Beam Monte-Carlo Tree Search

Hendrik Baier and Mark H. M. Winands

***Abstract*—*Monte-Carlo Tree Search* (MCTS) is a state-of-the-art stochastic search algorithm that has successfully been applied to various multi- and one-player games (puzzles). *Beam search* is a search method that only expands a limited number of promising nodes per tree level, thus restricting the space complexity of the underlying search algorithm to linear in the tree depth. This paper presents *Beam Monte-Carlo Tree Search* (BMCTS), combining the ideas of MCTS and beam search. Like MCTS, BMCTS builds a search tree using Monte-Carlo simulations as state evaluations. When a predetermined number of simulations has traversed the nodes of a given tree depth, these nodes are sorted by their estimated value, and only a fixed number of them is selected for further exploration. In our experiments with the puzzles SameGame, Clickomania and Bubble Breaker, BMCTS significantly outperforms MCTS at equal time controls. We show that the improvement is equivalent to an up to four-fold increase in computing time for MCTS.**

I. Introduction

Monte-Carlo Tree Search (MCTS) [1], [2] is a best-first tree search algorithm with Monte-Carlo evaluation of states. It effectively handles large search spaces by selectively deepening the search tree, guided by the outcomes of Monte-Carlo simulations. These simulations allow MCTS to take long-term rewards into account even with distant horizons. Combined with multi-armed bandit algorithms to balance exploration and exploitation, MCTS has been shown to guarantee asymptotic convergence to the optimal policy [2], while providing approximations when stopped at any time.

MCTS has achieved considerable success in e.g. the games of Go [3], [4], Amazons [5], LOA [6], and Ms. Pacman [7]; in General Game Playing [8], planning [9], [10], and optimization [11], [12], [13].

Beam search is a basic search method. It reduces the number of nodes at each tree level to a constant number, allowing for search effort linear in the tree depth. Since its first application in speech recognition [14], it has been used in a multitude of fields, such as machine translation [15], planning [16], and scheduling [17].

In this paper, we propose *Beam Monte-Carlo Tree Search* (BMCTS), combining the MCTS framework with the idea of beam search. We demonstrate the significantly stronger performance of BMCTS as compared to regular MCTS, at equal time controls, in the three one-player test domains SameGame, Clickomania, and Bubble Breaker.

This paper is organized as follows. Section II provides the necessary background on MCTS. After an overview of related work on beam search in Section III, Section IV proposes BMCTS. Section V introduces the test domains, and Section VI shows the experimental results. Conclusions and future work follow in Section VII.

Hendrik Baier and Mark H. M. Winands are with the Games and AI Group, Department of Knowledge Engineering, Maastricht University, The Netherlands, email: hendrik.baier | m.winands@maastrichtuniversity.nl

II. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [1], [2] is a best-first search algorithm using statistical sampling to evaluate states. For each move decision of the player, MCTS constructs a search tree T, starting from the current game state as root. This tree is selectively deepened into the direction of the most promising moves, which are determined by the success of simulated games starting with these moves. After n simulated games, the tree contains $n+1$ states, for which distinct value estimates are maintained.

MCTS works by repeating the following four-phase loop until computation time runs out [18]. Each loop represents one simulated game.

Phase one: *selection*. The tree is traversed from the root to one of the leaves. At each node, MCTS uses a selection policy to choose the move to sample from this state. Critical is a balance of exploitation of states with high value estimates and exploration of states with uncertain value estimates.

Phase two: *expansion*. When a leaf has been reached, one or more of its successors are added to the tree. In this paper, we always add the immediate successor of the leaf in the rollout (cf. [1]).

Phase three: *rollout*. A rollout (also called *playout*) policy is used to choose moves until the game ends. Uniformly random move choices are sufficient to achieve convergence of MCTS to the optimal move in the limit, but rollout policies utilizing basic domain knowledge can improve convergence speed considerably.

Phase four: *backpropagation*. The result of the finished game is used to update value estimates of all states traversed during the simulation.

Listing 1 shows pseudocode of MCTS for deterministic one-player games (puzzles), where not only the immediate next move choice is of interest, but also the best solution sequence found so far for the entire game. It assumes a uniformly random rollout policy.

In a variety of applications, a variant of MCTS called *Upper Confidence Bounds for Trees* (UCT) [2] has shown excellent performance. UCT uses the UCB1 formula, originally developed for the multi-armed bandit problem [19], to select moves in the tree and to balance exploration and exploitation. In this paper, a variant of UCT with the selection policy UCB1-TUNED is used. This policy takes the empirical variance of moves into account and has been shown to be empirically superior to UCB1 in several multi-armed bandit problems [19].

```
MCTS(startState) {
   bestResult ← -Infinity
   bestSolution ← {}
   for(numberOfIterations) {
      currentState ← startState
      solution ← {}
      while(currentState ∈ Tree) {
         currentState ← selectAction(currentState)
         solution ← solution + currentState
      }
      addToTree(currentState)
      while(simulationNotEnded) {
         currentState ← randomAction(currentState)
         solution ← solution + currentState
      }
      result = cumulativeReward(currentState)
      forall(state ∈ solution) {
         state.value ← backPropagate(state.value, result)
      }
      if(result > bestResult) {
         bestResult ← result
         bestSolution ← solution
      }
   }
   return (bestResult, bestSolution)
}
```

Listing 1. MCTS with random rollout policy.

Described in the framework of reinforcement learning, there are two interacting processes within MCTS.

Policy evaluation: In the backpropagation phase after each simulated game, the result of that game is used to update the value estimates of each visited state $s \in T$.

$$n_s \longleftarrow n_s + 1 \tag{1a}$$

$$\hat{V}^\pi(s) \longleftarrow \hat{V}^\pi(s) + \frac{r - \hat{V}^\pi(s)}{n_s} \tag{1b}$$

where n_s is the number of times state s has been traversed in all simulations so far, r is the reward received at the end of the current simulation, and $\hat{V}^\pi(s)$ is the current estimate of the value of state s for a player following policy π.

Policy improvement: During each simulation, the policy adapts to the current value estimates. In case of MCTS using UCB1-TUNED in the selection phase, and a uniformly random policy in the rollout phase, let

$$U^{\text{Var}}(s,m) = \left(\frac{1}{n_{s,m}} \sum_{t=1}^{n_{s,m}} r_{s,m,t}^2\right) - \left(\frac{1}{n_{s,m}} \sum_{t=1}^{n_{s,m}} r_{s,m,t}\right)^2 + \sqrt{\frac{2 \ln n_s}{n_{s,m}}}$$

be an upper confidence bound for the variance of move m in state s, where $n_{s,m}$ is the number of times move m has been chosen in state s in all simulations so far, and $r_{s,m,t}$ is the game result achieved when move m was chosen in state s for the t-th time. Let

$$U^{\text{Val}}(s,m) = \sqrt{\frac{2\ln(n_s)}{n_{s.m}} \min\left(\frac{1}{4}, U^{\text{Var}}(s,m)\right)}$$

be an upper confidence bound for the value of move m in state s. Then, the selection and rollout policies of the MCTS algorithm using UCB1-TUNED are given by

$$\pi(s) = \begin{cases} \underset{m \in A(s)}{\operatorname{argmax}} \left(\hat{V}^\pi\left(S_m(s)\right) + C \times U^{\text{Val}}(s,m)\right) & \text{if } s \in T \\ \operatorname{random}(s) & \text{otherwise} \end{cases}$$

where $S_m(s)$ is the state reached from s with move m, $\operatorname{random}(s)$ chooses one of the moves available in s with uniform probability, and C is an exploration coefficient, which has a domain-dependent optimum.

III. Beam Search

Beam search [14] is a technique that reduces the memory requirements of breadth-first or best-first search at the cost of completeness and optimality. Its basic idea is using heuristic value estimates to determine the most promising states at each level of the search tree. Only these states are then selected for further expansion, while all others are permanently pruned. Consequently, time and memory complexity of the search are linear in the beam width W and the tree depth D. By increasing or decreasing the beam width, memory can be traded off against solution quality, with $W = 1$ resulting in a greedy search, and $W = \infty$ resulting in a complete search.

Beam search has also been extended by combining it with depth-first search [20] as well as with limited discrepancy search [21]. These variants turn beam search into a *complete* search algorithm, i.e. an algorithm guaranteed to find a solution when there is one.

In the Monte-Carlo framework, [22] combines beam search with Nested Monte-Carlo Search (NMCS), a search algorithm that has shown good results in various single-player games [23]. Beam search has so far not been applied to MCTS. A similar idea to beam search, however, has been applied per node instead of per tree level. *Progressive widening* or *unpruning* [18], [24] reduces the number of children of rarely visited nodes. Heuristics are used to choose the most promising children. As the number of rollouts passing through the node increases, i.e. as the node is found to be more and more important by the search, the pruned children are progressively added again. This way, search effort for most nodes can be reduced, while retaining convergence to the optimal moves in the limit.

In the next section, we describe an application of the beam search idea in MCTS, reducing the time and space complexity of MCTS to linear in the tree depth. Our algorithm does not require heuristic knowledge.

IV. Beam Monte-Carlo Tree Search

In this section, we propose *Beam Monte-Carlo Tree Search* (BMCTS), our approach for combining beam search with MCTS. In addition to the MCTS tree, BMCTS maintains a counter for each tree depth, counting the number of simulated games that have passed through any tree node at this depth in the search so far. During backpropagation, these counters are

compared with the first parameter of BMCTS: the *simulation limit* L. If any tree depth d reaches this limit, the tree is pruned at level d.

Pruning restricts the number of tree nodes at depth d to the maximum number given by the second parameter of BMCTS: the *beam width* W. In order to do this, all tree nodes of depth d are first sorted for their heuristic values. Due to the large variance of Monte-Carlo value estimates at low simulation counts, we use the number of visits of a tree node instead of its estimated value as our heuristic—nodes that have seen the highest numbers of simulations are judged to be most promising by the search algorithm (cf. move selection in computer Go, e.g. in [18]). Then, the best W nodes at level d together with all of their ancestor nodes are retained, while all of their descendants as well as the less promising nodes of depth d are discarded.

When search continues, no new nodes up to depth d will be created anymore. The selection policy takes only those moves into account that lead to the retained nodes. Beyond depth d, the tree grows as usual.

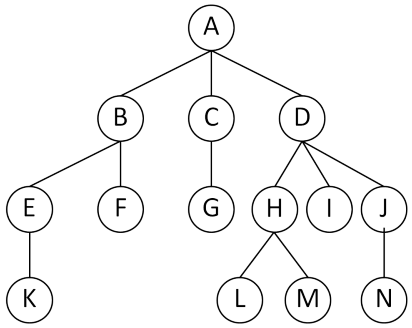
Listings 2 and 3 show pseudocode of BMCTS for deterministic one-player games, using a uniformly random rollout policy. Fig. 1 visualizes the three steps of the tree pruning method `pruneTree` outlined in Listing 3.

```
BMCTS(startState) {
   bestResult ← -Infinity
   bestSolution ← {}
   for(numberOfIterations) {
      currentState ← startState
      solution ← {}
      while(currentState ∈ Tree) {
         numberOfRolloutsThrough[currentState.depth]++
         currentState ← selectAction(currentState)
         solution ← solution + currentState
      }
      addToTree(currentState)
      while(simulationNotEnded) {
         currentState ← randomAction(currentState)
         solution ← solution + currentState
      }
      result = cumulativeReward(currentState)
      forall(state ∈ solution) {
         state.value ← backPropagate(state.value, result)
         if(numberOfRolloutsThrough[state.depth] = SIMLIMIT) {
            pruneTree(state.depth)
         }
      }
      if(result > bestResult) {
         bestResult ← result
         bestSolution ← solution
      }
   }
   return (bestResult, bestSolution)
}
```

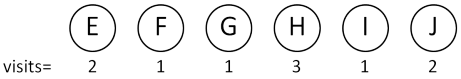
Listing 2. BMCTS with random rollout policy.

```
pruneTree(depth) {
   nodeSet ← treeNodesAtDepth(Tree, depth)
   nodeSet ← mostVisitedTreeNodes(nodeSet, BEAMWIDTH)
   Tree ← nodeSet + ancestorNodes(nodeSet)
}
```

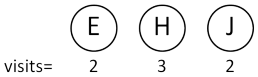
Listing 3. Tree pruning for BMCTS.



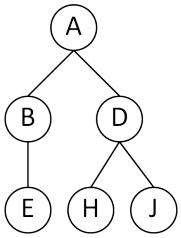
(a) The tree before pruning on depth $d = 2$.



(b) The nodes with $d = 2$ are collected.



(c) The $W = 3$ nodes with the highest visit count are retained, the rest is pruned.



(d) Ancestor nodes are added. Search continues on the new tree.

Fig. 1. Sketch of tree pruning (depth $d = 2$, beam width $W = 3$) for BMCTS.

V. Test Domains

We tested Beam Monte-Carlo Tree Search on three different domains: The puzzles named "SameGame", "Clickomania" and "Bubble Breaker", which are popular test domains for Monte-Carlo search approaches [25], [26], [27], [28], [29]. These puzzles have identical move rules, but different scoring rules, resulting in different distributions of high-scoring solutions. The decision problem associated with these optimization problems is NP-complete [30].

The rules of the puzzles are as follows. A two-dimensional board or grid is filled with $M \times N$ tiles of C different colors, usually randomly distributed, at the start. Each move consists of selecting a group of two or more vertically or horizontally connected, identically-colored tiles. When the move is executed, the tiles of this group are removed from the board. If there are tiles above the deleted group, they fall down. If an entire column of the board is emptied of tiles, the columns to the right shift to the left to close the gap. The

game ends when no moves are left to the player. The score the player receives depends on the specific variant of the puzzle:

- *Clickomania.* The goal of Clickomania is to clear the board of tiles as far as possible. At the end of each game, the player receives a score equivalent to the number of tiles removed.
- *Bubble Breaker.* The goal of Bubble Breaker is to create and then remove the largest possible groups of tiles. After each move removing a group of size `groupSize`, the player receives a score of $\texttt{groupSize}*(\texttt{groupSize}-1)$ points.
- *SameGame.* In SameGame, both the removal of large groups and the clearing of the board are rewarded. Each move removing a group of size `groupSize` results in a score of $(\texttt{groupSize}-2)^2$ points. Additionally, ending the game by clearing the board completely is rewarded with an extra 1000 points. If the game ends without clearing the board, the player receives a negative score. It is computed by assuming that all remaining tiles of the same color are connected into virtual groups, and subtracting points for all colors according to the formula $(\texttt{groupSize}-2)^2$.

We compared regular MCTS and BMCTS in all three domains, using a random rollout policy. For SameGame, we also employed a state-of-the-art informed rollout policy, consisting of the TabuColorRandomPolicy [28] (setting a "tabu color" at the start of each rollout that is not chosen as long as groups of other colors are available) in combination with a multi-armed bandit learning the best-performing tabu color for the position at hand (based on UCB1-TUNED).

VI. Experiments

In Subsection VI-A, we show the parameter landscape of BMCTS in our test domains, explaining how optimal parameter settings were found. Afterwards, these settings are used in Subsection VI-B to compare the performance of BMCTS and regular MCTS.

A. Parameter Optimization

In our first experiments, we examined the influence of the BMCTS parameters C, L and W, determining optimal values for each test domain. The experiments were conducted on 500 randomly generated 20×20 boards with 10 different tile colors in Clickomania, 250 randomly generated 15×15 boards with 5 different tile colors in Bubble Breaker and SameGame with random rollouts, and 1000 randomly generated 15×15 boards with 5 different tile colors in SameGame with informed rollouts. Computation time per board was 60 seconds. Rollout results were normalized to the interval $[0, 1]$.

As a first step, we determined optimal exploration factors C for regular MCTS in all test domains. 10 to 25 different values of C were tested per game. The best settings found were $C = 0.0009$ for SameGame with informed rollouts, $C = 0.0025$ for SameGame with random rollouts, $C = 0.0275$ for Bubble Breaker and $C = 0.012$ for Clickomania. Manual experimentation showed that the optimal setting of exploration factor C for BMCTS was in most cases identical to the optimal setting for MCTS at the same time controls. Only in Clickomania, BMCTS performed better at a slightly more explorative setting ($C = 0.0175$ instead of $C = 0.012$). Fig. 2, 3, 4 and 5 illustrate the influence of BMCTS parameters further by showing the performance for different settings of L and W. Here, C is set to the value that achieved best results with the optimal choice of L and W.

In two of the puzzle variants considered (Bubble Breaker and SameGame with random rollouts), the optimal beam width found was 1. This setting reduces beam search to a variant of move-by-move search as described in [26], a time-management technique that distributes search time over all moves in the game instead of globally searching from the initial position. In the two other variants however, optimal beam widths are larger: 100 in Clickomania and 25 in SameGame with informed rollouts. This demonstrates the positive effect of genuine beam search, examining a number of alternative moves instead of focusing on just one when proceeding with the next move of the game. The effect is most pronounced in Clickomania, as shown in Fig. 5. The specific class of problems for which large beam widths are most successful remains to be characterized in future work.

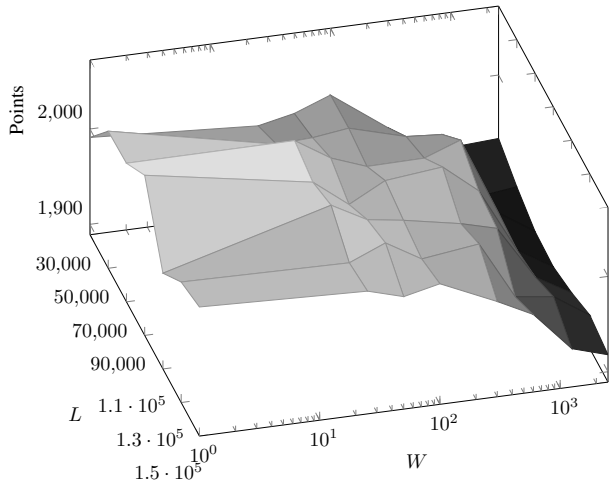


Fig. 2. Performance of BMCTS in SameGame with random rollouts. $C = 0.0009$ for all conditions.

B. Comparison to MCTS

After finding for each game and time setting the optimal settings for exploration coefficient C (both for MCTS and BMCTS), simulation limit L and beam width W, independent test sets of 1000 positions for each game were used to produce the results reported in this subsection. Clickomania boards had 20×20 boards with 10 different tile colors, while in all other test domains 15×15 boards with 5 different tile colors were used.

Fig. 6, 7, 8 and 9 show the results of our comparison. In all three puzzles—in SameGame using both random and informed rollouts—BMCTS significantly outperformed regular MCTS at 60 second time controls ($p<0.0001$ according to a two-tailed paired t-test). In Clickomania and in SameGame using

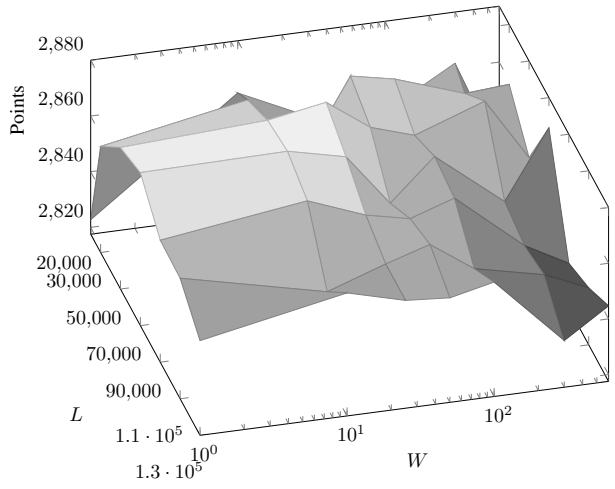


Fig. 3. Performance of BMCTS in SameGame with informed rollouts. $C = 0.0025$ for all conditions.

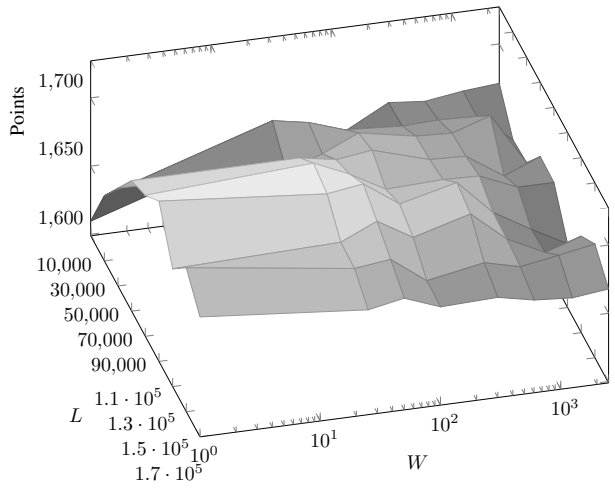


Fig. 4. Performance of BMCTS in Bubble Breaker. $C = 0.0275$ for all conditions.

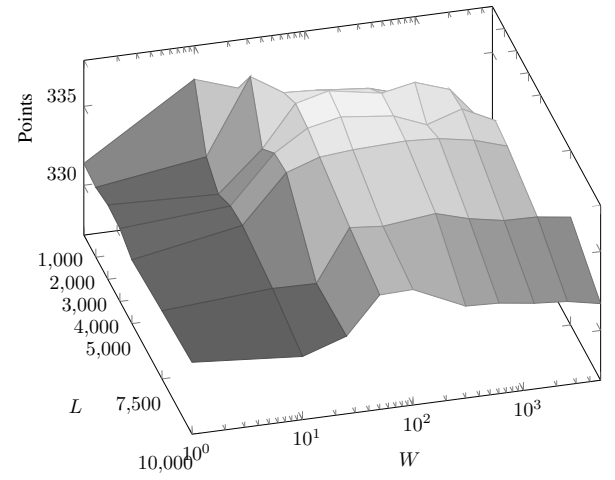


Fig. 5. Performance of BMCTS in Clickomania. $C = 0.0175$ for all conditions.

informed rollouts, the improvement is comparable to almost a doubling in search time for regular MCTS. In Bubble Breaker and in SameGame using random rollouts, it is comparable to roughly a four-fold increase in search time.

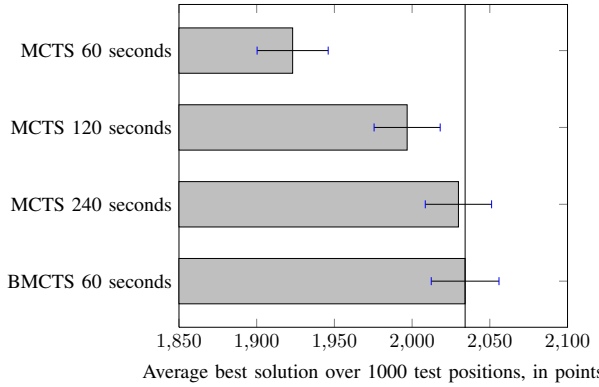


Fig. 6. Performance of MCTS vs. BMCTS in SameGame with random rollouts. MCTS settings: $C = 0.0009$ for 60 seconds, $C = 0.0016$ for 120 seconds, $C = 0.002$ for 240 seconds. BMCTS settings: $C = 0.0009$, $L = 90,000$, $W = 1$.

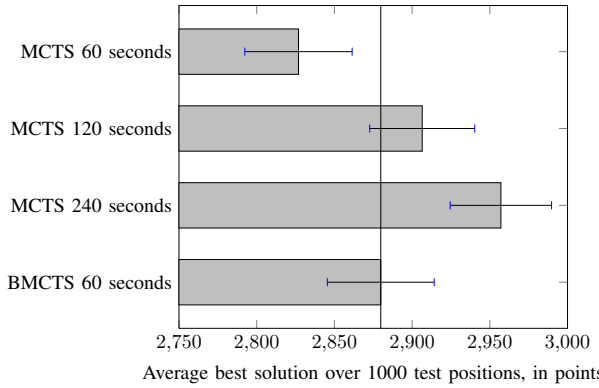


Fig. 7. Performance of MCTS vs. BMCTS in SameGame with informed rollouts. MCTS settings: $C = 0.0025$ for 60 seconds, $C = 0.0035$ for 120 seconds, $C = 0.0065$ for 240 seconds. BMCTS settings: $C = 0.0025$, $L = 50,000$, $W = 25$.

VII. Conclusions and Future Research

In this paper, we proposed *Beam Monte-Carlo Tree Search* (BMCTS), integrating the concept of beam search into an MCTS framework. Experimental results show that BMCTS significantly outperforms regular MCTS in the test domains of Bubble Breaker, Clickomania and SameGame. Depending on the domain, optimal parameter settings either result in a move-by-move time management scheme ($W = 1$), or in a new type of Monte-Carlo search that makes use of a predetermined number of alternative moves per tree level ($W > 1$). The performance improvement achieved by BMCTS is comparable to that achieved by a two-fold increase in computation time for regular MCTS in Clickomania and SameGame with informed rollouts. It is comparable to the effect of a four-fold increase

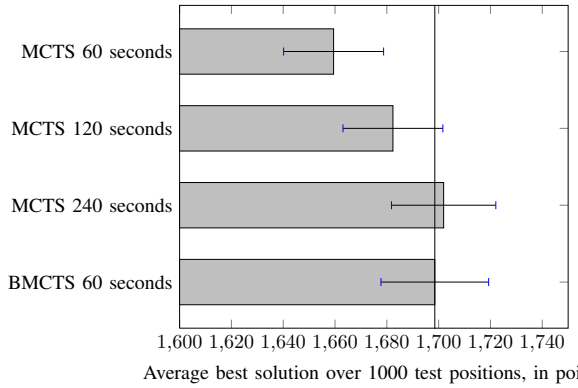


Fig. 8. Performance of MCTS vs. BMCTS in Bubble Breaker. MCTS settings: $C = 0.0275$ for 60 seconds, $C = 0.0475$ for 120 seconds, $C = 0.055$ for 240 seconds. BMCTS settings: $C = 0.0275$, $L = 110,000$, $W = 1$.

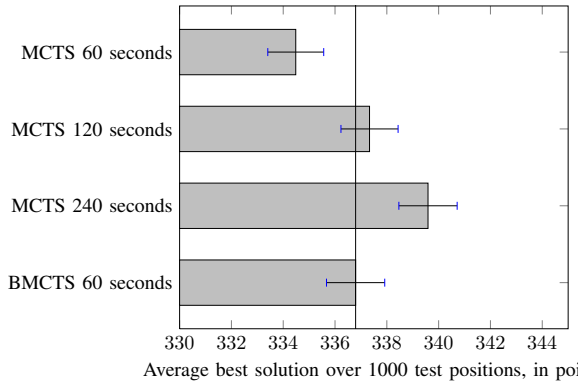


Fig. 9. Performance of MCTS vs. BMCTS in Clickomania. MCTS settings: $C = 0.012$ for 60 seconds, $C = 0.016$ for 120 seconds, $C = 0.022$ for 240 seconds. BMCTS settings: $C = 0.0175$, $L = 3,000$, $W = 100$.

in computation time in Bubble Breaker and SameGame with random rollouts.

Four directions appear promising for future work. First, BMCTS as presented in this paper does not retain the asymptotic properties of MCTS—due to the permanent pruning of nodes, optimal behavior in the limit cannot guaranteed. The addition of e.g. gradually increasing beam widths, similar to progressive widening [18], [24] but on a per-depth instead of per-node basis, could restore this important completeness property. Second, the basic BMCTS algorithm could be refined in various ways, for instance by automatically learning different simulation limits and beam widths for each tree depth. Any such refinements should ideally go along with characterizations of the classes of tasks for which they are most effective. Third, it would be interesting to further investigate the influence of the strength of the rollout policy on the performance of BMCTS—testing SameGame with both random and informed rollouts was a first step in this direction. Fourth, it would be worthwhile to compare our beam search variant of MCTS to the beam search variant of NMCS proposed in [22].

Acknowledgments

This work is funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project Go4Nature, grant number 612.000.938.

References

[1] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *5th International Conference on Computers and Games (CG 2006). Revised Papers*, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer, 2007, pp. 72–83.

[2] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *17th European Conference on Machine Learning, ECML 2006*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Springer, 2006, pp. 282–293.

[3] S. Gelly and D. Silver, "Achieving Master Level Play in 9 x 9 Computer Go," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, D. Fox and C. P. Gomes, Eds., 2008, pp. 1537–1540.

[4] C.-S. Lee, M.-H. Wang, G. M. J.-B. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong, "The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 73–89, 2009.

[5] R. J. Lorentz, "Amazons Discover Monte-Carlo," in *Proceedings of the 6th International Conference on Computers and Games*, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., 2008, pp. 13–24.

[6] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, "Monte Carlo Tree Search in Lines of Action," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 239–250, 2010.

[7] N. Ikehata and T. Ito, "Monte-Carlo Tree Search in Ms. Pac-Man," in *2011 IEEE Conference on Computational Intelligence and Games (CIG)*, 2011, pp. 39–46.

[8] H. Finnsson and Y. Björnsson, "Simulation-Based Approach to General Game Playing," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, D. Fox and C. P. Gomes, Eds., 2008, pp. 259–264.

[9] H. Nakhost and M. Müller, "Monte-Carlo Exploration for Deterministic Planning," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, C. Boutilier, Ed., 2009, pp. 1766–1771.

[10] D. Silver and J. Veness, "Monte-Carlo Planning in Large POMDPs," in *Advances in Neural Information Processing Systems 23, NIPS 2010*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds., 2010, pp. 2164–2172.

[11] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel, "Bandit-Based Optimization on Graphs with Application to Library Performance Tuning," in *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009*, 2009, pp. 729–736.

[12] A. Rimmel, F. Teytaud, and T. Cazenave, "Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows," in *Proceedings of Applications of Evolutionary Computation - EvoApplications 2011*, C. D. Chio, A. Brabazon, G. A. D. Caro, R. Drechsler, M. Farooq, J. Grahl, G. Greenfield, C. Prins, J. Romero, G. Squillero, E. Tarantino, A. Tettamanzi, N. Urquhart, and A. S. Etaner-Uyar, Eds., 2011, pp. 501–510.

[13] A. Sabharwal and H. Samulowitz, "Guiding Combinatorial Optimization with UCT," in *ICAPS 2011 Workshop on Monte-Carlo Tree Search: Theory and Applications*, 2011.

[14] B. T. Lowerre, "The Harpy Speech Recognition System," Ph.D. dissertation, Carnegie Mellon University, 1976.

[15] C. Tillmann and H. Ney, "Word Reordering and a Dynamic Programming Beam Search Algorithm for Statistical Machine Translation," *Computational Linguistics*, vol. 29, no. 1, pp. 97–133, 2003.

[16] R. Zhou and E. A. Hansen, "Breadth-First Heuristic Search," *Artificial Intelligence*, vol. 170, no. 4-5, pp. 385–408, 2006.

[17] I. Sabuncuoglu and M. Bayiz, "Job Shop Scheduling with Beam Search," *European Journal of Operational Research*, vol. 118, no. 2, pp. 390–412, 1999.

[18] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive Strategies for Monte-Carlo Tree Search," *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008.

[19] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-Time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[20] R. Zhou and E. A. Hansen, "Beam-Stack Search: Integrating Backtracking with Beam Search," in *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, S. Biundo, K. L. Myers, and K. Rajan, Eds. AAAI, 2005, pp. 90–98.

[21] D. Furcy and S. Koenig, "Limited Discrepancy Beam Search," in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 2005, pp. 125–131.

[22] T. Cazenave, "Monte-Carlo Beam Search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 68–72, 2012.

[23] ——, "Nested Monte-Carlo Search," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, C. Boutilier, Ed., 2009, pp. 456–461.

[24] R. Coulom, "Computing Elo Ratings of Move Patterns in the Game of Go," *ICGA Journal*, vol. 30, no. 4, pp. 198–208, 2007.

[25] S. Matsumoto, N. Hirosue, K. Itonaga, K. Yokoo, and H. Futahashi, "Evaluation of Simulation Strategy on Single-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem," in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2010*, vol. 3, 2010, pp. 2086–2091.

[26] M. P. D. Schadd, M. H. M. Winands, M. J. W. Tak, and J. W. H. M. Uiterwijk, "Single-Player Monte-Carlo Tree Search for SameGame," *Knowledge-Based Systems. In press*.

[27] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, and H. Aldewereld, "Addressing NP-Complete Puzzles with Monte-Carlo Methods," in *Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning*, vol. 9, 2008, pp. 55–61.

[28] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J.-B. Chaslot, and J. W. H. M. Uiterwijk, "Single-Player Monte-Carlo Tree Search," in *Proceedings of the 6th International Conference on Computers and Games*, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., 2008, pp. 1–12.

[29] F. W. Takes and W. A. Kosters, "Solving SameGame and its Chessboard Variant," in *Proceedings of the 21st Benelux Conference on Artificial Intelligence, BNAIC 2009*, T. Calders, K. Tuyls, and M. Pechenizkiy, Eds., 2009, pp. 249–256.

[30] T. C. Biedl, E. D. Demaine, M. L. Demaine, R. Fleischer, L. Jacobsen, and J. I. Munro, "The Complexity of Clickomania," in *More Games of No Chance, Proceedings of the MSRI Workshop on Combinatorial Games*, R. J. Nowakowski, Ed., 2002, pp. 389–404.