A Rollout-Based Search Algorithm Unifying MCTS and Alpha-Beta

Hendrik Baier

Advanced Concepts Team
European Space Agency
Noordwijk, The Netherlands
hendrik.baier@esa.int

Abstract. *Monte Carlo Tree Search* (MCTS) has been found to be a weaker player than minimax in some tactical domains, partly due to its highly selective focus only on the most promising moves. In order to combine the strategic strength of MCTS and the tactical strength of minimax, *MCTS-minimax hybrids* have been introduced in prior work, embedding shallow minimax searches into the MCTS framework. This paper continues this line of research by integrating MCTS and minimax even more tightly into one rollout-based hybrid search algorithm, *MCTS-$\alpha\beta$*. The hybrid is able to execute two types of rollouts: MCTS rollouts and alpha-beta rollouts, i.e. rollouts implementing minimax with alpha-beta pruning and iterative deepening. During the search, all nodes accumulate both MCTS value estimates as well as alpha-beta value bounds. The two types of information are combined in a given tree node whenever alpha-beta completes a deepening iteration rooted in that node—by increasing the MCTS value estimates for the best move found by alpha-beta. A single parameter, the probability of executing MCTS rollouts vs. alpha-beta rollouts, makes it possible for the hybrid to subsume both MCTS as well as alpha-beta search as extreme cases, while allowing for a spectrum of new search algorithms in between.
Preliminary results in the game of Breakthrough show the proposed hybrid to outperform its special cases of alpha-beta and MCTS. These results are promising for the further development of rollout-based algorithms that unify MCTS and minimax approaches.

1 Introduction

Monte Carlo Tree Search (MCTS) [8, 11] is a sampling-based tree search algorithm. Instead of taking all legal moves into account like traditional full-width minimax search, MCTS samples promising moves selectively. This is helpful in many large search spaces with high branching factors. Furthermore, MCTS can often take long-term effects of moves better into account than minimax, since it typically uses Monte-Carlo simulations of entire games instead of a static heuristic evaluation function for the evaluation of states. This can lead to greater positional understanding with lower implementation effort. If exploration and

exploitation are traded off appropriately, MCTS asymptotically converges to the optimal policy [11], while providing approximations at any time.

While MCTS has shown considerable success in a variety of domains [6], there are still games such as Chess and Checkers where it is inferior to minimax search with alpha-beta pruning [10]. One reason that has been identified for this weakness is the selectivity of MCTS, its focus on only the most promising lines of play. Tactical games such as Chess can have a large number of traps in their search space [16]. These can only be avoided by precise play, and the selective sampling of MCTS based on average simulation outcomes can easily miss or underestimate an important move.

In previous work [2–4], the tactical strength of minimax has been combined with the strategic and positional understanding of MCTS in *MCTS-minimax hybrids*, integrating shallow-depth minimax searches into the MCTS framework. These hybrids have shown promising results in tactical domains, both for the case where heuristic evaluation functions are unavailable [3], as well as for the case where their existence is assumed [2, 4]. In this paper, we continue this line of work by integrating MCTS and minimax even more closely. Based on Huang's formulation of alpha-beta search as a rollout-based algorithm [5], we propose a hybrid search algorithm *MCTS-*$\alpha\beta$ that makes use of both MCTS rollouts as well as alpha-beta rollouts. MCTS-$\alpha\beta$ can switch from executing a rollout in MCTS fashion to executing it in alpha-beta fashion at any node traversed in the tree. During the search, all nodes accumulate both MCTS value estimates as well as alpha-beta value bounds. A single rollout can collect both types of information. Whenever a deepening iteration of alpha-beta rooted in a given node is completed, the move leading to the best child found by this alpha-beta search is awarded a number of MCTS wins in that node. This allows the hybrid to combine both types of information throughout the tree.

Unlike previously proposed hybrid search algorithms, MCTS-$\alpha\beta$ subsumes both MCTS as well as alpha-beta search as extreme cases. It turns into MCTS when only using MCTS rollouts, and into alpha-beta when only using alpha-beta rollouts. By mixing both types of rollouts however, a spectrum of new search algorithms between those extremes is made available, potentially leading to better performance than either MCTS or alpha-beta in any given search domain.

This paper is structured as follows. Section 2 gives some background on MCTS and Huang's rollout-based alpha-beta as the baseline algorithms of this paper. Section 3 provides a brief overview of related work on hybrid algorithms combining features of MCTS and minimax. Section 4 outlines the proposed rollout-based hybrid MCTS-$\alpha\beta$, and Section 5 shows first experimental results in the test domain of Breakthrough. Section 6 finally concludes and suggests future research.

2 Background

The hybrid MCTS-$\alpha\beta$ proposed in this paper is based on two search methods as basic components: Monte Carlo Tree Search (MCTS) and minimax search with alpha-beta pruning.

2.1 MCTS

The first component of MCTS-$\alpha\beta$ is MCTS, which works by repeating the following four-phase loop until computation time runs out.

Phase one: *selection*. The tree is traversed from the root to one of its not fully expanded nodes, choosing the move to sample from each state with the help of a selection policy. The selection policy should balance the exploitation of states with high value estimates and the exploration of states with uncertain value estimates. In this paper, the popular UCT variant of MCTS is used, with the UCB1-TUNED policy as selection policy [1].

Phase two: *expansion*. When a not fully expanded node has been reached, one or more of its successors are added to the tree. In this paper, we always add the one successor chosen in the current rollout.

Phase three: *simulation*. A default policy plays the game to its end, starting from the state represented by the newly added node. MCTS converges to the optimal move in the limit even when moves are chosen randomly in this phase. Note that this phase is often also called "rollout" phase or "playout" phase in the literature. We are calling it simulation phase here, and refer to its policy as the default policy, while choosing "rollout" as the name for one entire four-phase loop. This is in order to draw a clearer connection between MCTS rollouts in this subsection and alpha-beta rollouts in the next one. It is also consistent with the terminology used in [6].

Phase four: *backpropagation*. The value estimates of all states traversed in the tree are updated with the result of the finished game.

Algorithm 1.1 shows pseudocode for a recursive formulation of MCTS used as a first starting point for this work. `gameResult(s)` returns the game-theoretic value of terminal state s. `backPropagate(s.value, score)` updates the MCTS value estimate for state s with the new result `score`. UCB1-TUNED for example requires a rollout counter, an average score and an average squared score of all previous rollouts passing through the state. Different implementations are possible for `finalMoveChoice()`; in this work, it chooses the move leading to the child of the root with the highest number of rollouts.

Many variants and extensions of this framework have been proposed in the literature [6]. In this paper, we are using MCTS with the *MCTS-Solver* extension [21] as a component of MCTS-$\alpha\beta$. MCTS-Solver is able to backpropagate not only regular simulation results such as losses and wins, but also game-theoretic values such as proven losses and proven wins whenever the search tree encounters a terminal state. The idea is marking a move as a proven loss if the opponent has a winning move from the resulting position, and marking a move as a proven win if the opponent has only losing moves from the resulting position. This

```
MCTS(root) {
   while(timeAvailable) {
      MCTSRollout(root)
   }
   return finalMoveChoice()
}

MCTSRollout(currentState) {
   if(currentState ∈ Tree) {
      # selection
      nextState ← takeSelectionPolicyMove(currentState)
      score = MCTSRollout(nextState)
   } else {
      # expansion
      addToTree(currentState)
      # simulation
      simulationState ← currentState
      while(simulationState.notTerminalPosition) {
         simulationState ← takeDefaultPolicyMove(simulationState)
      }
      score ← gameResult(simulationState)
   }
   # backpropagation
   currentState.value ← backPropagate(currentState.value, score)
   return score
}
```

Algorithm 1.1: Monte Carlo Tree Search.

avoids wasting time on the re-sampling of game states whose values are already known. Additionally, we use an informed default policy instead of a random one, making move choices based on simple knowledge about the domain at hand. It is described in Section 5. Both of these improvements are not essential to the idea of MCTS-$\alpha\beta$, but together allow for MCTS to win a considerable number of games against alpha-beta in our experiments. This makes combining their strengths more worthwhile than if alpha-beta utterly dominated MCTS (or the other way around).

2.2 Rollout-based Alpha-beta

The second component of MCTS-$\alpha\beta$ is alpha-beta search. Specifically, we base this work on the rollout-based formulation of alpha-beta presented by Huang [5]. It is strictly equivalent not to classic alpha-beta search, but to an augmented version *alphabeta2*. Alphabeta2 behaves exactly like classic alpha-beta if given only one pass over the tree without any previously stored information, but it can "outprune" (evaluate fewer leaf nodes than) classic alpha-beta when called as a subroutine of a storage-enhanced search algorithm such as MT-SSS* [15]. See [5] for a detailed analysis.

The basic idea of rollout algorithms is to repeatedly start at the root and traverse down the tree. At each node representing a game state s, a selection policy chooses a successor state c from the set $C(s)$ of all legal successor states or children of s. In principle, any child could be chosen. However, it is known from alpha-beta pruning that the minimax value of the root can be determined without taking *all* children into account. Based on this realization, rollout-based alpha-beta was constructed as a rollout algorithm that restricts the selection policy at each state s to a subset of $C(s)$. This enables the algorithm to visit the same set of leaf nodes in the same order as alpha-beta, if the selection policy is chosen correctly.

Algorithm 1.2 shows pseudocode for the rollout-based formulation of alpha-beta used as a second starting point for this work. It requires the tree to maintain a closed interval $[v_s^-, v_s^+]$ for every visited state s. These intervals are initialized with $[-\infty, +\infty]$ and contain at any point in time the true minimax value of the respective state. When $v_s^- = v_s^+$, the minimax value of state s is found. When the minimax value of the root is found and the search is over, `finalMoveChoice()` chooses an optimal move at the root. The result of Algorithm 1.2 is independent of the implementation of `takeSelectionPolicyMove(feasibleChildren)`; in order to achieve alpha-beta behavior however, this method always needs to return the left-most child in `feasibleChildren`. That is the implementation used in this work. MAX and MIN refer to states where it is the turn of the maximizing or minimizing player, respectively.

As mentioned by Huang, "it seems that [Algorithm 1.2] could be adapted to an 'incremental' rollout algorithm when incorporating admissible heuristic function at internal nodes (essentially an iterative deepening setting)" [5]. As shown in Section 4, we extended Algorithm 1.2 with an heuristic evaluation function and

```
1  alphaBeta(root) {
2     while(v^-_root < v^+_root) {
3        alphaBetaRollout(root, v^-_root, v^+_root)
4     }
5     return finalMoveChoice()
6  }
7
8  alphaBetaRollout(s, α_s, β_s) {
9     if( C(s) ≠ ∅ ) {
10       for each c ∈ C(s) do {
11          [α_c, β_c] ← [max{α_s, v^-_c}, min{β_s, v^+_c}]
12       }
13       feasibleChildren ← {c ∈ C(s) | α_c < β_c}
14       nextState ← takeSelectionPolicyMove(feasibleChildren)
15       alphaBetaRollout(nextState, α_nextState, β_nextState)
16    }
                 { gameResult(s)      if s is leaf
17    v^-_s ←    { max_{c∈C(s)} v^-_c  if s is internal and MAX
                 { min_{c∈C(s)} v^-_c  if s is internal and MIN
                 { gameResult(s)      if s is leaf
18    v^+_s ←    { max_{c∈C(s)} v^+_c  if s is internal and MAX
                 { min_{c∈C(s)} v^+_c  if s is internal and MIN
19 }
```

Algorithm 1.2: Rollout-based alpha-beta as proposed by Huang [5].

iterative deepening in order to create practical alpha-beta rollouts for MCTS-$\alpha\beta$. Furthermore, Huang predicted that "traditional pruning techniques and the recent Monte Carlo Tree Search algorithms, as two competing approaches for game tree evaluation, may be unified under the rollout paradigm" [5]. This is the goal of the work presented in this paper.

3 Related Work

The idea of combining the strengths of alpha-beta and MCTS in one search algorithm is motivated for instance by the work of Ramanujan et al. [16], who identified *shallow traps* as a feature of domains that are problematic for the selectively searching MCTS. Informally, Ramanujan et al. define a *level-k search trap* as the possibility of a player to choose an unfortunate move such that *after* executing the move, the opponent has a guaranteed winning strategy at most k plies deep. While such traps at shallow depths of 3 to 7 are not found in Go until the latest part of the endgame, they are relatively frequent in Chess games even at grandmaster level [16], partly explaining the success of MCTS in Go and its problems in Chess. Finnsson and Björnsson [9] discovered the similar problem of *optimistic moves*, which refers to seemingly strong moves that can be refuted right away by the opponent, but take MCTS prohibitively many simulations to evaluate correctly. The work presented in this paper is meant as a step towards

search algorithms that can successfully be used in both kinds of domains—those favoring MCTS and those favoring alpha-beta until now.

Previous work on developing algorithms influenced by both MCTS and minimax has taken two main approaches. The first approach is to embed minimax searches within the MCTS framework. Shallow minimax searches have for example been used in every step of the simulation phase for Lines of Action [20], Chess [17], and various multi-player games [14]. Baier and Winands studied approaches that use minimax search without evaluation functions nested into the selection/expansion phase, the simulation phase, and the backpropagation phase of MCTS [3], as well as approaches that use minimax search with evaluation functions in the simulation phase, for early termination of simulations, and as a prior for tree nodes [2, 4].

The second approach is to identify individual features of minimax such as minimax-style backups, and integrate them into MCTS. In the algorithm $UCTMAX_H$ [18] for example, MCTS simulations are replaced with heuristic evaluations and classic averaging MCTS backups with minimaxing backups. In *implicit minimax backups* [12], both minimaxing backups of heuristic evaluations and averaging backups of simulation returns are managed simultaneously.

This paper takes a new approach. While in our previous hybrids [2–4], alpha-beta searches were nested into the MCTS framework and had to complete before MCTS could continue—MCTS and alpha-beta functioned as combined, but separate algorithms—the newly proposed MCTS-$\alpha\beta$ tightly interleaves MCTS and alpha-beta. The formulation of alpha-beta as a rollout algorithm [5] allows MCTS-$\alpha\beta$ to decide about continuing a rollout in MCTS fashion or in alpha-beta fashion at every node encountered during the search. As opposed to $UCTMAX_H$ and *implicit minimax* mentioned above, MCTS-$\alpha\beta$ is not picking and combining individual features of MCTS and minimax. It subsumes both regular MCTS and regular alpha-beta when only MCTS rollouts or only alpha-beta rollouts are used, but results in a new type of search algorithm when both types are combined. A probability parameter p determines the mix.

Apart from Huang [5], several other researchers have proposed rollout-based formulations of minimax search. For example, Weinstein, Littman, and Goschin [19] presented a rollout algorithm that outprunes alpha-beta, and Chen et al. [7] proposed a rollout algorithm similar to MT-SSS*. We chose Huang's alpha-beta formulation as a basis for this work because of its clear formal characterization, unifying both alpha-beta and MT-SSS* under the rollout framework.

4 MCTS-$\alpha\beta$

The basic idea of MCTS-$\alpha\beta$ is to allow for a mix of MCTS and alpha-beta rollouts. A simple way of achieving this is by introducing a parameter $p \in [0, 1]$ as the probability of starting an MCTS rollout at the root. $1 - p$, conversely, is the probability of starting an alpha-beta rollout instead. Assume that an MCTS rollout is chosen at the root. At every recursive call of the MCTS rollout, the randomized decision is made again whether to continue with MCTS or whether

to switch to alpha-beta, using the same probabilities. If the search tree is left without switching to an alpha-beta rollout at any point, the simulation and backpropagation phases are executed just like in a regular MCTS rollout. MCTS value estimates are updated in all traversed nodes, and the next rollout begins.

If however any randomized decision indicates the start of an alpha-beta rollout—either at the root or at a later stage of an MCTS rollout—then the rollout continues in alpha-beta fashion, with the current node functioning as the root of the alpha-beta search. This is similar to starting an embedded alpha-beta search at the current node, like the MCTS-IP-M algorithm described in [2, 4]. But MCTS-$\alpha\beta$ does not necessarily execute the entire alpha-beta search. The newly proposed hybrid can execute only one alpha-beta rollout instead, and potentially continue this particular alpha-beta search at any later point during the search process—whenever the decision for an alpha-beta rollout is made again at the same node. The interleaving of MCTS and minimax is more fine-grained than in previous hybrids.

There are a few differences between the alpha-beta rollouts of Huang's Algorithm 1.2 and those of MCTS-$\alpha\beta$. First, MCTS-$\alpha\beta$ uses an evaluation function to allow for depth-limited search. Second, these depth-limited searches are conducted in an iterative deepening manner. Third, MCTS-$\alpha\beta$ can reduce the branching factor for alpha-beta rollouts with the help of move ordering and *k-best pruning* (only searching the k moves ranked highest by a move ordering function). Fourth, if an alpha-beta rollout of MCTS-$\alpha\beta$ was called from an ongoing MCTS rollout instead of from the root, it returns the evaluation value of its leaf node to that MCTS rollout for backpropagation. And fifth, if the alpha-beta rollout is finishing a deepening iteration in a state s—if it is completing a 1-ply, 2-ply, 3-ply search etc—MCTS-$\alpha\beta$ gives a bonus in MCTS rollouts to the MCTS value estimate of the best child of s found in that iteration. At the same time, the bonus given for the previous deepening iteration is removed, so that only the currently best child of s is boosted.

This last point makes it clear how MCTS-$\alpha\beta$ can subsume both MCTS and alpha-beta. If $p = 1$, only MCTS rollouts are executed, and MCTS-$\alpha\beta$ behaves exactly like regular MCTS. If $p = 0$, only alpha-beta rollouts are started immediately at the root, and only the best move found by the last deepening iteration has a positive MCTS value estimate due to its bonus. MCTS-$\alpha\beta$ therefore behaves exactly like alpha-beta (an iterative deepening version of Huang's augmented alphabeta2, to be precise). If $0 < p < 1$ however, MCTS-$\alpha\beta$ becomes a true hybrid, combining MCTS and minimax behavior throughout the search tree, and choosing moves at the root based on both real MCTS rollout counts as well as MCTS rollout bonuses from the last completed deepening iteration of alpha-beta.

Algorithm 1.3 shows pseudocode for MCTS-$\alpha\beta$. $D(s)$ is the current search depth for alpha-beta starting in state s, initialized to 1 for all states. $K(s)$ is the set of the k best successor states or children of state s as determined by the move ordering function. `random(0,1)` returns a random, uniformly distributed value in $[0, 1]$. $\left[v^-_{s,d}, v^+_{s,d}\right]$ is an interval containing the value of state s when searched

to depth d. `eval(s)` is a heuristic evaluation of state s, and `sigmoid(x)` is a sigmoid transformation used to spread out heuristic evaluations to the interval $[0, 1]$. `s.giveBonus(b)` adds b winning rollouts to the MCTS value estimate of state s, and `s.removeLastBonusGiven()` removes the last bonus given to s. `finalMoveChoice()` is the same as for regular MCTS, choosing the child with the most rollouts at the root.

The parameters of MCTS-$\alpha\beta$ are the MCTS rollout probability p, the bonus weight w, the bonus weight factor f that defines how much stronger bonuses become with the depth of the completed alpha-beta search, the number of moves k for k-best pruning, and the maximum minimax depth l. When a depth-l alpha-beta search starting from state s is completed, the search depth is not further increased there. Only MCTS rollouts will be started from s in the rest of the search time.

Note that while an MCTS rollout can turn into an alpha-beta rollout at any node, a mid-rollout switch from alpha-beta back to MCTS is not possible in MCTS-$\alpha\beta$.

5 Experimental Results

We conducted preliminary experiments with MCTS-$\alpha\beta$ in the deterministic, two-player, zero-sum game of *Breakthrough*. MCTS parameters such as the exploration factor C ($C = 0.8$) were optimized for the baseline MCTS-Solver and kept constant during testing. We used minimax with alpha-beta pruning, move ordering, and iterative deepening, but no other search enhancements. Every exprimental condition consisted of 1000 games, with each player playing 500 as White and 500 as Black. All algorithms were allowed to expand 2500 nodes before making each move decision, unless specified otherwise. A node limit was chosen instead of a time limit in order to first test whether the newly proposed hybrid can search more effectively than its special cases MCTS and alpha-beta, without taking into account the additional questions of using more or less computationally expensive evaluation functions and MCTS default policies.

Subsection 5.1 describes the rules of Breakthrough as well as the evaluation function, move ordering, and default policy used for it. Subsection 5.2 shows the results.

5.1 Test Domain

The variant of Breakthrough used in our experiments is played on a 6×6 board. The game was originally described as being played on a 7×7 board, but other sizes such as 8×8 are popular as well, and the 6×6 board preserves an interesting search space.

At the beginning of the game, White occupies the first two rows of the board, and Black occupies the last two rows of the board. The two players alternatingly move one of their pieces straight or diagonally forward. Two pieces cannot occupy the same square. However, players can capture the opponent's pieces by moving

```
1  MCTSAlphaBeta(root) {
2     while(timeAvailable) {
3        if(random(0,1)<p) {
4           MCTSRollout(root)
5        } else {
6           alphaBetaRollout(root, D(root), v^-_{root,D(root)}, v^+_{root,D(root)})
7        }
8     }
9     return finalMoveChoice()
10 }
11
12 alphaBetaRollout(s, d, α_s, β_s) {
13    if( K(s) ≠ ∅ and d > 0 ) {
14       for each c ∈ K(s) do {
15          [α_c, β_c] ← [max{α_s, v^-_{c,d-1}}, min{β_s, v^+_{c,d-1}}]
16       }
17       feasibleChildren ← {c ∈ K(s) | α_c < β_c}
18       nextState ← takeSelectionPolicyMove(feasibleChildren)
19       rolloutResult ← alphaBetaRollout(nextState, d-1, α_nextState, β_nextState)
20    }
21    v^-_{s,d} ← { gameResult(s)        if s is leaf
                  { eval(s)              if d = 0
                  { max_{c∈K(s)} v^-_{c,d-1}   if d > 0 and s is internal and MAX
                  { min_{c∈K(s)} v^-_{c,d-1}   if d > 0 and s is internal and MIN
22    v^+_{s,d} ← { gameResult(s)        if s is leaf
                  { eval(s)              if d = 0
                  { max_{c∈K(s)} v^+_{c,d-1}   if d > 0 and s is internal and MAX
                  { min_{c∈K(s)} v^+_{c,d-1}   if d > 0 and s is internal and MIN
23    if(K(s) = ∅ or d = 0) {
24       rolloutResult ← v^+_{s,d}
25    }
26    return rolloutResult
27 }
28
29 MCTSRollout(currentState) {
30    if(currentState ∈ Tree) {
31       nextState ← takeSelectionPolicyMove(currentState)
32       if(random(0,1)<p or D(nextState)= l {
33          score ← MCTSRollout(nextState)
34       } else {
35          score ← alphaBetaRollout(nextState, D(nextState),
36             v^-_{nextState,D(nextState)}, v^+_{nextState,D(nextState)})
37          score ← sigmoid(score)
38          if(v^-_{nextState,D(nextState)} = v^+_{nextState,D(nextState)}) {
39             bestChild ← bestChildFoundByAlphaBetaIn(nextState)
40             bonus ← score * w * f^{D(nextState)}
41             bestChild.removeLastBonusGiven()
42             bestChild.giveBonus(bonus)
43             D(nextState) ← D(nextState)+1
44          }
45       }
46    } else {
47       addToTree(currentState)
48       simulationState ← currentState
49       while(simulationState.notTerminalPosition) {
50          simulationState ← takeDefaultPolicyMove(simulationState)
51       }
52       score ← gameResult(simulationState)
53    }
54    currentState.value ← backPropagate(currentState.value, score)
55    return score
56 }
```

Algorithm 1.3: MCTS-$\alpha\beta$.

diagonally onto their square. The game is won by the player who succeeds first at advancing one piece to the home row of her opponent, i.e. reaching the first row as Black or reaching the last row as White.

The simple evaluation function we use for Breakthrough gives the player one point for each piece of her color. The opponent's points are subtracted, and the resulting value is then normalized to the interval $[0, 1]$.

The move ordering ranks winning moves first. Second, it ranks saving moves (captures of an opponent piece that is only one move away from winning). Third, it ranks captures, and fourth, all other moves. Within all four groups of moves, moves that are closer to the opponent's home row are preferred. When two moves are ranked equally by the move ordering, they are searched in random order.

The informed default policy used for the experiments always chooses the move ranked first by the above move ordering function.

5.2 Performance of MCTS-$\alpha\beta$

In our first set of experiments, we hand-tuned the five MCTS-$\alpha\beta$ parameters against two opponents: regular alpha-beta with the same evaluation function, move ordering, and k-best pruning ($k = 10$ was found to be the strongest setting, which confirms our observations in [2] for 6×6 Breakthrough), as well as regular MCTS with the same informed default policy as described in the previous subsection. The best parameter settings found were $k = 8$, $l = 6$, $w = 200$, $f = 8$, and $p = 0.95$. With these settings, the results for MCTS-$\alpha\beta$ were a winrate of 63.7% against alpha-beta and 58.2% against MCTS. This means MCTS-$\alpha\beta$ is significantly stronger than both of its basic components ($p < 0.001$). MCTS won 63.6% of 1000 games against alpha-beta, which means that also in the resulting round-robin competition between the three players MCTS-$\alpha\beta$ performed best with 1219 won games, followed by MCTS with 1054 and alpha-beta with 727 wins in total. The optimal parameters for MCTS-$\alpha\beta$ in this scenario include a high MCTS rollout probability ($p = 0.95$), resulting in few alpha-beta rollouts being carried out. This fact seems to be agree with the strong performance of regular MCTS against regular alpha-beta. MCTS rollouts with an informed default policy seem to be more effective than alpha-beta rollouts with the primitive evaluation function described above, especially when the higher computational cost of the MCTS simulations is not taken into consideration.

In a second set of experiments, we contrasted this with a different scenario where no strong default policy is available. Instead, MCTS simulations are stopped after 3 random moves, the heuristic evaluation function is applied to the current state, and the resulting value is backpropagated as MCTS simulation return. This technique is called MCTS-IC here for consistency with previous work [4]; a similar technique where the evaluation function value is rounded to either a win or a loss before backpropagation has also been studied under the name MCTS-EPT [13]. Alpha-beta rollouts remain unchanged. MCTS-IC with the simple evaluation function we are using is weaker than MCTS with the strong informed policy described above—in a direct comparison, MCTS-IC won only 20.5% of 1000 games. In this setting, and keeping all other parameters

constant, MCTS-$\alpha\beta$ performed best with $p = 0.3$. This confirms that as soon as MCTS is weakened in comparison to alpha-beta, alpha-beta rollouts become more effective for MCTS-$\alpha\beta$ than MCTS rollouts, and the optimal value for p is lower. The results of 1000 games against the baselines were 53.7% against regular alpha-beta (not significantly different), and 73.1% against MCTS-IC (here the hybrid is significantly stronger with $p < 0.001$). With alpha-beta winning 73.1% of games against MCTS-IC as well, this means a round-robin result of 1268 wins for MCTS-$\alpha\beta$, now followed by alpha-beta with 1194 and MCTS-IC with 538 wins. Although the lack of a strong MCTS default policy has pushed alpha-beta ahead of MCTS, the hybrid algorithm still leads.

Figures 1 and 2 illustrate the performance landscape of MCTS-$\alpha\beta$ with regard to the crucial p parameter, both in the scenario with informed simulations and in the scenario with MCTS-IC simulations. Each data point results from 1000 games against regular alpha-beta.

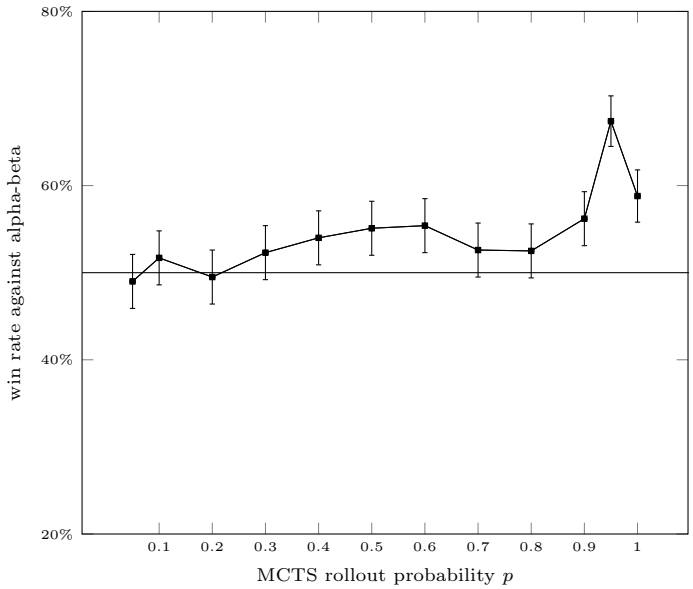

Fig. 1: Performance of MCTS-$\alpha\beta$ with informed MCTS simulations.

In a third and last set of experiments, we tested the generalization of the behavior of MCTS-$\alpha\beta$ to different time settings. In the scenario with MCTS-IC simulations, all parameter settings were left unchanged ($p = 0.3$, $k = 8$, $l = 6$, $w = 200$, $f = 8$), but all algorithms were now allowed 10000 nodes per move. MCTS-$\alpha\beta$ won 51.9% of 1000 games against regular alpha-beta, and 76.4% against regular MCTS-IC. Alpha-beta won 71.9% of games against MCTS-IC. The round-robin result is 1283 wins for MCTS-$\alpha\beta$, followed by alpha-beta with

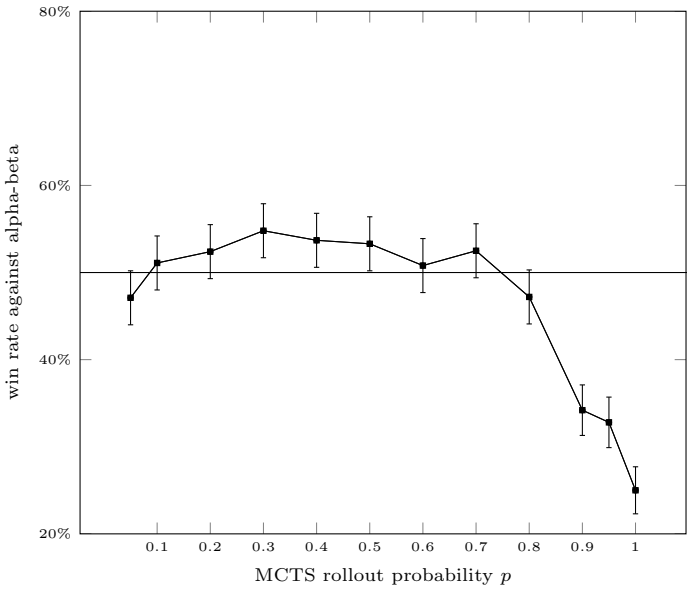

Fig. 2: Performance of MCTS-$\alpha\beta$ with MCTS-IC simulations.

1200 and MCTS-IC with 517 wins. The algorithms were also tested against each other with only 500 nodes per move—here parameter l was reduced to 4, while all other parameters stayed the same (experience with other MCTS-minimax hybrids has shown that shorter search times often profit from keeping alpha-beta more shallow [2]). For this setting, the results were 54.6% for MCTS-$\alpha\beta$ versus alpha-beta, 67.2% for MCTS-$\alpha\beta$ versus MCTS-IC, and 74.0% for alpha-beta versus MCTS-IC. Added up, this results in a round-robin with 1218 wins for MCTS-$\alpha\beta$, 1194 for alpha-beta, and 588 for MCTS-IC. In conclusion, the relative performance of MCTS-$\alpha\beta$ generalized to time settings 4 times longer as well as 5 times shorter without requiring extensive re-tuning.

6 Conclusion and Future Research

In this paper, we introduced the new hybrid search algorithm MCTS-$\alpha\beta$. It is based on MCTS rollouts and alpha-beta rollouts and unifies both search approaches under the same framework. While subsuming regular alpha-beta and regular MCTS as extreme cases, MCTS-$\alpha\beta$ opens a new space of search algorithms in between.

Preliminary results in the game of Breakthrough are promising, but do not constitute much more than a proof of concept yet. More work has to be done to gain an understanding of MCTS-$\alpha\beta$, and to further develop rollout-based MCTS-minimax hybrids. A first possible research direction is the exploration

of different design choices in the algorithm. Can alpha-beta and MCTS rollouts be more intelligently combined than by choosing them at random? How much playing strength comes from the backpropagated evaluation values, and how much from the MCTS bonuses given after alpha-beta finishes a search depth? A second direction is an analysis of the conditions under which MCTS-$\alpha\beta$ works best. Does it only show promise when the performance of MCTS and alpha-beta in the domain at hand are at least roughly comparable, or can it also improve an algorithm which is already clearly superior? How does MCTS-$\alpha\beta$ perform against MCTS and alpha-beta at equal time controls? And finally, a comparison of MCTS-$\alpha\beta$ with previously proposed hybrids would be of great interest.

Acknowledgment. The author thanks the Games and AI group, Department of Data Science and Knowledge Engineering, Maastricht University, for computational support.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-Time Analysis of the Multiarmed Bandit Problem. Machine Learning 47(2-3), 235–256 (2002)
2. Baier, H.: Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains. Ph.D. thesis, Maastricht University, Maastricht, The Netherlands (2015)
3. Baier, H., Winands, M.H.M.: Monte-Carlo Tree Search and Minimax Hybrids. In: 2013 IEEE Conference on Computational Intelligence and Games, CIG 2013. pp. 129–136 (2013)
4. Baier, H., Winands, M.H.M.: Monte-Carlo Tree Search and Minimax Hybrids with Heuristic Evaluation Functions. In: Cazenave, T., Winands, M.H.M., Björnsson, Y. (eds.) Computer Games Workshop at 21st European Conference on Artificial Intelligence, ECAI 2014. Communications in Computer and Information Science, vol. 504, pp. 45–63 (2014)
5. Bojun Huang: Pruning Game Tree by Rollouts. In: Bonet, B., Koenig, S. (eds.) Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI 2015. pp. 1165–1173. AAAI Press (2015)
6. Browne, C., Powley, E.J., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games 4(1), 1–43 (2012)
7. Chen, J., Wu, I., Tseng, W., Lin, B., Chang, C.: Job-Level Alpha-Beta Search. IEEE Transactions on Computational Intelligence and AI in Games 7(1), 28–38 (2015)
8. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) 5th International Conference on Computers and Games (CG 2006). Revised Papers. Lecture Notes in Computer Science, vol. 4630, pp. 72–83. Springer (2007)
9. Finnsson, H., Björnsson, Y.: Game-Tree Properties and MCTS Performance. In: IJCAI'11 Workshop on General Intelligence in Game Playing Agents (GIGA'11). pp. 23–30 (2011)
10. Knuth, D.E., Moore, R.W.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6(4), 293–326 (1975)

11. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) 17th European Conference on Machine Learning, ECML 2006. Lecture Notes in Computer Science, vol. 4212, pp. 282–293. Springer (2006)
12. Lanctot, M., Winands, M.H.M., Pepels, T., Sturtevant, N.R.: Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. In: 2014 IEEE Conference on Computational Intelligence and Games, CIG 2014. pp. 1–8. IEEE (2014)
13. Lorentz, R.: Early Playout Termination in MCTS. In: Plaat, A., van den Herik, H.J., Kosters, W.A. (eds.) 14th International Conference on Advances in Computer Games, ACG 2015. Lecture Notes in Computer Science, vol. 9525, pp. 12–19. Springer (2015)
14. Nijssen, J.A.M., Winands, M.H.M.: Playout Search for Monte-Carlo Tree Search in Multi-player Games. In: van den Herik, H.J., Plaat, A. (eds.) 13th International Conference on Advances in Computer Games, ACG 2011. Lecture Notes in Computer Science, vol. 7168, pp. 72–83. Springer (2011)
15. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Best-first fixed-depth minimax algorithms. Artificial Intelligence 87(1), 255–293 (1996)
16. Ramanujan, R., Sabharwal, A., Selman, B.: On Adversarial Search Spaces and Sampling-Based Planning. In: Brafman, R.I., Geffner, H., Hoffmann, J., Kautz, H.A. (eds.) 20th International Conference on Automated Planning and Scheduling, ICAPS 2010. pp. 242–245. AAAI (2010)
17. Ramanujan, R., Sabharwal, A., Selman, B.: Understanding Sampling Style Adversarial Search Methods. In: Grünwald, P., Spirtes, P. (eds.) 26th Conference on Uncertainty in Artificial Intelligence, UAI 2010. pp. 474–483 (2010)
18. Ramanujan, R., Selman, B.: Trade-Offs in Sampling-Based Adversarial Planning. In: Bacchus, F., Domshlak, C., Edelkamp, S., Helmert, M. (eds.) 21st International Conference on Automated Planning and Scheduling, ICAPS 2011. AAAI (2011)
19. Weinstein, A., Littman, M.L., Goschin, S.: Rollout-based Game-tree Search Outprunes Traditional Alpha-beta. In: Deisenroth, M.P., Szepesvári, C., Peters, J. (eds.) Tenth European Workshop on Reinforcement Learning, EWRL 2012. JMLR Proceedings, vol. 24, pp. 155–167 (2012)
20. Winands, M.H.M., Björnsson, Y.: Alpha-Beta-based Play-outs in Monte-Carlo Tree Search. In: Cho, S.B., Lucas, S.M., Hingston, P. (eds.) 2011 IEEE Conference on Computational Intelligence and Games, CIG 2011. pp. 110–117. IEEE (2011)
21. Winands, M.H.M., Björnsson, Y., Saito, J.T.: Monte-Carlo Tree Search Solver. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) 6th International Conference on Computers and Games, CG 2008. Lecture Notes in Computer Science, vol. 5131, pp. 25–36. Springer (2008)