PROGRESSIVE STRATEGIES FOR MONTE-CARLO TREE SEARCH

GUILLAUME M.J-B. CHASLOT, MARK H.M. WINANDS,
H. JAAP VAN DEN HERIK and JOS W.H.M. UITERWIJK
*MICC-IKAT, Games and AI Group, Faculty of Humanities and Sciences,
Universiteit Maastricht, P.O. Box 616, 6200 MD Maastricht, The Netherlands
{g.chaslot, m.winands, uiterwijk, herik}@micc.unimaas.nl*

BRUNO BOUZY
*Centre de Recherche en Informatique de Paris 5, Université Paris 5 Descartes,
45, rue des Saints Pères, 75270 Cedex 06, France
bouzy@math-info.univ-paris5.fr*

Monte-Carlo Tree Search (MCTS) is a new best-first search guided by the results of Monte-Carlo simulations. In this article we introduce two *progressive strategies* for MCTS, called progressive bias and progressive unpruning. They enable the use of relatively time-expensive heuristic knowledge without speed reduction. Progressive bias directs the search according to heuristic knowledge. Progressive unpruning first reduces the branching factor, and then increases it gradually again. Experiments assess that the two progressive strategies significantly improve the level of our Go program Mango. Moreover, we see that the combination of both strategies perform even better on larger board sizes.

1. Introduction

Over fifty years, two-person zero-sum games with perfect information have been addressed by many AI researchers with great success.[16] The classical approach is to use the alpha-beta framework, combined with a positional evaluation function. Such an evaluation function is applied to the leaf nodes of a search tree. If the node represents a terminal position (or a databased position) it produces an exact value. Otherwise, heuristic knowledge is used to estimate the value of the leaf node. This technique led to excellent results in many games (e.g., Chess and Checkers).[7,19]

However, in several games building an evaluation function based on heuristic knowledge for a non-terminal position is a difficult and time-consuming issue; the most notorious example is the game of Go.[2] It is probably one of the reasons why Go programs so far did not achieve a strong level, despite intensive research and additional use of knowledge-based methods.

Recently, researchers proposed to use Monte-Carlo simulations as an evaluation function.[5,6] Yet, this approach remained too slow to achieve a satisfying search depth. Even more recently, three slightly different uses of Monte-Carlo simulations within a tree-search context have been proposed.[9,12,17] The new general method, which we call "Monte-Carlo Tree Search" (MCTS) resulted from it. MCTS is not a classical tree search followed by a Monte-Carlo evaluation, but rather a best-first search guided by the results of Monte-Carlo simulations. This method uses two main strategies, which aim at different purposes described below. (1) A *selection strategy*, derived from the Multi-Armed Bandit problem, is able to increase the quality of the chosen moves in the tree when the number of simulations grows.[10,14] Yet, the strategy requires the results of several previous simulations. (2) When a sufficient amount of results is not available, a *simulation strategy* decides on the moves to be played.[1,14]

In this article, we propose two progressive strategies as a soft transition between the simulation strategy and the selection strategy. The strategies enable, among others, the use of time-consuming heuristic knowledge. Below, we use the game of Go as test domain. Go is challenging, because so far programs are not able to defeat expert humans, and thus it has been a testbed for artificial-intelligence techniques for over 30 years.

The article is organized as follows. In Section 2, we present the Monte-Carlo Tree Search method. In Section 3, we describe two progressive strategies. Section 4 presents the experiments, performed by our Go program MANGO. Section 5 summarizes the contributions, formulates conclusions, and gives an outlook on future research.

2. Monte-Carlo Tree Search

MCTS is a best-first search method which does not require a positional evaluation function. It is based on a randomized exploration of the search space: in the beginning of the search, exploration is performed fully at random. Then, using the results of previous explorations, the algorithm becomes able to predict the most promising moves more accurately, and thus, their evaluation becomes more accurate. The basic structure of MCTS is given in Subsection 2.1. Relevant pseudo-code is provided in Subsection 2.2. The four strategic tasks are discussed in Subsection 2.3.

2.1. *Structure of MCTS*

In MCTS, each node i represents a given position (also called a state) of the game. A node contains at least the following two pieces of information: (1) the current value v_i of the position (usually the average of the results of the simulated games that visited this node), and (2) the visit count of this position n_i. MCTS usually starts with a tree containing only the root node.

MCTS consists of four steps, repeated as long as there is time left. The steps are as follows. (1) The tree is traversed from the root node to a leaf node (L), using a *selection strategy.* (2) An *expansion strategy* is called to store one (or more) children of L in the tree. (3) A *simulation strategy* plays moves in self-play until the end of the game is reached. The result R of this "simulated" game is $+1$ in case of a win for Black (the first player in Go), and -1 in case of a win for White. (4) R

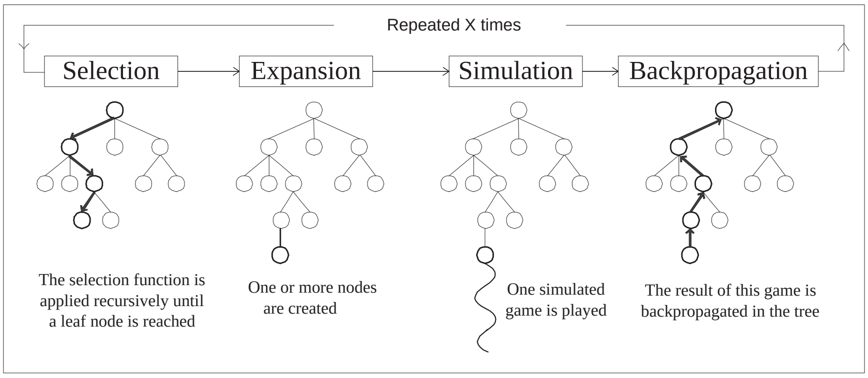

Fig. 1. Outline of a Monte-Carlo Tree Search.

is backpropagated in the tree according to a *backpropagation strategy*. Finally, the move played by the program is the child of the root with the highest visit count. The four steps of MCTS are explained in some detail in Figure 1, and more elaborated in 2.3.

2.2. *Relevant pseudo-code*

The pseudo-code for MCTS is given in Figure 2. In this algorithm, $\mathcal{T}$ is the set of all nodes (the search tree), *Select(Node N)* is the selection function, which returns one child of the node N. *Expand(Node N)* is the function that stores one more node in the tree, and returns this node. *Play_simulated_game(Node N)* is the function which plays a simulated game from the node N, and returns the result $R \in \{-1, 1\}$ of this game. *Backpropagate(Integer R)* is the procedure that updates the value of the node depending on the result R of the last simulated game. $\mathcal{N}_c$*(node N)* is the set of the children of the node N.

2.3. *The four strategic tasks*

As has been mentioned earlier, the four strategic tasks in MCTS are selection, expansion, simulation, and backpropagation. They are each discussed in detail below. Then, we will show how we use them in our Go program Mango.

2.3.1. *Selection*

Selection is the strategic task that selects one of the children of a given node. It controls the balance between exploitation and exploration. On the one hand, the task is often to select the move that led to the best results so far (exploitation). On the other hand, the least promising moves still have to be explored, due to the uncertainty of the evaluation (exploration). Similar balancing of exploitation and exploration has been studied in the literature, in particular with respect to the

```
void MCTS(Node root_node)

while(has_time)
{
    current_node ← root_node
    while (current_node ∈ T)
    {
        last_node ← current_node
        current_node ← Select(current_node)                          // Selection
    }
    last_node ← Expand(last_node)                                    // Expansion
    R ← Play_simulated_game(last_node)                               // Simulation
    while(current_node ∈ T)
    {
        current_node.Backpropagate(R)                                // Backpropagation
        current_node.visit_count ← current_node.visit_count + 1
        current_node ← current_node.parent
    }
}
return best_move = argmax_{N∈N_c(root_node)} N.visit_count
```

Fig. 2. Pseudo-code for Monte-Carlo Tree Search.

Multi-Armed Bandit (MAB) problem.[18] The MAB problem considers a gambling device and a player, whose objective is maximizing the reward from the device. At each time step, the player can select one of N arms of the device, which gives a reward. In most settings, the reward obeys a stochastic distribution. The selection problem of Monte-Carlo Tree Search could be viewed as a MAB problem for a given node: the problem is to select the next move (arm) to play, which will give an unpredictable reward (the outcome of a single random game). Knowing the past results, the problem is to find the optimal move. However, the main difference with the MAB problem is that Monte-Carlo Tree Search works by super-imposing several selections: the selection at the root node, the selection at depth one, the selection at depth two, etc. Several algorithms have been designed for this setup,[9,10,12] or have been adapted from MAB algorithms.[14,17]

Selection strategy used in MANGO

We use the strategy UCT (Upper Confidence bound applied to Trees).[17] This strategy is easy to implement, and used in many programs. UCT works as follows. Let I be the set of nodes reachable from the current node p. UCT selects the child k of the node p that satisfies formula 2.1:

$$k = argmax_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \right) \tag{2.1}$$

where v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a coefficient, which has to be tuned experimentally. In practice, this method is only applied in nodes of which the visit count is higher than a certain threshold T.[12] If the node has been visited fewer times than this threshold, the next node is selected according to the *simulation strategy*, discussed in 2.3.3.

2.3.2. *Expansion*

Expansion is the strategic task that, for a given leaf node L, decides whether this node will be expanded by storing one or more of its children in memory. The simplest rule is to expand one node per simulated game.[12] The expanded node corresponds to the first position encountered that was not stored yet.

Expansion strategy used in Mango
In addition to expanding one node per simulated game, we also expand all the children of a node when a node's visit count equals T.

2.3.3. *Simulation*

Simulation (also called *playout*) is the strategic task that selects moves in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves chosen according to a *simulation strategy.* Indeed, the use of an adequate simulation strategy has been shown to improve the level of play significantly.[1,15] The main idea is to play interesting moves by using patterns, capture considerations, and proximity to the last move. The simulation requires that the number of moves per game is limited. When considering the game of Go, extra rules are added to satisfy this condition: (1) a player should not play in his eyes, and (2) the game is stopped if it exceeds a given number of moves. Elaborating an efficient simulation strategy is a difficult issue. If the strategy is too stochastic (e.g., if it selects moves nearly randomly), then the moves played are often weak, and the level of the Monte-Carlo program is decreased. In contrast, if the strategy is too deterministic (e.g., if the selected move for a given position is almost always the same) then the exploration of the search space is too selective, and the level of the Monte-Carlo program is decreased too.

Simulation strategy used in Mango
Each possible move $j \in \mathcal{M}$ is given an urgency $U_j \geq 1$. The simulation strategy selects one move from $\mathcal{M}$. Each move's probability to be selected is $p_j = \frac{U_j}{\sum_{k \in \mathcal{M}} U_k}$. The urgency is the sum of three values.

(1) *Capture value.* Moves that capture stones or that prevent a capture are given a larger urgency, which depends on the number of captured or saved stones. Using a capture value was first proposed by Bouzy.[1]
(2) *3x3 Pattern value.* For each possible 3×3 pattern, the value of the central move has been learned by a dedicated algorithm.[4]

(3) *Proximity modification.* Moves adjacent to the previous move have their urgency multiplied by a large factor. This is similar to the strategy developed by Gelly and Wang.[15]

2.3.4. *Backpropagation*

Backpropagation is the procedure which backpropagates the *result* of a simulated game k from leaf node i to the nodes it had to traverse to reach this leaf node. This result is counted positively ($R_k = +1$) if the game was won, and negatively ($R_k = -1$) if the game was lost. Draws lead to a result $R_k = 0$. The *value* v_i of a node is computed by taking the average of the results of all simulated games made through this node, i.e., $v_i = \sum_k R_k$. Several backpropagation strategies have been proposed in the literature. However, the best results in game play have been obtained by using the plain average of the simulations.[8,10,12]

Backpropagation strategy used in MANGO
In MANGO we use the plain average strategy described above.

3. Progressive Strategies

When a node has been visited only a few times, a well-tuned simulation strategy chooses moves more accurately than a selection strategy. However, when a node has been visited quite often, the selection strategy is more accurate, because it is able to improve by the number of games played.[9,10,12,17]

We propose a "progressive strategy" that performs a soft transition between the simulation strategy and the selection strategy. Such a strategy uses (1) the information available for the selection strategy, and (2) some time-expensive domain knowledge. A progressive strategy is similar to a simulation strategy when a few games have been played, and converges to a selection strategy when numerous games have been played.

In the following two subsections we describe the two progressive strategies used in our Go-playing program MANGO: *progressive bias* (Subsection 3.1) and *progressive unpruning* (Subsection 3.2). Subsection 3.3 describes the heuristic domain knowledge used in MANGO. Subsection 3.4 discusses the time efficiency of these heuristics.

3.1. *Progressive bias*

The aim of the *progressive bias* strategy is to direct the search according to – possibly time-expensive – heuristic knowledge. 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 fast (when more games have been played) to ensure that the strategy converges to a selection strategy. We modified the UCT selection in the following way. Instead of selecting the move which satisfies formula 2.1, we select the node k which satisfies formula 3.2. We call this formula our enhancement.

$$k = argmax_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + f(n_i) \right) \tag{3.2}$$

In MANGO, we chose $f(n_i) = \frac{H_i}{n_i+1}$, where H_i is a coefficient representing heuristic knowledge, which depends only on the board configuration represented by the node i. The variables n_p and n_i, and coefficient C are the same as in Section 2. More details on the construction of H_i are in Subsection 3.3. Formula 3.2 has the following four properties.

(1) When the number of games n_p made through a node p equals T (30 in MANGO), the selection algorithm starts to be applied in this node. For all the children i of this node with $n_i = 0$, $\sqrt{\frac{\ln n_p}{n_i}}$ is replaced by a fixed number M satisfying $\forall i, M \gg v_i$. v_i is replaced by 0 when $n_i = 0$. Thereafter, the algorithm selects every unexplored child once. The order in which these children are selected is given by $f(n_i)$, i.e., the children with the highest heuristic values are selected first.
(2) If only a few simulations have been made through the node (e.g., from around 30 to 100 in MANGO), and if the heuristic value H_i is sufficiently high, the term $\frac{H_i}{n_i+1}$ is dominant. Hence, the number of simulations made depends more on the domain knowledge H_i than on the results of the simulated games. It is an advantage to use mainly the domain knowledge at this stage, because then the results of only a few simulated games are affected by a large uncertainty. The behavior of the algorithm is therefore close to the behavior of a simulation strategy.
(3) When the number of simulations increases (e.g., from around 100 to 500 in MANGO), both the results of the simulated games and the domain knowledge have a balanced impact on the selection.
(4) When the number of simulations is high (e.g., > 500 in MANGO), the influence of the domain knowledge is low compared to the influence of the previous simulations, because the domain knowledge decreases by $O(1/n_i)$, and the term corresponding to the simulation decreases by $O(\sqrt{\ln n_p / n_i})$. The behavior of the algorithm is, at this point, close to the behavior of a classical selection strategy (UCT). The only difference with plain UCT occurs if two positions i and j have the same value $v_i = v_j$, but different heuristic evaluations H_i and H_j. Then, the position with the highest heuristic evaluation will be selected more often.

An alternative enhancement has been proposed by Gelly and Silver.[13] It consists of introducing prior knowledge. The selected node k is the one, which satisfies formula 3.3:

$$k = argmax_{i\in I}\left(\frac{v_i \cdot n_i + n_{prior} \cdot Q_i}{n_i + n_{prior}} + C \times \sqrt{\frac{\ln n_p}{n_i + n_{prior}}}\right) \quad (3.3)$$

where Q_i is the prior estimation of the position. Gelly and Silver use a reinforcement learning algorithm, which learned the value from self-play on the 9×9 board. n_{prior} is a coefficient that was tuned experimentally.

On the 9×9 board, this technique successfully increased MoGo's winning percentage against GNU Go from 60 percent to 69 percent. However, learning the prior value Q_i was only done for the 9×9 board. So, the scalability of this approach to larger board sizes is an open question.

3.2. *Progressive unpruning*

We have seen in Mango that when there is not much time available and simultaneously the branching factor is high, MCTS performs poorly. Our solution, *progressive unpruning*, consists of (1) reducing the branching factor artificially when the selection function is applied, and (2) increasing it progressively as more time becomes available. When the number of games n_p in a node p equals the threshold T, progressive unpruning "prunes"[a] most of the children. The children, which are not pruned, are the k_{init} children with the highest heuristic values. In Mango k_{init} was set to 5. The children of the node i are progressively "unpruned". In Mango, k nodes are unpruned when the number of simulations in the parent surpasses $A \times B^{k-k_{init}}$ simulated games. A was set experimentally to 40 and B to 1.3. The scheme is showed in Figure 3. We would like to remark that a similar scheme, called *progressive widening*, has been proposed simultaneously by Coulom.[11]

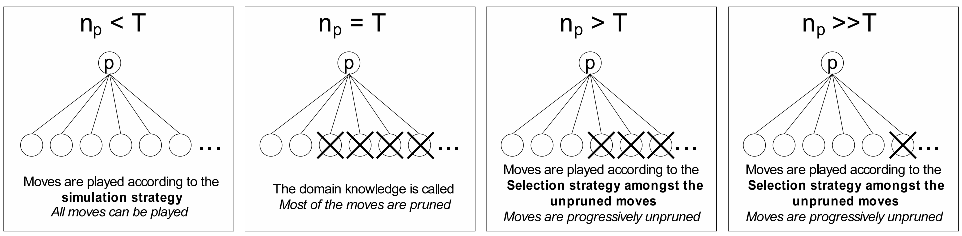

Fig. 3. Progressive unpruning in Mango.

3.3. *Heuristic knowledge used in Mango*

The two previous soft-transition strategies require to compute a heuristic value H_i for a given board configuration representing the node i. In this subsection we

[a]A node is pruned if it cannot be accessed in the simulated games.

describe the heuristic, which is based on the same ideas as seen in 2.3.3. However, the heuristic knowledge for H_i is much more elaborate than the one used for the urgency value U_i. In MANGO, H_i is composed of three elements: (i) a pattern value, (ii) a capture value, and (iii) the proximity to the last moves.

The pattern value is learned offline by using the pattern matching described by Bouzy and Chaslot.[3] This pattern matching was also implemented in the Go program INDIGO, and improved its level significantly.[b] In this research, each pattern assigns a value to the move that is in its center. The value corresponds to the probability that the move is played in professional games. The learning phase has been performed on $2,000$ professional games; $89,119$ patterns were learned. Each pattern contained between 0 stones (e.g., corner pattern) and 15 stones (e.g., joseki pattern). The size of the patterns was not bounded, so some patterns covered nearly the whole board, and some covered only a few intersections.

The capture value of each move depends on (1) the number of stones that could be captured by playing the move, or on (2) the number of stones that could escape a capture by playing the move.

The proximity coefficients are computed as the Euclidian distances to the last moves.

These elements are combined in the following formula to compute H_i.

$$H_i = (C_i + P_i) \sum_k \frac{1}{(2d_{k,i})^{\alpha_k}} \tag{3.4}$$

where P_i is the pattern value, C_i is the capture value of the move that leads to the position i, $d_{k,i}$ is the (Euclidean) distance to the k^{th} last move, and $\alpha_k = 1.25 + \frac{k}{2}$. Computing the P_i values is the time-consuming part of the heuristic.

3.4. *Time available for heuristics*

The time consumed to compute H_i is in the order of one millisecond, which is around 1000 times slower than playing a move in a simulated game. To avoid a speed reduction in the program's performance, we compute H_i only once per node, when a certain threshold of games has been played through this node. The threshold was set to $T = 30$ simulated games in MANGO. With this setting, the speed of the program was only reduced by 4 percent. The speed reduction is low because the number of nodes that have been visited more than 30 times is low compared to the number of moves played in the simulated games. It can be seen in Figure 4 that the number of calls to the domain knowledge is reduced fast as T increases. Even for $T = 9$, the number of calls to the domain knowledge is quite low compared to the number of simulated moves. The number of nodes having a certain visit count is plotted in Figure 5. The data has been obtained from a 19×19 initial position

[b] INDIGO was third out of 17 participants in the World Computer Go Championship 2006, see `http://computer-go.softopia.or.jp/gifu2006/English/index.html`

by performing a 30 seconds MCTS. We have also plotted a trend line that shows that this data can be approximated by a power law.

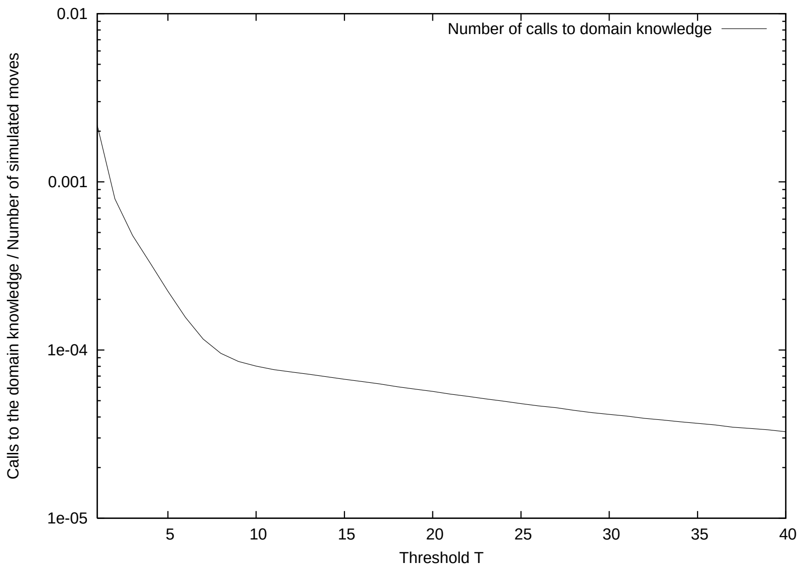

Fig. 4. Number of calls to the domain knowledge relative to the number of simulated moves, as a function of the threshold T.

4. Experiments

Three different series of experiments were conducted. Subsection 4.1 gives the impact of each progressive strategy against GNU Go. Subsection 4.2 shows that these methods also improve the level of our program in self-play. Subsection 4.3 assesses the strength of our program Mango in recent (internet) tournaments.

4.1. Mango *vs.* GNU Go

In the first series of experiments we tested the two progressive strategies in games against GNU Go version 3.6. The experiments were performed on the 9×9, 13×13, and 19×19 boards. Our program used $20,000$ simulations per move. It takes on average less than one second on a 9×9 board, two seconds on a 13×13 board, and five seconds on a 19×19 board. The level of GNU Go has been set to 10 on the 9×9 board and on the 13×13 board, and to 0 on the 19×19 board. The results are reported in Table 1, where PB stands for progressive bias and PU for progressive unpruning.

From these experiments, the results, and our observation, we may arrive at four conclusions. First, the plain MCTS framework does not scale well to the 13×13

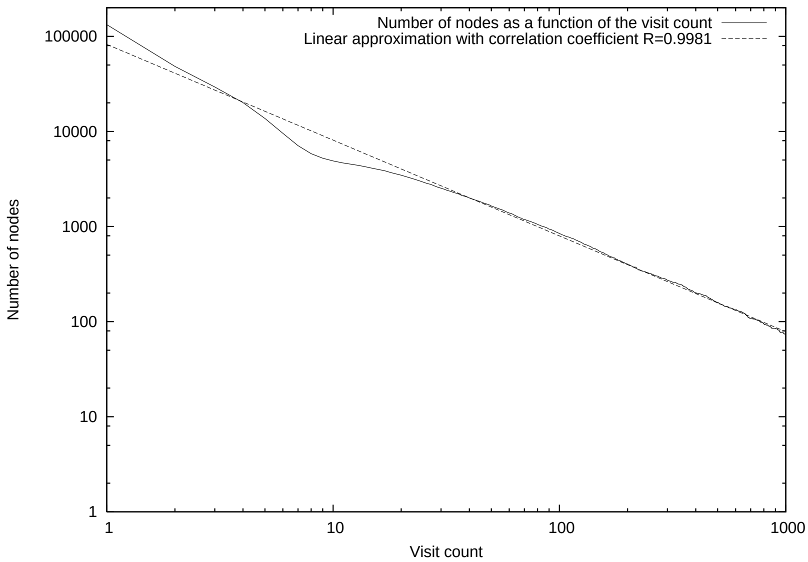

Fig. 5. Number of nodes with a given visit count.

board and the 19 × 19 board. Second, the progressive strategies increase MANGO's level of play on every board size. Third, on the 19 × 19 board size the combination of both strategies is much stronger than each strategy applied separately.

Table 1. Results of MANGO against GNU GO.

Board Size	Simulations/move	GNU GO's level	PB	PU	Winning rate	Games
19	20,000	0			0%	200
19	20,000	0		X	3.1%	200
19	20,000	0	X		4.8%	200
19	20,000	0	X	X	48.2%	500
13	20,000	10			8.5%	500
13	20,000	10		X	15.6%	500
13	20,000	10	X		30.0%	500
13	20,000	10	X	X	35.1%	500
9	20,000	10			33.2%	1000
9	20,000	10		X	37.2%	1000
9	20,000	10	X		58.3%	1000
9	20,000	10	X	X	61.7%	2000

4.2. *Self-play experiment*

To verify the previous results we performed a self-play experiment on the 13 × 13 board with a time setting of 10s per move. The version of MANGO using our progressive strategies won 81 percent of the 500 games played against MANGO without progressive strategies.

4.3. *Tournaments*

In the last series of experiments we tested MANGO's strength by competing in computer tournaments. Table 2 presents the results by MANGO in the tournaments entered in 2007. In this table, KGS stands for "KGS Go Server". This server is the most popular one for computer programmers, and most of the well-known programs have participated in one or more editions (e.g., MOGO, CRAZYSTONE, GO++, THE MANY FACES OF GO, GNU GO, INDIGO, AYA, DARIUSH, etc...)

As shown in the previous experiments, the progressive strategies are the main strength of MANGO. We remark that MANGO was always in the best half of the participants.

Table 2. Results by MANGO in 2007 tournaments.

Tournament	Board Size	Participants	MANGO's rank
KGS January 2007	13 × 13	10	2^{nd}
KGS March 2007	19 × 19	12	4^{th}
KGS April 2007	13 × 13	10	3^{rd}
KGS May 2007	13 × 13	7	2^{nd}
12^{th} Computer Olympiad	9 × 9	10	5^{th}
12^{th} Computer Olympiad	19 × 19	8	4^{th}
KGS July 2007	13 × 13	10	4^{th}

5. Conclusions and Future Research

In this paper we introduced the concept of progressive strategy. It enables a soft transition from a simulation to a selection strategy. Such a strategy uses (1) the information available for the selection strategy, and (2) some time-expensive domain knowledge. We have developed two progressive strategies: progressive bias and progressive unpruning. Progressive bias uses knowledge to direct the search. Progressive unpruning first reduces the branching factor, and then increases it gradually. This scheme is also dependent on knowledge. Based on the results of the experiments performed with our program MANGO, we may offer four conclusions. (1) The plain Monte-Carlo Tree Search method does not scale well to 13×13 Go, and performs even worse in 19×19 Go. (2) Progressive strategies increase the level of play of our program MANGO significantly, on every board size. (3) On the 19×19 board size, the combination of both strategies is much stronger than each strategy applied separately. (4) These strategies can use relatively expensive domain knowledge without hardly any speed reduction.

For future research we see four interesting directions. First, we will investigate other progressive strategies such as *RAVE* and *UCT with prior knowledge*.[13] They have been recently proposed to include knowledge in the Monte-Carlo Tree Search. It would be interesting to combine them with our current progressive strategies. Second, we are convinced that our strategies could give even better results by using more advanced knowledge, e.g., as developed by Coulom.[11] Third, we will also improve our heuristic knowledge by using life-and-death. Fourth, a remarkable idea will be to use the work by Coquelin and Munos[10] to improve the progressive bias.

Acknowledgments

This work is financed by the Dutch Organization for Scientific Research (NWO) for the project Go for Go, grant number 612.066.409. We thank Steven de Jong for exchanging some ideas that broadened our scope.

References

1. B. Bouzy. Associating Domain-Dependent Knowledge and Monte-Carlo Approaches within a Go program. *Information Sciences, Heuristic Search and Computer Game Playing IV*, 175(4):247–257, 2005.
2. B. Bouzy and T. Cazenave. Computer Go: An AI-oriented Survey. *Artificial Intelligence*, 132(1):39–103, 2001.
3. B. Bouzy and G.M.J-B. Chaslot. Bayesian Generation and Integration of K-nearest-neighbor Patterns for 19x19 Go. In G. Kendall and Simon Lucas, editors, *IEEE 2005 Symposium on Computational Intelligence in Games, Essex, UK*, pages 176–181, 2005.
4. B. Bouzy and G.M.J-B. Chaslot. Monte-Carlo Go Reinforcement Learning Experiments. In *IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA*, pages 187–194, 2006.
5. B. Bouzy and B. Helmstetter. Monte-Carlo Go Developments. In H.J. van den Herik, H. Iida, and E. A. Heinz, editors, *Proceedings of the 10th Advances in Computer Games Conference (ACG-10)*, pages 159–174. Kluwer Academic, 2003.
6. B. Brügmann. Monte Carlo Go. *White paper*, 1993.
7. M. Campbell, A.J. Hoane Jr., and F-H. Hsu. Deep Blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.
8. G.M.J-B. Chaslot, S. De Jong, J-T. Saito, and J.W.H.M. Uiterwijk. Monte-Carlo Tree Search in Production Management Problems. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 91–98, 2006.
9. G.M.J-B. Chaslot, J-T. Saito, B. Bouzy, J.W.H.M. Uiterwijk, and H.J. van den Herik.

Monte-Carlo Strategies for Computer Go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 83–90, 2006.
10. P-A. Coquelin and R. Munos. Bandit Algorithm for Tree Search. Technical Report 6141, INRIA, 2007.
11. R. Coulom. Computing Elo Ratings of Move Patterns in the Game of Go. In H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd, editors, *Proceedings of the Computers Games Workshop 2007 (CGW 2007)*, pages 113–124, 2007.
12. R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Proceedings of the 5th Computers and Games Conference (CG 2006)*, 2007. To be published in LNCS, Springer-Verlag, Berlin.
13. S. Gelly and D. Silver. Combining Online and Offline Knowledge in UCT. In Zoubin Ghahramani, editor, *Proceedings of the International Conference of Machine Learning (ICML 2007)*, pages 273–280, 2007.
14. S. Gelly and Y. Wang. Exploration Exploitation in Go: UCT for Monte-Carlo Go. In *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*, 2006.
15. S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.
16. H. J. van den Herik, J. W. H. M. Uiterwijk, and J. van Rijswijck. Games Solved: Now and in the Future. *Artificial Intelligence*, 134(1–2):277–311, 2002.
17. L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006, Lecture Notes in Artificial Intelligence 4212*, pages 282–293, 2006.
18. H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 55:527–535, 1952.
19. J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is Solved. *Science*, July, 2007.