Formal Verification of Minimax Algorithms: A Case Study in Dafny

Wieger Wesselink[1], Kees Huizing[1], and Huub van de Wetering[1]

Eindhoven University of Technology, The Netherlands
{j.w.wesselink,h.v.d.wetering,c.huizing}@tue.nl

Abstract. Minimax search algorithms play a central role in game-playing engines, where they are combined with alpha-beta pruning to reduce the search space and with transposition tables to cache intermediate results. The resulting algorithms are subtle and notoriously difficult to reason about, making non-obvious errors hard to detect. We present a verification case study of a range of minimax and negamax search algorithms using the Dafny auto-active verification system, covering basic variants, alpha-beta pruning, and depth-limited search with transposition tables. For the verification of algorithms with transposition tables, we introduce a witness-based correctness criterion that characterizes when a cached result can be justified by an explicit game-tree, a non-trivial requirement given that cache entries are reused across different search depths. Using this criterion, we obtain a fully mechanized correctness proof for two different transposition table implementations. In contrast, the same approach for the classic transposition-table formulation due to Marsland fails with our criterion. Guided by this Dafny verification failure, we construct a concrete counterexample demonstrating a violation of our criterion: it may return values for which no valid witness exists and which are inconsistent with minimax semantics. Notably, this inconsistency is independent of the chosen witness criterion. All verification artifacts, including the Dafny proofs and executable Python reference implementations used for validation, are publicly available.

Keywords: Formal Verification · Dafny · Minimax · Negamax · Alpha-Beta Pruning · Depth-limited search · Transposition Tables.

1 Introduction

Game-tree search algorithms based on minimax and alpha–beta pruning are a standard technique for computing optimal moves in deterministic two-player games. Since the seminal analysis of Knuth and Moore [5] and the AND/OR tree formulation of Stockman [10], these algorithms have been studied extensively and have become a standard technique for game-playing engines. Practical implementations further improve efficiency by using transposition tables, which cache search results so that positions reached through different move sequences need not be explored repeatedly. While highly effective in practice, this optimization substantially complicates formal reasoning about correctness because

cached information is reused across multiple search contexts rather than being derived from a single execution tree.

Recent work by Nipkow et al. [8] mechanically verifies several minimax variants, including an algorithm with transposition tables, in a functional setting using the Isabelle proof assistant. It also formally clarifies the relationship between fail-hard and fail-soft alpha–beta pruning, showing that fail-soft yields bounds at least as tight as fail-hard when the search falls outside the alpha–beta window. It considers, however, depth-unlimited search, in which cached entries are always derived from complete exploration of a subtree. Once depth limits are introduced, correctness becomes considerably more subtle: cached information may originate from searches performed with different depth limits or alpha–beta windows and must remain semantically valid when reused. This challenge also appears in the context of QBF-based game solving, where He et al. [4] observe that defining a sound transposition-table mechanism remains an open problem.

Other formalizations address game-tree search with different tool support; for example, Escobar and Insuasti [2] verify multi-threaded minimax behavior for Connect 4 in mCRL2, focusing on concurrency rather than the semantic reuse of partial results in sequential depth-limited search. More broadly, backtracking and memoized algorithms have been studied in proof assistants such as Isabelle/HOL and Coq (e.g., Wimmer et al. [14] on verified memoization). Depth-limited alpha–beta search with transposition tables, however, introduces a distinct problem: cached bounds are partial and context-dependent, so standard memoization invariants do not transfer directly.

Earlier work by Warnock and Wendroff [11] demonstrated that subtle implementation choices can lead to behavioral inconsistencies when alpha–beta pruning is combined with transposition tables. As we show in Section 4.3, however, semantic incorrectness can arise even in the absence of such inconsistencies and is therefore not captured by behavioral analyses alone. Correctness can no longer be understood solely in terms of the explored search tree, but must also account for the validity of cached information across search contexts.

To address this problem, we introduce a witness-based correctness criterion for depth-limited game-tree search with transposition tables. Rather than reasoning directly about the explored search tree, the criterion requires a suitable witness expansion that justifies every cached bound returned by the algorithm. This provides a foundation for verifying imperative game-tree search algorithms combining alpha–beta pruning, depth-limited search, and transposition tables.

We evaluate this criterion in a mechanized case study using the Dafny verification system [6]. The development covers classical minimax and negamax search, fail-hard and fail-soft alpha–beta pruning, depth-limited search, and three representative transposition-table algorithms: the classic formulation due to Marsland [7] (NegamaxTTM), a depth-limited negamax reformulation of the algorithm of Plaat et al. [9] (NegamaxTTP), and the Wikipedia formulation [13] (NegamaxTTW). While the Plaat and Wikipedia variants satisfy the witness-based correctness criterion, verification of the Marsland algorithm fails. Guided by the resulting proof obligations, we construct a concrete counterex-

ample showing that the algorithm can return a value for which no valid witness expansion exists, demonstrating a violation of our correctness criterion.

Contributions. This paper makes the following contributions:

- We introduce a witness-based correctness criterion for depth-limited game-tree search with transposition tables, based on witness expansions of truncated game trees.
- We mechanize this criterion in Dafny and verify a representative collection of classical game-tree search algorithms, including minimax, negamax, fail-hard and fail-soft alpha–beta pruning, depth-limited search, and three transposition-table formulations.
- We establish correctness of the Wikipedia and Plaat transposition-table variants, and isolate the semantic impact of individual design choices by incrementally porting the four distinguishing features of Marsland's formulation into the Wikipedia variant. Three features preserve correctness, while the fourth causes the algorithm to violate the witness criterion, yielding a concrete counterexample to Marsland's algorithm.
- We release a reusable Dafny artefact containing verified algorithms, executable reference implementations, and supporting proof infrastructure for the comparative verification of game-tree search algorithms.

Our case study demonstrates that modern auto-active verification can not only certify realistic game-tree search algorithms, but also provide reusable proof techniques for algorithms that combine imperative state with recursive search, while exposing subtle semantic flaws in long-standing algorithmic formulations.

2 Definitions

A game tree is a rooted directed acyclic graph representing the possible evolutions of a two-player game. Each node corresponds to a game position, and each edge to a legal move. Nodes are colored by the player to move: white for the maximizer and gray for the minimizer, as illustrated in Figure 1. Formally, a node is defined as a record with three attributes:

$$\text{Node} = \textbf{record} \begin{cases} \mathit{eval} & : \text{evaluation value} \\ \mathit{color} & : \text{player to move } (\pm 1) \\ \mathit{children} & : \text{ordered child list} \end{cases}$$

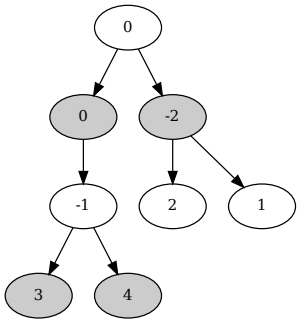


Fig. 1: Game tree with white maximizer nodes and gray minimizer nodes, labeled with evaluation values.

The *eval* field encodes an integer value of a game state: the maximizer ($color$=1) favors higher, while the minimizer ($color$=-1) favors lower values. Internal nodes use heuristic evaluations, while leaves reflect game-theoretic values. The *children* list defines the successors of a node u in a fixed order, inducing edges (u, v) iff $v \in u.children$. A fixed order is not strictly necessary, but it is commonly used

and enables deterministic execution. We assume acyclicity, as repetition rules in games like chess and draughts preclude infinite loops in practice. Nodes abstract from concrete game states: equality is structural, requiring identical *eval*, *color*, and isomorphic subtrees. This suffices for search correctness while avoiding full state representation. Since a node u implicitly defines the subtree rooted at u, we use the terms node and game tree interchangeably. For a game tree u we recursively define $\mathsf{minimax}(u)$ as

$$\begin{cases} u.\mathit{eval} & \text{if } |u.\mathit{children}| = 0 \\ \min\{\mathsf{minimax}(v) \mid v \in u.\mathit{children}\} & \text{if } |u.\mathit{children}| > 0 \text{ and } u.\mathit{color} = -1 \\ \max\{\mathsf{minimax}(v) \mid v \in u.\mathit{children}\} & \text{if } |u.\mathit{children}| > 0 \text{ and } u.\mathit{color} = 1 \end{cases} \tag{1}$$

and $\mathsf{negamax}(u)$ as

$$\begin{cases} u.\mathit{color} \cdot u.\mathit{eval} & \text{if } |u.\mathit{children}| = 0 \\ \max\{-\mathsf{negamax}(v) \mid v \in u.\mathit{children}\} & \text{if } |u.\mathit{children}| > 0. \end{cases} \tag{2}$$

For minimax, we allow the same player to move consecutively. For negamax, the game must be *turn-based*, i.e., for each node u the following holds:

$$\forall v \in u.\mathit{children} : v.\mathit{color} = -u.\mathit{color}. \tag{3}$$

Note that the two functions are the same up to a sign inversion.

$$\mathsf{negamax}(u) = u.\mathit{color} \cdot \mathsf{minimax}(u). \tag{4}$$

In depth-limited search, we work with truncated game trees. For a game tree u with root node r, the *depth-d truncation* $\lfloor u \rfloor_d$ is the subtree consisting of all nodes whose path from r has length at most d, along with all edges connecting them. The root r is at distance 0; nodes at distance greater than d are excluded.

2.1 Alpha-beta Pruning

Alpha-beta pruning is an optimization of minimax search that avoids exploring parts of the game tree that cannot affect the final decision. The algorithm maintains a window (α, β) of values that are relevant to the outcome. If the exact minimax (or negamax) value lies strictly within this window, it must be computed precisely; if it lies outside the window, any value that preserves the appropriate bound is sufficient. This allows the search to terminate early in branches that are irrelevant to the root decision. To formalize this behavior, we introduce several predicates used in the verification of alpha-beta pruning:

$$\mathsf{is\text{-}ab\text{-}result}(x, e, \alpha, \beta) = (e \le x \le \alpha) \ \vee \ (\alpha < e = x < \beta) \ \vee \ (\beta \le x \le e), \tag{5}$$

$$\mathsf{is\text{-}negamax\text{-}ab\text{-}result}(x, u, \alpha, \beta) = \mathsf{is\text{-}ab\text{-}result}(x, \mathsf{negamax}(u), \alpha, \beta) \tag{6}$$

where the predicate $\mathsf{is\text{-}ab\text{-}result}$, adapted from Fishburn [3], specifies when a value x is an acceptable result for a true minimax or negamax value e under the

window (α, β). The three cases correspond to returning an upper bound ($e \leq x$), an exact value ($e = x$), or a lower bound ($x \leq e$), respectively. To support reasoning about partial computations, we also define

$$\mathsf{is\text{-}partial\text{-}negamax\text{-}ab\text{-}result}(x, u, i, \alpha, \beta) = \mathsf{is\text{-}ab\text{-}result}(x, \mathsf{pnm}(u, i), \alpha, \beta), \quad (7)$$

where $\mathsf{pnm}(u, i) = \max\{-\mathsf{negamax}(u.\mathit{children}[j]) \mid 0 \leq j < i\}$ denotes the partial negamax value obtained after evaluating the first i children of u. This notion is used to express loop invariants and intermediate correctness conditions.

2.2 Transposition Tables

A transposition table (TT) is a cache used to avoid repeated computations in game-tree search by storing results of previously visited states. Formally, it represents a partial mapping from states (nodes) to search results. We define a table entry as follows:

$$\text{TableEntry} = \textbf{record} \begin{cases} \mathit{value} & \text{estimated minimax value,} \\ \mathit{depth} & \text{search depth at evaluation time,} \\ \mathit{flag} & \text{value classification (see below).} \end{cases}$$

The *flag* attribute identifies the relationship between the stored *value* and an exact minimax or negamax value of the state:

$$\begin{cases} \mathsf{exact} \ \ (\mathsf{EX}) & \mathit{value} \text{ is an exact value,} \\ \mathsf{lowerbound} \ \ (\mathsf{LB}) & \mathit{value} \text{ is a lower bound,} \\ \mathsf{upperbound} \ \ (\mathsf{UB}) & \mathit{value} \text{ is an upper bound.} \end{cases}$$

Despite their widespread use, the formal semantics of transposition tables is typically underspecified. While the flag indicates how the stored value relates to an exact minimax value, it does not specify *with respect to which game tree* this exact value is defined. In particular, table entries may be produced by searches performed at different depths, under different alpha-beta windows, and in different parts of the game tree. As a result, the meaning of a table entry cannot be understood in isolation: its validity depends implicitly on a search context that is not recorded in the table itself. This lack of a precise semantics complicates formal reasoning about the correctness of transposition-table-based search.

In this work, we intentionally restrict the contents of the transposition table to values, depths, and flags, and do not store best moves. Best moves are commonly used to reorder successor nodes and thereby improve search efficiency, but they do not affect the correctness of returned values. Since our focus is on establishing value correctness rather than on performance characteristics, we exclude best-move information from the formal model.

3 Algorithm Pseudocode

This section introduces a selection of the game tree search algorithms that we have formally verified using Dafny. The classical Minimax algorithm (Algorithm 1) recursively evaluates a game tree to determine the optimal value at the

Algorithm 1 Minimax

Input: A game tree u.
Output: $\mathsf{minimax}(u)$.

```
MINIMAX(u):
 1: if |u.children| = 0 then
 2:   return u.eval
 3: else if u.color = −1 then
 4:   value := ∞
 5:   for v ∈ u.children do
 6:     value := min(value, MINIMAX(v))
 7:   return value
 8: else
 9:   value := −∞
10:   for v ∈ u.children do
11:     value := max(value, MINIMAX(v))
12:   return value
```

Algorithm 2 Negamax

Input: A game tree u.
Output: $\mathsf{negamax}(u)$.

```
NEGAMAX(u):
1: if |u.children| = 0 then
2:   return u.color · u.eval
3: else
4:   value := −∞
5:   for v ∈ u.children do
6:     value := max(value, −NEGAMAX(v))
7:   return value
```

root under perfect play. Each internal node loops over its children to compute either the minimum or maximum value, depending on the player to move. Instead of handling separate cases for minimizing and maximizing nodes NEGAMAX (Algorithm 2), is a reformulation of minimax that exploits symmetry between the players, using negation to unify both into a single recursive structure. Negamax simplifies implementation, making it the de facto standard. We further consider two negamax algorithms that combine depth-limited search, alpha-beta pruning, and transposition tables (TT), see Algorithm 3. The first, which we refer to as NEGAMAXTTW, is a variant commonly described in community sources such as Wikipedia (W), though its precise origin is unclear and likely influenced by work such as [7, 1]. The second, which we refer to as NEGAMAXTTM, is a related algorithm from Marsland (M) [7] that uses a different transposition table strategy. Special handling of the best move has been removed from NEGAMAXTTM. Both algorithms consist of three distinct phases: a table lookup, a negamax search and a table update. In Section 4.4 the differences between them will be discussed.

4 Verification using Dafny

The modeling of data types in Dafny is straightforward, see Listing 1. While in the pseudocode we used the value ∞, in our Dafny modeling we use a bounded integer type, where `INFINITY` is an unspecified positive integer constant. This is also common in practical implementations. The verifier automatically checks that variables remain within the allowed domain. Note that inductive datatypes in Dafny, such as `Node`, are free from cycles by construction. To verify correctness,

Algorithm 3 Transposition-table-based Negamax search variants. Algorithm NegamaxTTW follows Wikipedia [13], while Algorithm NegamaxTTM follows Marsland [7]. Both algorithms take as input a node u, an alpha-beta window (α, β), a search depth d, and a transposition table T. Both algorithms are subdivided in three phases: Table Lookup, Negamax Search, and Table Update.

```
function NegamaxTTW(u, α, β, d, T):
    α₀ := α                                                          ▷ Table Lookup
    if u ∈ T then
        t := T[u]
        if t.depth ≥ d then
            if t.flag = exact then return t.value
            else if t.flag = lowerbound ∧ t.value ≥ β then return t.value
            else if t.flag = upperbound ∧ t.value ≤ α then return t.value
    if d = 0 ∨ |u.children| = 0 then return u.color · u.eval         ▷ Negamax Search
    value := −∞
    for v ∈ u.children do
        value := max(value, −NegamaxTTW(v, −β, −α, d − 1, T))
        α := max(α, value)
        if α ≥ β then break
    if value ≤ α₀ then T[u] := (value, d, upperbound)                ▷ Table Update
    else if value ≥ β then T[u] := (value, d, lowerbound)
    else T[u] := (value, d, exact)
    return value
```

```
function NegamaxTTM(u, α, β, depth, T):
    if u ∈ T then                                                    ▷ Table Lookup
        t := T[u]
        if t.depth ≥ d then
            if t.flag = exact then return t.value
            else if t.flag = lowerbound then α := max(α, t.value)
            else if t.flag = upperbound then β := min(β, t.value)
            if α ≥ β then return t.value
    if d = 0 ∨ |u.children| = 0 then return u.color · u.eval         ▷ Negamax Search
    value := −∞
    for v ∈ u.children do
        value := max(value, −NegamaxTTM(v, −β, − max(α, value), d − 1, T))
        if value ≥ β then break
    flag := exact                                                    ▷ Table Update
    if value ≤ α then flag := upperbound
    if value ≥ β then flag := lowerbound
    if u ∉ T ∨ T[u].depth ≤ d then T[u] := (value, d, flag)
    return value
```

Listing 1 The datatypes used in Dafny

```
type bounded_int = x:int | -INFINITY <= x <= INFINITY
datatype Color = White | Black
datatype Node = Node_(eval:bounded_int, color:Color, children:seq<Node>)
datatype Flag = Lowerbound | Upperbound | Exact
datatype TableEntry = TableEntry_(depth:nat, value:bounded_int, flag:Flag)
type TranspositionTable = map<Node, TableEntry>
```

we translated each algorithm from pseudocode into Dafny syntax. For example, NegamaxTTW in Algorithm 3 corresponds to Dafny Listing 2.

4.1 Correctness of Transposition Table Search

For depth-limited transposition-table-based algorithms NegamaxTT[W,M], the returned results may depend on values computed earlier in unrelated parts of the game tree, potentially at greater depth than required at the current node. Effectively, the algorithm explores an *expansion* of the depth-d truncated tree by greedily reusing deeper search results via the transposition table. Figure 2(c) illustrates this behavior: the primary search unfolds as a triangle-shaped subtree rooted at the current node. Upon a transposition table hit, the algorithm may reuse a value previously computed in a deeper search of a different subtree. So not only the nodes in the depth-d truncated tree contribute to the search result, but also recursively the nodes in these (reused) subtrees. This motivates our notion of a *witness*, a concrete expansion of the truncated tree that merges all such reused subtrees into a single game tree. Such an expansion u' can be obtained from u by removing all children of some nodes in u at depth at least d. Correctness is then characterized by the existence of a witness, an expansion whose minimax (or negamax) value equals the value returned by the algorithm. To formalize this notion, we impose four conditions on a witness:

1. The witness u' is a subtree of the original tree u.
2. The witness contains the complete depth-d truncation of u: $\lfloor u \rfloor_d = \lfloor u' \rfloor_d$.
3. For any node z' in u' corresponding to z in u, either *all* children of z are included in u', or *none*.
4. The algorithm's returned value matches the negamax alpha-beta result on the witness u', as defined in (6).

Condition 3 is crucial: it prevents a witness from including arbitrary subsets of a node's children, which would produce subtrees that do not correspond to any valid minimax or negamax search. Conditions 1-3 are formalized in the predicate is-expansion. Let $C = u.children$ and $C' = u'.children$:

$$\textsf{is-expansion}(u', u, d) \;=\; u.\text{eval} = u'.\text{eval} \;\wedge\; u.\text{color} = u'.\text{color} \;\wedge \tag{8}$$

$$\begin{cases} (|C| = |C'| \;\wedge\; \forall\, i \in [0, |C|) :\; \textsf{is-expansion}(C'_i, C_i, d-1))\,, & \text{if } d > 0 \\ (|C'| = 0 \;\vee\; (|C| = |C'| \;\wedge\; \forall\, i \in [0, |C|) :\; \textsf{is-expansion}(C'_i, C_i, 0)))\,, & \text{if } d = 0 \end{cases}$$

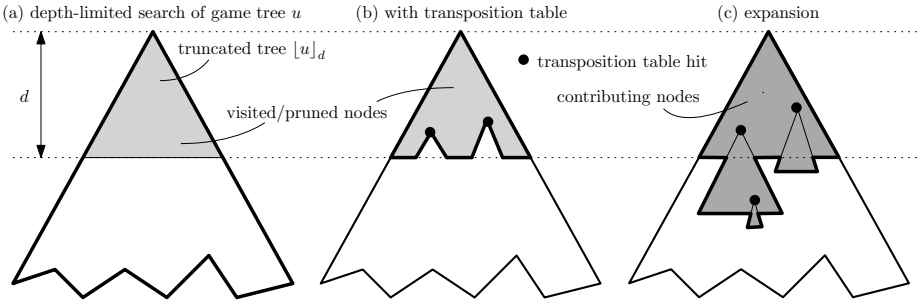


Fig. 2: Schematic truncated game tree showing the visited/pruned nodes for depth-d searching without (a) and with (b) transposition table. The expansion (c) with all nodes directly and indirectly contributing to the table-based search.

The first case ensures that the truncated tree $\lfloor u \rfloor_d$ is included, while the second enforces the "all-or-none children" requirement of Condition 3.

Correctness of the algorithm is then defined as:

$$\begin{aligned} &\mathsf{is\text{-}negamax\text{-}tt\text{-}result}(\mathit{result}, u, \alpha, \beta, \mathit{depth}) = \\ &\quad \exists u' : \ \mathsf{is\text{-}expansion}(u', u, \mathit{depth}) \wedge \mathsf{is\text{-}negamax\text{-}ab\text{-}result}(\mathit{result}, u', \alpha, \beta). \end{aligned} \tag{9}$$

Moreover, for each entry t in the transposition table, a witness must exist justifying that entry:

$$\begin{aligned} \mathsf{is\text{-}valid\text{-}table\text{-}entry}(t, u) = \exists u' : \mathsf{is\text{-}expansion}(u', u, t.\mathit{depth}) \wedge \\ \begin{cases} t.\mathit{value} \geq \mathsf{negamax}(u'), & \text{if } t.\mathit{flag} = \mathsf{upperbound} \\ t.\mathit{value} = \mathsf{negamax}(u'), & \text{if } t.\mathit{flag} = \mathsf{exact} \\ t.\mathit{value} \leq \mathsf{negamax}(u'), & \text{if } t.\mathit{flag} = \mathsf{lowerbound} \end{cases} \end{aligned} \tag{10}$$

We adopt a permissive model of the transposition table. At every iteration of the algorithm, we allow the table's contents to change arbitrarily, as long as the entries satisfy (10). This reflects practical implementations, where transposition tables are typically realized using hash tables: insertions may overwrite existing entries due to collisions and replacement strategies.

Importantly, this model does not restrict how a witness is obtained. Any witness satisfying Conditions 1-3 can be justified compositionally by valid transposition table entries, independently of a specific execution trace. Intuitively, the witness may be viewed as being assembled incrementally from shallow (e.g., one-ply) table lookups, but the correctness argument relies only on the existence of such a witness, not on its constructive derivation.

4.2 Verification of NegamaxTTW

We formally verified the Wikipedia variant of transposition-table-based negamax search, NegamaxTTW, using Dafny against the correctness criteria defined in

Listing 2 Dafny verification of NegamaxTTW

```
class NegamaxAlgorithm
{
  var T: TranspositionTable

  method Negamax(u: Node, alpha0: bounded_int, beta0: bounded_int, depth:nat)
    returns (result: bounded_int)
    modifies this`T
    requires alpha0 < beta0
    requires turn_based()
    requires is_valid_table(T)
    ensures is_negamax_tt_result(result, u, alpha0, beta0, depth)
    ensures is_valid_table(T)
  {
    var alpha, beta := alpha0, beta0;
    if u in T.Keys
    {
      var t := T[u];
      if t.depth >= depth && ((t.flag == Lowerbound && t.value >= beta) ||
          (t.flag == Exact) || (t.flag == Upperbound && t.value <= alpha))
      {
        TableLookupReturnLemma(u, alpha, beta, depth, t, T);
        return t.value;
      }
    }
    if depth == 0 || |u.children| == 0
    {
      DepthZeroReturnLemma(u, depth, alpha0, beta0);
      return color(u) * u.eval;
    }
    var value: bounded_int := -INFINITY;
    for i := 0 to |u.children|
      invariant is_valid_table(T)
      invariant 0 <= i <= |u.children|
      invariant i == 0 ==> value == -INFINITY && alpha == alpha0
      invariant alpha0 <= alpha < beta0
      invariant value <= alpha0 ==> alpha == alpha0
      invariant alpha0 < value < beta0 ==> value == alpha
      invariant i > 0 ==> exists u': Node :: is_expansion(u', u, depth) &&
                    is_partial_negamax_ab_result(value, u', i, alpha0, beta0)
      invariant i == |u.children| ==>
              is_negamax_tt_result(value, u, alpha0, beta0, depth)
    {
      ghost var old_alpha := alpha;
      ghost var old_value := value;
      var v := u.children[i];
      var negamax_v := Negamax(v, -beta, -alpha, depth - 1);
      value := max(value, -negamax_v);
      alpha := max(alpha, value);
      if alpha >= beta
      {
        LoopBreakLemma(u, v, i, depth, alpha0, beta0, old_alpha, old_value,
                       alpha, value, negamax_v);
        break;
      }
      LoopMaintenanceLemma(u, v, i, depth, alpha0, beta0, old_alpha, old_value,
          alpha, value, negamax_v);
    }
    TableUpdateLemma(value, u, alpha0, beta0, depth, T);
    if value <= alpha0      {T := T[u:=TableEntry_(depth,value,Upperbound)];}
    else if alpha0 < value < beta0 {T:=T[u:=TableEntry_(depth,value,Exact)];}
    else if value >= beta0  {T := T[u:=TableEntry_(depth,value,Lowerbound)];}
    return value;
  }
}
```

Equations (9) and (10). The Dafny development for this algorithm comprises approximately 850 lines of code, and verification completes in about 5 seconds on an Intel Core i7-13700 system with 128 GiB RAM (with Dafny running a single verification thread). The core algorithm is shown in Listing 2.

The verification effort involved several nontrivial challenges. First, correctness had to be expressed in terms of witnesses rather than concrete execution traces, requiring the specification of structural properties of expanded game trees. Second, suitable loop invariants were needed to relate intermediate search results, transposition table contents, and partially constructed witnesses. Third, the verification required reasoning about the incremental assembly of a witness tree that satisfies the expansion constraints introduced in Section 2.

Dafny proved effective for establishing many basic properties largely automatically. In particular, the is-expansion predicate admits a rich collection of lemmas that were relatively straightforward to verify, including reflexivity and transitivity, preservation under subtree inclusion, inclusion of the depth-truncated tree, and preservation of the expansion property when a subtree is replaced by a witness justified by a transposition table lookup.

At the same time, the verification process exposed limitations of automated reasoning in the presence of deeply nested quantifiers, which arise naturally when formalizing game trees, expansions, and transposition-table invariants. Without careful structuring, the verifier would often time out or exhaust memory. To address this, we systematically separated proof obligations and discharged them via explicit lemma invocations rather than inline reasoning, particularly at return points, loop exits, and invariant maintenance boundaries. For NegamaxTTW, this strategy resulted in five dedicated lemma calls in the implementation (Listing 2), covering early returns after table lookups, the $depth = 0$ base case, loop maintenance and termination, and updates to the transposition table. This decomposition proved essential for extending the verification to realistic algorithm variants.

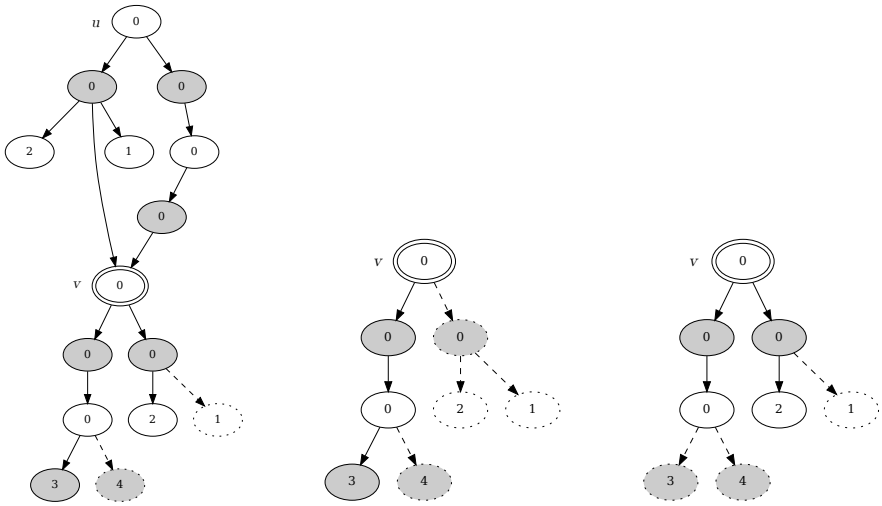
4.3 Verification of NegamaxTTM

Our attempt to verify Marsland's transposition-table algorithm NegamaxTTM was unsuccessful: for a specific configuration the Dafny proof could not establish the postcondition of is-negamax-tt-result. Analysis of that unprovable case revealed a concrete counterexample, showing that NegamaxTTM violates our witness-based correctness criterion.

The problematic pattern occurs when a transposition-table lookup raises the lower bound α beyond the eventual return value x while staying below the original upper bound β_0:

$$\alpha_0 < x \leq \alpha < \beta_0.$$

This suggests that the previously stored lower bound α may cause premature pruning, preventing the exploration of subtrees that would otherwise affect the minimax value.



(a) Root call on u: depth 6, window $(0, 5)$, returns 2 (exact)

(b) First call on v: depth 4, window $(0, 2)$, returns 3 (lower bound)

(c) Second call on v: depth 2, window $(0, 5)$, reuses (b)'s result and returns 2 (upper bound)

Fig. 3: Counterexample to NegamaxTTM. Node v (with double border) is searched twice. Call (b) stores a lower bound of 3 in the transposition table. Call (c) reuses this result in a shallower search with a wider window, leading to an unexpected return value. Dotted nodes are not visited during search.

Guided by this insight, we constructed the concrete instance shown in Figure 3(c). The algorithm is called on node v with window $(0, 5)$, depth 2, and the transposition table

$$T = \{v \mapsto (\mathit{value} = 3, \mathit{depth} = 4, \mathit{flag} = \mathsf{LB})\}.$$

On this input, NegamaxTTM returns 2. However, no valid witness expansion of $\lfloor v \rfloor_2$ yields this value: the only admissible expansions (the truncated tree itself and the full tree) have negamax alpha-beta results 1 and 4, respectively. The value 2 cannot be justified. The node with value 1 is never explored because the stored lower bound 3 shrinks the window to $(3, 5)$, causing an early cutoff.

$$\begin{aligned}
&\text{(a)}\ \textsc{NegamaxTTM}(u, \alpha = 0, \beta = 5, \mathit{depth} = 6, T = \emptyset) &&= 2 \quad (\mathsf{exact}) \qquad (11)\\
&\text{(b)}\ \textsc{NegamaxTTM}(v, \alpha = 0, \beta = 2, \mathit{depth} = 4, T) &&= 3 \quad (\mathsf{lowerbound})\\
&\text{(c)}\ \textsc{NegamaxTTM}(v, \alpha = 0, \beta = 5, \mathit{depth} = 2, T) &&= 2 \quad (\mathsf{upperbound})
\end{aligned}$$

To rule out that the counterexample relies on an artificially pre-filled table, we extend the run to start with an empty table (Figure 3(a)). The root call on u at depth 6, window $(0, 5)$, produces two recursive searches of v; the first

stores the lower bound 3, and the second reuses it, again returning 2 with no witness. The calls are summarized in Equation 11; a detailed execution trace is given in the supplementary material. The root cause is the reuse of a lower bound computed under a narrow window $(0, 2)$ in a wider window $(0, 5)$, leading to premature pruning. Thus, no witness exists for the returned value, and the algorithm violates our correctness criterion.

Moreover, the returned value is surprising even independently of the witness criterion. Under standard minimax semantics, a minimizing node should never return the value 2 when another child with value 1 lies within both the search horizon and the current alpha–beta window. Yet this is precisely the behavior exhibited by NegamaxTTM. It remains unclear whether there is a correctness notion for Marsland's algorithm that is both faithful to the algorithm's behavior and practically meaningful.

4.4 Differences between NegamaxTTW and NegamaxTTM

The algorithms NegamaxTTW and NegamaxTTM in Algorithm 3 both consist of three conceptual phases: a table lookup, a negamax search, and a table update. While structurally similar, the two algorithms differ in each phase:

Table Lookup: NegamaxTTM uses lower and upper bounds stored in the transposition-table to narrow the alpha-beta window before continuing the search, whereas NegamaxTTW uses them only for immediate returns.

Negamax Search: NegamaxTTM follows Fishburn's fail-soft negamax scheme [3], which differs in how intermediate bounds are propagated during recursion.

Table Update: NegamaxTTM uses the current value of α to classify search results, while NegamaxTTW uses the initial value of α for that. In addition, NegamaxTTM preserves existing table entries when their recorded search depth exceeds that of the newly computed result.

We analyzed the impact of each of these differences on correctness with respect to our witness-based criterion. The verification attempt for NegamaxTTM shows that the deviation in the *Table Lookup* phase is decisive: reusing bounds to shrink the search window can lead to results for which no admissible witness expansion exists, and therefore violates our correctness notion.

The remaining differences were studied in isolation by incorporating them into NegamaxTTW and re-verifying the resulting variants in Dafny. The additional depth condition in the *Table Update* phase does not affect correctness, since our transposition table model constrains only the validity of stored entries, not their presence or replacement policy.

Similarly, adopting Fishburn's fail-soft negamax variant in the *Negamax Search* phase does not compromise correctness. Proving this required adaptations to the loop maintenance and loop break lemmas, but no changes to the underlying correctness argument.

In contrast, classifying results against the *current* value of α rather than the initial one turned out to be more subtle. Under this modification, the three cases

in the table-update are no longer mutually exclusive. Correctness could only be re-established by imposing a different ordering on these cases, ensuring that the correct flag (lower bound, exact or upper bound) is assigned to the result.

4.5 Verification of NegamaxTTP

Plaat et al. [9] present a transposition-table algorithm in minimax form for depth-unlimited search. To make it comparable with the other verified algorithms in our development, we first reformulated it as a negamax algorithm and extended it with explicit depth-limited search, yielding the variant NegamaxTTP. These transformations preserve the essential transposition-table interface while bringing the algorithm into our paper's framework.

Compared with NegamaxTTW, NegamaxTTP differs in two principal respects. First, following Plaat et al., transposition-table entries store explicit lower and upper bounds instead of a single value annotated with an exact/lower/upper flag. Second, table lookups refine the current search window, while recursive calls retain the original upper search bound rather than the dynamically updated one used by NegamaxTTW. Despite these differences in table representation and search discipline, the witness-based correctness criterion remains applicable.

The verification of NegamaxTTP reuses much of the proof infrastructure developed for NegamaxTTW. In particular, many of the supporting lemmas and loop invariants could be adapted with only local modifications to account for the different transposition-table interface and recursive search discipline.

The resulting Dafny development establishes that NegamaxTTP satisfies the witness-based correctness criterion. Together with the verification of NegamaxTTW, this demonstrates that the proposed framework transfers naturally to non-trivial variations in transposition-table design, while the contrasting result for NegamaxTTM highlights that such variations are not merely syntactic but can affect semantic correctness.

5 Results

We applied the witness-based correctness criterion to three representative depth-limited transposition-table algorithms. Both NegamaxTTW and NegamaxTTP were successfully verified in Dafny, providing, to our knowledge, the first mechanically verified depth-limited negamax algorithms combining alpha-beta pruning with transposition tables. In contrast, Marsland's algorithm (NegamaxTTM, with best-move tracking omitted) does not satisfy the criterion. The failed verification led to a concrete counterexample demonstrating that the algorithm may return a value for which no valid witness expansion exists. This identifies a semantic mismatch between Marsland's algorithm and the witness-based correctness criterion. Whether alternative correctness notions should admit such behaviour remains an interesting direction for future work.

Table 1: Structure and verification times (seconds) of the Dafny verification artefact, for Dafny 4.10 and Dafny 4.11.

	Structure			Time (s)	
File	**LoC**	**Lemmas**	**Algorithms**	**4.10**	**4.11**
`definitions.dfy`	205	1	0	0.87	0.87
`lemmas.dfy`	206	20	0	1.89	1.55
`minimax-alpha-beta.dfy`	301	6	3	3.74	2.47
`minimax.dfy`	67	1	1	0.84	0.83
`negamax-alpha-beta.dfy`	584	15	7	5.00	4.81
`negamax.dfy`	56	1	1	0.90	0.87
`negamax-ttp.dfy`	1045	28	4	5.71	8.07
`negamax-ttw.dfy`†	391	15	1	2.15	2.92
`negamax-ttw-variants.dfy`†	340	5	3	2.16	2.30

†Under Dafny 4.11, `negamax-ttw{,-variants}.dfy` required additional supporting modules (sharing code via `import opened` instead of `refines`) to work around an internal tool error; this accounts for the modest increase in LoC, while the Lemmas/Algorithms counts, and the underlying proofs, are unchanged.

5.1 Verification artefact and statistics.

The complete Dafny development, together with executable Python implementations and randomized tests, is available on GitHub [12]. Table 1 summarizes the structure of the artefact and its verification times under Dafny 4.10 and Dafny 4.11. The artefact consists of a library of definitions and lemmas together with a family of verified minimax and negamax implementations, including depth-limited variants and selected transposition-table algorithms.

Most lemmas require little manual guidance beyond their specification and are discharged automatically by Dafny's SMT backend. The main exceptions involve quantified properties of the witness relation and alpha–beta correctness, which are handled by decomposing larger proofs into reusable helper lemmas. This modular proof structure improves solver performance and makes proofs more robust to implementation refactorings by explicitly documenting proof obligations. The central proof obligation is the loop-maintenance lemma for NegamaxTT[W,P], which establishes that processing one additional child preserves the partial witness invariant maintained throughout the search. The proof combines the witness expansion obtained from the recursive call with the current partial expansion, showing that replacing a single child by its witness preserves the semantic interpretation of the explored subtree. Once all children have been processed, the partial invariant immediately yields the correctness of the completed search result.

The file `negamax-ttw-variants.dfy` contains three verified variants obtained by selectively incorporating design choices from Marsland's algorithm. Marsland's original algorithm (NegamaxTTM) itself is absent from the verified

artefact because it does not satisfy the witness-based correctness criterion; its failed verification directly motivated the counterexample of Section 4.3.

Overall, verification times remain modest, with all verified files discharging within a few seconds. The increase in verification time for certain files under Dafny 4.11 reflects changes in the verifier and, in one case, the need to restructure code to work around an internal error. Importantly, the verification effort scales with the complexity of the specification rather than with algorithmic depth or branching factor, reinforcing that the primary challenge lies in specification design rather than raw automation cost.

6 Conclusions

This work illustrates the value of formal verification in reasoning about correctness properties of game tree search algorithms, particularly when optimizations such as transposition tables are employed. Using Dafny, we verify a representative collection of classical game-tree search algorithms, including minimax, negamax, fail-hard and fail-soft alpha–beta pruning, depth-limited search, and three transposition-table formulations.

We introduce a general and intuitive witness-based correctness criterion for depth-limited search using transposition tables, applicable both with and without alpha-beta pruning. We demonstrate the breadth and discriminative power of this criterion by considering three distinct depth-limited negamax algorithms. First, we verify the Wikipedia-style variant NegamaxTTW. Second, we show that the criterion extends beyond a single implementation by successfully verifying NegamaxTTP, a variant derived from the work of Plaat et al. that uses a different transposition-table representation and recursive search discipline. Third, we show that the criterion is discerning enough to reject an established algorithm: the closely related Marsland variant NegamaxTTM fails to satisfy the criterion, and we present a concrete counterexample demonstrating its semantic unsoundness under our model. This contrast highlights how subtle algorithmic differences can invalidate seemingly plausible correctness arguments, underscoring the necessity of precise semantic models when verifying optimized search techniques. More broadly, our work illustrates Dafny's suitability for verifying recursive, stateful algorithms with nontrivial invariants, as commonly found in game tree search.

A methodological lesson is that explicit lemma decomposition is essential for scaling Dafny verification to algorithms with nested quantifiers and deeply recursive state, as the SMT backend often times out when reasoning about large monolithic proof obligations.

Several directions for future work emerge from this study. A natural next step is to extend the witness-based framework to other search algorithms that rely intrinsically on the persistence of transposition table state, such as SSS^* [10, 9] and MTD(f) [9]. Another goal is to identify if a correctness notion exists for NegamaxTTM that faithfully captures its behavior while remaining practically meaningful.

References

1. Breuker, D.: Memory versus Search in Games. Ph.d. thesis, Maastricht University, Maastricht (1998). https://doi.org/10.26481/dis.19981016db
2. Escobar, D., Insuasti, J.: Formal verification of multi-thread minimax behavior using mcrl2 in the connect 4. Mathematics **13**(1), 96 (2024)
3. Fishburn, J.P.: Another optimization of alpha-beta search. SIGART Newsl. **84**, 37–38 (1983). https://doi.org/10.1145/1056623.1056628
4. He, Y., Saffidine, A., Thielscher, M.: Verification of general games with QBF solvers. In: Rapberger, A., Rudolph, S. (eds.) Proceedings of the 23rd International Workshop on Non-Monotonic Reasoning (NMR 2025) co-located with the 22nd International Conference on Principles of Knowledge Representation and Reasoning (KR 2025), Melbourne, Australia, November 11-13, 2025. pp. 142–156. CEUR Workshop Proceedings, CEUR-WS.org (2025), https://ceur-ws.org/Vol-4071/paper11.pdf
5. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artif. Intell. **6**(4), 293–326 (1975). https://doi.org/10.1016/0004-3702(75)90019-3
6. Leino, R.: Dafny: An automatic program verifier for functional correctness. In: 16th International Conference, LPAR-16, Dakar, Senegal. pp. 348–370. Springer Berlin Heidelberg (April 2010). https://doi.org/10.1007/978-3-642-17511-4_20
7. Marsland, T.A.: A review of game-tree pruning. ICCA Journal **9**(1), 3–19 (1986). https://doi.org/10.3233/ICG-1986-9102
8. Nipkow, T. (ed.): Functional Data Structures and Algorithms: A Proof Assistant Approach, vol. 67. Association for Computing Machinery, New York, NY, USA, 1 edn. (2025)
9. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Best-first fixed-depth minimax algorithms. Artif. Intell. **87**(1-2), 255–293 (1996). https://doi.org/10.1016/0004-3702(95)00126-3
10. Stockman, G.C.: A minimax algorithm better than alpha-beta? Artif. Intell. **12**(2), 179–196 (1979). https://doi.org/10.1016/0004-3702(79)90016-X
11. Warnock, T., Wendroff, B.: Search tables in computer chess. ICCA J. **11**(1), 10–13 (1988). https://doi.org/10.3233/ICG-1988-11103
12. Wesselink, W.: Minimax verification. https://github.com/wiegerw/minimax (2025), accessed: 2026-01-25
13. Wikipedia contributors: Negamax — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Negamax (2025), accessed: 2025-06-09
14. Wimmer, S., Hu, S., Nipkow, T.: Verified memoization and dynamic programming. In: International Conference on Interactive Theorem Proving. pp. 579–596. Springer (2018)

A Execution of the Minimal Counterexample

This appendix gives the detailed execution traces underlying the counterexample of Section 4.3. It first analyzes the minimal counterexample for NegamaxTTM, then compares it with the corresponding execution of NegamaxTTW. Both algorithms are called with identical inputs:

$$\texttt{NegamaxTT[W,M]}(v, \alpha = 0, \beta = 5, \mathit{depth} = 2, T),$$

where $T = \{v \mapsto (\mathit{value} = 3, \mathit{depth} = 4, \mathit{flag} = \mathsf{LB})\}$.

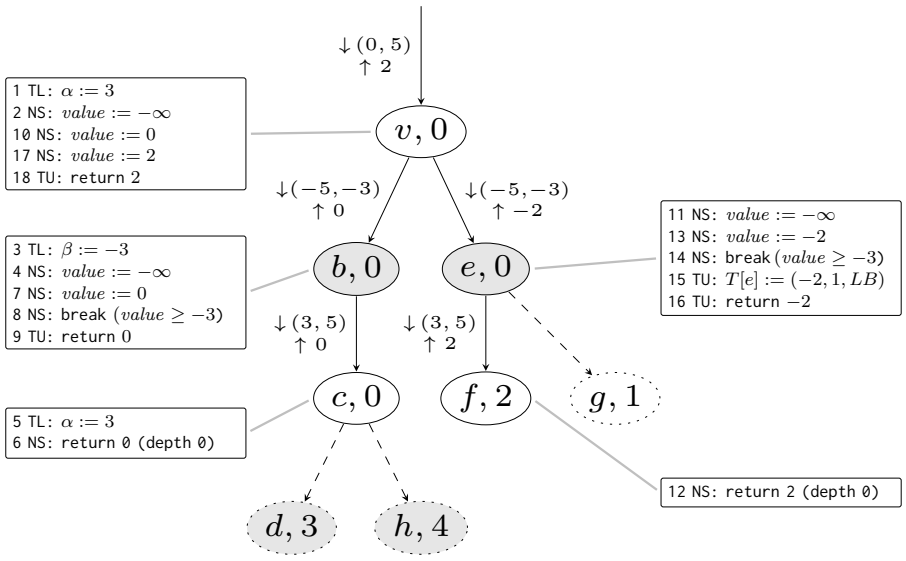


Fig. 4: Execution trace of NegamaxTTM on the minimal counterexample from Figure 3(c), with transposition table $\{v \mapsto (\mathit{value} = 3, \mathit{depth} = 4, \mathit{flag} = \mathsf{LB})\}$. Nodes contain an identifier and an evaluation value. Rectangular blocks show statements executed at each node, labeled by algorithmic phase (TL/NS/TU) and global execution order. Edge labels show the alpha-beta interval passed downward (↓) and the value returned upward (↑).

A.1 NegamaxTTM

Figure 4 shows the recursive depth-first search of NegamaxTTM through nodes v, b, c, e, and f. For each node, a text block lists the relevant statements executed during its search, prefixed with their global execution order. The statements are labeled by phase: **TL** (Table Lookup), **NS** (Negamax Search), and **TU** (Table Update). Edges are labeled with two pieces of information:

- $\downarrow (\alpha, \beta)$: the alpha-beta window passed to the child, after the standard negation and swapping of bounds $(-\beta, -\alpha)$,
- $\uparrow$ *value*: the value returned by the child.

Dashed edges and nodes (e.g., d, g, h) are not visited during this search. The key steps that lead to the problematic return value 2 are:

1. At the root v, the table lookup increases α from 0 to 3 (step 1). This narrowing of the window is caused by the stored lower bound for v.
2. When searching child b, the window becomes $(-5, -3)$ after negation. Its child c returns its own value 0 (step 6), since it is searched at depth 0. The search of b propagates this value 0 (step 9) to the root v (step 10).
3. Child e is searched with window $(-5, -3)$. Its leaf child f returns 2, which becomes -2 after negation (step 13). Since $-2 \geq -3$, a beta-cutoff occurs (step 14), skipping the better child g with value 1. The value -2 is stored in the TT (step 15) and propagated back to the root v (step 16). After negation (step 17) the final return value becomes 2.

The stored lower bound 3 for v thus causes a premature cutoff at e. Consequently, the algorithm never visits the subtree that would reveal the better minimax value 1 available at depth 2. The return value 2 cannot be justified by any witness expansion of $\lfloor v \rfloor_2$, confirming the violation of our correctness criterion.

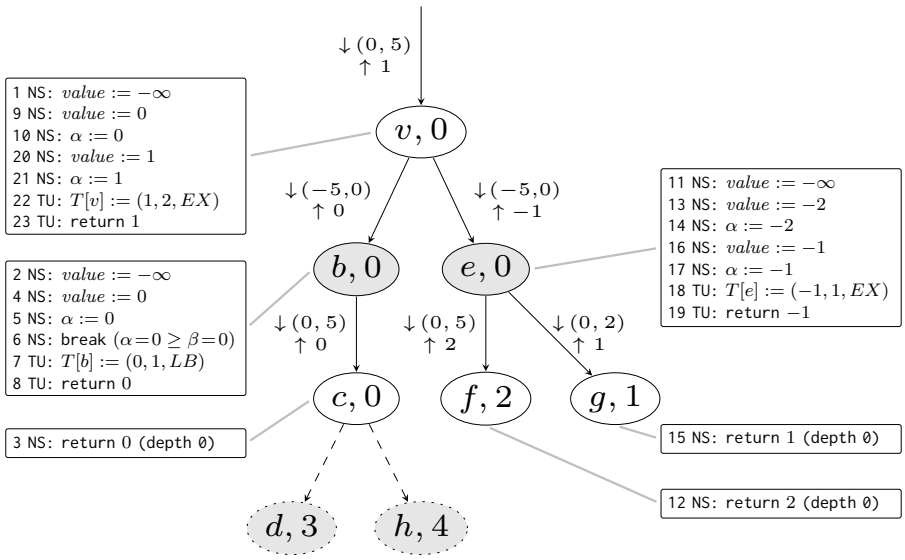


Fig. 5: Execution trace of NegamaxTTW on the minimal counterexample from Figure 3(c), with transposition table $\{v \mapsto (value = 3, depth = 4, flag = \mathsf{LB})\}$.

A.2 NegamaxTTW

Figure 5 illustrates the execution of NegamaxTTW. The table lookup at v also finds the stored lower bound 3, but the check $t.value \geq \beta$ (i.e., $3 \geq 5$) fails, so the lookup does *not* cause an early return. Consequently, the search proceeds with the original window $(0, 5)$, explores all relevant children, and returns the expected negamax value 1. The stored entry 3 is effectively ignored because it does not guarantee a cutoff in the current, wider window.

A.3 Comparison

The two traces highlight the semantic difference between the table-lookup strategies. NegamaxTTM uses stored bounds to narrow the search window, which can lead to incorrect pruning when a bound computed under a narrow window is reused in a wider one. NegamaxTTW only accepts a stored value if it immediately guarantees a cutoff; otherwise it discards the entry and performs a full search. This more conservative policy preserves the existence of a witness tree, explaining why NegamaxTTW satisfies our correctness criterion while NegamaxTTM does not.