Artificial Intelligence 283 (2020) 103262

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

On pruning search trees of impartial games

Piotr Beling *, Marek Rogalski

Faculty of Mathematics and Computer Science, University of Łódź, Banacha 22, Łódź 90-238, Poland

ARTICLE INFO

Article history:
Received 16 July 2018
Received in revised form 8 October 2019
Accepted 18 March 2020
Available online 24 March 2020

Keywords:
Combinatorial game theory
Game tree
Sprague-Grundy value
Nimber
Mex function
Impartial game
Nim
Chomp
Cram

ABSTRACT

In this paper we study computing Sprague-Grundy values for short impartial games under the normal play convention. We put forward new game-agnostic methods for effective pruning search trees of impartial games. These algorithms are inspired by the α-β, a well-known pruning method for minimax trees. However, our algorithms prune trees whose node values are assigned by the mex function instead of min/max.
We have verified the effectiveness of our algorithms experimentally on instances of some standard impartial games (that is Nim, Chomp, and Cram). Based on the results of our experiments we have concluded that: (1) our methods generally perform well, especially when transposition table can store only a small fraction of all game positions (which is typical when larger games are concerned); (2) one of our algorithms constitutes a more universal alternative to the state-of-the-art algorithm proposed by Lemoine and Viennot.

1. Introduction

We wish to present a new algorithm for calculating the Sprague-Grundy values. We calculate this value only for short impartial games under the normal play convention, that is combinatorial games which are finite (with a finite number of positions), non-loopy (each run of a game is finite), impartial (both players have the same moves in any position), and with the normal play convention (the last player able to move is the winner).

A game of this type can be defined as a tuple $(\mathbb{S}, s_i, N)$, where:

- $\mathbb{S}$ is a finite set of legal game states (positions);
- $s_i \in \mathbb{S}$ is the initial position from which the game begins;
- $N : \mathbb{S} \to 2^{\mathbb{S}}$ is a function which defines the successors of a given position, $q \in N(p)$ means that q can be reached from p in one move.
  N must satisfy the condition: There is no sequence of states $p_0, p_1, \ldots p_n$ with $n > 0$, $p_i \in N(p_{i-1})$ (for each $i \in \{1, \ldots, n\}$) and $p_0 = p_n$. In other words, there are no "cycles" that return to an identical position.

A game starts at a position s_i, and players make moves alternately. If the game reaches a position $s \in \mathbb{S}$ such that $N(s) = \emptyset$, then the game ends and the player to move loses.

* Corresponding author.
E-mail addresses: piotr.beling@wmii.uni.lodz.pl (P. Beling), marek.rogalski@wmii.uni.lodz.pl (M. Rogalski).

https://doi.org/10.1016/j.artint.2020.103262
0004-3702/

```
1  function is_losing(s):
2      for m ∈ N(s):
3          if is_losing(m): return false
4      return true
```

Listing 1: Returns true if and only if s is losing.

Many authors (for example [1, page 4]) give an equivalent definition of a game as a set of games, instead of using the successor function N. Such definition does not differentiate between a game and its initial position. Since this is comfortable in many contexts, we often equate them in our paper as well.

A more in-depth introduction to impartial games can be found in [2, pages 61–78]. Examples of rules of a few such games are included in the appendix.

From any given position there exists a winning strategy either for the player to move or for their opponent. If the winning strategy is for the player to move, we will say that the position is winning. In the other case, we will say that the position is losing. It is easy to prove, that a position is losing if and only if it does not have any losing successors. The algorithm in Listing 1 directly implements this idea.

Sprague [3] and Grundy [4] independently developed a theory that allows us to efficiently identify a position as winning or losing if it is divisible into a set of disjoint positions. The theory divides the set $\mathbb{S}$ into equivalence classes by assigning the so-called *Sprague-Grundy value* to each position. The Sprague-Grundy value is also called a nimber, since this value for a Nim-heap of n tokens is n.

For every position $s \in \mathbb{S}$, the Sprague-Grundy value is defined by:

$$G(s) = \operatorname{mex}(\{G(t) : t \in N(s)\}), \tag{1}$$

where $\operatorname{mex}(X)$ is the minimum excluded value from a set X — the least non-negative integer not in X. (The above definition is adopted from [1, page 54], and the algorithm which is based on it is shown at Listing 3 and discussed in detail in Section 2.)

Note that calculating this value directly by definition may be computationally intensive. For example, computation tree for Cram with a board 4×4 has 6,257,129 nodes. Increasing the board size to 5×4 scales its up nearly 285 times, up to 1,780,847,574 nodes. Since standard techniques such as board symmetry or transposition tables help only up to a certain point, new methods of pruning the search tree are desirable.

There are two reasons to calculate the Sprague-Grundy value. Firstly, this value correctly identifies the outcome of optimal play. A position is losing if and only if its value is zero.

Secondly, the Sprague-Grundy value of decomposable positions can be computed by combining the values of the disjoint positions, by the following formula:

$$G(s) = G(s_1) \oplus G(s_2) \oplus \cdots \oplus G(s_k), \tag{2}$$

where:

- $\oplus$ is the nim-sum, which operates like an exclusive or (xor) operation on binary integers [5, page 74][4],
- s consists of disjoint positions $s_1, s_2, \ldots, s_k$. To make a move in s, a player first has to chose one of $s_1, s_2, \ldots, s_k$, and then make a move in it.

Decomposable positions arise naturally in certain games, such as Nim or Cram. The formula (2) gives a way to compute the Sprague-Grundy value of such a position usually faster than by Definition (1), since smaller components are considered independently. Authors of [6] prove that even in order to compute only the outcome of a sum of positions, it is always more efficient to 'compute separately the nimber of at least one of the independent positions, rather than to develop directly the game tree of the sum'.

Furthermore, an analysis of components of a decomposable position is useful also in partisan games, which constitute a superset of impartial games (where each player can have a different set of moves in any position). For instance, an application of this approach to Go 'has demonstrated perfect play in long endgame problems, which far exceed the capabilities of conventional game tree search methods' [7,8]. However, in partisan games, the analysis of each component separately is not always enough to determine optimal move in their sum. Then their combinatorial summation might provide the move, but it may also lead to a combinatorial explosion, becoming impractical. Even then the analysis of components can be used in an $\alpha - \beta$ search to prune moves, order moves or heuristically evaluate positions [9].

Some games (like Nim [10,4] or Kayles [11]) have been solved analytically. Such analyses usually depend on the form of each game, rather than on general ideas in the whole field of short impartial games. To give an example, Chomp $n \times n$ and $2 \times n$ or Cram $2n \times 2n$ can be easily solved through analytical methods, but despite thorough research, a general solution cannot be found for both games – and finding a solution for Chomp potentially gives us very little insight into Cram (and vice versa). That is why there are so few game-agnostic results, and why computer analysis is such a popular theme in this field.

```
1  function lv(s):
2      for n ← 0, 1, ...:
3          if is_losing(s+⟨n⟩_nim): return n
```

Listing 2: Returns Sprague-Grundy value of the state s. Algorithm proposed by Lemoine and Viennot.

One of the most important examples is the Glop project [12], authored by Lemoine and Viennot. Glop is a software developed to compute the winning strategies of combinatorial games. It is able to compute the winning strategies of three games: Sprouts, Cram, and Dots and Boxes.

The authors of Glop [12] have also written several papers on the topic of computing Sprague-Grundy values [6,13], as well as misère variants of Sprouts [14]. They proposed the algorithm for computing the Sprague-Grundy value, which is shown in Listing 2. It sequentially tests if a given position s has the value $n = 0, 1, \ldots$. Each test consists of checking if a game $s + \langle n \rangle_{\text{nim}}$ is losing, where $s + \langle n \rangle_{\text{nim}}$ is a game composed of s and a Nim-heap of size n (in this game, player can move either in s or $\langle n \rangle_{\text{nim}}$: $N(s + \langle n \rangle_{\text{nim}}) = \{m + \langle n \rangle_{\text{nim}} : m \in N(s)\} \cup \{s + \langle i \rangle_{\text{nim}} : i \in \{0, 1, \ldots, n-1\}\}$). The algorithm is based on the following:

$$\begin{aligned} s + \langle n \rangle_{\text{nim}} \text{ is losing} &\Leftrightarrow G(s + \langle n \rangle_{\text{nim}}) = 0 \\ &\Leftrightarrow G(s) \oplus G(\langle n \rangle_{\text{nim}}) = 0 \qquad \text{(by (2))} \\ &\Leftrightarrow G(s) = n. \qquad \text{(since } G(\langle n \rangle_{\text{nim}}) = n\text{)} \end{aligned} \tag{3}$$

Note that performance of this algorithm strongly depends on the order in which `is_losing` iterates over `N(s)`. It is easy to see that in order to prove that a position is winning, it is enough to find its one losing successor. So, for the algorithm at Listing 1 to perform well, the losing successor of each winning position needs to be considered as early as possible. But this algorithm is used by `lv` to analyze positions of type $s + \langle n \rangle_{\text{nim}}$. At the same time, a position $s + \langle n \rangle_{\text{nim}}$ is losing only when $G(s) = n$, which follows from (3). So heuristically checking if $s + \langle n \rangle_{\text{nim}}$ is losing requires guessing $G(s)$, which is generally very difficult (harder than guessing if s itself is losing) and require very deep game-specific knowledge. It is also beneficial (and easier to implement) to consider as first moves which lead to possibly small subtrees (which are generally easier to compute) in order to increase the chance of pruning larger branches of the search tree.

Just like `lv`, the algorithms presented in the next section are game-agnostic (and they can use game-specific knowledge to arrange successors as well). It means that they can be applied to virtually any short impartial game. Other examples of game agnostic techniques include transposition tables or endgame databases (successfully applied to, for example, Cram [15–17]). This stands in contrast to game-specific algorithms, for instance using rotational symmetries in Cram to equate some moves. Most of the work done in this field exploits unique features of games in order to solve it, or at least compute larger boards. Examples of these kinds of papers include calculating the Sprague-Grundy values in: Euclid [18,19], Wythoff's game [20], Nimhoff Games [21], Grundy's game [22,11], two-dimensional generalization of Grundy's game [23], and periodicity of nim-values for one-dimensional duotaire [24]. Aside from calculating the Sprague-Grundy values for games, there are also papers that describe calculating winning or losing positions only: in Chomp [25,26], Raleigh [27], Nimbi [28], Hexad [29], and Pentominoes [30]. Papers [16,17] present an efficient solver for Cram, which combines combinatorial game theory with $\alpha - \beta$ pruning, a transposition table, and Enhanced Transposition Cutoff (ETC). The solver uses an endgame database filled with nimbers, and heuristically orders moves to quickly decompose a board and use values from the database.

Our contributions are presented in the next sections. In Section 2 we put forward several new algorithms for pruning search trees of impartial games. In Section 3 we show how our algorithms can be extended to use the Sprague-Grundy theorem for decomposable positions. In Section 4 we discuss some details of implementation of our methods as well as the one proposed by Lemoine and Viennot. We measured the effectiveness of all these methods. The results are presented and analyzed in Section 5. Finally, Section 6 contains conclusions and suggestions for further research.

2. The algorithms

In this section we show different algorithms for calculating Sprague-Grundy value of a given position. We present them in order of increasing complexity. We start from the simplest one (Listing 3), which is based on the definition of Sprague-Grundy value (1). Next, we discuss new elements introduced by each subsequent version.

In Listing 3, P is a set of potential Sprague-Grundy values of position s given as argument. The algorithm successively eliminates values from P. Exactly one value is excluded from P after visiting each successor of s in the loop at lines 5–10. If the value $v = \text{def}(m) = G(m)$ (see line 6) of the successor is included in P (see line 7) then, by the definition of mex, the value excluded is v (see line 8). If $v \notin P$, the maximal value is removed from P (at line 10). In such a case, the value of s cannot be equal to $\max(P)$ since not enough successors remain to exclude all values less than $\max(P)$ from P.

The size of P is always equal to one plus the number of successors of s remaining to visit. It has exactly $|N(s)| + 1$ elements at the beginning (at line 4) and 1 element after visiting all $|N(s)|$ successors. This element (see line 11) is the Sprague-Grundy value of s, and it is returned at line 13.

```
1  function def(s):
2      if (x, v) ∈ TT where x=s:
3          return v
4      P ← {0, 1, ..., |N(s)|}
5      for m ∈ N(s):
6          v ← def(m)
7          if v ∈ P:
8              P ← P \ {v}
9          else:
10             P ← P \ {max(P)}
11     result ← the only one element of P
12     TT[s] ← result
13     return result
```

Listing 3: Returns Sprague-Grundy value of the state s.

```
1  function scut(s, R):
2      if (x, v) ∈ TT where x=s:
3          return v
4      P ← {0, 1, ..., |N(s)|}
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← scut(m, P \ {max(P)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12     result ← the only one element of P
13     TT[s] ← result
14     return result
```

Listing 4: Mex tree search with simplified pruning. Returns the Sprague-Grundy value of the state s, or -1 (only if the value is not included in R).

```
1  function cut(s, R):
2      if (x, v) ∈ TT where x=s:
3          return v
4      P ← {0, 1, ..., |N(s)|}
5      exact ← true
6      for m ∈ N(s):
7          if P ∩ R = ∅: return −1
8          v ← cut(m, (P \ {max(P)}) ∩ {0, 1, ..., max(R)})
9          if v ∈ P:
10             P ← P \ {v}
11         else:
12             P ← P \ {max(P)}
13             if v = −1: exact ← false
14     result ← the only one element of P
15     if exact or result ≤ max(R):
16         TT[s] ← result
17         return result
18     else:
19         return −1
```

Listing 5: Mex tree search with pruning. Returns Sprague-Grundy value of the state s, or -1 (only if the value is not included in R).

Just before returning, the value of s is added to TT at line 12. TT is a transposition table. It stores Sprague-Grundy values of positions that have already been evaluated in order to avoid redundant calculations when many different sequences of moves lead to the same position. The value of position s is immediately returned (at line 3) if it is in TT (see line 2). Transposition table is an optional part of the algorithms in Listings 3, 4 and 5. It can be dropped by removing lines 2, 3, 12 from Listing 3, lines 2, 3, 13 from Listing 4, and lines 2, 3, 16 from Listing 5.

The modification introduced in Listing 4 (which we called *simplified pruning*, abbreviated `scut`) is based on an observation[1] that very often it is unnecessary to know the accurate value v of the considered successor of s. If only $v \notin P$, the algorithm takes the same action (exclude $\max(P)$ from P – see line 11 in Listing 4), regardless of the precise value of v.

[1] This observation is inspired by α-β pruning [31].

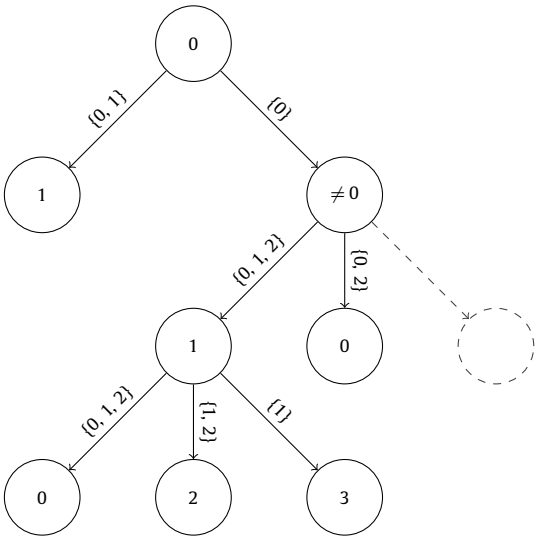

Fig. 1. An example fragment of `scut` search tree and an illustration of the pruning done by this algorithm. Successors of each position are evaluated from left to right. The root has two successors, so its P is initialized to $\{0, 1, 2\}$. Since P is passed without its maximal element to the evaluation of each successor, the set $\{0, 1\}$ is passed to the first call (the sets passed are given above the arrows and, for clarity, the figure does not include the details of this and some other recursive evaluations). This call returns 1, and since $1 \in \{0, 1, 2\}$, 1 is deleted from the set of possible values of the root, leaving $\{0, 2\}$. This set is passed without the maximal element to the calculation of the right successor of the root. To ascertain the value of the root, it is enough to know whether its right successor has the value of 0 or non-0. The algorithm is applied recursively to all successors of this successor. If any of them has the value 0, the rest can be cut-off, since the value of the right successor of the root cannot be equal to 0 (cannot be a member of the set $\{0\}$ obtained from the root). The right successor of the root has three successors. The first has the value 1, and `scut` has to evaluate its whole subtree to calculate this value. The second evaluates to 0, and so the third can be pruned.

The same action is also taken at line 9, when $v = \max(P)$. So, accurate value of v is needed only if $v \in P \setminus \max(P)$ (to exclude v from P at line 9). In other cases (to exclude $\max(P)$ from P) it is enough to know that $v \notin P \setminus \max(P)$.

The algorithm passes the set $P \setminus \max(P)$ as the second argument of the recursive call at line 7. This call is obligated to return an accurate Sprague-Grundy value of the first argument only if this value is included in the set given as the second argument (denoted as R). In other cases it may return either the accurate value or the special value -1 (which is never included in any set of Sprague-Grundy values).

The algorithm takes advantage of the argument R to cut the branches of the search tree at line 6. It immediately returns -1 if it can prove that the Sprague-Grundy value of s is not included in R (when the set of potential values of s is disjoint from R). An example of such pruning is shown in Fig. 1.

Note that performance of this and the following algorithms depend on the order in which successors are visited. It is beneficial to visit as first:

- moves which lead to possibly small subtrees (which are generally easier to compute) in order to increase the chance of pruning larger branches of the search tree,
- moves with nimbers included in $P \cap R$, in order to quickly eliminate all members of R from P and perform a cut-off as a result.

Algorithm in Listing 5 (which we called *mex tree search with pruning*, abbreviated `cut`) tries to increase the number of cut-offs in the search tree. It restricts the set passed to the recursive call (at line 8) to values not greater than $\max(R)$, by intersecting it with the set $\{0, 1, \ldots, \max(R)\}$. This may lead to an increase in the number of cut-offs in subtrees, but also to inaccuracies in the set P (see Fig. 2). Let's consider a successor m, such that $G(m) \in P \setminus \{\max(P)\}$ and $G(m) > \max(R)$. Since $G(m) \notin (P \setminus \{\max(P)\}) \cap \{0, 1, \ldots, \max(R)\}$, recursive call at line 8 might return -1 instead of $G(m)$, and therefore $\max(P)$ (at line 12) instead of $G(m)$ (at line 10) may be removed from P. A similar situation would not be possible if $G(m) \leq \max(R)$. Thus P might be inaccurate, but it is still partially correct – for values lower than or equal to $\max(R)$. So, if *result* (the only element of P after the loop – see line 14) is less than or equal to $\max(R)$ (this is checked at line 15), then $result = G(s)$. In such a situation, *result* is added to TT (at line 16) and returned (at line 17). If $result > \max(R)$, then $G(s) > \max(R)$. In such a situation, $G(s) \notin R$ and hence -1 can be safely returned (at line 19).

In order to increase the chance of finding the nimber of s and adding it to TT, the algorithm checks if all recursive calls at line 8 return exact values (obviously, P and *result* are accurate in such a situation, even if $result > \max(R)$). To this aim, it uses *exact* flag. The flag is initialized to `true` in line 5, and possibly changed to `false` (in line 13) after returning -1 by any of the recursive calls. If it remains `true` until line 15, then $result = G(s)$. Note that above-mentioned usage of *exact* flag is optional and can be dropped, especially if TT is not used.

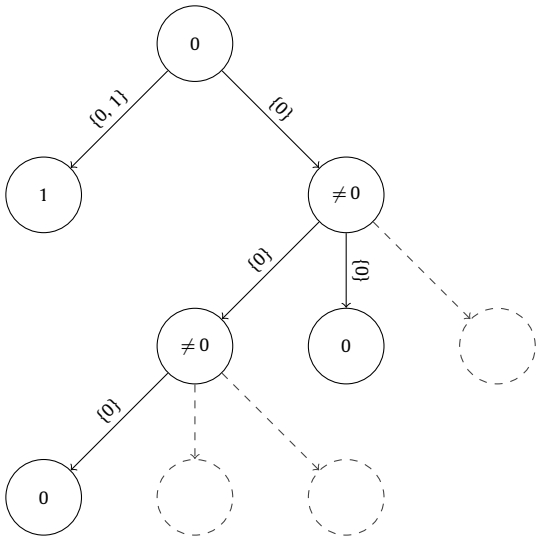

Fig. 2. An example fragment of `cut` search tree and an illustration of the pruning done by this algorithm. Successors of each position are evaluated from left to right. We use the same game tree as in the Fig. 1 to illustrate the additional cut-offs performed by `cut`. The algorithm operates by limiting the sizes of sets passed recursively down the tree (these sets are shown above the arrows), by intersecting them with the set $\{0, 1, \ldots, \max(R)\}$, with R being the set obtained from its parent. In this example, this limits to $\{0\}$ the sets passed to successors of right successor of the root. Consequently, the algorithm can stop exploring the first of these successors just after eliminating 0 from its potential values P.

```
1     M ← ∅
2     for m ∈ N(s):
3         //if P ∩ R = ∅: return −1
4         if (x, v) ∈ TT where x=m:
5             if v ∈ P:
6                 P ← P \ {v}
7             else:
8                 P ← P \ {max(P)}
9         else:
10            M ← M ∪ {m}
```

Listing 6: Enhanced Transposition Cutoff (ETC). Can be placed between lines 4 and 5 in Listing 4, or between lines 4 and 5 in Listing 5. In both listings, the loop in line 5 and 6 respectively, should iterate over M instead of $N(s)$.

Listing 6 contains an adaptation of the Enhanced Transposition Cut-off (ETC), which is a well-known improvement of α-β pruning algorithm, proposed in [32,33]. Our version can be applied to algorithm at Listing 4 as well as 5. Its idea is to exclude as many values from P as possible, before recursively visiting any of the successors by the main loop (the one at lines 5 and 6 in Listings 4 and 5 respectively). It searches for the values of subsequent successors in TT, and eliminates elements of P in the same way as the main loop does. Successors not in TT are stored in set M. Next, the main loop iterates only over them instead of all the successors.

Code in line 3 is optional. It allows the algorithm to return earlier, but it also prevents the nimber of s from being determined (when all successors of s are in TT).

Note that ETC typically lowers the number of nodes visited, at the cost of additional TT lookups. Authors of [32,33] suggest using ETC only near the root of the search tree (where the potential benefits are the greatest), in order to reach an optimal compromise between the amount of cut-offs and TT lookups.

Algorithms in Listings 4 and 5 can waste some calculations. When they return at line 6 and 7 respectively, before computing the exact value of a position, then they drop the knowledge encoded in P about the successors visited so far. Consequently, this knowledge has to be recalculated when the position is visited again.

In Listing 7 we present a version of the algorithm at Listing 5 with an extension, which we called *resuming*. The new version avoids aforementioned recalculations, by saving P in TT at line 14. This requires TT to store a set of values for a position, instead of a single value (which might increase memory usage). Condition at line 13 ensures that only fully accurate sets are stored in TT.

The sequence M of successors to be visited depends on what is found in TT at line 2. When nothing is found, the algorithm considers all successors, just as without the modification. When a singleton is found, its only member can be safely returned, because it represents an exact value of the position stored in TT (this is also similar to the unmodified version). In the remaining case, we retrieve a partially computed set P, which was constructed after evaluating first $|N(s)| - |P| + 1$ successors of s. It is enough for the algorithm to resume the calculation and consider only the remaining $|P| - 1$ successors. Note that resuming assumes that the successors of each given position are always considered in the same order

```
1  function cut_R(s, R):
2      if (x, V) ∈ TT where x=s:
3          if |V| = 1:
4              return the only element of V
5          P ← V
6          M ← last |V|−1 elements of N(s)
7      else:
8          P ← {0, 1, ..., |N(s)|}
9          M ← N(s)
10     exact ← true
11     for m ∈ M:
12         if P ∩ R = ∅:
13             if exact or max(P) ≤ max(R):
14                 TT[s] ← P
15             return −1
16         v ← cut_R(m, (P \ {max(P)}) ∩ {0, 1, ..., max(R)})
17         if v ∈ P:
18             P ← P \ {v}
19         else:
20             P ← P \ {max(P)}
21             if v = −1: exact ← false
22     result ← the only one element of P
23     if exact or result ≤ max(R):
24         TT[s] ← P
25         return result
26     else:
27         return −1
```

Listing 7: Mex tree search with pruning and resuming. Returns Sprague-Grundy value of the state s, or -1 (only if the value is not included in R).

```
1  function alg_A(s):
2      for c ← 0, 1, ...:
3          v ← alg(s, {c})
4          if v ≠ −1: return v
```

Listing 8: Aspiration sets. Returns Sprague-Grundy value of the state s. `alg` is any of the presented algorithms which take two arguments, but we recommend variants of non-simplified `cut`.

and that the values in TT are computed based on the first few successors. As ETC violates the second assumption, it cannot be straightforwardly used with resuming.

An analogous method of resuming calculation can be applied to the algorithm in Listing 4. The modification is even simpler, since in this algorithm P is always accurate, and so the condition analogous to one found at line 13 in Listing 7 is not needed.

In order to find the $G(s)$, one can call `alg(s, {0, 1, ..., |N(s)|})`, where: `alg` is any of the presented algorithms which take two arguments. Alternatively, `alg` can also be called with smaller second argument. For instance, checking if $G(s) \in X$ can be done by testing if $alg(s, X) \in X$. This gives another method for finding the Sprague-Grundy value of a position, by checking if $G(s)$ is included in successive sets. Algorithm in Listing 8 is based on this method (note that it can use different sequences of sets instead of $\{0\}, \{1\}, \{2\}, \ldots$, for instance $\{0\}, \{1, 2\}, \{3, 4, 5, 6\}, \ldots$). We called it *aspiration sets* (by analogy to α-β aspiration windows). Note that the algorithm works correctly if `alg` is a variant of `scut` or `cut`, but we strongly recommend using the latter. In the case of using `scut`, the potential benefits of passing the smaller sets do not propagate deeper than one level into the search tree.

3. Decomposable positions

Lemoine and Viennot show (in [6]) that their algorithm (presented in Listings 1 and 2) can be extended to use the Sprague-Grundy theorem for decomposable positions. The extension directly changes the algorithm in Listings 1 and indirectly the algorithm in Listings 2 (as it calls the former).

Let s be a decomposable position which consists of k disjoint positions $s_1, \ldots, s_k$.

For any $n \in \mathbb{N}$, we have the following equivalence:

$$\begin{aligned} G(s) = n &\Leftrightarrow G(s_1) \oplus \cdots \oplus G(s_{k-1}) \oplus G(s_k) = n \quad (\text{by } (2)) \\ &\Leftrightarrow G(s_k) = n \oplus (G(s_1) \oplus \cdots \oplus G(s_{k-1})) \\ &\Leftrightarrow G(s_k) = n \oplus r, \end{aligned} \tag{4}$$

where $r = G(s_1) \oplus \cdots \oplus G(s_{k-1})$.

```
1  function is_losing_ETC(s+⟨n⟩nim):
2      if (x, v) ∈ TT where x=s:
3          return v = n
4      M ← ∅
5      for m ∈ N(s): //ETC
6          if (x, v) ∈ TT where x=m:
7              if v = n: return false
8          else:
9              M ← M ∪ {m}
10     for i ← 0, 1, ..., n−1:
11         if is_losing_ETC(s+⟨i⟩nim): return false
12     for m ∈ M:
13         if is_losing_ETC(m+⟨n⟩nim): return false
14     TT[s] ← n
15     return true
```

Listing 9: Efficient implementation (with TT and ETC) of algorithm in Listing 1. Returns true if and only if $s + \langle n \rangle_{\text{nim}}$ is losing.

So, in order to check if $s + \langle n \rangle_{\text{nim}}$ is losing (which is equivalent to checking if $G(s) = n$, as shown in (3)), the extended version of the algorithm in Listing 1 can check if $s_k + \langle n \oplus r \rangle_{\text{nim}}$ is losing (which is equivalent to checking whether $G(s_k) = n \oplus r$). That is, the extension changes the argument of `is_losing`, from $s + \langle n \rangle_{\text{nim}}$ to $s_k + \langle n \oplus r \rangle_{\text{nim}}$. To calculate r, the algorithm has to calculate $G(s_1), \ldots, G(s_{k-1})$, which can be done by indirectly extended version of the method in Listing 2 (that algorithm can suppose that its argument is already decomposed and does not have to try to decompose it further).

Analogous extension can be applied to the algorithms presented in Section 2. It can be directly applied to the methods which take a pair of arguments (and indirectly to the rest): a position s, and a set of nimbers R. Using the same symbols as above, the pair which consists of decomposable s and any R, can be replaced by: s_k and $\{n \oplus r : n \in R\}$. Next, if the result differs from -1, it has to be xored with r.

To prove that the output of the extended algorithm is correct, let us analyze the possible outputs (which we denote as o) of the call with arguments $s_k, \{n \oplus r : n \in R\}$:

- If $o = -1$, we know that $G(s_k) \notin \{n \oplus r : n \in R\}$). From this and from $G(s_k) = G(s) \oplus r$ (which follows from (4)) we conclude that $G(s) \notin R$, and finally that -1 is the proper output of the whole algorithm.
- If $o \neq -1$ (which means that $o = G(s_k)$) then the whole algorithm correctly returns $o \oplus r = G(s_k) \oplus (G(s_1) \oplus \cdots \oplus G(s_{k-1})) = G(s)$.

To test if a decomposable position is losing, one can call the extended algorithm with $R = \{0\}$ and check whether the output is 0.

Thanks to the fact that the nimber of the last component (s_k) must be determined precisely only if it is included in a given set (or equal to a given value in case of the method by Lemoine and Viennot), additional cuts can usually be made in the search tree of s_k. That is why it is generally better to consider the component with the largest search tree at the end.

4. Implementations

The algorithms described in Section 2 operate on sets of nimbers. These sets can be efficiently implemented with bitfields (i-th bit is set only if a nimber i is a member of a particular set) and bitwise operations. For a game $(\mathbb{S}, s_i, N)$, the bitfields have to have a length of at least $\max_{s \in \mathbb{S}}(|N(s)|) + 1$.

Lemoine and Viennot proposed an algorithm which is presented in Listings 1 and 2, but their papers do not provide too many details about the implementation they use. Luckily, we managed to get them from sources of Glop (available at [12]). Our reimplementation is presented in Listing 9. It constitutes extended and more detailed version of Listing 1 (part presented in Listing 2 is implemented by us directly, but calls `is_losing_ETC` instead of `is_losing`). Loops at lines 10–13 are equivalent to the loop at lines 2–3 in Listing 1. The implementation uses a transposition table (TT) and Enhanced Transposition Cutoff (ETC). TT stores nimbers of positions (nimbers of their first parts as far as composed positions are concerned). When nimber of any $s \in \mathbb{S}$ is known, then outcome of $s + \langle n \rangle_{\text{nim}}$ (for any n) can be easily determined by (3): $s + \langle n \rangle_{\text{nim}}$ is losing if and only if $G(s) = n$. That is why the algorithm returns (at line 3) either `true` (if $v = n$) or `false` (if $v \neq n$), when the value v of s is found in TT at line 2. The same fact is used by ETC (see lines 5–9), in order to find outcome of all[2] successors of $s + \langle n \rangle_{\text{nim}}$, which are included in TT. Consequently, ETC is able to:

- immediately return (at line 7) when it finds a losing successor (then $s + \langle n \rangle_{\text{nim}}$ is winning),
- skip winning successors (by not adding them to set M, which is considered by the loop at line 12).

[2] The loop in lines 5–9 iterates only over successors from set $\{m + \langle n \rangle_{\text{nim}} : m \in N(s)\}$. Rest are considered in lines 2–3.

The value n of position s is appended to TT (in line 14) just after proving that $s + \langle n \rangle_{\mathrm{nim}}$ is losing (then $G(s) = n$ by (3)).

5. Benchmarks

We implemented[3] the aforementioned algorithms, and tested their efficiency against our reimplementation of the Lemoine and Viennot algorithm, which is probably the best known method so far, in calculating nimbers of well known impartial games (Nim, Chomp and Cram — the rules of these games can be found in the appendix). In this section we present the methodology of tests and the most important results. We chose the number of evaluated nodes as measurement of efficiency of the algorithms, since it is independent of implementation details of test programs. We also distinguished the amount of nodes actually evaluated, from those which value was simply looked up in a transposition table. Such a distinction enables a much finer analysis of relation of pruning by ETC to pruning done by presented algorithms.

5.1. Algorithms tested

We tested the following algorithms:

def is based directly on definition, see Listing 3,
scut mex tree search with simplified pruning, see Listing 4,
cut mex tree search with pruning, see Listing 5,
lv reimplementation of algorithm proposed by Lemoine and Viennot, see Listings 1, 2 and 9,

with the following extensions (listed after the "_" symbol):

A aspiration sets, see Listing 8,
E ETC without optional line 3, see Listing 6,
cE ETC with optional line 3, see Listing 6,
R resuming, see Listing 7.

5.2. Methods

It is obvious that the cut-offs made by algorithms presented may have impacted the efficiency of the search process. Usually, this effect should be positive, however in the case of algorithms which use the transposition table, we suspected that it also could be negative. When the exact value of a position is not found and stored in the transposition table, it might be explored multiple times. On the other hand, for games with large state space, we usually do not have enough memory to store values for all positions anyway. For this reason, we investigated the performance of our algorithms for four transposition table capacities which equal: 0 (i.e. without TT), about $\frac{1}{4}|\mathbb{S}|$, about $\frac{1}{2}|\mathbb{S}|$, and $|\mathbb{S}|$ (i.e. TT is effectively unlimited as it can store nimbers for all positions). When the number of positions stored in TT exceeds capacity, the position appended earliest is erased from TT (FIFO strategy).

Most algorithms perform a lookup in TT exactly once per node visited. However, ETC can perform additional lookups. It typically lowers the number of nodes visited, at the cost of possibly increasing in the number of TT searches. That is why we separately measured both numbers.

As previously mentioned, the performance of algorithms depends on the order in which they visit successors. For that reason, we tested algorithms with heuristic moves ordering for Nim and Chomp.

Our program does not use any other heuristics, especially game-specific ones like board symmetries.

5.3. Results

Each table presented in this section contains results obtained for a particular variant of game, for all considered combinations of algorithms (rows) and capacities of transposition table (columns). Each result is given in two forms: absolute value, and relative to `def`. Since absolute values are of different, sometimes very large orders of magnitude, we shown different numbers of significant figures (precision is shown in the units row). The results are divided into two categories: the numbers of nodes expanded and amounts of searches in transposition table (note that they can differ only for algorithms that use ETC). In each column, $_{\mathrm{N}}$ is put next to the lowest number of nodes expanded, and $^{\mathrm{T}}$ is put next to the lowest number of searches in TT. In both categories, all numbers less or equal than twice the lowest are made **bold**.

The value in the last cell of the 'TT size' row is equal to the number of different positions in the game. The title of each table contains the nimber of the initial position.

Tables 1, 2, 3 and 4 present results for the game of Nim played with three stacks of sizes either 9, 8, 5 (Tables 1 and 2) or 9, 7, 7 (Tables 3 and 4).

[3] The source code of our benchmark program is available at https://data.mendeley.com/datasets/3j8rzxshgw/draft?a=7142aedc-68dd-44da-86b2-ca52b6b5a96b.

Table 1
Nim 9, 8, 5. Nimber: 4.

TT size:	0		135		270		540	
units:	10^9	%	10^3	%	10^3	%	10^3	%
Number of nodes expanded = number of searches in TT:								
def	489	100	63 601	100	839	100	**6**	**100**
scut	252	52	55 567	87	849	101	**7**	**112**
cut	135	28	50 445	79	1 212	144	**7**	**116**
cut_A	$^{T}_{N}$**15**	**3**	39 863	63	16 987	2 025	16 412	276 243
scut_R	n/a		52 058	82	782	93	^{T}**6**	**100**
cut_R	n/a		50 440	79	1 119	133	**6**	**102**
cut_RA	n/a		46 831	74	12 278	1 463	9 446	158 993
Number of nodes expanded:								
lv	n/a		**107**	**0**	**5**	**1**	5	76
scut_E	n/a		4 900	8	71	8	$_{N}$**1**	**14**
scut_cE	n/a		5 325	8	80	10	**1**	**18**
cut_E	n/a		3 328	5	53	6	**1**	**14**
cut_cE	n/a		3 534	6	61	7	**1**	**18**
cut_EA	n/a		212	0	**4**	**1**	2	34
cut_cEA	n/a		$_{N}$**98**	**0**	$_{N}$**3**	**0**	3	52
Number of searches in TT:								
lv	n/a		**705**	**1**	**32**	**4**	32	534
scut_E	n/a		33 381	52	579	69	**8**	**127**
scut_cE	n/a		33 606	53	600	72	**9**	**155**
cut_E	n/a		23 525	37	452	54	**8**	**129**
cut_cE	n/a		22 422	35	456	54	**9**	**158**
cut_EA	n/a		1 848	3	50	6	24	406
cut_cEA	n/a		^{T}**676**	**1**	^{T}**23**	**3**	23	380

Table 2
Nim 9, 8, 5 with heuristic moves ordering. Nimber: 4.

TT size:	0		135		270		540	
units:	10^9	%	10^3	%	10^3	%	10^3	%
Number of nodes expanded = number of searches in TT:								
def	489	100	34 471	100	414	100	**6**	**100**
scut	39	8	8 590	25	157	38	**7**	**120**
cut	13	3	4 981	14	150	36	**7**	**122**
cut_A	$^{T}_{N}$**2**	**0**	1 339	4	1 011	244	1 011	17 014
scut_R	n/a		10 224	30	182	44	^{T}**6**	**94**
cut_R	n/a		5 502	16	161	39	**6**	**96**
cut_RA	n/a		1 712	5	754	182	677	11 388
Number of nodes expanded:								
lv	n/a		**14**	**0**	**4**	**1**	4	62
scut_E	n/a		301	1	7	2	$_{N}$**1**	**9**
scut_cE	n/a		327	1	8	2	**1**	**14**
cut_E	n/a		119	0	6	1	$_{N}$**1**	**9**
cut_cE	n/a		128	0	6	2	**1**	**15**
cut_EA	n/a		25	0	$_{N}$**3**	**1**	2	25
cut_cEA	n/a		$_{N}$**10**	**0**	**3**	**1**	3	46
Number of searches in TT:								
lv	n/a		**92**	**0**	**25**	**6**	25	416
scut_E	n/a		2 494	7	75	18	**7**	**110**
scut_cE	n/a		2 352	7	69	17	**8**	**138**
cut_E	n/a		1 012	3	61	15	**7**	**110**
cut_cE	n/a		906	3	52	12	**8**	**140**
cut_EA	n/a		247	1	**31**	**7**	19	320
cut_cEA	n/a		^{T}**69**	**0**	^{T}**18**	**4**	18	305

Table 3
Nim 9, 7, 7. Nimber: 9.

TT size:	0		160		320		640	
units:	10^9	%	10^3	%	10^3	%	10^3	%
Number of nodes expanded = number of searches in TT:								
def	2 403	100	137 274	100	**1 410**	**100**	**7**	**100**
scut	**1 226**	**51**	123 683	90	**1 427**	**101**	**8**	**111**
cut	**771**	**32**	120 587	88	1 550	110	**8**	**111**
cut_A	$^{T}_{N}$**754**	**31**	223 017	162	33 233	2 357	27 919	379 277
scut_R	n/a		115 616	84	**1 312**	**93**	^{T}**7**	**100**
cut_R	n/a		115 759	84	**1 407**	**100**	**7**	**100**
cut_RA	n/a		226 519	165	22 715	1 611	16 135	219 192
Number of nodes expanded:								
lv	n/a		39 380	29	477	34	11	153
scut_E	n/a		**10 467**	**8**	**115**	**8**	**1**	**13**
scut_cE	n/a		**11 432**	**8**	**131**	**9**	**1**	**17**
cut_E	n/a		**8 509**	**6**	**101**	**7**	$_{N}$**1**	**13**
cut_cE	n/a		**9 181**	**7**	**117**	**8**	**1**	**17**
cut_EA	n/a		$_{N}$**6 917**	**5**	**93**	**7**	3	43
cut_cEA	n/a		**6 945**	**5**	$_{N}$**90**	**6**	6	78
Number of searches in TT:								
lv	n/a		261 466	190	3 597	255	87	1 188
scut_E	n/a		**74 526**	**54**	**983**	**70**	**9**	**121**
scut_cE	n/a		**75 035**	**55**	**1 019**	**72**	**11**	**150**
cut_E	n/a		**62 358**	**45**	**889**	**63**	**9**	**122**
cut_cE	n/a		**60 377**	**44**	**907**	**64**	**11**	**151**
cut_EA	n/a		**52 778**	**38**	**908**	**64**	41	556
cut_cEA	n/a		^{T}**46 169**	**34**	^{T}**719**	**51**	46	620

Table 4
Nim 9, 7, 7 with heuristic moves ordering. Nimber: 9.

TT size:	0		160		320		640	
units:	10^9	%	10^3	%	10^3	%	10^3	%
Number of nodes expanded = number of searches in TT:								
def	2 403	100	72 332	100	614	100	**7**	**100**
scut	293	12	32 994	46	553	90	**10**	**141**
cut	**141**	**6**	34 634	48	613	100	**11**	**146**
cut_A	$^{T}_{N}$**139**	**6**	37 692	52	642	104	71	963
scut_R	n/a		36 207	50	562	92	^{T}**7**	**99**
cut_R	n/a		33 158	46	575	94	**8**	**103**
cut_RA	n/a		20 585	28	383	62	41	561
Number of nodes expanded:								
lv	n/a		$_{N}$**623**	**1**	$_{N}$**7**	**1**	6	85
scut_E	n/a		**1 108**	**2**	25	4	$_{N}$**1**	**9**
scut_cE	n/a		1 274	2	31	5	**1**	**16**
cut_E	n/a		**815**	**1**	21	3	$_{N}$**1**	**9**
cut_cE	n/a		**889**	**1**	25	4	**1**	**17**
cut_EA	n/a		**648**	**1**	**14**	**2**	1	19
cut_cEA	n/a		**706**	**1**	15	2	3	35
Number of searches in TT:								
lv	n/a		^{T}**4 091**	**6**	^{T}**51**	**8**	44	593
scut_E	n/a		9 627	13	270	44	**8**	**110**
scut_cE	n/a		9 472	13	277	45	**12**	**165**
cut_E	n/a		**7 240**	**10**	226	37	**8**	**110**
cut_cE	n/a		**6 529**	**9**	214	35	**13**	**175**
cut_EA	n/a		**5 807**	**8**	152	25	18	247
cut_cEA	n/a		**5 153**	**7**	129	21	21	288

Table 5
Chomp 6 × 4. Nimber: 16.

TT size:	0		52		105		209	
units:	10^9	%	10^3	%	10^3	%	10^0	%
Number of nodes expanded = number of searches in TT:								
def	161	100	275 930	100	5 363	100	^{T}**2 312**	**100**
scut	**83**	**51**	206 810	75	4 752	89	**2 589**	**112**
cut	$^{T}_{N}$**46**	**28**	136 675	50	3 535	66	**2 600**	**112**
cut_A	120	75	382 281	139	11 921	222	1 403 634	60 711
scut_R	n/a		205 916	75	4 666	87	^{T}**2 312**	**100**
cut_R	n/a		145 415	53	3 759	70	**2 318**	**100**
cut_RA	n/a		375 337	136	14 788	276	839 553	36 313
Number of nodes expanded:								
lv	n/a		1 361 853	494	5 341	100	3 930	170
scut_E	n/a		**15 276**	**6**	**223**	**4**	$_{N}$**360**	**16**
scut_cE	n/a		**15 886**	**6**	**234**	**4**	**452**	**20**
cut_E	n/a		$_{N}$**9 029**	**3**	$_{N}$**145**	**3**	**362**	**16**
cut_cE	n/a		**9 191**	**3**	**145**	**3**	**455**	**20**
cut_EA	n/a		**17 402**	**6**	**236**	**4**	981	42
cut_cEA	n/a		18 161	7	**224**	**4**	1 419	61
Number of searches in TT:								
lv	n/a		8 066 002	2 923	37 244	694	29 990	1 297
scut_E	n/a		**98 167**	**36**	**1 855**	**35**	**3 148**	**136**
scut_cE	n/a		**93 384**	**34**	**1 758**	**33**	**3 819**	**165**
cut_E	n/a		**59 664**	**22**	**1 235**	**23**	**3 169**	**137**
cut_cE	n/a		^{T}**54 269**	**20**	^{T}**1 086**	**20**	**3 830**	**166**
cut_EA	n/a		116 710	42	**2 029**	**38**	12 354	534
cut_cEA	n/a		**107 728**	**39**	**1 683**	**31**	11 299	489

Tables 2 and 4 show results for methods which ordered moves heuristically. For each subsequent heap, moves were considered in the order of increasing number of tokens left. Consequently, moves which lead to possibly small subtrees (which are generally easier to compute) were visited first and so larger branches of the search tree were more likely to have been pruned.

In version without this heuristic (Tables 1 and 3), all calculations were done 10 times, using different variants of move ordering. Each was deterministic, but unpredictable.[4] Obtained numbers of nodes visited and TT lookups were independently averaged over variants, which approximates their expected values for the random move ordering.

Tables 5 and 6 present results for the game of Chomp played on a board with 6 columns and 4 rows. The program used moves ordering similar to Nim. Version with heuristic moves ordering examined them in the order of increasing number of fields left, for each subsequent row. Rows were considered from bottom to top.

Tables 7 and 8 present results for the game of Cram played on boards 5×4 and 6×4 respectively. Here, results are also averaged over 10 runs. Each run used a different but fixed move order.[4]

In most cases, the algorithms presented effectively reduced number of nodes visited and TT lookups in comparison to `def`, especially when TT capacity was limited.

In more capacious TT, each stored information is available for longer. So the benefits of calculating the value of a position and then storing it in TT are generally greater. Consequently, cut-offs are less desirable (which is generally confirmed by our data). Especially the methods which try to prune more aggressively (for example these that use aspiration sets) lose relative to `def` even more than others. That is why `def` is hard to improve on when TT is unlimited, especially in terms of number of TT lookups. However, games with state spaces small enough to fit in TT are not challenging in practice. Realistically, TT is very small in relative to $|\mathbb{S}|$.

We have not found a universally best method, as none outperformed all others in all tests. Neither resuming (`R`) nor ETC (`E`) is unambiguously better than the other. We tend to favor `E`, especially if TT has small capacity or access time. In remaining cases `R` might be better, but do not forget that it might use more memory per item stored in TT.

There is no big difference between performances of ETC variants: with (`cE`) and without (`E`) optional line 3 in Listing 6. Benefits of using this line tend to increase as the TT capacity decreases (which was expected since the line increases the chance for finding an exact value of a position at the cost of extra TT lookups).

[4] Our program used C++ `std::shuffle` algorithm with pseudo-random generator seeded by hash of the given position increased by the variant's number.

Table 6
Chomp 6×4 with heuristic moves ordering. Nimber: 16.

TT size:	0		52		105		209	
units:	10^9	%	10^3	%	10^3	%	10^0	%
Number of nodes expanded = number of searches in TT:								
def	161	100	241 161	100	4 334	100	**2 312**	**100**
scut	17	11	55 916	23	1 651	38	**3 595**	**155**
cut	$^{T}_{N}$**3**	**2**	10 939	5	443	10	**3 665**	**159**
cut_A	9	6	33 915	14	1 692	39	346 819	15 001
scut_R	n/a		69 198	29	2 029	47	^{T}**2 310**	**100**
cut_R	n/a		16 860	7	710	16	**2 343**	**101**
cut_RA	n/a		53 519	22	2 354	54	228 273	9 873
Number of nodes expanded:								
lv	n/a		398 544	165	1 536	35	3 625	157
scut_E	n/a		2 228	1	63	1	$_{N}$**225**	**10**
scut_cE	n/a		2 214	1	64	1	**394**	**17**
cut_E	n/a		**450**	**0**	$_{N}$**16**	**0**	$_{N}$**225**	**10**
cut_cE	n/a		$_{N}$**430**	**0**	**17**	**0**	**416**	**18**
cut_EA	n/a		1 138	0	41	1	1 025	44
cut_cEA	n/a		1 058	0	42	1	1 452	63
Number of searches in TT:								
lv	n/a		2 410 604	1 000	10 490	242	26 943	1 165
scut_E	n/a		16 852	7	612	14	**2 769**	**120**
scut_cE	n/a		14 769	6	545	13	**4 149**	**179**
cut_E	n/a		**3 564**	**1**	**161**	**4**	**2 769**	**120**
cut_cE	n/a		^{T}**2 861**	**1**	^{T}**139**	**3**	**4 302**	**186**
cut_EA	n/a		9 068	4	418	10	13 256	573
cut_cEA	n/a		7 042	3	352	8	11 779	509

Table 7
Cram 5×4. Nimber: 2.

TT size:	0		14 708		29 415		58 830	
units:	10^6	%	10^3	%	10^3	%	10^3	%
Number of nodes expanded = number of searches in TT:								
def	1 781	100	2 642	100	1 731	100	**424**	**100**
scut	991	56	3 400	129	2 253	130	**616**	**145**
cut	938	53	3 360	127	2 222	128	**616**	**145**
cut_A	$^{T}_{N}$**15**	**1**	**1 133**	**43**	**648**	**37**	**617**	**146**
scut_R	n/a		2 559	97	1 673	97	^{T}**422**	**100**
cut_R	n/a		2 536	96	1 658	96	**423**	**100**
cut_RA	n/a		**1 030**	**39**	**610**	**35**	**424**	**100**
Number of nodes expanded:								
lv	n/a		**151**	**6**	141	8	141	33
scut_E	n/a		519	20	325	19	**76**	**18**
scut_cE	n/a		775	29	490	28	141	33
cut_E	n/a		513	19	319	18	**76**	**18**
cut_cE	n/a		764	29	481	28	141	33
cut_EA	n/a		$_{N}$**119**	**4**	$_{N}$**67**	**4**	$_{N}$**66**	**16**
cut_cEA	n/a		**123**	**5**	**89**	**5**	**89**	**21**
Number of searches in TT:								
lv	n/a		**702**	**27**	**658**	**38**	**658**	**155**
scut_E	n/a		3 921	148	2 592	150	**673**	**159**
scut_cE	n/a		4 889	185	3 257	188	1 035	244
cut_E	n/a		3 870	147	2 547	147	**674**	**159**
cut_cE	n/a		4 799	182	3 186	184	1 032	244
cut_EA	n/a		**826**	**31**	**506**	**29**	**503**	**119**
cut_cEA	n/a		^{T}**603**	**23**	^{T}**447**	**26**	**447**	**106**

Table 8
Cram 6 × 4. Nimber: 0.

TT size:	0		151 938		303 876		607 752	
units:	10^6	%	10^3	%	10^3	%	10^3	%
Number of nodes expanded = number of searches in TT:								
def	760 429	100	42 512	100	26 586	100	5 293	100
scut	381 192	50	60 470	142	38 227	144	8 257	156
cut	344 124	45	58 940	139	37 175	140	8 245	156
cut_A	$^{\mathrm{T}}_{\mathrm{N}}$**81**	**0**	8 433	20	8 433	32	8 433	159
scut_R	n/a		41 225	97	25 919	97	5 287	100
cut_R	n/a		40 552	95	25 414	96	5 296	100
cut_RA	n/a		7 127	17	5 590	21	5 590	106
Number of nodes expanded:								
lv	n/a		**546**	**1**	**546**	**2**	**546**	**10**
scut_E	n/a		7 220	17	4 387	17	**820**	**15**
scut_cE	n/a		13 332	31	8 543	32	2 084	39
cut_E	n/a		6 991	16	4 234	16	**821**	**16**
cut_cE	n/a		12 818	30	8 184	31	2 071	39
cut_EA	n/a		$_{\mathrm{N}}$**432**	**1**	$_{\mathrm{N}}$**432**	**2**	$_{\mathrm{N}}$**432**	**8**
cut_cEA	n/a		**501**	**1**	**501**	**2**	**501**	**9**
Number of searches in TT:								
lv	n/a		**2 412**	**6**	**2 412**	**9**	**2 412**	**46**
scut_E	n/a		66 433	156	42 348	159	8 587	162
scut_cE	n/a		102 126	240	68 302	257	18 033	341
cut_E	n/a		64 177	151	40 814	154	8 593	162
cut_cE	n/a		97 490	229	65 016	245	17 857	337
cut_EA	n/a		**3 353**	**8**	**3 353**	**13**	**3 353**	**63**
cut_cEA	n/a		$^{\mathrm{T}}$**2 317**	**5**	$^{\mathrm{T}}$**2 317**	**9**	$^{\mathrm{T}}$**2 317**	**44**

Aspiration sets (`A`) as well as `lv` can perform very well, especially when the nimber of the initial position is small. However, these algorithms perform significantly worse when this nimber is large since they have to check more potential root values (please compare Tables 1 with 3, 2 with 4, and 7 with 8). An example of this poor performance is visible in Chomp 6 × 4 (Tables 5 and 6) and Nim 9, 7, 7 (Tables 3 and 4), which had the largest values (16 and 9 respectively) of games tested. Only `cut_EA` and `cut_cEA` behaved relatively well, in particular they were often significantly better than `lv`. At the same time, they performed similarly to `lv` in other tests (without significant drawbacks, especially `cut_cEA`). That is why, we think that they constitute a more universal alternative to `lv`.

As expected, heuristic moves ordering significantly increased performance of all methods which perform cut-offs (please compare Tables 1 with 2, 3 with 4, and 5 with 6).

6. Conclusions

In the paper we put forward new game-agnostic methods for pruning search trees of impartial games. We confirmed that they are effective, especially when the transposition table has limited capacity. The benchmarks suggest that our `cut_cEA` algorithm constitutes a more universal alternative to the method proposed by Lemoine and Viennot (which seems to have been the only technique other than direct usage of the definition).

We think that pruning search trees of impartial games is a relatively unexplored subject. The promising results strongly indicate that our methods are definitely worth further investigation. Supplementary research might include an analysis on:

- how to choose aspiration sets (in Listing 8) to speed up the search process;
- possibilities of combining ETC and resuming;
- how to reach an optimal compromise between the number of cut-offs and TT lookups done by ETC;
- how to order moves to maximally reduce the size of the search tree (since successors of Nim positions can be easily generated in any desired order of nimbers, the issue can be studied empirically);
- interaction with other techniques (especially game-specific), in solving larger games;
- time and memory complexity in relation to size of the game tree.

It might also be interesting to explore variants of the algorithms in which return values meet conditions that are different from the ones presented. For instance, it can be beneficial to allow the algorithms to return just when they prove that the nimber of the position visited is a member of the set given as the second parameter.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Game rulesets

In our paper we consider short impartial games under normal play only. This means that in all of them, the first person without a move loses.

A.1. Nim

Nim [10] is an impartial game played on n heaps of items. A move consists of choosing any non-empty heap and removing any amount of items from the heap. A complete solution for every position can be found in [10,4].

A.2. Chomp

Chomp [34,35] is usually played on a rectangular $n \times m$ board with square fields, and the bottom left corner deleted. A move consists of choosing a field on the board and removing it together with those that are above it and to its right.

A.3. Cram

Cram [36, pages 502–506] is usually played on a rectangular $n \times m$ board with square fields. A move consists of choosing two horizontally or vertically neighboring fields, and erasing them from further play.

References

[1] J.H. Conway, On Numbers and Games, CRC Press, 2000.
[2] R.K. Guy, Impartial games, in: R.J. Nowakowski (Ed.), Games of No Chance, vol. 29, Cambridge University Press, 1998, pp. 61–78.
[3] R. Sprague, Über mathematische Kampfspiele, Tôhoku Math. J 41 (1935) 438–444.
[4] P.M. Grundy, Mathematics and games, Eureka 2 (5) (1939) 6–8.
[5] E.R. Berlekamp, J.H. Conway, R.K. Guy, Winning Ways for Your Mathematical Plays, vol. 1, AK Peters, 2001.
[6] J. Lemoine, S. Viennot, Nimbers are inevitable, Theor. Comput. Sci. 462 (2012) 70–79, https://doi.org/10.1016/j.tcs.2012.09.002.
[7] M. Müller, Decomposition search: a combinatorial games approach to game tree search, with applications to solving go endgames, in: IJCAI, 1999.
[8] M. Müller, Global and local game tree search, in: Heuristic Search and Computer Game Playing, Inf. Sci. 135 (3) (2001) 187–206, https://doi.org/10.1016/S0020-0255(01)00136-0, http://www.sciencedirect.com/science/article/pii/S0020025501001360.
[9] M. Müller, Z. Li, Locally informed global search for sums of combinatorial games, in: H.J. van den Herik, Y. Björnsson, N.S. Netanyahu (Eds.), Computers and Games, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 273–284.
[10] C.L. Bouton, Nim, a game with a complete mathematical theory, Ann. Math. 3 (1/4) (1901) 35–39, http://www.jstor.org/stable/1967631.
[11] R.K. Guy, C.A. Smith, The G-values of various games, in: Mathematical Proceedings of the Cambridge Philosophical Society, vol. 52, Cambridge Univ Press, 1956, pp. 514–526.
[12] J. Lemoine, S. Viennot, Glop project homepage, a solver for combinatorial games, http://sprouts.tuxfamily.org/, 2011.
[13] J. Lemoine, S. Viennot, Computer analysis of Sprouts with nimbers, in: R.J. Nowakowski (Ed.), Games of No Chance 4, Cambridge University Press, 2015, pp. 161–183.
[14] J. Lemoine, S. Viennot, Analysis of misere Sprouts game with reduced canonical trees, preprint, arXiv:0908.4407.
[15] J.W.H.M. Uiterwijk, Construction and investigation of Cram endgame databases, ICGA J. 40 (4) (2018) 425–437, https://doi.org/10.3233/ICG-180064.
[16] J.W.H.M. Uiterwijk, Solving Cram using Combinatorial Game Theory, in: Proceedings of the 16th Advances in Computer Games Conference (ACG2019), ICGA, submitted for publication, https://cris.maastrichtuniversity.nl/en/publications/solving-cram-using-combinatorial-game-theory.
[17] J.W.H.M. Uiterwijk, L. Kroes, Combining combinatorial game theory with an alpha-beta solver for cram, in: M. Atzmueller, W. Duivesteijn (Eds.), BNAIC 2018: 30th Benelux Conference on Artificial Intelligence, Jheronimus Academy of Data Science, Netherlands, 2018, pp. 267–280.
[18] G. Cairns, N.B. Ho, T. Lengyel, The Sprague–Grundy function of the real game Euclid, Discrete Math. 311 (6) (2011) 457–462, https://doi.org/10.1016/j.disc.2010.12.011.
[19] G. Nivasch, The Sprague–Grundy function of the game Euclid, Discrete Math. 306 (21) (2006) 2798–2800, https://doi.org/10.1016/j.disc.2006.04.020.
[20] U. Blass, A.S. Fraenkel, The Sprague-Grundy function for Wythoff's game, Theor. Comput. Sci. 75 (3) (1990) 311–333, https://doi.org/10.1016/0304-3975(90)90098-3.
[21] A.S. Fraenkel, M. Lorberbom, Nimhoff games, J. Comb. Theory, Ser. A 58 (1) (1991) 1–25, https://doi.org/10.1016/0097-3165(91)90070-W.
[22] A. Flammenkamp, Grundy's game, http://wwwhomes.uni-bielefeld.de/achim/grundy.html, 2002.
[23] G. Schrage, A two-dimensional generalization of Grundy's game, Fibonacci Q. 23 (4) (1985) 325–329.
[24] J. Grossman, Periodicity in one-dimensional peg duotaire, Theor. Comput. Sci. 313 (3) (2004) 417–425, https://doi.org/10.1016/j.tcs.2002.11.003.
[25] X. Sun, Improvements on Chomp, Integers 2 (G01) (2002) 8.
[26] D. Zeilberger, Three-rowed Chomp, Adv. Appl. Math. 26 (2) (2001) 168–179, https://doi.org/10.1006/aama.2000.0714.
[27] A.S. Fraenkel, The Raleigh game, Integers 7 (2) (2007) A13.
[28] H.H. Aviezri, S. Fraenkel, Never rush to be first in playing Nimbi, Math. Mag. 53 (1) (1980) 21–26, https://doi.org/10.1080/0025570X.1980.11976820.
[29] J. Kahane, A.J. Ryba, The hexad game, Electron. J. Comb. 8 (2) (2001) R11.
[30] H.K. Orman, Pentominoes: a first player win, in: R.J. Nowakowski (Ed.), Games of No Chance, vol. 29, Cambridge University Press, 1998, pp. 339–344.
[31] D.E. Knuth, R.W. Moore, An analysis of alpha-beta pruning, Artif. Intell. 6 (4) (1976) 293–326, https://doi.org/10.1016/0004-3702(75)90019-3.
[32] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin, Nearly optimal minimax tree search?, Tech. rep., Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada, dec 1994.

[33] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin, Exploiting graph properties of game trees, in: W.J. Clancey, D.S. Weld (Eds.), Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, vol. 1, AAAI 96, IAAI 96, AAAI Press/The MIT Press, Portland, Oregon, 1996, pp. 234–239, http://www.aaai.org/Library/AAAI/1996/aaai96-035.php.
[34] F. Schuh, Spel van delers (the game of divisors), Nieuw Tijdschrift voor Wiskunde, vol. 39, 1952, pp. 299–304.
[35] D. Gale, A curious Nim-type game, Am. Math. Mon. 81 (8) (1974) 876–879.
[36] E.R. Berlekamp, J.H. Conway, R.K. Guy, Winning Ways for Your Mathematical Plays, vol. 3, AK Peters, 2001.