# Generalisation of alpha-beta search for AND-OR graphs with partially ordered values[*]

**Junkang Li**[1,2] **Bruno Zanuttini**[2]
**Tristan Cazenave**[1,3] **Véronique Ventos**[1]
[1]NukkAI, Paris, France
[2]Normandie Univ.; UNICAEN, ENSICAEN, CNRS, GREYC, 14 000 Caen, France
[3]LAMSADE, Université Paris-Dauphine, PSL, CNRS, France
junkang.li@nukk.ai bruno.zanuttini@unicaen.fr
tristan.cazenave@lamsade.dauphine.fr vventos@nukk.ai

**Résumé**

Nous proposons un cadre pour l'évaluation de graphes ET-OU (orientés, acycliques) portant des valeurs partiellement ordonnées. De tels graphes apparaissent naturellement dans la résolution de jeux à information incomplète (par exemple, la plupart des jeux de cartes, comme le bridge) ou multicritères. En particulier, notre cadre généralise l'évaluation standard de graphes ET-OU et le calcul de stratégies optimales pour les jeux à information complète.

Dans ce cadre, nous proposons un nouvel algorithme, qui utilise l'élagage alpha-beta et un cache des valeurs déjà calculées. Cet article présente l'algorithme, démontre sa correction, et fournit des résultats expérimentaux sur des jeux aléatoires et sur un jeu de cartes à information incomplète.

**Abstract**

We define a new setting related to the evaluation of AND-OR directed acyclic graphs with partially ordered values. Such graphs arise naturally when solving games with incomplete information (e.g. most card games such as Bridge) or games with multiple criteria. In particular, this setting generalises standard AND-OR graph evaluation and computation of optimal strategies in games with complete information.

Under this setting, we propose a new algorithm which uses both alpha-beta pruning and cached values. In this paper, we present our algorithm, prove its correctness, and give experimental results on random games and on a card game with incomplete information.

[*]This article is a long version with full proofs of the article published in the proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI 2022).

## 1 Introduction

Search in graphs containing AND- and OR-nodes is used as a basis of many algorithmic solutions to Artificial Intelligence problems. In such graphs, OR-nodes typically model choice nodes where an agent can choose a successor, while AND-nodes model an opponent. For instance, in robust planning with nondeterministic actions, an AND-node models the outcome of an action: a strategy must be valid whatever the outcome [11]. Similarly, when solving a zero-sum two-player (sequential) game for a player, OR-nodes are those at which it is her turn to play, while AND-nodes correspond to her opponent [22]: the value of an OR-node is 1 if and only if at least one move leads to a node with value 1; dually, at AND-nodes, the values of the children are conjoined. More generally, in games that can have more than two outcomes, such as chess or checkers [20], AND-nodes (respectively OR-nodes) correspond to a minimum (respectively maximum) operator on the values of their children.

A fundamental question is that of evaluating rooted AND-OR directed acyclic graphs (DAGs), which means computing the value of their root given a value for each of their leaves. For instance, in games with complete information, selecting the best move for the current turn amounts to evaluate each of the children of the root. This problem has been thoroughly studied in the literature [16] under the setting of totally ordered values (Boolean or real) and the standard AND/OR or min/max operators. Following [8], we investigate here a more general setting, where the values are taken from a distributive lattice $(V, \wedge, \vee)$ (i.e. a partially ordered set with least upper bound and greatest lower bound for any two elements), and operators for AND- and OR nodes are taken to be the meet $\wedge$ and join $\vee$, respectively. This setting arises naturally in many applications, in particular in games with incomplete information. Example of such games are Skat [13, 19, 5], Bridge [14, 9, 2], Hearts and Spades [21].

A well-known technique for evaluating AND-OR graphs is alpha-beta pruning, which maintains a lower bound $\alpha$ (respectively upper bound $\beta$) on the value of each OR-nodes (respectively AND-nodes), and uses them to prune some of their successors. This technique is currently used in strong chess programs [10] combined with sophisticated evaluation functions such as NNUE neural networks that were first used in Shogi [17]. However, the generalisation of alpha-beta pruning to AND-OR DAGs with partially ordered values is nontrivial since two values from a lattice are not always comparable. We build on the seminal work by [8] and generalise it by proving the correctness of lattice-valued alpha-beta pruning with the consideration for heuristic functions.

Orthogonally, we investigate caching techniques for alpha-beta pruning in lattice-valued DAGs. The question is again nontrivial because nodes are in general revisited with different $\alpha$ and $\beta$ than during previous visits. For this, we propose a new algorithm called 'alpha-beta duo'. We state its correctness and experimentally evaluate its efficiency.

The paper is organised as follows. Preliminaries are given in Sections 2 and 3. We extend the work by [8] in Section 4, and present alpha-beta duo in Section 5. We then report experimental results and conclude.

## 2 Preliminaries

The following definitions on posets and lattices are based on [4].

**Definition 1.** *Let $V$ be a set and $\leq$ be a binary relation on $V$. Then $(V, \leq)$ is called a* partially ordered set *(poset) if $\leq$ is a partial order (i.e. reflexive, transitive, and antisymmetric).*

For a poset $(V, \leq)$ and $S \subseteq V$, an element $x \in V$ is called an *upper bound* (UB) of $S$ if $s \leq x$ holds for all $s \in S$, and $x$ is called a *least upper bound* (LUB) if in addition $x \leq y$ holds for any UB $y$ of $S$. If $S$ has an LUB, then it is unique. The greatest lower bound (GLB) of $S$ is defined dually. For $x, y \in V$, we write $x \vee y$ ('$x$ *join* $y$') and $x \wedge y$ ('$x$ *meet* $y$') respectively for the LUB and GLB of $\{x, y\}$, when they exist.

**Definition 2.** *A poset $(V, \leq)$ is called a* distributive lattice *if*

- *for all $x, y \in V$, $x \vee y$ and $x \wedge y$ exist,*
- *for all $x, y, z \in V$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$,*
- *for all $x, y, z \in V$, $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.*

*It is moreover said to be* bounded *if there are elements $\bot, \top \in V$ satisfying $\bot \leq x$ and $x \leq \top$ for any $x \in V$.*

In the remainder of this paper, we denote by $(V, \leq, \wedge, \vee)$ an arbitrary bounded distributive lattice.

**Example 1.** *Let $S$ be a set and let $2^S$ denote its powerset. Then $(2^S, \subseteq, \cap, \cup)$ is a bounded distributive lattice with set inclusion $\subseteq$ as partial order, set intersection $\cap$ and set union $\cup$ respectively as meet and join, $\emptyset$ as $\bot$, and $S$ as $\top$.*

We denote any directed acyclic graph (DAG) by $G = (N, C)$, where $N$ is the set of nodes, and $C : N \to 2^N$ is a function that yields the set of *children* of each node. A *root* $r$ is a node without predecessor (i.e. for any $n \in N$, $r \notin C(n)$), and a *leaf* is a node without child. We denote the set of leaves of a DAG $G$ by $L_G$. We only consider *rooted* DAGs, which contain a (necessarily unique) root $r$ such that there exists a directed path to every vertex from $r$.

An *AND-OR DAG* is a rooted DAG $(N, C, r)$ equipped with a labelling function $\ell : N \to \{\mathrm{A}, \mathrm{O}\}$. Nodes labelled by A and O are respectively called *AND-nodes* and *OR-nodes*. Note that we do not impose nodes to be alternating.

## 3 Problem setting

We are interested in the problem of evaluating the root value of an AND-OR DAG, given values for all its leaves. Formally, given an AND-OR DAG $G = (N, C, r, \ell)$, a bounded distributive lattice $(V, \leq, \wedge, \vee)$, and an evaluation function $e : L_G \to V$ assigning a value in $V$ to each leaf of $G$, the goal is to compute $v(r)$, where the value $v(n)$ of $n \in N$ is defined recursively by:

- for a leaf node $n$, $v(n) := e(n)$;
- for an internal AND-node $n$, $v(n) := \bigwedge_{c \in C(n)} v(c)$;
- for an internal OR-node $n$, $v(n) := \bigvee_{c \in C(n)} v(c)$.

Since $G$ is a DAG, the function $v : N \to V$ is well-defined.

**Example 2.** *Consider the DAGs in Figure 1, where circle and square nodes represent AND-nodes and OR-nodes, respectively. On the left, the lattice is the set of Boolean vectors of length* 4 *(denoted as words), with bitwise AND and bitwise OR as meet and join, respectively. One can easily verify that $v(r) = 1100$. On the right, the lattice is the set of Boolean vectors of length* 3*, and $v(R) = 001$.*

### Example applications

Many important problems are in fact AND-OR DAG evaluation in disguise. The simplest one is Boolean circuit evaluation. Here AND- and OR-nodes model AND and OR gates, the lattice is the Boolean algebra (i.e. $V = \{0, 1\}$ with $0 \leq 1$ and logical conjunction and disjunction as meet

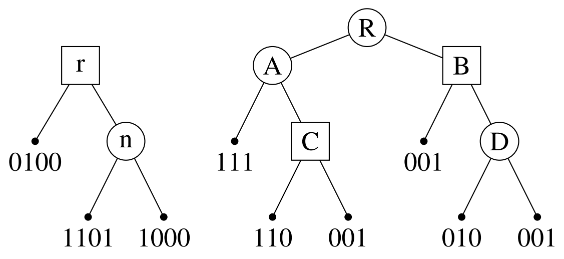

Figure 1: Two AND-OR DAGs with partially ordered values.

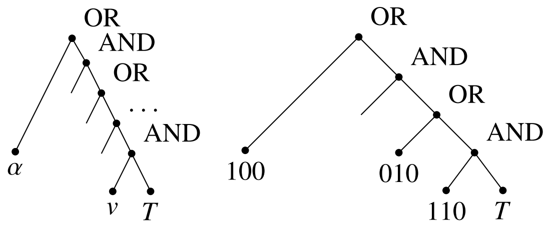

Figure 2: Deep pruning *vs* expected pruning.

and join), and the evaluation function encodes the inputs of the circuit.

Solving a game with complete information typically involves computing the minimax value of a game tree, which can be regarded as evaluating an AND-OR DAG: AND- and OR-nodes are respectively choice nodes of player MIN and of player MAX, the lattice is $(V, \leq, \min, \max)$ with $V$ a totally ordered set such as $\mathbb{Z}$ or $\mathbb{R}$, and the evaluation function gives the value of terminal nodes, or a heuristic value if the search is cut at some depth. Then the root value is the minimax value of the game.

In games with incomplete information, nontrivial lattices come into play. For example, [9] shows that computing the maxmin value of a player amounts to evaluate the game DAG with the lattice $(2^{2^S}, \leq, \sqcap, \cup)$[1], where $S$ is a finite set and $f \sqcap g = \{\alpha \cap \beta \mid \alpha \in f, \beta \in g\}$ for any $f, g \in 2^{2^S}$ (i.e. $f$ and $g$ are sets of subsets of $S$). We will discuss more about this in Section 6.

## 4 Alpha-beta pruning under partial order

Most of the literature on alpha-beta pruning concerns only totally ordered values, such as real numbers. Since AND-OR DAGs with partially ordered values are useful to model richer problems, [3] proposed alpha-beta pruning in this new setting for multi-criteria game. [8] gave the first thorough study on this subject, and proved in particular that deep pruning is sound for rational players if and only if the set of values is a distributive lattice. [15] showed that deep pruning is sound for tropical algebras if rationality is relaxed.

The form of deep $\alpha$ pruning considered by [8] is given in Figure 2 (left). If $v \leq \alpha$, then the subtree $T$ can be deeply pruned. To show why this definition of deep pruning does not capture every cut an alpha-beta search should perform when values are partially ordered, consider Figure 2 (right). The lattice is again the set of Boolean vectors of length 3, with bitwise AND and bitwise OR as meet and join, respectively. When an alpha-beta search algorithm descends to the bottommost AND-node, the value of $\alpha$ would be 110, which is the join of the value of an already explored child of two ancestor OR-nodes. We would like the algorithm to prune subtree $T$ since the value of its parent node cannot be better than the current value of $\alpha$ (due to the sibling of $T$). However, deep pruning, as it is defined in the literature such as [8], does not apply since the value of $\alpha$ does not come from a child of one single ancestor node. Note that this phenomenon is specific to lattices that are *not* totally ordered, since otherwise the meet or join (i.e. min or max) of two values is always one of them, hence deep pruning captures any pruning in a standard alpha-beta search.

Another question not formally addressed in the literature is the initialisation of node values. In standard alpha-beta search, one typically initialises the value of an OR-node (respectively AND-node) to be $\alpha$ (respectively $\beta$) [12] or $-\infty$ (respectively $+\infty$) [16] (note that $-\infty$ and $+\infty$ translate to $\perp$ and $\top$ in our context). However, one may have access to a heuristic evaluation of nodes, typically by evaluating a relaxed version of the problem which is easier to solve. For instance, a player can do no better in a game with incomplete information than in the same game but with complete information. The latter being much easier to solve, the value obtained can be used as a heuristic in the original game with incomplete information. Ideally, initialising values with an accurate heuristic should accelerate the search by finding cuts earlier.

In order to fill these two gaps in the literature, we first formalise alpha-beta search under partial order with initialisation function in Algorithm 1. We denote by $h$ the initialisation function. In general, its value depends on the current $\alpha$ and $\beta$, so we define it to yield a value $h(n, \alpha, \beta)$ for any node $n$ and bounds $\alpha$ and $\beta$. Note that since non-trivial initial value can be used for a node $n$ (Line 3), a cut may happen even before the first child of $n$ is explored, hence we update $\alpha$ and $\beta$ (Lines 7 and 9) and determine whether there is a cut (Line 10) at the beginning of the main loop. This is otherwise the same algorithm as in the literature [16, for instance].

It can be seen that Algorithm 1 will perform the wished pruning in the example in Figure 2 (right). By mimicking the proof by [12], we prove that such pruning is indeed sound, thereby extending the result by [8] to its full form,

[1]We abuse the notation to denote by $2^{2^S}$ the set of subsets of $S$ closed under subsets, i.e. the set of down-sets of $2^S$.

**Algorithm 1:** Alpha-beta search

```
def AlphaBeta(node $n$, $\alpha$, $\beta$):
    $I = h(n,\alpha,\beta)$
    $v \leftarrow I$
    determine the successor nodes $n_1,\ldots,n_b$ of $n$
    for $i$ in $\{1,\ldots,b\}$:
        if $n$ is an OR-node:
            $\alpha \leftarrow \alpha \vee v$
        else:
            $\beta \leftarrow \beta \wedge v$
        if $\alpha \geq \beta$:
            break
        $v_{\text{child}} \leftarrow$ AlphaBeta($n_i$, $\alpha$, $\beta$)
        if $n$ is an OR-node:
            $v \leftarrow v \vee v_{\text{child}}$
        else:
            $v \leftarrow v \wedge v_{\text{child}}$
    return $v$
```

provided that the initialisation function $h$ satisfies a certain admissibility condition:

**Definition 3.** *A heuristic function h is said to be* admissible *for G and V if for any node n in G and any $\alpha,\beta \in V$,*

$$h(n,\alpha,\beta)\begin{cases} = v(n) & \text{if } n \text{ is a leaf node;}\\ \geq v(n)\wedge\beta & \text{if } n \text{ is an AND-node;}\\ \leq v(n)\vee\alpha & \text{if } n \text{ is an OR-node.}\end{cases}$$

Note that this condition is satisfied for the initialisation values usually used in the literature, such as $\alpha$ or $-\infty$ for OR-nodes (and $\beta$ or $+\infty$ for AND-nodes). It is also satisfied when $h(n,\alpha,\beta)$ overestimates $v(n)$ for internal AND-nodes and underestimates it for internal OR-nodes.

We denote the value returned by Algorithm 1 with input $n$, $\alpha$, $\beta$ by $f(n,\alpha,\beta)$. The correctness of Algorithm 1 is a consequence of the following central result.

**Proposition 1.** *If h is an admissible heuristic function for G and V, then for any node n of G and any $\alpha,\beta \in V$, we have*

$$\beta \wedge \big(\alpha \vee f(n,\alpha,\beta)\big) = \beta \wedge \big(\alpha \vee v(n)\big). \qquad (1)$$

*Proof.* The proof is based on structural induction. We will only focus on OR nodes, since the case for AND nodes is completely symmetric. So let $n$ be an OR node and let us consider the execution of the function call AlphaBeta($n,\alpha,\beta$).

If $n$ is a leaf, then $f(n,\alpha,\beta) = h(n,\alpha,\beta) = v(n)$ since $h$ is admissible, hence equality (1) holds trivially.

Now consider an internal OR node $n$. Let $b \geq 1$ be the number of children of $n$ and let $n_1,\ldots,n_b$ be the children of $n$, listed in the same order as in Algorithm 1. By definition of the value function,

$$v(n) = \bigvee_{j=1}^{b} v(n_j).$$

We assume by induction that all function calls on the children of $n$ satisfy equality (1).

Let $k$ be the index of loop during which a break happens (i.e. a cut is found). If no break happens, then $k$ is taken to be $b+1$. For $0 \leq i < k$, let $v_i$ and $\alpha_i$ denote the value of $v$ and $\alpha$ after the $i$th loop[2]. Then we have

$$v_i = I \vee \bigvee_{j=1}^{i} f(n_j,\alpha_j,\beta).$$

In particular, $v_0 = I$. In addition, $\alpha_0 = \alpha$, and $\alpha_i = \alpha \vee v_{i-1}$ for $1 < i < k$.

To prove equality (1) for node $n$, we need the following lemma:

**Lemma 1.** *For any $0 \leq i < k$, we have*

$$\beta \wedge (\alpha \vee v_i) = \beta \wedge \left(\alpha \vee I \vee \bigvee_{j=1}^{i} v(n_j)\right). \qquad (2)$$

*Proof.* We prove equality (2) by an induction on $i$. When $i = 0$, both sides of equality (2) equals $\beta \wedge (\alpha \vee I)$, hence the equality holds.

For $i \geq 1$, $f(n_i,\alpha_i,\beta)$ satisfies equality (1) by induction from the main proposition, which means

$$\beta \wedge \big(\alpha_i \vee f(n_i,\alpha_i,\beta)\big) = \beta \wedge \big(\alpha_i \vee v(n_i)\big). \qquad (3)$$

Notice that $\alpha_i = \alpha \vee v_{i-1}$, we have

$$\begin{aligned}\beta \wedge (\alpha \vee v_i) &= \beta \wedge \big(\alpha \vee v_{i-1} \vee f(n_i,\alpha_i,\beta)\big)\\ &= \beta \wedge \big(\alpha \vee v_{i-1} \vee v(n_i)\big)\\ &= \beta \wedge \left(\alpha \vee I \vee \bigvee_{j=1}^{i-1} v(n_j) \vee v(n_i)\right)\\ &= \beta \wedge \left(\alpha \vee I \vee \bigvee_{j=1}^{i} v(n_j)\right),\end{aligned}$$

where the second line is due to equality (3), the third line is due to distributivity and equality (2) applied to $v_{i-1}$. Therefore, equality (2) holds for all $0 \leq i < k$. □

Now we can complete the proof of Proposition 1. For a non-leaf OR-node $n$ with $b \geq 1$ children, two cases are possible:

[2]By after the 0th loop, we mean before the beginning of the first loop.

- No break has taken place, which means $k = b + 1$ and the algorithm has looped through all children of $n$. Then we have $f(n, \alpha, \beta) = v_b$. Plugging $i = b$ into equality (2), we get

$$\begin{aligned}\beta \wedge (\alpha \vee v_b) &= \beta \wedge \left(\alpha \vee I \vee \bigvee_{j=1}^{b} v(n_j)\right)\\ &= \beta \wedge \big(\alpha \vee I \vee v(n)\big)\\ &= \beta \wedge \big(\alpha \vee v(n)\big)\end{aligned}$$

where the last equality is due to $I = h(n, \alpha, \beta) \leq v(n) \vee \alpha$ for an OR node since $h$ is admissible. Hence, equality (1) holds for node $n$.

- A break happens during the $k$th loop where $1 \leq k \leq b$, which means $\alpha_k = \alpha \vee v_{k-1} \geq \beta$ (Line 10 in Algorithm 1) and $f(n, \alpha, \beta) = v_{k-1}$. On the right hand side of equality (1), we have

$$\begin{aligned}\beta \wedge \big(\alpha \vee v(n)\big) &= \beta \wedge \big(\alpha \vee I \vee v(n)\big)\\ &= \beta \wedge \left(\alpha \vee I \vee \bigvee_{j=1}^{b} v(n_j)\right)\\ &\geq \beta \wedge \left(\alpha \vee I \vee \bigvee_{j=1}^{k-1} v(n_j)\right)\\ &= \beta \wedge (\alpha \vee v_{k-1}),\end{aligned}$$

where the first line is due to $I \leq \alpha \vee v(n)$, the second one is by definition of $v(n)$, and the last line is due to equality (2) applied to $i = k - 1$. Hence,

$$\beta \geq \beta \wedge \big(\alpha \vee v(n)\big) \geq \beta \wedge (\alpha \vee v_{k-1}) = \beta,$$

which means all inequalities are equalities. Hence,

$$\beta \wedge \big(\alpha \vee f(n, \alpha, \beta)\big) = \beta \wedge \big(\alpha \vee v_{k-1}\big) = \beta \wedge \big(\alpha \vee v(n)\big).$$

which means, equality (1) holds for node $n$.

□

Intuitively, Proposition 1 states that the value returned by Algorithm 1 is the exact value of $n$ up to a factor of $\alpha$ and $\beta$. So even for partially ordered values, alpha-beta search can be interpreted as search with a pruning window.

Now it follows that Algorithm 1 is correct in the sense that if we use a lower and an upper bound as the initial search window at a node, we can recover its exact value using the returned value of the algorithm.

**Corollary 1.** *If $h$ is admissible for $G$ and $V$, then for any node $n$ of $G$ and any $\underline{v}, \overline{v} \in V$ satisfying $\underline{v} \leq v(n) \leq \overline{v}$, we have $v(n) = \overline{v} \wedge \big(\underline{v} \vee f(n, \underline{v}, \overline{v})\big)$. In particular, $v(n) = f(n, \bot, \top)$.*

*Proof.* Plug $\alpha = \underline{v}$ and $\beta = \overline{v}$ into the equality in Proposition 1, and use the fact that $\underline{v} \vee v(n) = v(n) = \overline{v} \wedge v(n)$. □

Importantly, contrary to the case of totally ordered values, it is *not* true in general that the stronger equality $v(n) = f(n, \underline{v}, \overline{v})$ holds, as the example in Figure 1 (left) shows. Let $h(r, \cdot, \cdot) = \bot = 0000$ and $h(n, \cdot, \cdot) = \top = 1111$, so that $h$ is admissible. Recall that $v(r) = 1100$. For $\underline{v} = 1000$ and $\overline{v} = 1110$, indeed $\underline{v} \leq v(r) \leq \overline{v}$. However, $f(r, \underline{v}, \overline{v}) = 1101 \neq v(r)$ since an $\alpha$-cut happens in the call on $n$ (Line 10) after examining its first child, since at this point $\alpha = \beta = 1100$. However, since $\alpha = \underline{v}$ and $\beta = \overline{v}$ yield no constraint on the fourth component of the values, we still have $v(r) = \overline{v} \wedge \big(\underline{v} \vee f(r, \underline{v}, \overline{v})\big)$, as stated in Corollary 1.

## 5 Alpha-beta duo algorithm

We now come to the main contribution of our work, namely a caching scheme for reusing previously computed values in an alpha-beta search for partially ordered values. The trouble of standard alpha-beta search is that the returned value is equal to the exact value only up to a factor of $\alpha$ and $\beta$. It is therefore a nontrivial question how to reuse a previously computed value, since subsequent revisits of a node may come with different search windows.

In alpha-beta search with cache for totally ordered values [16, for instance], one can exploit the fact that with usual value initialisation, the value $f(n, \alpha, \beta)$ satisfies $f(n, \alpha, \beta) < \beta \Rightarrow v(n) \leq f(n, \alpha, \beta)$ and $f(n, \alpha, \beta) > \alpha \Rightarrow v(n) \geq f(n, \alpha, \beta)$. In particular, if $\alpha < f(n, \alpha, \beta) < \beta$, then $f(n, \alpha, \beta) = v(n)$. Hence by comparing $f(n, \alpha, \beta)$ to $\alpha$ and $\beta$, one can determine whether it is exact, a lower, or an upper bound, and store it in the cache with an appropriate flag.

However, this does not hold in general for partially ordered values, as shown on Figure 1 (right). For $\alpha = 010$ and $\beta = 110$, a $\beta$-cut happens after evaluating the first child of $C$, and an $\alpha$-cut after evaluating the first child of $D$. Hence, the algorithm returns 010 for $R$, which is neither a lower nor an upper bound of the exact value 001. In fact, these two values are incomparable in the lattice. If $R$ is an internal node in a DAG, then caching this returned value 010 may cause an evaluation error when the algorithm revisits $R$.

To tackle this difficulty, we propose a new algorithm named ‘alpha-beta duo’, which computes a pair of values for all nodes instead of one single value. The algorithm is presented in Algorithm 2, where Cache refers to a transposition table the entries of which are pairs of values indexed by nodes of $G$, and $(\underline{h}, \overline{h})$ refers to a pair of initialisation functions for which we assume the following property.

**Definition 4.** *A pair $(\underline{h}, \overline{h})$ of initialisation functions is said to be* admissible *for $G$ and $V$ if for any node $n$ in $G$, $\underline{h}(n) \leq v(n) \leq \overline{h}(n)$ holds, and in addition, if $n$ is a leaf node, $\underline{h}(n) = \overline{h}(n) = v(n)$ holds.*

In other words, admissible $\underline{h}$ and $\overline{h}$ respectively underestimates and overestimates the value of a node. Note that $\underline{h}$ and $\overline{h}$ that assign respectively $\bot$ and $\top$ to all internal nodes form an admissible pair, which can always be used if one does not have better heuristic functions.

**Algorithm 2:** Alpha-beta duo search

```
def AlphaBetaDuo(node $n$, $\alpha$, $\beta$):
    if there is an entry for $n$ in the cache:
        $(\underline{c}, \overline{c}) \leftarrow \mathsf{Cache}(n)$
    else:
        $(\underline{c}, \overline{c}) \leftarrow (\underline{h}(n), \overline{h}(n))$
    if $\underline{c} = \overline{c}$:
        return $(\underline{c}, \overline{c})$
    $\alpha \leftarrow \alpha \vee \underline{c}$
    $\beta \leftarrow \beta \wedge \overline{c}$
    if $n$ is an OR-node:
        $(\underline{v}, \overline{v}) \leftarrow (\bot, \bot)$
    else:
        $(\underline{v}, \overline{v}) \leftarrow (\top, \top)$
    determine the children $n_1, \ldots, n_b$ of $n$
    for $i$ in $\{1, \ldots, b\}$:
        if $\alpha \geq \beta$:
            if $n$ is an OR-node:
                $\overline{v} = \overline{c}$
            else:
                $\underline{v} = \underline{c}$
            break
        $\underline{v}', \overline{v}' \leftarrow$ AlphaBetaDuo($n_i$, $\alpha$, $\beta$)
        if $n$ is an OR-node:
            $\underline{v} \leftarrow \underline{v} \vee \underline{v}'$
            $\overline{v} \leftarrow \overline{v} \vee \overline{v}'$
            $\alpha \leftarrow \alpha \vee \underline{v}'$
        else:
            $\underline{v} \leftarrow \underline{v} \wedge \underline{v}'$
            $\overline{v} \leftarrow \overline{v} \wedge \overline{v}'$
            $\beta \leftarrow \beta \wedge \overline{v}'$
    $\underline{v} \leftarrow \underline{v} \vee \underline{c}$
    $\overline{v} \leftarrow \overline{v} \wedge \overline{c}$
    store $(\underline{v}, \overline{v})$ in the cache under an entry for $n$
    return $(\underline{v}, \overline{v})$
```

Alpha-beta duo search works in the following manner:

- First, variables $\underline{c}$ and $\overline{c}$ denote respectively the best lower and upper bound of $v(n)$ available to the algorithm before this call. If $n$ has already been visited, then $\underline{c}$ and $\overline{c}$ are retrieved from the cache. Otherwise, they are given by the initialisation functions $\underline{h}$ and $\overline{h}$. If $\underline{c} = \overline{c}$, $(\underline{c}, \overline{c})$ is returned immediately.
- Otherwise, by symmetry consider the case when $n$ is an OR-node. During the main loop, $\underline{v}$ and $\overline{v}$ are respectively the cumulative lower and upper bound of $v(n)$ (notice that they are both initialised to $\bot$ for an OR-node). If a cut ever happens, it means not all children of $n$ have been evaluated, hence $\overline{v}$ is not a valid upper bound of $v(n)$. Then we take $\overline{v}$ to be $\overline{c}$, the best upper bound previously known. On the other hand, $\underline{v}$, which is the join of lower bounds of children of $n$ that have been evaluated, is a valid lower bound so we keep it.
- Finally, after the main loop, $\underline{v}$ and $\overline{v}$ are the lower and upper bounds of $v(n)$ computed by the current call. Hence they are combined with the previously known bounds $\underline{c}$ and $\overline{c}$ to yield to best currently known bounds on $v(n)$ and they are cached.

We now prove that alpha-beta duo is correct. For this, we first need the following notion.

**Definition 5.** *A cache* Cache *is said to be* coherent *for $G$ and $V$ if for any node $n$ in $G$, if there is an entry for $n$ in the cache, then* $\mathsf{Cache}(n) = (\underline{c}, \overline{c})$ *where $\underline{c} \leq v(n) \leq \overline{c}$, and in addition, if $n$ is a leaf node, then $\underline{c}(n) = \overline{c}(n) = v(n)$.*

Obviously, an empty cache is coherent for any $G$ and $V$.

In the following, we denote the pair of values returned by Algorithm 2 with input $n$, $\alpha$, $\beta$ by $(\underline{f}(n, \alpha, \beta), \overline{f}(n, \alpha, \beta))$. We first show that if the cache is initially coherent, then it remains coherent after the execution, and that any interval stored in it cannot become looser.

**Proposition 2.** *If $(\underline{h}, \overline{h})$ is admissible and* Cache *is initially coherent for $G$ and $V$, then for any node $n$ in $G$ and any $\alpha, \beta \in V$, we have*

$$\underline{f}(n, \alpha, \beta) \leq v(n) \leq \overline{f}(n, \alpha, \beta). \tag{4}$$

*Moreover, if there is an entry $(\underline{c}, \overline{c})$ in the cache for $n$ before the call, then $\underline{c} \leq \underline{f}(n, \alpha, \beta)$ and $\overline{f}(n, \alpha, \beta) \leq \overline{c}$.*

*Proof.* $\underline{c} \leq \underline{f}(n, \alpha, \beta)$ and $\overline{f}(n, \alpha, \beta) \leq \overline{c}$ are direct consequence of Line 31 and 32.

The proof of inequality (4) is based on structural induction. We will only focus on OR nodes since the case for AND nodes is completely symmetric. So let $n$ be an OR node and let us consider the execution of the function call AlphaBetaDuo($n$, $\alpha$, $\beta$).

If $n$ is a leaf node, then whether or not there is an entry for $n$ in the cache, on Line 6 we have $\underline{c} = \overline{c} = v(n)$ since $(\underline{h}, \overline{h})$ is admissible and Cache is coherent. Hence the function returns immediately, so inequality (4) holds and the cache remains coherent.

Otherwise, $n$ is an internal OR-node. Again, whether or not there is an entry for $n$ in the cache, on Line 6 and forward we have $\underline{c} \leq v(n) \leq \overline{c}$, since $(\underline{h}, \overline{h})$ is admissible and Cache is coherent.

Let $b \geq 1$ be the number of children of $n$. We assume by induction that all function calls on the children of $n$ satisfy inequality (4) and maintain the coherence of the cache. Let $k$ be the index of loop during which a break happens (if no break happens, then $k$ is taken to be $b+1$). For $1 \le i \le k-1$, let $\underline{v}^i$ and $\overline{v}^i$ denote the values returned by the function call on the child $n_i$ during the $i$th loop. Then by induction assumption, inequality (4) yields $\underline{v}^i \le v(n_i) \le \overline{v}^i$ for $1 \le i \le k-1$.

Hence after the loop (i.e. just before Line 31),

$$\underline{v} = \bigvee_{i=1}^{k-1} \underline{v}^i \le \bigvee_{i=1}^{k-1} v(n_i) \le \bigvee_{i=1}^{b} v(n_i) = v(n).$$

As for $\overline{v}$, we have two cases.

- No break happens, i.e. $k = b+1$. Then

$$\overline{v} = \bigvee_{i=1}^{b} \overline{v}^i \ge \bigvee_{i=1}^{b} v(n_i) = v(n).$$

- Otherwise, a break happens, and $\overline{v} = \overline{c} \ge v(n)$.

So in both cases, $\overline{v} \ge v(n)$.

Therefore, after the final updates on Line 31 and 32, we have $\underline{v} \le v(n) \le \overline{v}$. As a result, the returned values of the function call $\texttt{AlphaBetaDuo}(n, \alpha, \beta)$ satisfy inequality (4). And the cache remains coherent after the function call. □

We can now prove results parallel to those in Section 4.

**Proposition 3.** *If $(\underline{h}, \overline{h})$ is admissible and* Cache *is initially coherent for $G$ and $V$, then for any node $n$ in $G$ and any $\alpha, \beta \in V$ we have*

$$\beta \wedge \big(\alpha \vee \underline{f}(n,\alpha,\beta)\big) = \beta \wedge \big(\alpha \vee \overline{f}(n,\alpha,\beta)\big). \qquad (5)$$

*Proof.* Again, we will only focus on OR nodes since the case for AND nodes is symmetric.

If $n$ is a leaf node, then $\underline{f}(n,\alpha,\beta) = \overline{f}(n,\alpha,\beta) = v(n)$ since $(\underline{h}, \overline{h})$ is admissible and Cache is coherent. Hence equality (5) holds.

Otherwise, let $b \ge 1$ be the number of children of $n$. We assume by induction that all function calls on the children of $n$ satisfy inequality (5). Let $k$ be the index of loop during which a break happens (if no break happens, then $k$ is taken to be $b+1$). For $1 \le j < k$, let $\underline{v}^j$ and $\overline{v}^j$ denote the values returned by the function call on $n_j$ during the $j$th loop. For $0 \le i < k$, let $\underline{v}_i$, $\overline{v}_i$, and $\alpha_i$ denote the value of $\underline{v}$, $\overline{v}$, and $\alpha$ after the $i$th loop. Then for $0 \le i < k$, we have $\underline{v}_i = \bigvee_{j=1}^{i} \underline{v}^j$, $\overline{v}_i = \bigvee_{j=1}^{i} \overline{v}^j$, and $\alpha_i = \alpha \vee \underline{c} \vee \underline{v}_i$. In particular, $\underline{v}_0 = \overline{v}_0 = \bot$ and $\alpha_0 = \alpha \vee \underline{c}$. For $1 \le j < k$, since function call on the child $n_j$ has the form $\texttt{AlphaBetaDuo}(n_j, \alpha_j, \beta \wedge \overline{c})$, by equality (5), we have

$$\beta \wedge \overline{c} \wedge \big(\alpha_j \vee \underline{v}^j\big) = \beta \wedge \overline{c} \wedge \big(\alpha_j \vee \overline{v}^j\big).$$

Hence by distributivity, we have

$$\beta \wedge \overline{c} \wedge \bigvee_{j=1}^{k-1} \big(\alpha_j \vee \underline{v}^j\big) = \beta \wedge \overline{c} \wedge \bigvee_{j=1}^{k-1} \big(\alpha_j \vee \overline{v}^j\big). \qquad (6)$$

We distinguish two cases.

- No break happens (i.e. $k = b+1$). Then

$$\underline{f}(n,\alpha,\beta) = \underline{c} \vee \underline{v}_b = \underline{c} \vee \bigvee_{j=1}^{b} \underline{v}^j,$$

$$\overline{f}(n,\alpha,\beta) = \overline{c} \wedge \overline{v}_b = \overline{c} \wedge \bigvee_{j=1}^{b} \overline{v}^j.$$

First notice that the distributivity of the lattice implies that for any $x, y, z \in V$,

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) = x \vee \big(y \wedge (x \vee z)\big), \qquad (7)$$

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) = x \wedge \big(y \vee (x \wedge z)\big). \qquad (8)$$

Applying equalities (7) and (8), one has

$$\begin{aligned}\beta \wedge \big(\alpha \vee \overline{f}(n,\alpha,\beta)\big) &= \beta \wedge \left(\alpha \vee \left(\overline{c} \wedge \bigvee_{j=1}^{b} \overline{v}^j\right)\right)\\ &= \beta \wedge \left(\alpha \vee \left(\overline{c} \wedge \left(\alpha \vee \bigvee_{j=1}^{b} \overline{v}^j\right)\right)\right)\\ &= \beta \wedge \left(\alpha \vee \left(\beta \wedge \overline{c} \wedge \left(\alpha \vee \bigvee_{j=1}^{b} \overline{v}^j\right)\right)\right).\end{aligned}$$

We will first focus on the term $\beta \wedge \overline{c} \wedge (\alpha \vee \bigvee_{j=1}^{b} \overline{v}^j)$. Our goal is to massage it into a form to which the induction assumption (6) can apply.

Recall that for $j \le b$, $\alpha_j = \alpha_0 \vee \underline{v}_j = \alpha_0 \vee \bigvee_{l=1}^{j} \underline{v}^l$. Hence,

$$\begin{aligned}\bigvee_{j=1}^{b} (\alpha_j \vee \overline{v}^j) &= \bigvee_{j=1}^{b} \left(\alpha_0 \vee \bigvee_{l=1}^{j} \underline{v}^l \vee \overline{v}^j\right)\\ &= \alpha_0 \vee \bigvee_{j=1}^{b} \bigvee_{l=1}^{j} \underline{v}^l \vee \bigvee_{j=1}^{b} \overline{v}^j\\ &= \alpha_0 \vee \bigvee_{j=1}^{b} \underline{v}^j \vee \bigvee_{j=1}^{b} \overline{v}^j\\ &= \alpha_0 \vee \bigvee_{j=1}^{b} \overline{v}^j,\end{aligned}$$

where in the last equality we use the fact that by Proposition 2, we have $\underline{v}^j \leq \overline{v}^j$ for any $j \leq b$. Similarly, we have

$$\bigvee_{j=1}^{b}(\alpha_j \vee \underline{v}^j) = \bigvee_{j=1}^{b}\left(\alpha_0 \vee \bigvee_{l=1}^{j} \underline{v}^l \vee \underline{v}^j\right) = \alpha_0 \vee \bigvee_{j=1}^{b} \underline{v}^j.$$

Hence, by $\alpha_0 = \alpha \vee \underline{c} \geq \alpha$, the two previous equalities, distributivity, and the induction assumption (6),

$$\begin{aligned}\beta \wedge \overline{c} \wedge \left(\alpha \vee \bigvee_{j=1}^{b} \overline{v}^j\right) &\leq \beta \wedge \overline{c} \wedge \left(\alpha_0 \vee \bigvee_{j=1}^{b} \overline{v}^j\right)\\ &= \beta \wedge \overline{c} \wedge \bigvee_{j=1}^{b}(\alpha_j \vee \overline{v}^j)\\ &= \beta \wedge \overline{c} \wedge \bigvee_{j=1}^{b}(\alpha_j \vee \underline{v}^j)\\ &= \beta \wedge \overline{c} \wedge \left(\alpha_0 \vee \bigvee_{j=1}^{b} \underline{v}^j\right)\\ &\leq \beta \wedge \left(\alpha_0 \vee \bigvee_{j=1}^{b} \underline{v}^j\right).\end{aligned}$$

Therefore, applying again equalities (7) and (8) yields

$$\begin{aligned}&\beta \wedge \big(\alpha \vee \overline{f}(n,\alpha,\beta)\big)\\ &= \beta \wedge \left(\alpha \vee \left(\beta \wedge \overline{c} \wedge \left(\alpha \vee \bigvee_{j=1}^{b} \overline{v}^j\right)\right)\right)\\ &\leq \beta \wedge \left(\alpha \vee \left(\beta \wedge \left(\alpha_0 \vee \bigvee_{j=1}^{b} \underline{v}^j\right)\right)\right)\\ &= \beta \wedge \left(\alpha \vee \alpha_0 \vee \bigvee_{j=1}^{b} \underline{v}^j\right)\\ &= \beta \wedge \left(\alpha \vee \underline{c} \vee \bigvee_{j=1}^{b} \underline{v}^j\right)\\ &= \beta \wedge \big(\alpha \vee \underline{f}(n,\alpha,\beta)\big).\end{aligned}$$

- A break happens (i.e. $1 \leq k \leq b$). Then $\overline{f}(n,\alpha,\beta) = \overline{c}$ and $\underline{f}(n,\alpha,\beta) = \underline{c} \vee \underline{v}_{k-1}$. Since a break happens during the $k$th loop, according to Line 16 in Algorithm 2 we have $\beta \wedge \overline{c} \leq \alpha_{k-1} = \alpha \vee \underline{c} \vee \underline{v}_{k-1}$. Hence by distributivity,

$$\begin{aligned}\beta \wedge \big(\alpha \vee \overline{f}(n,\alpha,\beta)\big) &= \beta \wedge (\alpha \vee \overline{c})\\ &= \beta \wedge \big(\alpha \vee (\beta \wedge \overline{c})\big)\\ &\leq \beta \wedge \big(\alpha \vee (\alpha \vee \underline{c} \vee \underline{v}_{k-1})\big)\\ &= \beta \wedge \big(\alpha \vee \underline{f}(n,\alpha,\beta)\big).\end{aligned}$$

Therefore, no matter a break happens or not, we have

$$\beta \wedge \big(\alpha \vee \overline{f}(n,\alpha,\beta)\big) \leq \beta \wedge \big(\alpha \vee \underline{f}(n,\alpha,\beta)\big).$$

On the other hand, by Proposition 2, $\overline{f}(n,\alpha,\beta) \geq \underline{f}(n,\alpha,\beta)$, which means

$$\beta \wedge \big(\alpha \vee \overline{f}(n,\alpha,\beta)\big) \geq \beta \wedge \big(\alpha \vee \underline{f}(n,\alpha,\beta)\big).$$

As a consequence, equality (5) holds. □

**Corollary 2.** *If $(\underline{h}, \overline{h})$ is admissible and* Cache *is initially coherent for $G$ and $V$, then for any node $n$ in $G$ and any $\underline{v}, \overline{v} \in V$ satisfying $\underline{v} \leq v(n) \leq \overline{v}$, we have $v(n) = \overline{v} \wedge \big(\underline{v} \vee \underline{f}(n,\underline{v},\overline{v})\big) = \overline{v} \wedge \big(\underline{v} \vee \overline{f}(n,\underline{v},\overline{v})\big)$. In particular, $v(n) = \underline{f}(n,\bot,\top) = \overline{f}(n,\bot,\top)$.*

*Proof.* By Proposition 2, $\underline{f}(n,\underline{v},\overline{v}) \leq v(n) \leq \overline{f}(n,\underline{v},\overline{v})$. Using $\overline{v} \wedge \big(\underline{v} \vee v(n)\big) = v(n)$, we get

$$\overline{v} \wedge \big(\underline{v} \vee \underline{f}(n,\underline{v},\overline{v})\big) \leq v(n) \leq \overline{v} \wedge \big(\underline{v} \vee \overline{f}(n,\underline{v},\overline{v})\big).$$

These inequalities are in fact equalities since from Proposition 3 we have $\overline{v} \wedge \big(\underline{v} \vee \underline{f}(n,\underline{v},\overline{v})\big) = \overline{v} \wedge \big(\underline{v} \vee \overline{f}(n,\underline{v},\overline{v})\big)$. □

## 6 Experiments

To assess the efficiency of alpha-beta duo (hereafter 'ABD'), we ran experiments comparing it to three other algorithms:

- alpha-beta without cache (Algorithm 1, 'AB' for short);
- an alpha-beta search which only caches the value computed for a node if no cut is found during the search in the subtree rooted at this node (hereafter 'ABC');
- a minimax search algorithm without alpha-beta pruning, but with a cache (hereafter 'MMC').

The code of ABD was slightly optimized by refining the values computed on Lines 31 and 32 with the corresponding bounds of all fully explored children. It is easy to show that this preserves the correctness of the algorithm.

For all experiments, we measured the number of nodes of the DAG visited at least once, the total number of node visits (equivalently, the total number of recursive calls), and the time taken for solving the problem. Intuitively, we expect ABD to be better than ABC, ABC to be better than MMC (because MMC does not use alpha-beta pruning), and MMC to be better than AB (because the latter does not cache its results and hence, recomputes several times for the same node).

We used two synthetic sets of benchmarks. The first (hereafter 'random') consists of random DAGs of the same kind as the one in Figure 1. Random DAGs with parameters $d$ (depth), $b$ (branching factor), and $v$ (number of variables) are generated in the following manner:

- $d$ layers $0, 1, \ldots, d-1$ are built: layer $i$ consists of $3^{\min(i, \frac{3d}{2}-i)}$ nodes (which yields diamond-shapes DAGs);
- from each node $n$, a set of $b$ nodes is randomly chosen from nodes in the next layer to be $C(n)$;
- each internal node is labelled AND or OR at random;
- the value of each terminal node is a uniformly drawn subset of $\{1, \ldots, v\}$, or equivalently a random Boolean vector of length $v$.

We also consider strictly alternating DAGs, in which all nodes in layer $i$ are OR-nodes (respectively AND-nodes) if $i$ is even (respectively odd). In particular, the root is an OR-node.

The second set of benchmarks consists of a simplified version of the card game Bridge that we call 'racing'. There are two players, MIN and MAX. Each has a hand of $h$ cards drawn uniformly from the deck $\{1, \ldots, d\}$ where $d \geq 2h$. Players only see their own hand. MIN begins the game. During each trick, the player who begins plays a card from her hand, the other sees it and plays a card in turn. The player who played the highest card wins this trick and starts the next one. No new card is ever drawn from the deck. The game ends when the players have no more cards or when one has won in total $g$ tricks. MAX wins if she is the first one to reach $g$ tricks. For the benchmark, each instance with parameter $h$, $d$, and $g$ consists of a randomly drawn hand with $h$ cards for MAX and a randomly drawn card that is supposed to be played by MIN during the first trick. Notice that when $d > 2h$, each player has incomplete information. We use evaluation of AND-OR DAGs to compute optimal strategies for MAX against the best defence adversary model defined in [6].

In games with incomplete information where $S$ is the set of all possible hidden configurations, [6, 7] define the maxmin value of player MAX to be the set of all subsets $S'$ of $S$ such that there is a uniform strategy winning in any configuration of $S'$. [9] shows that computing this value amounts to evaluate the game DAG with the lattice $(2^{2^S}, \preceq, \sqcap, \cup)$. Intuitively, using $\cup$ at OR-nodes models the fact that player MAX can choose any child as her strategy, and $\sqcap$ at AND-nodes models the fact that a strategy must be robust to all adversarial strategies, hence a strategy wins in $s \in S$ if and only if it wins in $s$ whatever action her opponent chooses.

In the same setting of games with incomplete information, one can also be interested in non-uniform strategies which allow player MAX's actions to depend on the hidden information. This can be useful for computing heuristic values of for the game with incomplete information. It can be seen that the set of all configurations for which there is a non-uniform winning strategy can be computed as the value of the game DAG with the lattice $(2^S, \subseteq, \cap, \cup)$.

Hence, for both benchmarks, we consider the two lattices $(2^{2^S}, \sqcap, \cup)$ and $(2^S, \cap, \cup)$. In 'random', $S = \{1, \ldots, v\}$, while in 'racing' $S$ is the set of all possible hands of player MIN.

For space reasons, we only give the most representative results, in terms of computation time. For each parameter setting and each algorithm, we averaged over 10 runs. Figure 3 shows two examples where, as can be expected, it is more efficient to cache bounds, even more to perform cuts, and still more to compute and store two bounds per node. On the top plot, the gain of using ABD is exponential: with the branching factor increasing, ABD gets a better advantage of computing and caching two bounds. On the bottom plot, ABC and AB (not represented) are exponentially worse, and ABD is better than MMC when the branching factor is high.

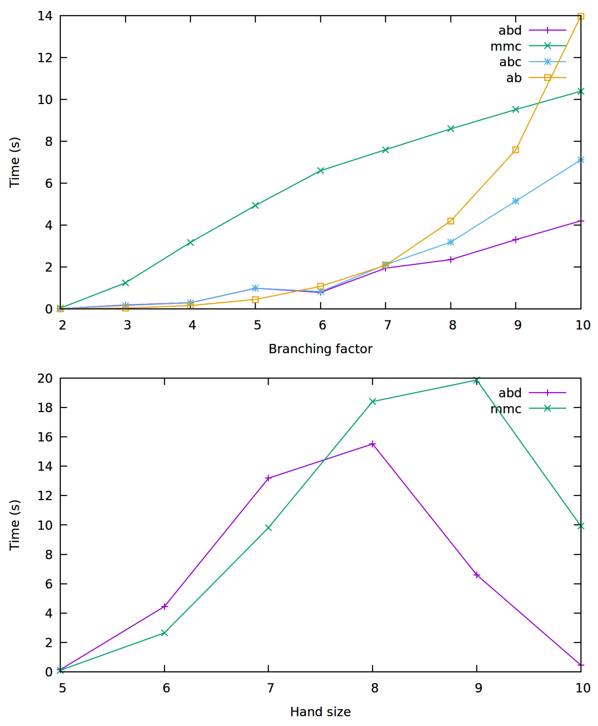

Figure 3: Experimental results on random (top) and racing (bottom). Top: $d = 15$, $v = 10$ (varying $b$), alternating DAGs. Bottom: $d = 20$, $g = 5$ (varying $h$). The lattice is $2^S$ in both cases.

Now Figure 4 shows two examples where it turns out that it is not always better to use alpha-beta pruning with cache.

On the top plot, not caching results at all turns out to be better: the overhead due to the additional operations from the lattice $2^{2^S}$ (which are necessary to maintain the cache) seems to compensate the advantage of ABC or ABD in terms of number of visited nodes and recursive calls (the curves are reversed for this metrics, not shown here).

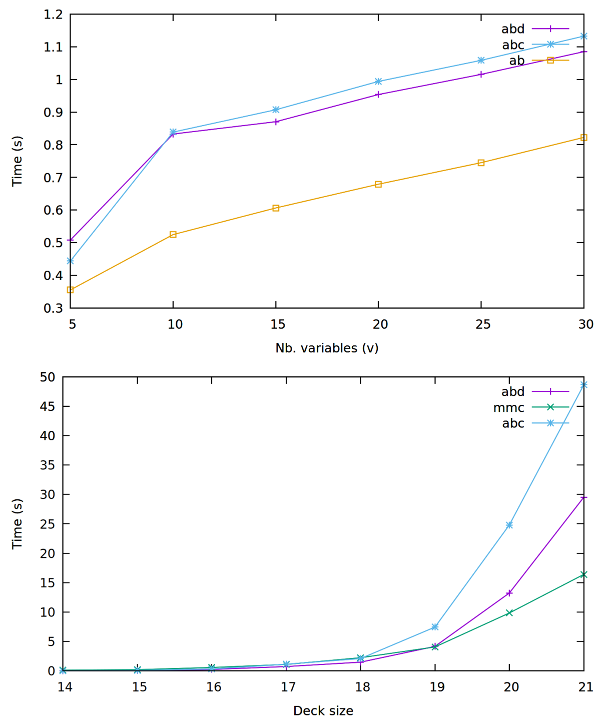

Figure 4: Experimental results on random (top) and racing (bottom). Top: $d = 15$, $b = 4$ (varying $v$), alternating DAGs, lattice $2^{2^S}$. Bottom: $h = 7$, $g = 5$ (varying $d$), lattice $2^S$.

On the bottom plot, it turns out that sometimes alpha-beta pruning even degrades performance. Again, this is due to the overhead of manipulating values from a large lattice (for a fixed hand size, the lattice grows exponentially with the deck size).

To complete these results, let us mention that in most experiments, the numbers of nodes explored and visited by each method are ordered as expected, with ratios varying from linear to exponential. In particular, for these metrics, ABD is most of the time better, and often much better, than the other three algorithms.

Summarising, ABD seems to provide a real gain in (brute) performance for DAGs with high branching factors. Contrastingly, when $\wedge$ and $\vee$ from the lattice are too expensive to compute (as is the case in some large lattices), it may sometimes be better not to use cache and alpha-beta pruning together, due to the overhead to maintain the coherence of the cache.

## 7 Conclusion

We investigated alpha-beta search for AND-OR DAGs with values from a lattice, which has direct applications such as solving games with incomplete information. We have extended previous formal analyses, in particular to the use of heuristic as initialisation functions. Then we have proposed a new algorithm named 'alpha-beta duo', which caches both a lower and an upper bound of the value of each visited node, and we have formally proved its correctness. Experiments show that it is more efficient than other algorithms in terms of number of visited nodes and recursive calls. As for time efficiency, alpha-beta duo turns out to be more efficient than other algorithms for DAGs with large branching factors and reasonably-sized lattices. As an interesting conclusion, our experiments also put forth that in other cases, it may be better not to use a cache with alpha-beta pruning.

Our future work includes algorithmic optimisations for alpha-beta search with cache applied to games with incomplete information. We will also investigate the use of efficient knowledge representations [1, 18] to accelerate lattice operations in such context. Another perspective is to apply our work to games with multiple criteria instead of scalar outcomes.

## References

[1] Meghyn Bienvenu, Hélène Fargier, and Pierre Marquis. Knowledge Compilation in the Modal Logic S5. In *Proc. 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, pages 261–266, 2010.

[2] Tristan Cazenave and Véronique Ventos. The $\alpha\mu$ search algorithm for the game of bridge. In *Proc. Monte Carlo Search International Workshop*, pages 1–16. Springer, 2020.

[3] Pallab Dasgupta, P. P. Chakrabarti, and S. C. De Sarkar. Searching game trees under a partial order. *Artif. Intell.*, 82(1-2):237–257, 1996.

[4] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

[5] Stefan Edelkamp. Representing and reducing uncertainty for enumerating the belief space to improve endgame play in skat. In *Proc. 24th European Conference on Artificial Intelligence (ECAI 2020)*, pages 395–402. IOS Press, 2020.

[6] Ian Frank and David A. Basin. Search in games with incomplete information: A case study using bridge card play. *Artif. Intell.*, 100(1-2):87–123, 1998.

[7] Ian Frank and David A. Basin. A theoretical and empirical investigation of search in imperfect information games. *Theor. Comput. Sci.*, 252(1-2):217–256, 2001.

[8] M Ginsberg and Alan Jaffray. Alpha-beta pruning under partial orders. In Richard Nowakowski, editor,

*More Games of No Chance*, number 42 in Mathematical Sciences Research Institute Publications, pages 37–48. Cambridge University Press, 2002.

[9] Matthew L. Ginsberg. GIB: imperfect information in a computationally challenging game. *J. Artif. Intell. Res.*, 14:303–358, 2001.

[10] Guy Haworth and Nelson Hernandez. The 20th top chess engine championship, TCEC20. *ICGA Journal*, 43(1):62–73, 2021.

[11] Peter Kissmann and Stefan Edelkamp. Solving fully-observable non-deterministic planning problems via translation into a general game. In Bärbel Mertsching, Marcus Hund, and Muhammad Zaheer Aziz, editors, *Proc. 32nd Annual German Conference on Artificial Intelligence (KI 2009)*, pages 1–8. Springer, 2009.

[12] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artif. Intell.*, 6(4):293–326, 1975.

[13] Sebastian Kupferschmid and Malte Helmert. A skat player based on monte-carlo simulation. In *Proc. 5th International Conference on Computers and Games (CG 2006)*, pages 135–147. Springer, 2006.

[14] David NL Levy. The million pound bridge program. *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, pages 95–103, 1989.

[15] Jean-Vincent Loddo and Luca Saiu. How to correctly prune tropical trees. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *Proc. 10th International Conference on Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2010.

[16] T. A. Marsland. A review of game-tree pruning. *J. Int. Comput. Games Assoc.*, 9(1):3–19, 1986.

[17] Yu Nasu. Efficiently updatable neural-network-based evaluation functions for computer shogi. *The 28th World Computer Shogi Championship Appeal Document*, 2018.

[18] Alexandre Niveau and Bruno Zanuttini. Efficient Representations for the Modal Logic S5. In *Proc. 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 2016.

[19] Douglas Rebstock, Christopher Solinas, Michael Buro, and Nathan R Sturtevant. Policy based inference in trick-taking card games. In *Proc. 2019 IEEE Conference on Games (CoG 2019)*, pages 1–8. IEEE, 2019.

[20] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[21] Nathan R Sturtevant and Adam M White. Feature construction for reinforcement learning in hearts. In *Proc. 5th International Conference on Computers and Games (CG 2006)*, pages 122–134. Springer, 2006.

[22] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. *Artif. Intell.*, 134(1-2):277–311, 2002.