
 2:
 3: if terminal(position) or depth==0 then
 4:    return turn × eval();
 5: end if
 6:
 7: best = −∞;
 8: for all Moves i do
 9:    doMove(position, i);
10:    best = max(best, −negamax(position, depth−1, −turn));
11:    undoMove(position, i);
12: end for
13:
14: return best;
```

Algorithm 2.1: Negamax pseudo code.

2.3 $\alpha\beta$ Search

It is not necessary to investigate every node of the search tree to determine the minimax value of the root. The process of eliminating branches of the search tree from consideration without examining them is called *pruning*. The most-famous pruning method is $\alpha\beta$ pruning (Knuth and Moore, 1975).

$\alpha\beta$ produces a cutoff in a node if a returned value indicates that the remaining children cannot alter the value of the root. To prune the remaining children, the returned value has to be greater than or equal to the value of its ancestors of the same type (MAX or MIN). Knuth and Moore (1975) proved that in the best case $O(b^{d/2})$ nodes will be expanded, where b is the average branching factor, and d is the search depth.

The pseudo code for $\alpha\beta$ in the negamax framework is shown in Algorithm 2.2. Only small modifications of the minimax pseudo code (Algorithm 2.1) are required. At the root node, α and β are initialized to $-\infty$ and ∞, respectively.

```
 1: alphabeta(position, depth, turn, alpha, beta)
 2:
 3: if terminal(position) or depth==0 then
 4:   return turn × eval();
 5: end if
 6:
 7: for all Moves i do
 8:   doMove(position, i);
 9:   alpha = max(alpha, −alphabeta(position, depth−1, −turn, −beta, −alpha));
10:   undoMove(position, i);
11:   if alpha ≥ beta then
12:     return beta;
13:   end if
14: end for
15:
16: return alpha;
```

Algorithm 2.2: $\alpha\beta$ pseudo code.

Figure 2.2 depicts an example of $\alpha\beta$ pruning (the initial position is identical to the example in Figure 2.1). In this tree, both players have an optimal move ordering, i.e., the strongest move is always investigated first. $\alpha\beta$ is able to prune twice in this example. After node A has been searched, the value of the root is ≥ 0, because the other two moves may have a higher value. After edge 7 has been investigated, the value of the node B is ≤ -1 (actually equals -1 because it is the smallest possible value). The MAX player now always prefers A above B. Therefore, all other moves under B are irrelevant to the value of the root and do not require to be searched anymore. With the same reasoning, the right subtree of C can be pruned.

We remark that three types of nodes can be distinguished in $\alpha\beta$ search (Knuth and Moore, 1975; Marsland and Popowich, 1985). (1) *PV nodes* form the *principal variation* of the search (i.e., the expected line of play). At a PV node, all the children have to be investigated. The best move found in a PV node leads to a successor PV node, while all other investigated children are CUT nodes. (2) At *CUT nodes* a cutoff takes place. Ideally, only one child has to be explored. At a CUT node the child causing the cutoff is an ALL node. (3) In an *ALL node* all children have to be explored. The successors of an ALL node are CUT nodes.

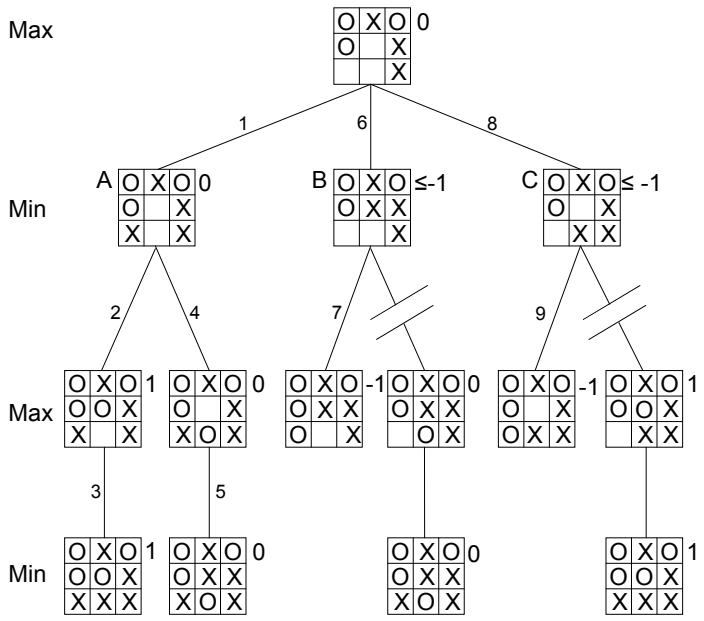

Figure 2.2: An example $\alpha\beta$ tree.

2.4 Move Ordering

The success of $\alpha\beta$ search is strongly dependent on the move ordering (Marsland, 1986). Therefore, researchers have investigated methods to examine the strongest move first. We distinguish two types of move ordering. (1) *Static* move ordering is independent of the search. These techniques rely on domain knowledge (e.g., capturing a queen in chess) or by learning techniques (e.g., the Neural MoveMap Heuristic, Kocsis, Uiterwijk, and Van den Herik, 2001). (2) *Dynamic* move ordering is dependent on the search. These techniques rely on information gained during the search. We describe two dynamic move-ordering techniques in the following subsections. Subsection 2.4.1 gives an introduction to the killer heuristic. In Subsection 2.4.2 the history heuristic is explained.

2.4.1 Killer Heuristic

The basic assumption of the *killer heuristic* (Akl and Newborn, 1977) is that a move which produces a cutoff in one position, often produces a cutoff in a similar position, if the move is legal. The killer heuristic stores for every ply at least one killer move which has produced a pruning before at a node in the corresponding ply. When a new node is searched, the killer moves for that ply are examined first if they are legal. A cutoff may occur even before all possible moves are generated. When a move produces a pruning which is not a killer move for that ply, it becomes a new

killer move, overwriting an old one. The killer move selected for deletion, is the one that has not been used for the longest time.

2.4.2 History Heuristic

The *history heuristic* is a dynamic move-ordering technique based on the number of cutoffs caused by a move, irrespective of the position in which the move has been made. It was proposed by Schaeffer (1983) and has been adopted in several game-playing programs.

For every move seen in the game tree a history is maintained. Because there is only a limited number of legal moves, it is possible to maintain a score for each move in n tables (n is the number of players). Moves are typically indexed by their coordinates on the board, independent of how the rest of the board looks like (e.g., 64 from squares $\times$ 64 to squares for chess). When an interior node is searched, the history-table entry for the best move found is incremented by some value (e.g., 2^d, where d is the search depth of the subtree under the node). This move either caused an $\alpha\beta$ cutoff or had the best score after all moves had been searched.

When in a node all possible moves have been generated, they are ordered according to their history-heuristic scores. This may lead to more $\alpha\beta$ cutoffs. The scores in the tables can be maintained during the whole game. Each time a new search is started, the scores are decremented by a factor (e.g., divided by 2). They are only reset to zero or to some default values at the beginning of a complete new game. The two drawbacks of the history heuristic are (1) that it assumes that all moves occur equally often and (2) for illegal moves memory is reserved in the tables (Hartmann, 1988).

There exist three variations on the history heuristic. (1) To counter somewhat the two disadvantages Hartmann (1988) proposed an alternative for the history heuristic, the *butterfly heuristic*. This heuristic takes the move frequencies in the search trees into account. Instead of history tables, *butterfly boards* are used. They are defined in the same way as in the history heuristic. Any move that is not cut is recorded. Each time a move is executed in the search tree, its corresponding entry in the butterfly board is also incremented by a value. (2) The countermove heuristic (Uiterwijk, 1992) is based on the assumption that many moves have a "natural" response irrespective of the actual position in which the moves occur. (3) Winands *et al.* (2006) proposed the *relative history heuristic* which divides the history-heuristic score by the butterfly-heuristic score. Their experiments reveal that it reduces the number of nodes searched in the games of Lines of Action (LOA) and Go even more.

2.5 Iterative Deepening

Usually $\alpha\beta$ investigates the search tree up to a predefined depth. However, it is not straightforward to predict the running time. *Iterative deepening* overcomes this problem by gradually increasing the search depth, typically by one ply per iteration. This is done until time runs out. Although this may seem inefficient, the overhead is rather small. We provide the following example. To finish a search of depth d and

average branching factor b, b^d nodes are required. When using iterative deepening, the root is expanded d times, the 1-ply deep nodes $d-1$ times, up to the d-ply deep nodes, which are only expanded once. Therefore, iterative deepening expands $\sum_{i=0}^{d}(d+1-i)\times b^i$ nodes. With an average branching factor of 10 and search depth 5, the regular search expands $10^5 = 100,000$ nodes and iterative deepening expands $6+50+400+3,000+20,000+100,000 = 123,456$ nodes. The larger the branching factor, the smaller the overhead of repeatedly expanded nodes. In spite of this overhead, iterative deepening is able to reduce the number of nodes searched (Slate and Atkin, 1977). The reason for this is that information from previous iterations is used by the killer and history heuristic and is re-used to improve the quality of the move ordering at the next iteration. Also the transposition table benefits from iterative deepening, which is introduced in the next section.

2.6 Transpositions

In many games, there exist multiple paths to the same position. In these games, it is possible to reduce search effort by storing earlier search results. If an identical position is encountered during the search, it is called a *transposition*. This implies that the search tree can be considered a *graph*, because of transpositions. In a transposition, the information of the previous search may be reused to reduce search effort. While the position may be identical, this history of the game leading to the position is not. For some games (e.g., chess), a three-times repetition draw rule makes the identity of a position dependent on the history (Slate and Atkin, 1977). This is known as the *graph-history-interaction* (GHI) problem (Palay, 1983; Campbell, 1985).

Transposition tables are explained in Subsection 2.6.1. Subsection 2.6.2 describes Enhanced Transposition Cutoffs.

2.6.1 Transposition Tables

When a position has been investigated, positional information is stored in a *transposition table* (Greenblatt, Eastlake, and Crocker, 1967). Because memory limitations do not allow every position to be stored, the transposition table is implemented as a finite *hash table* (Knuth, 1973). A *hash value* is computed for the position using some hashing method, such as Zobrist (1970) hashing. If the transposition table consists of 2^n entries, the n low-order bits of the hash value are used as a *hash index*. The remaining bits (the *hash key*) are used to distinguish between positions with the same hash index. The size of the hash index should be chosen sufficiently large (Hyatt, Grover, and Nelson, 1990).

A typical entry consists of the following five components. (1) The hash key. (2) The best move for this position. This move either obtained the highest score or produced a cutoff. (3) The score of the best move. In $\alpha\beta$ search, this may be an exact value, an upper or a lower bound. (4) A *flag* that indicates whether the score is an exact value, upper or lower bound. (5) A search depth that indicates how deep this position has been investigated.

When a transposition is encountered, the information in the transposition table may be used in three ways, depending on the depth and flag stored in the entry. Let the depth stored in the transposition table be ttd and the depth still to be searched d. (1) If $ttd \geq d$ and an exact value is available, the search is aborted and the transposition table score is returned. (2) If $ttd \geq d$ and the score is not exact, but an upper or a lower bound, the current $\alpha\beta$ window is adapted accordingly and the stored move is used for move ordering. (3) If $ttd < d$, the stored move is used for move ordering.

If two different positions have the same hash value, a so-called *type-1 error* occurs. This is a serious error which is difficult to detect. Let N be the number of distinguishable positions and M be the number of different positions which have to be stored in the transposition table. The probability that all M positions will have different hash values is given by

$$P\left(no\ \ errors\right) \approx e^{-\frac{M^2}{2N}} \tag{2.1}$$

If the size of the transposition table is sufficiently large, the probability of a type-1 error is negligible (Breuker, 1998).

If two different positions have the same hash index, but different hash key, a so-called *type-2 error* occurs. This is also known as *collision* (Knuth and Moore, 1975). When a collision occurs, one has to choose which position to store. This choice is based on a *replacement scheme* (Breuker, Uiterwijk, and Van den Herik, 1994; Breuker, Uiterwijk, and Van den Herik, 1996). For an overview of replacement schemes, we refer to Beal and Smith (1996) and Breuker (1998). The most commonly used scheme is the two-deep replacement scheme (Breuker, 1998).

2.6.2 Enhanced Transposition Cutoff

Enhanced Transposition Cutoff (ETC) (Schaeffer and Plaat, 1996) is a method to produce a cutoff using the transposition table, even though an entry does not exist for the current position. Children of the current position may be stored in the transposition table and one of these may produce a cutoff without further expanding the current node. After the moves of the current positions have been created, all children are queried in the transposition table. If a child is stored in the table, the α value of the parent may be updated, possibly resulting in a cutoff. Because calculating a hash value for a position and examining in the transposition table takes time, ETC is only applied if the search depth is large enough.

2.7 Monte-Carlo Tree Search

Recently, Monte-Carlo (MC) methods have become a popular approach for intelligent play in games. MC simulations have first been used as an evaluation function inside a classic search tree (Brügmann, 1993; Bouzy and Helmstetter, 2003). In this role, MC simulations have been applied to Backgammon (Tesauro and Galperin, 1997), Poker (Billings *et al.*, 1999), Scrabble (Sheppard, 2002b), Morpion Solitaire

and Gaps (Helmstetter, 2007). Due to the expensive evaluation, the search is not able to investigate the search tree sufficiently deep in some games (Chaslot, 2010).

Therefore, the MC simulations have been placed into a tree-search context in multiple ways (Chaslot *et al.*, 2006b; Coulom, 2007a; Kocsis and Szepesvári, 2006). The resulting general method is called Monte-Carlo Tree Search (MCTS) (Coulom, 2007a; Kocsis and Szepesvári, 2006). MCTS is a best-first search method guided by Monte-Carlo simulations. In contrast to classic tree-search algorithms such as $\alpha\beta$ (Knuth and Moore, 1975) and A* (Hart *et al.*, 1968), MCTS does not require a heuristic evaluation function. MCTS is particularly interesting for domains where building an evaluation function is a difficult or time-consuming task, such as the game of Go. MCTS builds a search tree employing MC simulations at the leaf nodes. Each node in the tree represents a position and typically stores the average score of the simulations played through this node, and the number of visits. MCTS constitutes a family of tree-search algorithms not only applicable to games, but also to scheduling and optimization problems (cf. Chaslot *et al.*, 2006a; Mesmay *et al.*, 2009)

Subsection 2.7.1 describes the structure of MCTS. Thereafter, Subsection 2.7.2 explains three enhancements for MCTS: Progressive Bias, Progressive Widening and RAVE. Finally, Subsection 2.7.3 discusses how MCTS may be parallelized.

2.7.1 Structure of MCTS

MCTS consists of two parts: a relatively shallow search tree and deep simulated games. The tree structure determines the first moves of the simulated games. The results of these simulated games shape the tree. In general, MCTS consists of four steps (Chaslot *et al.*, 2008d). These steps are visualized in Figure 2.3. (1) During the selection step the search tree is traversed starting from the root node until a position is encountered which is not stored in the tree. (2) Subsequently, during the play-out step moves are played until the end of the game. (3) Next, in the expansion step a number of nodes is added to the tree. (4) Finally, in the backpropagation step the result of a simulated game is propagated backwards, through the previously traversed nodes. We discuss strategies to perform the four basic steps of MCTS in the remainder of this subsection.

The four steps are iterated until the time runs out. When this happens, a final move selection is used to determine which move should be played in the actual game.

Selection Step

From the root node, a *selection strategy* is applied recursively until a position is reached that is not a part of the tree yet. The selection strategy controls the balance between *exploitation* and *exploration* (Chaslot, 2010). On the one hand, the moves that lead to the best results so far are selected (exploitation). On the other hand, the less promising moves still must be tried, due to the uncertainty of the evaluation (exploration).

Several selection strategies have been designed for MCTS. Kocsis and Szepesvári (2006) proposed the selection strategy UCT (Upper Confidence bounds applied to

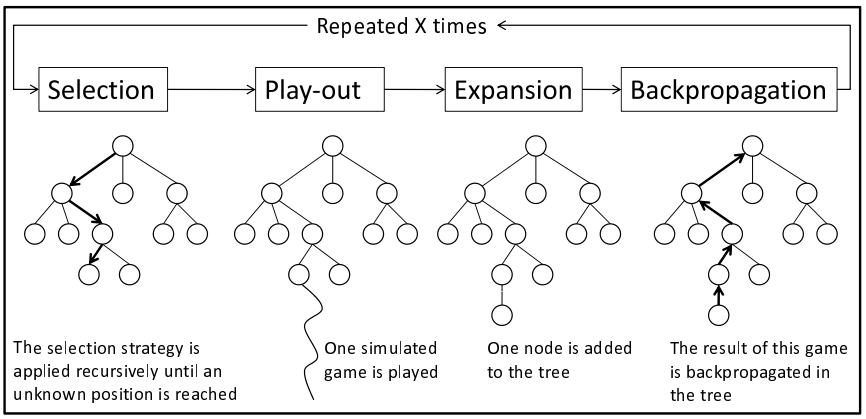

Figure 2.3: The four steps of MCTS (slightly adapted from Chaslot *et al.*, 2008d).

Trees). This strategy is straightforward to implement, and used in many programs. At node p with children i, the child is chosen that maximizes the following formula.

$$v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \tag{2.2}$$

v_i is the value of the child i, n_i is the visit count of i, and n_p is the visit count of node p. C is a coefficient, which has to be tuned experimentally. Other selection strategies are OMC (Chaslot *et al.*, 2006b), UCB1-TUNED (Gelly and Wang, 2006), PBBM (Coulom, 2007a) and MOSS (Audibert and Bubeck, 2009).

Here we remark that Coulom (2007a) chooses a move according to the selection strategy only if n_p is above a certain threshold. Before the threshold is reached, the simulation strategy is used. The latter is explained below.

Play-Out Step

The play-out step begins when the search enters a position that is not part of the tree yet. Moves are then played until the end of the game. These moves are chosen according to a *simulation strategy*. A completed game is called a *play-out*. The play-outs are an estimate for the values of the nodes in the MCTS tree. The use of an adequate simulation strategy (instead of playing randomly) has been shown to improve the level of play significantly (Bouzy, 2005; Gelly *et al.*, 2006; Chen and Zhang, 2008).

The moves may be chosen quasi-randomly based on heuristic knowledge (Gelly and Silver, 2007). A simulation strategy is subject to two tradeoffs (Chaslot, 2010). The first tradeoff is between search and knowledge. While knowledge increases the playing strength, it decreases the simulation speed. The second tradeoff deals with exploration and exploitation. If the strategy is too explorative, the search tree is too shallow and if the strategy is too exploitative, the search will become too selective. Ideally, the used heuristic knowledge should be fast to compute, increase the quality

of the play-out significantly and at the same time allow enough randomness for exploration.

Expansion Step

Usually the whole game tree cannot be stored in memory. An *expansion strategy* decides whether a node is expanded by storing one or more of its children in memory. Coulom (2007a) proposed to store the first encountered position that was not stored in the tree, expanding one node per simulation.

Backpropagation Step

During the backpropagation step, the result of the simulation at a leaf node is propagated backwards to the root. A *backpropagation strategy* decides how the value of a node is used to change the values of its ancestors.

There exist several backpropagation strategies. The most-applied one is Average, which takes the average of all simulations made through that node (Coulom, 2007a). The Max strategy (Knuth and Moore, 1975) backpropagates values in a negamax way, but was proven to be too noisy (Chaslot *et al.*, 2006b; Coulom, 2007a). The Informed-Average strategy (Chaslot *et al.*, 2006b) aims to converge faster to the value of the best move than Average by assigning a bigger weight to the best moves. Coulom (2007a) proposed the Mix strategy, which initially resembles the Average strategy, but converges to the Max strategy. The MCTS-Solver strategy (Winands, Björnsson, and Saito, 2008) also backpropagates game-theoretic values, allowing MCTS to play narrow tactical lines better in sudden-death games.

2.7.2 Enhancements

This subsection introduces three well-known enhancements for MCTS. First, progressive bias is introduced. Second, progressive widening is discussed. Finally, an introduction to RAVE is given.

Progressive Bias

The aim of the *progressive-bias* strategy is to direct the search according to heuristic knowledge (Chaslot *et al.*, 2008d). For that purpose, the selection strategy is modified according to that knowledge. The influence of this modification is important when a few games have been played, but decreases with a growing number of games. The UCT formula (Equation 2.2) is adapted in the following way.

$$v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + \frac{H_i}{n_i + 1} \tag{2.3}$$

where H_i represents heuristic knowledge, which depends only on the board configuration represented by the node i. When n_i is small, the heuristic factor is most dominant. With increasing n_i, a balance is found between the results of the simulated games and the heuristic knowledge. When n_i is large, the results of the simulated games are the dominant factor.

Progressive Widening

The heuristic score H_i, used by progressive bias, may also be used for a method called *progressive widening* (Coulom, 2007b; Chaslot *et al.*, 2008d). When the number of simulations is small, the simulation strategy is used as selection strategy. Progressive widening reduces the branching factor in this phase based on the heuristic knowledge H_i and gradually increases the branching factor when more simulations are performed.

Rapid Action-Value Estimation

Brügmann (1993) proposed to acquire results from simulations quicker by the *all-move-as-first heuristic* (AMAF). AMAF assigns the result of a simulated game not only to the first move played, but to all moves during the game. $\text{AMAF}_{s,m}$ is the AMAF value for move m in position s. Gelly and Silver (2007) proposed a method called *Rapid Action-Value Estimation* (RAVE), that uses the AMAF value in combination with MCTS. The UCT formula (Formula 2.2) is adapted in the following way.

$$(1 - \beta(n_p)) \times (v_i + C \times \sqrt{\frac{\ln n_p}{n_i}}) + \beta(n_p) \times AMAF_{p,i} \tag{2.4}$$

Gelly and Silver (2007) proposed to use $\beta(N) = \sqrt{\frac{k}{3N+k}}$ with k subject to tuning, which has led to good results. RAVE was successfully applied in Go (Gelly and Silver, 2007) and Havannah (Teytaud and Teytaud, 2010). In games where pieces are continually moving across the board, such as Chinese Checkers, it is more difficult for RAVE to work effectively (Sturtevant, 2008b).

2.7.3 Parallelization

Just as for $\alpha\beta$ search, it holds for MCTS that the more time is spent for selecting a move, the better the game play is. The recent developments in hardware have gone into the direction that nowadays even personal computers contain several cores. To get the most out of the available time, one has to parallelize AI techniques to use all available hardware (Chaslot, 2010).

Cazenave and Jouandeau (2007) proposed two parallelization methods, leaf and root parallelization, which will be introduced first. Thereafter, tree parallelization (Chaslot, Winands, and Van den Herik, 2008b) will be described.

Leaf Parallelization

Leaf parallelization (Cazenave and Jouandeau, 2007) is a straightforward way to parallelize MCTS. One thread traverses the tree just as in regular MCTS. Next, starting from the leaf node, one play-out is performed for each available thread. When all games are finished, the results of all these play-outs are propagated backwards through the tree as usual.

Leaf parallelization has two problems. (1) Play-outs have a variable length, and the search has to wait until the longest game has finished. (2) Information between the play-outs is not shared. When the first play-outs indicate that the node most likely leads to a loss, the remaining play-outs may be a waste of computational time.

Root Parallelization

Root parallelization (Cazenave and Jouandeau, 2007) consists of building multiple MCTS trees in parallel, with one thread per tree. The threads do not share information with each other. Each individual thread performs a regular MCTS. When the available time is spent, the separate trees are combined and the choice of which move to play is based on all results. Root parallelization is a straightforward and efficient way to parallelize MCTS. Chaslot *et al.* (2008b) show that root parallelization nearly has a linear speedup for a small number of threads.

Tree Parallelization

Tree parallelization (Chaslot *et al.*, 2008b) is a parallelization method in which threads share information with each other. This method uses one shared tree from which several simultaneous play-outs are played. Each thread can modify the information contained in the tree. Mutexes are required to lock certain parts of the tree to prevent data corruption. Based on the location of the mutexes in the tree, we distinguish two mutex-location methods: (1) using a *global mutex* and (2) using several *local mutexes*. With a global mutex the complete tree is locked when a thread edits information. With a local mutex only the node is locked that requires updating. This makes it possible that multiple threads edit the tree simultaneously, decreasing waiting time. Enzenberger and Müller (2010) showed that a mutex-free tree parallelization outperformed a global-mutex tree parallelization for the Go program Fuego. To prevent that all threads select the same node in the tree, a *virtual loss* may be assigned to a node if a thread selects it (Chaslot *et al.*, 2008b). The virtual loss is removed when the thread that gave the virtual loss starts backpropagating the result of the play-out.