Pruning Game Tree by Rollouts

Bojun Huang
Microsoft Research
bojhuang@microsoft.com

Abstract

In this paper we show that the α-β algorithm and its successor MT-SSS*, as two classic minimax search algorithms, can be implemented as *rollout algorithms*, a generic algorithmic paradigm widely used in many domains. Specifically, we define a family of rollout algorithms, in which the rollout policy is restricted to select successor nodes only from a certain subset of the children list. We show that *any* rollout policy in this family (either deterministic or randomized) guarantees to evaluate the game tree correctly with a finite number of rollouts. Moreover, we identify simple rollout policies in this family that "implement" α-β and MT-SSS*. Specifically, given any game tree, the rollout algorithms with these particular policies always visit the same set of leaf nodes in the same order with α-β and MT-SSS*, respectively. Our results suggest 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.

Introduction

Game tree evaluation formulates a logic process to make optimal *worst-case plans* for sequential decision making problems, a research topic usually benchmarked by two-player board games. Historically, classic game-tree search algorithms like α-β and its successors have successfully demonstrated human-champion-level performance in tactical games like Chess, but their computational complexity suffers from exponential growth with respect to the depth of the game tree. In order to reduce the size of the game tree under consideration, these traditional game-tree search algorithms have to be complemented by domain-specific *evaluation functions*, a task that is very difficult in some domains, such as GO and General Game Playing.

Recently, *rollout algorithms* have been introduced as a new paradigm to evaluate large game trees, partially because of their independence of domain-specific evaluation functions. In general, rollout algorithm is a generic algorithmic paradigm that has been widely used in many domains, such as combinatorial optimization (Bertsekas, Tsitsiklis, and Wu 1997) (Glover and Taillard 1993), stochastic optimization (Bertsekas and Castanon 1999), planning in Markov Decision Process (Péret and Garcia 2004) (Kocsis and Szepesvári 2006), and game playing (Tesauro and Galperin 1996) (Abramson 1990). In the context of game tree evaluation, a *rollout* is a process that simulates the game from the current state (the root of the game tree) to a terminating state (a leaf node), following a certain *rollout policy* that determines each move of the rollout in the state space. A rollout algorithm is a sequence of rollout processes, where information obtained from one rollout process can be utilized to reinforce the policies of subsequent rollouts, such that the long-term performance of the algorithm may converge towards the (near-)optimal policy.

In particular, a specfic class of rollout algorithms, called Monte Carlo Tree Search (MCTS), has substantially advanced the state-of-art in Computer GO (Gelly et al. 2012) and General Game Playing (Finnsson and Björnsson 2008). The key idea of MCTS is to use the average outcome of rollouts to approximate the minimax value of a game tree, which is shown to be effective in dynamic games (Coulom 2006). On the other hand, however, people also observed that existing MCTS algorithms appear to lack the capability of making "narrow lines" of tactical plans as traditional game-tree search algorithms do. For example, Ramanujan, Sabharwal, and Selman (2012) show that *UCT*, as a popular rollout algorithms in game playing, is prone to being misled by over-estimated nodes in the tree, thus often being trapped in sub-optimal solutions in some situations.

As a result, researchers have been trying to combine MCTS algorithms with traditional game tree search algorithms, in order to design unified algorithms that share the merits of both sides (Baier and Winands 2013) (Lanctot et al. 2013) (Coulom 2006). Unfortunately, one of the major difficulties for the unification is that most traditional α-β-like algorithms are based on minimax search, which seems to be a different paradigm from rollout.

In this paper, we show that two classic game-tree search algorithms, α-β and MT-SSS*, can be implemented as rollout algorithms. The observation offers a new perspective to understand these traditional game-tree search algorithms, one that unifies them with their modern "competitors" under the generic framework of rollout. Specifically,

- We define a broad family of rollout algorithms, and prove that any algorithm in this family is guaranteed to correctly evaluate game trees by visiting each leaf

A shorter version of this paper is published in AAAI 2015.

node at most once. The correctness guarantee is *policy-oblivious*, which applies to arbitrary rollout policy in the family, either deterministic or probabilistic.

- We then identify two simple rollout policies in this family that implement the classic minimax search algorithms α-β and MT-SSS* under our rollout framework. We prove that given any game tree, the rollout algorithms with these particular policies always visit the same set of leaf nodes in the same order as α-β and an "augmented" version of MT-SSS*, respectively.
- As a by-product, the"augmented" versions of these classic algorithms identified in our equivalence analysis are guaranteed to *outprune* their original versions.

Preliminaries

Game tree model

A game tree T is defined by a tuple $(\mathcal{S}, \mathcal{C}, \mathcal{V})$, where $\mathcal{S}$ is a finite state space, $\mathcal{C}(\cdot)$ is the successor function that defines an *ordered* children list $\mathcal{C}(s)$ for each state $s \in \mathcal{S}$, and $\mathcal{V}(\cdot)$ is the *value function* that defines a *minimax value* for each state $s \in \mathcal{S}$.

We assume that $\mathcal{C}(\cdot)$ imposes a tree topology over the state space, which means there is a single state that does not show in any children list, which is identified as the root node of the tree. A state s is a leaf node if its children list is empty (i.e. $\mathcal{C}(s) = \emptyset$), otherwise it is an internal node.

The value function $\mathcal{V}(\cdot)$ of a given game tree can be specified by labeling each state $s \in \mathcal{S}$ as either MAX or MIN, and further associating each leaf-node state s with a deterministic reward $R(s)$. The minimax value $\mathcal{V}(s)$ of each state $s \in \mathcal{S}$ is then defined as

$$\mathcal{V}(s) = \begin{cases} R(s) & \text{if } s \text{ is Leaf;} \\ \max_{c \in \mathcal{C}(s)} \mathcal{V}(c) & \text{if } s \text{ is Internal \& MAX;} \\ \min_{c \in \mathcal{C}(s)} \mathcal{V}(c) & \text{if } s \text{ is Internal \& MIN.} \end{cases} \quad (1)$$

To align with previous work, we assume that the reward $R(s)$ for any s is ranged in a finite set of integers, and we use the symbols $+\infty$ and $-\infty$ to denote a *finite* upper bound and lower bound that are larger and smaller than any possible value of $R(s)$, respectively. [1]

Explicit game trees often admit compact specifications in the real world (for example, as the rules of the game). Given the specification of a game tree, our goal is to compute the minimax value of the root node, denoted by $\mathcal{V}(root)$, as quickly as possible. Specifically, we measure the efficiency of the algorithm by the number of times the algorithm calls the reward function $R(\cdot)$, i.e. the number of leaf-node evaluations, which is often an effective indicator of the computation time of the algorithm in practice (Marsland 1986).

As a convenient abstraction, we suppose that any algorithm in the game tree model can access an *external storage* (in unit time) to retrieve/store a closed interval $[v_s^-, v_s^+]$ as the value range for any specified state $s \in S$. Initially, we have $[v_s^-, v_s^+] = [-\infty, +\infty]$ in the storage for all states. We remark that such a "whole-space" storage only serves as a conceptually unified interface that simplifies the presentation and analysis in this paper. In practice, we do not need to allocate physical memory for nodes with the trivial bound $[-\infty, +\infty]$. For example, the storage is physically empty if the algorithm does not access the storage at all. Such a storage can be easily implemented based on standard data structures such as a *transposition table* (Zobrist 1970).

[1]The algorithms and analysis proposed in this paper also apply to more general cases, such as when $R(s)$ is ranged in an infinite set of integers or in the real field.

Depth-first algorithms and α-β pruning

Observe that Eq. (1) implies that there must exist at least one leaf node in the tree that has the same minimax value as the root. We call any such leaf node a *critical leaf*. A *game-tree search* algorithm computes $\mathcal{V}(root)$ by searching for a critical leaf in the given game tree.

Depth-first algorithm is a specific class of game-tree search algorithms that compute $\mathcal{V}(root)$ through a depth-first search. It will evaluate the leaf nodes strictly from left to right, under the order induced by the successor function $\mathcal{C}(\cdot)$. It is well known that the α-β algorithm is the optimal depth-first algorithm in the sense that, for any game tree, no depth-first algorithm can correctly compute $\mathcal{V}(root)$ by evaluating fewer leaf nodes than α-β does (Pearl 1984).

The key idea of the α-β algorithm is to maintain an open interval (α, β) when visiting any tree node s, such that *a critical leaf is possible to locate in the subtree under s only if $\mathcal{V}(s) \in (\alpha, \beta)$*. In other words, whenever we know that the value of s falls outside the interval (α, β), we can skip over all leaf nodes under s without compromising correctness. Algorithm 1 gives the psuedocode of the α-β algorithm, which follows Figure 2 in (Plaat et al. 1996). Given a tree node s and an open interval (α, β), the alphabeta procedure returns a value g, which equals the exact value of $\mathcal{V}(s)$ if $\alpha < g < \beta$, but only encodes a lower bound of $\mathcal{V}(s)$ if $g \le \alpha$ (a situation called *fail-low*), or an upper bound of $\mathcal{V}(s)$ if $g \ge \beta$ (a situation called *fail-high*).

Note that Algorithm 1 accesses the external storage at Line 4 and Line 22, which are unnecessary because the algorithm never visits a tree node more than once. Indeed, the basic version of the α-β algorithm does not require the external storage at all. Here, we provide the "storage-enhanced" version of α-β for the sake of introducing its successor algorithm MT-SSS*.

Best-first algorithms and MT-SSS*

To a great extent, the pruning effect of the α-β algorithm depends on the *order* that the leaf nodes are arranged in the tree. In general, identifying and evaluating the "best" node early tends to narrow down the (α, β) window more quickly, thus more effectively pruning suboptimal nodes in the subsequent search. In the best case, the optimal child of any internal node is ordered first, and thus is visited before any other sibling node in the depth-first search. Knuth and Moore (1975) prove that in this case the α-β algorithm only needs to evaluate $n^{1/2}$ leaf nodes, assuming n leaves in total. In comparison, Pearl (1984) shows that this number degrades to around $n^{3/4}$ if the nodes are randomly ordered. In the worst case, it is possible to arrange the nodes in such

Algorithm 1: The α-β algorithm enhanced with storage.

```
1  $g \leftarrow$ alphabeta($root$, $-\infty$, $+\infty$) ;
2  return $g$
3  Function alphabeta($s$, $\alpha$, $\beta$)
4      retrieve $[v_s^-, v_s^+]$ ;
       if $v_s^- \geq \beta$ then return $v_s^-$ ;
       if $v_s^+ \leq \alpha$ then return $v_s^+$;
5      if $s$ is a leaf node then
6          $g \leftarrow R(s)$ ;
7      else if $s$ is a MAX node then
8          $g \leftarrow -\infty$ ; $\alpha_s \leftarrow \alpha$ ;
9          foreach $c \in \mathcal{C}(s)$ do
10             $g \leftarrow \max\{ g$ , alphabeta$(c, \alpha_s, \beta) \}$ ;
11             $\alpha_s \leftarrow \max\{ \alpha_s , g \}$ ;
12             if $g \geq \beta$ then break;
13         end
14     else if $s$ is a MIN node then
15         $g \leftarrow +\infty$; $\beta_s \leftarrow \beta$ ;
16         foreach $c \in \mathcal{C}(s)$ do
17             $g \leftarrow \min\{ g$ , alphabeta$(c, \alpha, \beta_s) \}$ ;
18             $\beta_s \leftarrow \min\{ \beta_s , g \}$ ;
19             if $g \leq \alpha$ then break;
20         end
21     end
22     if $g < \beta$ then $v_s^+ \leftarrow g$;
       if $g > \alpha$ then $v_s^- \leftarrow g$;
       store $[v_s^-, v_s^+]$ ;
23     return $g$
```

a way that the α-β algorithm has to evaluate all of the n leaf nodes in the tree.

Best-first algorithm is a class of algorithms that always try to evaluate the node that currently looks more "promising" (to be or contain a critical leaf). In particular, *The SSS* algorithm*, first proposed by Stockman (1979) and later revised by Campbell (1983) , is a classic best-first algorithm that is guaranteed to "*outprune*" the α-β algorithm in the sense that: SSS* never evaluates a leaf node that is not evaluated by α-β, while for some problem instances SSS* manages to evaluate fewer leaf nodes than α-β. The SSS* algorithm is based on the notion of *solution tree*. The basic idea is to treat each MAX node as a cluster of solution trees, and to always prefer to search the leaf node that is nested in the solution-tree cluster currently with the best upper bound.

Interestingly, Plaat et al. (1996) show that SSS*, as a best-first algorithm, can be implemented as *a series of* storage-enhanced depth-first searches. Each pass of such a depth-first search is called a *Memory-enhanced Test*, so this version of SSS*is also called MT-SSS*. Plaat et al. prove that for any game tree, MT-SSS* visits the same set of leaf nodes in the same order with the original SSS* algorithm proposed by Stockman.

Algorithm 2 gives the psuedocode of MT-SSS*. Instead of directly determining the exact value of $\mathcal{V}(root)$ in a single pass, the algorithm iteratively calls the alphabeta procedure (in Algorithm 1) to refine the value bounds of the root until the gap between the bounds is closed. Each pass of the alphabeta procedure is for examining the same question: *Is the current upper bound of the root tight*? The algorithm calls the alphabeta procedure with the *minimum window* $(v_{root}^+ - 1, v_{root}^+)$, [2] which forces the procedure to only visit the leaf nodes "relevant" to this question. In this case the alphabeta procedure will answer this question by returning either a new upper bound that is lower than the original bound v_{root}^+, or a matching lower bound that equals v_{root}^+. Note that the alphabeta procedure stores value bounds in the external storage, so latter iterations can re-use the results gained in previous iterations, avoiding repeated work. Since SSS* has been proven to outprune α-β, the total number of leaf-node evaluations over all passes of minimum-window searches in MT-SSS* will never exceed the number made by a single pass of full-window search. On the other hand, in cases when the given game tree is in a "bad" order, the best-first working style of MT-SSS* can help to significantly reduce the number of leaf-node evaluations.

Algorithm 2: The MT-SSS* algorithm.

```
1  $[v_s^-, v_s^+] \leftarrow [-\infty, +\infty]$ for every $s \in \mathcal{S}$;
2  while $v_{root}^- < v_{root}^+$ do
3      alphabeta($root, v_{root}^+ - 1, v_{root}^+$) ;
4  end
5  return $v_{root}^-$
```

Monte Carlo Tree Search and UCT

While α-β and MT-SSS* can offer substantial improvement over exhaustive tree search, both of them still have to run in time exponential to the depth of the tree (Pearl 1984), which limits the tree size they can directly deal with. In practice, these algorithms typically need to be complemented with a static *evaluation function* that can make heuristic estimation on the minimax value of an arbitrarily given non-leaf node, resulting in the "bounded look-ahead" paradigm (Reinefeld and Marsland 1994). In such a paradigm, an internal node s may be considered as a "virtual leaf node" (or *frontier node*) under certain conditions, and in that case the evaluation function is applied to give the reward value of this virtual leaf node, with the whole sub-tree under s being cut-off from the tree search. The hope is that the evaluation function can reasonably approximate $\mathcal{V}(s)$ at these virtual leaf nodes such that the result of searching only the partial tree above these virtual leaves is similar to the result of a complete tree search. However, depending on domains it can sometimes be highly challenging to design a satisfactory evaluation function and cut-off conditions.

Monte Carlo Tree Search (MCTS) is an alternative algorithmic paradigm that can evaluate large game trees without sophisticated evaluation functions (Coulom 2006). As

[2]Recall that the rewards are assumed to be integers, in which case the open interval $(x-1, x)$ is essentially empty if x is integer.

Algorithm 3: The UCT algorithm, under given time budget T and parameter λ.

```
1  $n_s \leftarrow 0$ , $\mu_s \leftarrow 0$ for every $s \in \mathcal{S}$;
2  while $n_{root} < T$ do rollout($root$) ;
3  return $\mu_{root}$
4  Function rollout($s$)
5      if $s$ is a leaf node then
6          $g \leftarrow R(s)$ ;
7      else if $s$ is a MAX node then
8          $c^* \leftarrow \arg\max_{c \in \mathcal{C}(s)} \ \mu_c + \lambda\sqrt{\ln n_s / n_c}$ ;
9          $g \leftarrow$ rollout($c^*$) ;
10     else if $s$ is a MIN node then
11         $c^* \leftarrow \arg\min_{c \in \mathcal{C}(s)} \ \mu_c - \lambda\sqrt{\ln n_s / n_c}$ ;
12         $g \leftarrow$ rollout($c^*$) ;
13     end
14     $\mu_s \leftarrow \frac{n_s}{n_s+1}\mu_s + \frac{1}{n_s+1} g$ ;
15     $n_s \leftarrow n_s + 1$ ;
16     return $g$
```

a specific class of rollout algorithms, MCTS algorithms repeatedly perform rollouts in the given game tree and use the average outcome of the rollouts to approximate the minimax value of the tree (Abramson 1990).

Among others, the *UCT* algorithm is a particular instance of MCTS algorithms that has drawn a lot of attention in the community (Kocsis and Szepesvári 2006). Algorithm 3 shows the pseudo-code of UCT. At a tree node s, the algorithm uses a deterministic rollout policy that computes for each successor node c a score $UCT(c) = \mu_c + \lambda\sqrt{\ln n_s / n_c}$, where μ_c is the average reward of the previous rollouts passing through c, n_c is the number of such rollouts, and n_s is the number of rollouts through s. Then the algorithm simply chooses the successor node with the highest score. One can check that the UCT score will approach to the average reward μ_c if the current node s has been extensively visited. In contrast, when the sample size n_s is small, less-visited successor nodes can get substantial bonus in their score, thus may get chance to be explored even if it has a low average reward μ_c. The trade-off between exploitation and exploration can be controlled by fine-tuning the parameter λ, such that the resulting footprints of the rollouts are "softly" biased to the most promising variations. Kocsis and Szepesvári (2006) proved that the outcome of UCT always converges to the minimax value $\mathcal{V}(root)$ if given infinite time.

A Family of Rollout Algorithms

Rollout algorithms perform a rollout by iteratively selecting a successor node at each node s along a top-down path starting from the root. In general, a rollout algorithm may select the successor node according to an arbitrary probability distribution over the *whole* children list $\mathcal{C}(s)$. However, the idea of α-β pruning suggests that it may be unnecessary to consider every successor node for computing the value of $\mathcal{V}(root)$. In this section we present a family of rollout algorithms that follows this observation by restricting the successor node selection to be over a *subset* of $\mathcal{C}(s)$. As shown later, this family of rollout algorithms naturally encompasses the ideas of traditional minimax search algorithms.

Recall that at any time we know from the external storage a value range $[v_c^-, v_c^+]$ for each tree node $c \in \mathcal{S}$. We start with identifying an important property of the knowledge in the storage. Specifically, we say that the storage is *valid* with respect to a given game tree T if $v_s^- \leq \mathcal{V}(s) \leq v_s^+$ for any node s in T. Moreover, we define that a valid storage is *coherent* to the given game tree if its validity is robust to the uncertainty itself claims.

Definition 1. *Given a game tree $T = (\mathcal{S}, \mathcal{C}, \mathcal{V})$, a storage $M = \{[v_s^-, v_s^+]\}_{s \in \mathcal{S}}$ is **coherent**, with respect to T, if i) M is valid to T; and ii) For any leaf node $c \in \mathcal{S}$ and for any $r \in [v_c^-, v_c^+]$, let $T' = (\mathcal{S}, \mathcal{C}, \mathcal{V}')$ be the game tree obtained by setting $R(c) = r$ in the original tree T, M is valid to T'.*

A coherent storage enables a sufficient condition to ignore a tree node c (as well as the subtree rooted at c) in the search for the value of $\mathcal{V}(root)$. Specifically, let $\mathcal{P}(c)$ be the set of tree nodes between and including the root node and node c (so $\mathcal{P}(root) = \{root\}$). For each tree node c, define $[\alpha_c, \beta_c]$ as the intersection interval of the value ranges of all nodes in $\mathcal{P}(c)$. That is,

$$[\alpha_c, \beta_c] = \bigcap_{s \in \mathcal{P}(c)} [v_s^-, v_s^+], \tag{2}$$

or equivalently, in practice we can compute α_c and β_c by

$$\alpha_c = \max_{s \in \mathcal{P}(c)} v_s^- \quad , \quad \beta_c = \min_{s \in \mathcal{P}(c)} v_s^+. \tag{3}$$

The following lemma shows that if the uncertainty in the storage is coherent, then under certain condition, not only is the value range $[v_{root}^-, v_{root}^+]$ stable to the "disturbance" from lower layers of the tree, but the exact minimax value $\mathcal{V}(root)$ is also stable. The key insight is to see that both max and min are *monotone* value functions.

Lemma 1. *Given any game tree $T = (\mathcal{S}, \mathcal{C}, \mathcal{V})$, let $M = \{[v_s^-, v_s^+]\}_{s \in \mathcal{S}}$ be a storage coherent to T. For any leaf node $c \in \mathcal{S}$ and for any $r \in [v_c^-, v_c^+]$, let $T' = (\mathcal{S}, \mathcal{C}, \mathcal{V}')$ be the game tree obtained by setting $R(c) = r$ in the original tree T, then $\mathcal{V}'(root) = \mathcal{V}(root)$ if $\alpha_c \geq \beta_c$.*

Proof. First observe that the lemma trivially holds when the leaf node c is at depth 1, i.e. when it is the root node – in that case we have $[\alpha_c, \beta_c] = [v_{root}^-, v_{root}^+]$, and thus $\alpha_c \geq \beta_c$ implies $v_{root}^- = v_{root}^+$ if the storage M is valid. For leaf node c with a depth larger than 1, by induction we only need to prove the lemma assuming that it holds for all ancestor nodes of the c (or equivalently, for each such ancestor node we assume the lemma holds for another tree in which the subtree of this ancestor node is pruned).

Let s be the parent node of c. For contradiction assume c *could* affect the value of root, i.e., for leaf node c with $\alpha_c \geq \beta_c$, $\mathcal{V}'(root) \neq \mathcal{V}(root)$ when $R(c)$ changes to some $r \in [v_c^-, v_c^+]$. In that case we must have $\alpha_s < \beta_s$, because otherwise s cannot affect the value of the root (which is assumed by induction), and neither can its successor node c.

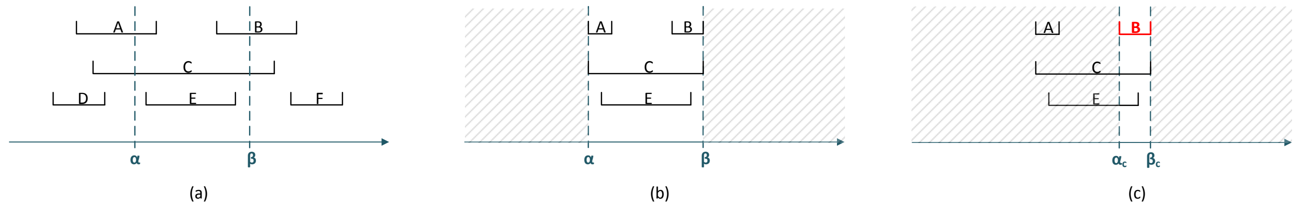

Figure 1: Illustration of one downward step in the rollout process of the algorithm family.

Now we have $\alpha_s < \beta_s$ and $\alpha_c \geq \beta_c$. Recall that $[\alpha_c, \beta_c] = [\alpha_s, \beta_s] \cap [v_c^-, v_c^+]$, which means we have either (i) $v_c^- = v_c^+$, in which case the lemma trivially holds; or (ii) $\beta_s \leq v_c^-$; or (iii) $v_c^+ \leq \alpha_s$. We only discuss case (ii) in the following, and the argument for case (iii) is symmetrical.

Since $\beta_s \leq v_c^-$, and by Eq.(3), $\beta_s = \min_{t \in \mathcal{P}(s)} v_t^+$, there must exist a node $s^* \in \mathcal{P}(s)$ such that $v_{s^*}^+ \leq v_c^-$. That is, there is no overlap between the value ranges of c and s^* (except the boundary). Now we suppose $R(c) = v_c^-$ in T, and prove that $\mathcal{V}'(s^*) = \mathcal{V}(s^*)$ if $R(c)$ increases from v_c^- to any $r > v_c^-$. In that case it immediately follows that the value of s^* must also remain constant if $R(c)$ is further changing between such r's (because any of them equals the $\mathcal{V}(s^*)$ when $R(c) = v_c^-$).

The key insight is to see that both the max function and the min function are *monotone*, and so does any recursive function defined by Eq.(1). Specifically, because $v_{s^*}^+ \leq v_c^-$, and because $v_{s^*}^+$ is valid upper bound, we have $\mathcal{V}(s^*) \leq v_c^-$.

Case 1: When $\mathcal{V}(s^*) = v_c^-$. Because $\mathcal{V}'(\cdot)$ is monotone and $r > v_c^-$, we have $\mathcal{V}'(s^*) \geq \mathcal{V}(s^*) = v_c^-$. On the other hand, recall that we already have $v_{s^*}^+ \leq v_c^-$, and because the storage M is valid to T', we must have $\mathcal{V}'(s^*) \leq v_{s^*}^+ \leq v_c^-$. For both the inequalities about $\mathcal{V}'(s^*)$ to be true, the only possibility is $\mathcal{V}'(s^*) = v_c^-$, and thus $\mathcal{V}'(s^*) = \mathcal{V}(s^*)$.

Case 2: When $\mathcal{V}(s^*) < v_c^-$. Because $\mathcal{V}(c) = v_c^-$, we have $\mathcal{V}(s^*) < \mathcal{V}(c)$. One can check that in this case there must exist an s' and its successor node c_1, both on the path between s^* and c (included), such that $\mathcal{V}(s') < v_c^- \leq \mathcal{V}(c_1)$. Notice that this can only happen when s' is a MIN node and there is another successor node of s', denoted by c_2, such that $\mathcal{V}(c_2) = \mathcal{V}(s') < \mathcal{V}(c_1)$. In other words, the value of s' must be currently dominated by c_2, and not by c_1. Again, due to the monotonicity of the minimax function, when $R(c)$ increases from v_c^- to $r > v_c^-$, the value of c_1 must become even larger, if ever changed. On the other hand, all the other successor nodes of s^* – including c_2 – are not ancestors of c, so their values will not change when $R(c)$ changes. Therefore, we know that the value of s' must still be dominated by c_2 after $R(c)$ changes, i.e. $\mathcal{V}'(s') = \mathcal{V}'(c_2) = \mathcal{V}(c_2) = \mathcal{V}(s')$. Since s' is on the path between s^* and c, we also have $\mathcal{V}'(s^*) = \mathcal{V}(s^*)$.

Finally, since we have proven that the leaf node c cannot affect the value of its ancestor node s^*, it immediately follows that c cannot affect the root node either, a contradiction to the assumption made at the beginning of the proof. □

Since Lemma 1 guarantees that a successor node c with $\alpha_c \geq \beta_c$ cannot affect $\mathcal{V}(root)$ (nor does any node in the subtree of c), it is safe for rollout algorithms to select successor nodes, at any node s, *only* from the collection of successor nodes with $\alpha_c < \beta_c$, denoted by

$$\mathcal{A}_s = \{c \in \mathcal{C}(s) \mid \alpha_c < \beta_c\}. \tag{4}$$

Algorithm 4 presents a family of rollout algorithms that embodies this idea. Algorithms in this family keep performing rollouts until the value range $[v_{root}^-, v_{root}^+]$ is closed. In each round of rollout, the specific rollout trajectory depends on the SelectionPolicy() routine, which selects a successor node c^* from the subset $\mathcal{A}_s$ for the next move. The selection of c^* can be either based on deterministic rules or sampled from a probability distribution over $\mathcal{A}_s$. An algorithm instance of this family is fully specified once the SelectionPolicy routine is concretely defined.

To keep the storage coherent, Algorithm 4 updates the value range $[v_s^-, v_s^+]$ for each node s along the trajectory of rollout, in a bottom-up order. The value bounds are updated directly based on the minimax function defined by Eq.(1). It is not hard to see that the storage of Algorithm 4 is always in a coherent state after each round of the rollout. Finally, Algorithm 4 computes α_c and β_c in an incremental way, as illustrated by Figure 1.

In the following, we present some nice properties that are shared between *all* algorithms in the family of Algorithm 4. First, Lemma 2 shows that all data structures used in Algorithm 4 changes monotonically over time.

Lemma 2. *Given any game tree T, and under any selection policy (deterministic or randomized), for any $s \in \mathcal{S}$, the set of $\mathcal{A}_s$ in Algorithm 4 is non-increasing over time, and so do for the intervals $[v_s^-, v_s^+]$ and $[\alpha_s, \beta_s]$.*

Proof. It can be directly seen that the interval $[v_s^-, v_s^+]$ can never increase in Algorithm 4. By definition, i.e. Eq.(3), this implies the non-increasing monotonicity of $[\alpha_s, \beta_s]$, which in turn implies the monotonicity of $\mathcal{A}_s$, due to Eq.(4). □

Lemma 2 suggests that once a node is excluded from $\mathcal{A}_s$, it cannot come back. In that sense, the rollout algorithms of Algorithm 4 is indeed using $\mathcal{A}_s$ to *prune* the game tree. On the other hand, one might worry that some algorithm in this family could potentially be "stuck" at some point, when there is no candidate in the subset $\mathcal{A}_s$. It turns out that this can never happen, regardless of the selection policy the algorithm is using.

Algorithm 4: A family of rollout algorithms.

```
$[v_s^-, v_s^+] \leftarrow [-\infty, +\infty]$ for every $s \in \mathcal{S}$;
while $v_{root}^- < v_{root}^+$ do
    rollout($root, v_{root}^-, v_{root}^+$) ;
end
return $v_{root}^-$

Function rollout($s$, $\alpha_s$, $\beta_s$)
    if $\mathcal{C}(s) \neq \emptyset$ then
        foreach $c \in \mathcal{C}(s)$ do
            $[\alpha_c, \beta_c] \leftarrow [\max\{\alpha_s, v_c^-\}, \min\{\beta_s, v_c^+\}]$ ;
        end
        $\mathcal{A}_s = \{c \in \mathcal{C}(s) \mid \alpha_c < \beta_c\}$ ;
        $c^* \leftarrow$ SelectionPolicy($\mathcal{A}_s$) ;
        rollout($c^*, \alpha_{c^*}, \beta_{c^*}$) ;
    end
    $v_s^- \leftarrow \begin{cases} R(s) & \text{if } s \text{ is Leaf} \\ \max_{c \in \mathcal{C}(s)} v_c^- & \text{if } s \text{ is Internal \& MAX} \\ \min_{c \in \mathcal{C}(s)} v_c^- & \text{if } s \text{ is Internal \& MIN} \end{cases}$ ;
    $v_s^+ \leftarrow \begin{cases} R(s) & \text{if } s \text{ is Leaf} \\ \max_{c \in \mathcal{C}(s)} v_c^+ & \text{if } s \text{ is Internal \& MAX} \\ \min_{c \in \mathcal{C}(s)} v_c^+ & \text{if } s \text{ is Internal \& MIN} \end{cases}$ ;
    return
```

Lemma 3. *Given any game tree T, and under any selection policy (deterministic or randomized), Algorithm 4 always runs the rollout() procedure on node s such that $\mathcal{A}_s = \emptyset$ if and only if $\mathcal{C}(s) = \emptyset$.*

Proof. It is sufficient to prove that if Algorithm 4 visits a non-leaf node s (i.e. $\mathcal{C}(s) \neq \emptyset$), then $\mathcal{A}_s \neq \emptyset$. We prove this by showing that for any non-leaf node s that Algorithm 4 visits, there always exists at least one successor node $c \in \mathcal{C}(s)$ such that $[v_c^-, v_c^+] \supseteq [v_s^-, v_s^+]$, i.e. c has a wider (or the same) value range than s.

We only discuss the case when s is a non-root MAX node. The argument is similar in other cases. If s is an internal MAX node, according to Algorithm 4 we have $v_s^+ = \max_{c \in \mathcal{C}(s)} v_c^+$, so there exists a successor node c^* such that $v_{c^*}^+ = v_s^+$. On the other hand, since $v_s^- = \max_{c \in \mathcal{C}(s)} v_c^-$, we have $v_{c^*}^- \leq v_s^-$. Thus, $[v_{c^*}^-, v_{c^*}^+] \supseteq [v_s^-, v_s^+]$.

Given such a successor node c^*, let t be the parent node of s, according to Algorithm 4 we have $\alpha_s = \max\{\alpha_t, v_s^-\} = \max\{\alpha_t, v_s^-, v_{c^*}^-\} = \max\{\alpha_s, v_{c^*}^-\} = \alpha_{c^*}$. Similarly we also have $\beta_s = \beta_{c^*}$. Because Algorithm 4 visits node s, we must have $\alpha_s < \beta_s$, thus $\alpha_{c^*} < \beta_{c^*}$, thus c^* is in $\mathcal{A}_s$. □

Finally, we observe that the family of Algorithm 4 is *consistent*, in the sense that all algorithm instances in the family always return the same result, and this policy-independent result of Algorithm 4 is always correct, as Theorem 1 shows.

Theorem 1. *Given any game tree T and under any selection policy (deterministic or randomized), Algorithm 4 never visit a leaf node more than once, and always terminates with $v_{root}^- = v_{root}^+ = \mathcal{V}(root)$.*

Proof. First, we see that Algorithm 4 never visits a node c with $v_c^- = v_c^+$, because in that case $\alpha_c = \beta_c$. The first time Algorithm 4 visits a leaf node s, it sets $v_s^- = v_s^+ = R(s)$, so the algorithm never re-visits a leaf node, which means it will have to terminate, at its latest, after visiting every leaf node. According to Line 2, we have $v_{root}^- = v_{root}^+$ at that time. Since the way Algorithm 4 updates the value bounds guarantees that they are always valid – that is, at any time we have $v_s^- \leq \mathcal{V}(s) \leq v_s^+$ for any node s – we know that v_{root}^- must equal $\mathcal{V}(root)$ when $v_{root}^- = v_{root}^+$. □

Note that Theorem 1 shows a stronger "consistency" property than that of some other rollout algorithms, such as UCT (Kocsis and Szepesvári 2006), which only guarantees to *converge to* $\mathcal{V}(root)$ if given infinite time. In contrast, Algorithm 4 never re-visits a leaf even under a probabilistic rollout policy, thus always terminating in finite time.

Two Rollout Policies: α-β and MT-SSS*

Since the rollout family of Algorithm 4 uses an α-β window to prune tree nodes during the computation, one may wonder how the classic game-tree pruning algorithms are compared to Algorithm 4. In this section we show that two simple "greedy" policies in the family of Algorithm 4 are equivalent, in a strict way, to an "augmented" version of the classic α-β and MT-SSS* algorithm, respectively.

To establish the strict equivalence, we introduce a variant of the classic alphabeta procedure, as Algorithm 5 shows, which differs from the alphabeta procedure of Algorithm 1 only in two places: (1) The classic alphabeta procedure in Algorithm 1 only returns a single value of g, but the alphabeta2 procedure in Algorithm 5 transmits *a pair of* values $\{g^-, g^+\}$ between its recursive calls. (2) The classic alphabeta procedure in Algorithm 1 initializes the (α_s, β_s) window directly with the received argument (α, β), while the alphabeta2 procedure in Algorithm 5 will further trim $[\alpha_s, \beta_s]$ with the value range $[v_s^-, v_s^+]$.

Before comparing these two versions more carefully, we first establish the equivalence between the alphabeta2 procedure, as well as the MT-SSS* algorithm based on the alphabeta2 procedure, with two simple rollout policies of the algorithm family proposed in the last section. Specifically, Theorem 2 shows that the alphabeta2 procedure is equivalent to a "left-first" policy of Algorithm 4, in the sense not only that they have the same footprints in leaf evaluation, but also that given *any* coherent storage (not necessarily an "empty" storage), they always leave identical content in their respective storages when terminating. This means they are still equivalent in a *reentrant* manner, even when used as subroutines by other algorithms.

Theorem 2. *Given any game tree $T = (\mathcal{S}, \mathcal{C}, \mathcal{V})$ and any coherent storage $M = \{[v_s^-, v_s^+]\}_{s \in \mathcal{S}}$, Algorithm 4 always evaluates the same set of leaf nodes in the same order as the "augmented" α-β algorithm (Algorithm 5) does, if Algorithm 4 is using the following selection policy:*

$$c^* = \text{the leftmost } c \text{ in } \mathcal{A}_s. \quad (5)$$

Moreover, let $M_{rollout}$ and $M_{\alpha\beta}$ be the storage states when Algorithm 4 and Algorithm 5 terminate, respectively. We have $M_{rollout} = M_{\alpha\beta}$, when the policy of Eq.(5) is used.

The key insight here is to see that under the "left-first" policy, Algorithm 4 can *locally* compute the new window $[\alpha_s, \beta_s]$ of the next round right at node s, without updating the ancestor nodes of s through back-propagation, a trick that turns the rollout algorithm into a backtracking algorithm. The complete proof consists of a series of equivalent transformations of algorithms, which is given in a later section due to its length.

It is easy to see from Theorem 2 that the MT-SSS* algorithm, if using the "augmented" alphabeta2 procedure, can also be implemented by a sequence of rollouts with the "left-first" policy, although such a rollout algorithm will not belong to the family of Algorithm 4. Interestingly, however, it turns out that the "augmented" MT-SSS algorithm can be encompassed by the rollout paradigm in a more *unified* way. In fact, Theorem 3 shows that the MT-SSS* algorithm is strictly equivalent to another policy of the same rollout family of Algorithm 4. Instead of selecting the leftmost node as the rollout policy of α-β does, the rollout policy of MT-SSS* selects the node with the largest β_c.

Theorem 3. *Given any game tree $T = (\mathcal{S}, \mathcal{C}, \mathcal{V})$ and any coherent storage $M = \{[v_s^-, v_s^+]\}_{s \in \mathcal{S}}$, Algorithm 4 always evaluates the same set of leaf nodes in the same order as the "augmented" MT-SSS* algorithm (Algorithm 2 + Algorithm 5) does, if Algorithm 4 is using the following selection policy:*

$$c^* = \text{ the leftmost c in } \arg\max_{c \in \mathcal{A}_s} \beta_c \qquad (6)$$

Moreover, let $M_{rollout}$ and M_{SSS^*} be the storages when Algorithm 4 and Algorithm 5 terminate, respectively. We have $M_{rollout} = M_{SSS^*}$, when the policy of Eq.(6) is used.

The "augmented" α-β and MT-SSS*

Given the strict equivalence between the rollout algorithms of Algorithm 4 and classic tree-search algorithms based on the alphabeta2 procedure, we now examine the relationship between the two variants of α-β presented in Algorithms 1 and 5. As mentioned before, the alphabeta2 procedure in Algorithm 5 captures every aspect of the original alphabeta procedure in Algorithm 1 except for two differences.

The alphabeta procedure in Algorithm 1 returns a single value of g, while the alphabeta2 procedure in Algorithm 5 returns a value pair $\{g^-, g^+\}$. From the psuedo-code one can check that $g = g^- = v_s^-$ when it is a fail-high and $g = g^+ = v_s^+$ when it is a fail-low, otherwise $g = g^- = g^+ = \mathcal{V}(s)$. This is consistent with the well-known protocol of the original α-β algorithm. The difference is that the alphabeta2 procedure tries to update *both* bounds even in fail-high and fail-low cases. As a result, we can expect that in some cases the alphabeta2 procedure will result in a storage with tighter value bounds than the one of the classic alphabeta procedure.

Meanwhile, the classic alphabeta procedure in Algorithm 1 initializes the (α_s, β_s) window directly with the received argument (α, β), while the alphabeta2 procedure in Algorithm 5 will further trim $[\alpha_s, \beta_s]$ with the value range $[v_s^-, v_s^+]$. In other words, even given the same storage, the alphabeta2 procedure may have a tighter pruning window than its counterpart.

Algorithm 5: A variant of the alphabeta procedure, which returns a pair of value bounds.

```
1  return alphabeta2(root, $-\infty$, $+\infty$)
2  Function alphabeta2(s, $\alpha$, $\beta$)
3      retrieve $[v_s^-, v_s^+]$ ;
4      $[\alpha_s, \beta_s] \leftarrow [\max\{\alpha, v_s^-\}, \min\{\beta, v_s^+\}]$ ;
5      if $\alpha_s \geq \beta_s$ then return $[v_s^-, v_s^+]$;
6      if s is a Leaf node then
7          $[g^-, g^+] \leftarrow [R(s), R(s)]$ ;
8      else if s is a MAX node then
9          $[g^-, g^+] \leftarrow [-\infty, -\infty]$ ;
10         foreach $c \in \mathcal{C}(s)$ do
11             $\{g_c^-, g_c^+\} \leftarrow$ alphabeta2(c, $\alpha_s$, $\beta_s$) ;
12             $[g^-, g^+] \leftarrow [\max\{g^-, g_c^-\}, \max\{g^+, g_c^+\}]$;
13             $[\alpha_s, \beta_s] \leftarrow [\max\{\alpha_s, g_c^-\}, \beta_s]$ ;
14         end
15     else if s is a MIN node then
16         $[g^-, g^+] \leftarrow [+\infty, +\infty]$ ;
17         foreach $c \in \mathcal{C}(s)$ do
18             $\{g_c^-, g_c^+\} \leftarrow$ alphabeta2(c, $\alpha_s$, $\beta_s$) ;
19             $[g^-, g^+] \leftarrow [\min\{g^-, g_c^-\}, \min\{g^+, g_c^+\}]$ ;
20             $[\alpha_s, \beta_s] \leftarrow [\alpha_s, \min\{\beta_s, g_c^+\}]$ ;
21         end
22     end
23     $[v_s^-, v_s^+] \leftarrow [g^-, g^+]$ ;
       store $[v_s^-, v_s^+]$ ;
24     return $\{g^-, g^+\}$
```

While the second difference may look like a small trick at the implementation level, we believe that the "single-bound v.s. double-bound" disparity is an inherent difference between the two versions. Since the storage-enhanced α-β algorithm requires maintaining a pair of bounds anyway (even for the "single-bound" version), it makes sense to update both of them effectively at run time.

Interestingly, the "single-bound" vesion of the alphabeta procedure performs exactly as well as its "double-bound" version *if* they are working on an empty storage with only one pass. This is directly followed from the well-known fact that the classic alphabeta procedure is per-instance optimal in all *directional* algorithms (Pearl 1984). However, when used as subroutines, they will behave differently. It turns out that the MT-SSS* algorithm using the alphabeta2 procedure can *outprune* the one based on the single-bound version, in the same way as the SSS* algorithm outprunes the classic α-β algorithm.

Theorem 4. *Given any game tree T, let L_{sss^*} be the sequence of leaf nodes evaluated by the original MT-SSS* algorithm that calls Algorithm 1, and let L_{sss^+} be the sequence of leaf nodes evaluated by the "augmented" MT-*

SSS algorithm that calls Algorithm 5, then L_{sss^+} is a subsequence of L_{sss^*}.*

Proof of Theorem 2

In this section we prove Theorem 2, which states that the "left-first" policy in the family of Algorithm 4 is equivalent to the "augmented" α-β algorithm shown by Algorithm 5. The key insight here is to see that under the "left-first" policy, Algorithm 4 can *locally* compute the new window $[\alpha_s, \beta_s]$ of the next round right at node s, without updating the ancestor nodes of s through back-propagation, a trick that turns the backtracking algorithm into a rollout algorithm. The complete proof consists of a series of equivalent transformations of algorithms, along Algorithm 4 $\rightarrow$ 6 $\rightarrow$ 7 $\rightarrow$ 8 $\rightarrow$ 9 $\rightarrow$ 10 $\rightarrow$ 5.

Proof. To prepare the proof, we need to define a "wrapper" procedure that generalizes Algorithm 4 a little bit, as Algorithm 6 shows. It is easy to see that Algorithm 6 is identical to Algorithm 4 when $s = root$, $\alpha_s = -\infty$, $\beta_s = +\infty$. On the other hand, Algorithm 6 may terminate without closing the range of $[v^-_{root}, v^+_{root}]$ if $\mathcal{V}(root)$ is outside the open interval (α_s, β_s).

Algorithm 6: A wrapper procedure of Algorithm 4.

```
1 Function wrapper1(s, $\alpha_s$, $\beta_s$)
2     while $\alpha_s < \beta_s$ do
3         rollout(s, $\alpha_s$, $\beta_s$) ;
4         $[\alpha_s, \beta_s] \leftarrow [\max\{\alpha_s, v^-_s\}, \min\{\beta_s, v^+_s\}]$ ;
5     end
6     return
```

In this proof, we say two algorithms $A1$ and $A2$ are *equivalent* if for any input (T, M, α, β) their behaviors – including both the leaf-node footprints and the terminating state of the storage – are identical. That is, let L_{A1} and L_{A2} be the sequence of leafs evaluated by $A1$ and $A2$ (respectively), and let M_{A1} and M_{A2} be the states of storage when $A1$ and $A2$ terminate (respectively), we say $A1$ and $A2$ are equivalent if we have $(L_{A1}, M_{A1}) = (L_{A2}, M_{A2})$ for any input (T, M, α, β). As another shortcut, we say a tree node c is *active* if at the given time we have $\alpha_c < \beta_c$.

To prove the theorem, it is sufficient to prove that the wrapper1 procedure in Algorithm 6 is equivalent to the alphabeta2 procedure in Algorithm 5 if $[\alpha, \beta] \subseteq [v^-_s, v^+_s]$ in the alphabeta2 procedure. Observe that Algorithm 5 works in the backtracking manner, while Algorithm 6 works in the rollout manner. Consider the moment when the execution of the rollout procedure at a node s is about to end (for example, imagine we are at Line 16 of Algorithm 4). According to the rollout paradigm, the algorithm will now update the value bounds of all ancestor nodes of s, then re-start another rollout round from the root. Under the specific policy of Eq.(5), the rollout process will always select, at each layer of the tree, the leftmost *active* node c. Notice that s is the chosen node at its own layer for the current round of rollout, which means all nodes at the left side of s (in the same layer) are already "inactive". Since Lemma 2 shows that the $[\alpha, \beta]$ window is non-increasing for any node, we know that the current node s will still be chosen in the next round of rollout if and only if $\alpha_s < \beta_s$ in the next round.

The key insight of the proof is to see that for the rollout algorithm of Algorithm 4, we can *locally* compute at node s the new window $[\alpha_s, \beta_s]$ of the next round, without updating any ancestor node of s through back-propagation. This enables us to make lossless decision at node s: If we "foresee" that $\alpha_s < \beta_s$ in the next round, we can immediately start a new rollout from node s (rather than from the root node), as the rollout from the root will go through s anyway; Otherwise if $\alpha_s \geq \beta_s$ in the next round, we just leave node s and continue the back-up phase, in which case we know that the rollout will never come back to s later. Note that the nodes below s can also play this trick, "pretending" that it is running a single round of rollout in the view of s and other ancestors. Extending this to the whole tree, we essentially re-write the original rollout algorithm into a recursion algorithm. Further combined with some other optimizations, we finally arrive at the alphabeta2 procedure in Algorithm 5.

The complete proof consists of a series of equivalent transformations between algorithms. We start with transforming Algorithm 6 to the Algorithm 7 shown below, which simply replaces the rollout procedure in Algorithm 6 with the left-first policy of Eq.(5). Algorithm 7 runs in a loop until s becomes inactive. In each round, the algorithm iterates over $\mathcal{C}(s)$ to find the leftmost active node c, issues a rollout on c, then updates $[v^-_s, v^+_s]$ and $[\alpha_s, \beta_s]$ immediately. The sentence $[v^-_s, v^+_s] \leftarrow [\mathcal{V}^-(s), \mathcal{V}^+(s)]$ is a shortcut of Lines 14 and 15 of Algorithm 4.

Now consider Algorithm 8, as shown below. The algorithm works by iterating over $\mathcal{C}(s)$, from left to right. For each active node $c \in \mathcal{C}(s)$, the algorithm keeps running rollouts on c until c becomes inactive. It terminates when all successor nodes are inactive. The intervals $[v^-_s, v^+_s]$ and $[\alpha_s, \beta_s]$ are updated only when the algorithm switches to another node. Finally, the value range $[v^-_s, v^+_s]$ is updated again before terminating, in case s is leaf node.

Proposition 1. *Algorithm 8 is equivalent to Algorithm 7.*

Proof. In fact, it is sufficient to prove that in each round and for the same node c, the window $[\alpha_c, \beta_c]$ computed at Line 5 of Algorithm 8 is always identical to the window $[\alpha_c, \beta_c]$ computed at Line 3 of Algorithm 7. Note that the same $[\alpha_c, \beta_c]$ will lead to the same node c chosen for the rollout, and also lead to the same terminating condition between the two algorithms – by definition $[\alpha_s, \beta_s] \supseteq [\alpha_c, \beta_c]$, so the node s must be active if some successor node c is active; the reverse is also true, due to Lemma 3.

Since Algorithm 8 will update $[v^-_s, v^+_s]$ and $[\alpha_s, \beta_s]$ when it changes the successor node for rollout, we only need to prove that the $[\alpha_c, \beta_c]$ in Algorithm 8 is consistent to the one in Algorithm 7 at the second time when c has been chosen for rollout. For any such round t, by mathematical induction we can assume that the window $[\alpha_c, \beta_c]$ is consistent in all previous rounds, in particular, for the last round $t-1$. Notice

Algorithm 7: An implementation of Algorithm 6 when using the "left-first" policy.

```
1  Function wrapper1($s$, $\alpha_s$, $\beta_s$)
2      while $\alpha_s < \beta_s$ do
3          foreach $c \in \mathcal{C}(s)$ do
               $[\alpha_c, \beta_c] \leftarrow [\max\{\alpha_s, v_c^-\}, \min\{\beta_s, v_c^+\}]$ ;
               if $\alpha_c < \beta_c$ then
                   rollout($c$, $\alpha_c$, $\beta_c$) ;
                   break;
               end
           end
           $[v_s^-, v_s^+] \leftarrow [\mathcal{V}^-(s), \mathcal{V}^+(s)]$ ;
4          $[\alpha_s, \beta_s] \leftarrow [\max\{\alpha_s, v_s^-\}, \min\{\beta_s, v_s^+\}]$ ;
5      end
6      return $\{v_s^-, v_s^+\}$
```

Algorithm 8: Another implementation of Algorithm 7.

```
1  Function wrapper2($s$, $\alpha_s$, $\beta_s$)
2      foreach $c \in \mathcal{C}(s)$ do
3          $[\alpha_c, \beta_c] \leftarrow [\max\{\alpha_s, v_c^-\}, \min\{\beta_s, v_c^+\}]$ ;
4          if $\alpha_c < \beta_c$ then
5              while $\alpha_c < \beta_c$ do
                   rollout($c$, $\alpha_c$, $\beta_c$) ;
                   $[\alpha_c, \beta_c] \leftarrow [\max\{\alpha_c, v_c^-\}, \min\{\beta_c, v_c^+\}]$ ;
               end
6              $[v_s^-, v_s^+] \leftarrow [\mathcal{V}^-(s), \mathcal{V}^+(s)]$ ;
7              $[\alpha_s, \beta_s] \leftarrow [\max\{\alpha_s, v_s^-\}, \min\{\beta_s, v_s^+\}]$ ;
8          end
9      end
10     $[v_s^-, v_s^+] \leftarrow [\mathcal{V}^-(s), \mathcal{V}^+(s)]$ ;
11     return $\{v_s^-, v_s^+\}$
```

that in Algorithm 7 we have $[\alpha_c^{(t)}, \beta_c^{(t)}] =$

$$[\alpha_s^{(t-1)}, \beta_s^{(t-1)}] \cap [v_s^{-(t)}, v_s^{+(t)}] \cap [v_c^{-(t)}, v_c^{+(t)}], \quad (7)$$

and in Algorithm 8 we have $[\alpha_c^{(t)}, \beta_c^{(t)}] =$

$$[\alpha_s^{(t-1)}, \beta_s^{(t-1)}] \cap [v_c^{-(t-1)}, v_c^{+(t-1)}] \cap [v_c^{-(t)}, v_c^{+(t)}]. \quad (8)$$

By Lemma 2, $[v_c^{-(t)}, v_c^{+(t)}] \subseteq [v_c^{-(t-1)}, v_c^{+(t-1)}]$, so the $[v_c^{-(t-1)}, v_c^{+(t-1)}]$ in Eq.(8) is unnecessary, and so to prove the equivalence of Algorithms 7 and 8, we only need to prove that in Algorithm 7 we always have

$$\begin{array}{l} [\alpha_s^{(t-1)}, \beta_s^{(t-1)}] \cap [v_s^{-(t)}, v_s^{+(t)}] \cap [v_c^{-(t)}, v_c^{+(t)}] = \\ [\alpha_s^{(t-1)}, \beta_s^{(t-1)}] \cap \qquad\qquad\qquad [v_c^{-(t)}, v_c^{+(t)}] \end{array}. \quad (9)$$

That is, we only need to prove that $[v_s^{-(t)}, v_s^{+(t)}]$ is useless for updating $[\alpha_c^{(t)}, \beta_c^{(t)}]$ in Algorithm 7. This can be shown by observing that for both the max and min functions, when one of its arguments changes, the value of the function either remains unchanged or is equal to the value of the new argument. So, when $[v_c^{-(t-1)}, v_c^{+(t-1)}]$ changes to $[v_c^{-(t)}, v_c^{+(t)}]$, the bounds of $[v_s^{-(t)}, v_s^{+(t)}]$ either remains unchanged, or is equal to the bounds of $[v_c^{-(t)}, v_c^{+(t)}]$ accordingly. In the former case $[v_s^{-(t)}, v_s^{+(t)}]$ is "masked" by $[\alpha_s^{(t-1)}, \beta_s^{(t-1)}]$ in Eq.(9), while in the latter case $[v_s^{-(t)}, v_s^{+(t)}]$ is masked by $[v_c^{-(t)}, v_c^{+(t)}]$ in Eq.(9). □

Algorithm 9: Recursion-based version of Algorithm 8.

```
1  Function wrapper2($s$, $\alpha_s$, $\beta_s$)
2      foreach $c \in \mathcal{C}(s)$ do
3          $[\alpha_c, \beta_c] \leftarrow [\max\{\alpha_s, v_c^-\}, \min\{\beta_s, v_c^+\}]$ ;
4          if $\alpha_c < \beta_c$ then
5              wrapper2($c$, $\alpha_c$, $\beta_c$)
6              $[v_s^-, v_s^+] \leftarrow [\mathcal{V}^-(s), \mathcal{V}^+(s)]$ ;
               $[\alpha_s, \beta_s] \leftarrow [\max\{\alpha_s, v_s^-\}, \min\{\beta_s, v_s^+\}]$ ;
7          end
8      end
9      $[v_s^-, v_s^+] \leftarrow [\mathcal{V}^-(s), \mathcal{V}^+(s)]$ ;
10     return $\{v_s^-, v_s^+\}$
```

Algorithm 10: Another implementation of Algorithm 9.

```
1  Function wrapper3($s$, $\alpha_s$, $\beta_s$)
2      foreach $c \in \mathcal{C}(s)$ do
3          $[\alpha_c, \beta_c] \leftarrow [\max\{\alpha_s, v_c^-\}, \min\{\beta_s, v_c^+\}]$ ;
4          if $\alpha_c < \beta_c$ then
5              wrapper3($c$, $\alpha_c$, $\beta_c$) ;
6              if $s$ is a MAX node then
                   $[\alpha_s, \beta_s] \leftarrow [\max\{\alpha_s, v_c^-\}, \beta_s]$ ;
               else if $s$ is a MIN node then
                   $[\alpha_s, \beta_s] \leftarrow [\alpha_s, \min\{\beta_s, v_c^+\}]$ ;
               end
7          end
8      end
9      $v_s^- \leftarrow \begin{cases} R(s) & \text{if } s \text{ is Leaf} \\ \max_{c \in C(s)} v_c^- & \text{if } s \text{ is Internal \& MAX} \\ \min_{c \in C(s)} v_c^- & \text{if } s \text{ is Internal \& MIN} \end{cases}$ ;
10     $v_s^+ \leftarrow \begin{cases} R(s) & \text{if } s \text{ is Leaf} \\ \max_{c \in C(s)} v_c^+ & \text{if } s \text{ is Internal \& MAX} \\ \min_{c \in C(s)} v_c^+ & \text{if } s \text{ is Internal \& MIN} \end{cases}$ ;
11     return $\{v_s^-, v_s^+\}$
```

Now, observe that Line 5 of Algorithm 8 (the shadowed part) is actually identical to the wrapper1 procedure in Algorithm 6, which we have just proven to be equivalent to the wrapper2 procedure. As a result, we can replace the logic block with a subroutine call of wrapper2(c, α_c, β_c), as shown by Algorithm 9. Note that this replacement has turned Algorithm 9 into a recursion procedure. In the following we further transform the wrapper2 procedure in Algorithm 9 to the wrapper3 procedure in Algorithm 10.

Proposition 2. *Algorithm 9 is equivalent to Algorithm 10.*

Proof. Algorithm 10 is different from Algorithm 9 only in the method for updating $[\alpha_s^{(t)}, \beta_s^{(t)}]$ when the algorithm switches the node for rollout. Without loss of generality, we only discuss cases in which s is MAX node, and there is an active node c' that is behind c in $\mathcal{C}(s)$ and that is the node for rollout in the next round. We see that Algorithm 10 does not use $[v_s^{-(t)}, v_s^{+(t)}]$ to update $[\alpha_s^{(t)}, \beta_s^{(t)}]$ at all. Instead, it updates with $\alpha_s = \max\{\alpha_s, v_c^-\}$. This is equivalent to $\alpha_s = \max\{\alpha_s, v_s^-\}$ again because v_s^- can either be v_c^- or be itself, in the latter case it is masked by α. To see why β_s does not need to update at all, observe that our goal is to correctly compute $[\alpha_{c'}, \beta_{c'}]$ in the next round, and at that time v_c^+ is either masked by β_s or by $v_{c'}^+$, depending on whether $v_c^+ > v_{c'}^+$ or $v_c^+ \leq v_{c'}^+$. □

So far we have made a series of equivalent transformations from Algorithm 6 to Algorithm 10. As the final step, just by following the code it is straightforward to verify the equivalence between the wrapper3 procedure in Algorithm 10 and the alphabeta2 procedure in Algorithm 5 *if* $[\alpha, \beta] \subseteq [v_s^-, v_s^+]$. Interestingly, by comparing Algorithm 10 and Algorithm 5 we can find that the window (α_s, β_s) used in the rollout procedure is conceptually *different* from the classic (α, β) window used in the alphabeta procedure. Specifically, we have

$$[\alpha_s, \beta_s]_{rollout} = [\alpha, \beta]_{minimax} \cap [v_s^-, v_s^+]. \qquad (10)$$

□

Proof of Theorem 3

In this section we prove Theorem 3, which states that the "max-β_c" policy in the family of Algorithm 4 is equivalent to the MT-SSS* algorithm based on the alphabeta2 procedure defined in Algorithm 5.

Proof. Since we already prove the strict equivalence between the wrapper1 procedure in Algorithm 6 and the alphabeta2 procedure in Algorithm 5, it is easy to first write MT-SSS* into rollout algorithm, as Algorithm 11 shows, which repeatedly calling $wrapper1(root, v_{root}^+ - 1, v_{root}^+)$ until the range $[v_{root}^-, v_{root}^+]$ is closed. Note that the null window has guaranteed the condition that $[\alpha, \beta] \subseteq [v_{root}^-, v_{root}^+]$.

Algorithm 11: A rollout version of MT-SSS*, which is based on the wrapper1 procedure in Algorithm6.

```
1 while $v_{root}^- < v_{root}^+$ do
2 |   wrapper1($root$, $v_{root}^+ - 1$, $v_{root}^+$) ;
3 end
```

Now we only need to prove that, starting from the empty storage where $[v_s^-, v_s^+] = [-\infty, +\infty]$ for all $s \in \mathcal{S}$, and for every round of rollout, Algorithm 11 chooses exactly the same rollout trajectory with Algorithm 4 if they are using the policy of Eq.(5) and Eq.(6), respectively. Note that both of them call the same rollout procedure to update the storage, which must act the same given the same rollout trajectory.

On the side of Algorithm 11, because it is using a minimal window $[v_{root}^+ - 1, v_{root}^+]$, no active node selected in the rollout can further reduce the upper-influence-bound β_c, thus the algorithm is always selecting the leftmost node with $v_c^+ \geq v_{root}^+$. Moreover, recall that Lemma 3 guarantees that such an active node can always be found along the rollout (as long as the value range of root is open).

On the other side, consider how Algorithm 4 selects successor nodes in the rollouts: The root node is a MAX node, so $v_{root}^+ = \max_{s \in \mathcal{C}(root)} v_s^+$, which means there exists an active successor node s of the root with $v_s^+ = v_{root}^+$. The leftmost of them will be selected by the algorithm, with the upper-influence-bound $\beta_s = v_{root}^+$. At the node s thus selected, because s is a MIN node, every successor node c of s will have $v_c^+ \geq v_s^+$. Thus, the algorithm will just select the leftmost one in $\mathcal{C}(s)$, still with the upper-influence-bound $\beta_c = v_{root}^+$. In that way, it is easy to see that Algorithm 4, if under the policy of Eq.(6), will also always select the leftmost node with $v_c^+ \geq v_{root}^+$, thus having exactly the same rollout trajectory. □

Related Work

The gaming tree models encompass computational problems that are inherently hard. It is known that the problem of evaluating the game trees of Chess and GO are both EXPTIME-Complete (Fraenkel and Lichtenstein 1981) (Robson 1983), implying a provable intractability to solve them in polynomial time, due to the Time Hierarchy Theorem (Demaine 2001). As important special cases, if each MIN internal node has only one single child node, the game tree degenerates to a backtracking tree of combinatorial optimization. In this case the problem could still be polynomial-time intractable, due to the widely believed P≠NP hypothesis (Fortnow 2013).

Historically, practical game tree evaluation algorithms were mostly in the depth-first search style at the early stage, partially because at that time computers had very limited physical memory. For example, the original implementation of SSS* in (Stockman 1979) needs to explicitly maintain a list of "open nodes", which is a huge burden on computation resources, enough to make it effectively impractical at the time when it was proposed (Pearl 1984).

However, the memory size of computers has grown exponentially for decades since then, and modern computers are now often equipped with enough memory to match the storage demand of online planning (typically in the order of millions or billions). Indeed, it has become a standard practice to even enhance depth-first algorithms with external storage in their modern implementations (Plaat et al. 1996). In particular, *transposition table* is one of the most popular data structures in such storages. The original purpose of the transposition table is to "transpose" searching to avoid repeatedly visiting a state when the state corresponds to multiple nodes in the game tree (in which case the topology of the state space is not strictly a tree, but actually a DAG). Meanwhile, the transposition table can also be used in the *iterative deepening* paradigm to store search results for improving move ordering in later iterations (Reinefeld and Marsland 1994).

On the other hand, rollout algorithms were originally used as a straightforward sampling method to estimate the expected outcome of stochastic games (Tesauro and Galperin 1996), where the policy at player nodes consists of heuristic rules and the policy at chance nodes just follows the probability distribution as the rule of the game prescribes. But it turns out that rollout algorithms can also be useful for even *deterministic* games. In 1990, Abramson reported that for several popular deterministic games there is an interesting correlation between the minimax value and the average outcome of *random* rollouts that simply choose successor states according to uniform distribution (Abramson 1990). This observation implies that repeated simulations based on the random rollout policy may approximately evaluate the minimax value of deterministic game trees. Later on, researchers further developed *adaptive* rollout policies that may dynamically change their rollout preference given outcomes from previous rollout simulations (Bouzy 2006), coined as Monte Carlo Tree Search in (Coulom 2006). Similar rollout ideas were also proposed in other domains, such as metaheuristics for combinatorial optimization (Bertsekas, Tsitsiklis, and Wu 1997) (Glover and Taillard 1993).

A popular way to design rollout policy is by recasting successor node selection into a *Multi-Armed Bandit* (MAB) problem – every time when the rollout simulation visits a MAX node s, we select one successor node from the children list $\mathcal{C}(s)$ and obtain a reward of this choice (from the rollout outcome of this round), and the goal is to maximize the average reward of repeated rollouts in the long term. A symmetric formulation applies to MIN nodes. By induction it is not hard to see that the average reward of each node s under such a bandit policy will "eventually" converge to $\mathcal{V}(s)$ as long as the average reward of every successor node $c \in \mathcal{C}(s)$ converges to $\mathcal{V}(c)$, which is true for the leaf nodes.

In particular, the rollout policy of UCT was directly borrowed from the *UCB* algorithm, a renowned algorithm for the stochastic MAB problem. In the specific model of *stochastic* MAB, the UCB index of any bandit arm, which is identical to the UCT score, is guaranteed to be an upper bound of the expected reward of the arm, with the confidence level of $1 - 1/n_s^{\lambda^2}$. Note that the confidence levels for different bandit arms are the same. Then by simply choosing the bandit arm with the best confidence-upper-bound, the UCB algorithm manages to achieve sublinear *regrets* for any problem instance of stochastic MAB, and was proven to achieve the asymptotically optimal performance of the problem (Auer, Cesa-Bianchi, and Fischer 2002). [3] In general, such a *best-upper-bound-first* strategy has been widely used in combinatorial optimization and single-player games, such as in the classic A^* algorithm. In many domains the upper bound of each node is typically computed by solving a relaxed (and easier) problem, which is thus inherently domain-specific. In the game tree model, however, these upper bounds can be bootstrapped directly from the tree search itself, without any domain-specific heuristics.

[3]However, we note that the underlying theoretical assumptions of game tree evaluation are different from the ones of stochastic MAB, which means that the UCT algorithm needs a different mathematical justification about its performance from the UCB's.

In terms of the techniques used in this paper, the idea of maintaining pairs of value bounds in an in-memory tree (and updating the bounds in a bottom-up manner) was proposed by Hans Berliner in the B* algorithm (Berliner 1979). More recently, Walsh, Goschin, and Littman (2010) proposed the FSSS algorithm, a rollout algorithm that updates the value bounds $[v_s^-, v_s^+]$ in the same way as Algorithm 4, in order to have a theoretical guarantee of its performance when used in reinforcement-learning applications. An algorithm with similar idea was also proposed in the context of game-tree evaluation (Cazenave and Saffidine 2011).

Weinstein, Littman, and Goschin (2012) further adapted the FSSS algorithm into the game tree model and proposed a rollout algorithm that outprunes the α-β algorithm. Their algorithm also uses an (α, β) window to filter successor nodes, but the window is manipulated in a different way from the algorithm family proposed in this paper. Furthermore, there is no domination between MT-SSS* and their algorithm, while in this paper we argue that α-β and MT-SSS* themselves can be unified under the rollout framework of Algorithm 4.

Chen et al. (2014) have recently presented a rollout algorithm that captures the idea of the MT-SSS* algorithm. Their algorithm uses the null window $(v_{root}^+ - 1, v_{root}^+)$ to filter nodes, and thus does not manipulate the window at all during rollouts. Besides, they did not formally characterize the relationship between MT-SSS* and their null-window rollout algorithm. Interestingly, the analysis of this paper suggests that their algorithm is not exactly equivalent to the original MT-SSS* algorithm.

Conclusion

Results from this paper suggest that the rollout paradigm could serve as a unified framework to study game-tree evaluation algorithms. In particular, Theorems 2 and 3 show that some classic minimax search algorithms could be implemented by rollouts. This observation implies that we could collect information in a single rollout for both minimax pruning and MCTS sampling. In light of this, we may design new hybrid algorithms that naturally combine MCTS algorithms with traditional game-tree search algorithms.

For example, by Theorem 3 we see that MT-SSS* corresponds to a rollout algorithm that prefers successor node with the largest "absolute" upper-influence-bound β_c, which is powerful in completely pruning nodes without compromising correctness. But the values of β_c propagate upwards slowly in large game trees. As a result, most nodes in the upper layers of the tree may be left with the non-informative bound $+\infty$ at the early running stage of the algorithm, in which case the MT-SSS* policy is essentially blind. On the other hand, the UCT score can be seen as an upper bound "with some confidence", which is able to provide informative guidances with much less rollouts, but could respond slowly to the discriminative knowledge collected in the search, probably due to its "amortizing" nature (Ramanujan, Sabharwal, and Selman 2012).

In light of this complementary role between β_c and the UCT score, Algorithm 12 demonstrates a natural way to combine the ideas of UCT and MT-SSS*. The algorithm selects successor nodes based on their upper bound information, with a prioritized rule to consider the bounds with different confidences. It first checks the upper bound with the absolute confidence (i.e., β_c). When there is a tie, the algorithm turns to consider bounds with less confidence, and selects the node with the largest UCT score. Given a game tree, we expect that such an algorithm will first follow the UCT score at the beginning, as essentially all tree nodes have the same upper bound $\beta_c = +\infty$ at that time. The hope is that the UCT score may be more likely to guide the rollouts to good "successor nodes" than the blind tie-breaking rule of the original MT-SSS* algorithm does, and thus better "ordering" the search/rollout trajectories. If these preferred nodes turn out to be sub-optimal, their absolute upper bounds β_c will later drop below α_c, turning the rollout to other candidates immediately. In the end, the algorithm will completely shift to the MT-SSS* mode. It may be an interesting future work to test the performance of this hybrid algorithm in real-world benchmarks.

Meanwhile, it would also be interesting to see how the observations in this paper generalize to other models. For example, it seems that Algorithm 4 could be adapted to an "incremental" rollout algorithm when incorporating admissible heuristic function at internal nodes (essentially an *iterative-deepening* setting). Moreover, one could also consider the problem in more general state-space topologies (such as DAG) and/or value functions including operators other than $\max$ and $\min$ (such as the *Weighted-Sum* operator formulated in stochastic games).

Algorithm 12: A rollout procedure that combines UCT and MT-SSS*.

```
1  $n_s, \mu_s \leftarrow 0$, $[v_s^-, v_s^+] \leftarrow [-\infty, +\infty]$ for every $s \in \mathcal{S}$;
2  Function rollout2($s$, $\alpha_s$, $\beta_s$)
3      if $\mathcal{C}(s) \neq \emptyset$ then
4          foreach $c \in \mathcal{C}(s)$ do
5              $[\alpha_c, \beta_c] \leftarrow [\alpha, \beta] \cap [v_c^-, v_c^+]$ ;
6          end
7          $\mathcal{A}_s = \{c \in \mathcal{C}(s) \mid \alpha_c < \beta_c\}$ ;
8          $\mathcal{B}_s = \arg\max_{c \in \mathcal{A}_s} \beta_c$;
           $c^* \leftarrow \arg\max_{c \in \mathcal{B}_s} \mu_c + \lambda\sqrt{\ln n_s / n_c}$ ;
           $g \leftarrow$ rollout2($c^*, \alpha_{c^*}, \beta_{c^*}$) ;
9      else
10         $g \leftarrow R(s)$
11     end
12     $v_s^- \leftarrow \begin{cases} R(s) & \text{if } s \text{ is Leaf} \\ \max_{c \in \mathcal{C}(s)} v_c^- & \text{if } s \text{ is Internal \& MAX} \\ \min_{c \in \mathcal{C}(s)} v_c^- & \text{if } s \text{ is Internal \& MIN ;} \end{cases}$
13     $v_s^+ \leftarrow \begin{cases} R(s) & \text{if } s \text{ is Leaf} \\ \max_{c \in \mathcal{C}(s)} v_c^+ & \text{if } s \text{ is Internal \& MAX} \\ \min_{c \in \mathcal{C}(s)} v_c^+ & \text{if } s \text{ is Internal \& MIN ;} \end{cases}$
14     $\mu_s \leftarrow \frac{n_s}{n_s+1}\mu_s + \frac{1}{n_s+1}g$ ;
15     $n_s \leftarrow n_s + 1$ ;
16     return $g$
```

Acknowledgements

We thank Thomas Moscibroda (Microsoft Research) for his inspiring comments during this work. We thank Aske Plaat (Leiden University) for his valuable feedbacks on the paper. We also thank the anonymous reviewers of AAAI-15 for their constructive suggestions on the paper.

References

Abramson, B. 1990. Expected-outcome: A general model of static evaluation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 12(2):182–193.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2-3):235–256.

Baier, H., and Winands, M. H. 2013. Monte-carlo tree search and minimax hybrids. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 1–8. IEEE.

Berliner, H. 1979. The b tree search algorithm: A best-first proof procedure. *Artificial Intelligence* 12(1):23–40.

Bertsekas, D. P., and Castanon, D. A. 1999. Rollout algorithms for stochastic scheduling problems. *Journal of Heuristics* 5(1):89–108.

Bertsekas, D. P.; Tsitsiklis, J. N.; and Wu, C. 1997. Rollout algorithms for combinatorial optimization. *Journal of Heuristics* 3(3):245–262.

Bouzy, B. 2006. Associating shallow and selective global tree search with monte carlo for 9× 9 go. In *Computers and Games*. Springer. 67–80.

Campbell, M. S., and Marsland, T. A. 1983. A comparison of minimax tree search algorithms. *Artificial Intelligence* 20(4):347–367.

Cazenave, T., and Saffidine, A. 2011. Score bounded monte-carlo tree search. In *Computers and Games*. Springer. 93–104.

Chen, J.; Wu, I.; Tseng, W.; Lin, B.; and Chang, C. 2014. Job-level alpha-beta search. *IEEE Transactions on Computational Intelligence and AI in Games*.

Coulom, R. 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*. Springer. 72–83.

Demaine, E. D. 2001. Playing games with algorithms: Algorithmic combinatorial game theory. In *Mathematical Foundations of Computer Science 2001*. Springer. 18–33.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI*, volume 8, 259–264.

Fortnow, L. 2013. *The golden ticket: P, NP, and the search for the impossible*. Princeton University Press Princeton.

Fraenkel, A. S., and Lichtenstein, D. 1981. Computing a perfect strategy for n* n chess requires time exponential in n. In *Proceedings of the 8th Colloquium* *on Automata, Languages and Programming*, 278–293. Springer-Verlag.

Gelly, S.; Kocsis, L.; Schoenauer, M.; Sebag, M.; Silver, D.; Szepesvári, C.; and Teytaud, O. 2012. The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM* 55(3):106–113.

Glover, F., and Taillard, E. 1993. A user's guide to tabu search. *Annals of operations research* 41(1):1–28.

Knuth, D. E., and Moore, R. W. 1975. An analysis of alpha-beta pruning. *Artificial intelligence* 6(4):293–326.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*. Springer. 282–293.

Lanctot, M.; Saffidine, A.; Veness, J.; Archibald, C.; and Winands, M. H. 2013. Monte carlo*-minimax search. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 580–586. AAAI Press.

Marsland, T. A. 1986. A review of game-tree pruning. *ICCA journal* 9(1):3–19.

Pearl, J. 1984. *Heuristics*. Addison-Wesley Publishing Company Reading, Massachusetts.

Péret, L., and Garcia, F. 2004. On-line search for solving markov decision processes via heuristic sampling. *learning* 16:2.

Plaat, A.; Schaeffer, J.; Pijls, W.; and de Bruin, A. 1996. Best-first fixed-depth minimax algorithms. *Artificial Intelligence* 87(1):255–293.

Ramanujan, R.; Sabharwal, A.; and Selman, B. 2012. Understanding sampling style adversarial search methods. *arXiv preprint arXiv:1203.4011*.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 16(7):701–710.

Robson, J. M. 1983. The complexity of go. In *IFIP Congress*, 413–417.

Stockman, G. C. 1979. A minimax algorithm better than alpha-beta? *Artificial Intelligence* 12(2):179–196.

Tesauro, G., and Galperin, G. R. 1996. On-line policy improvement using monte-carlo search. In *NIPS*, volume 96, 1068–1074.

Walsh, T. J.; Goschin, S.; and Littman, M. L. 2010. Integrating sample-based planning and model-based reinforcement learning. In *AAAI*.

Weinstein, A.; Littman, M. L.; and Goschin, S. 2012. Rollout-based game-tree search outprunes traditional alpha-beta. In *EWRL*, 155–167. Citeseer.

Zobrist, A. L. 1970. A new hashing method with application for game playing. *ICCA journal* 13(2):69–73.