COMPUTER ANALYSIS OF SPROUTS WITH NIMBERS

JULIEN LEMOINE - SIMON VIENNOT

Abstract. Sprouts is a two-player topological game, invented in 1967 in the University of Cambridge by John Conway and Michael Paterson. The game starts with p spots, and ends in at most $3p-1$ moves. The first player who cannot play loses.

The complexity of the p-spot game is very high, so that the best hand-checked proof only shows who the winner is for the 7-spot game, and the best previous computer analysis reached $p = 11$.

We have written a computer program, using mainly two new ideas. The nimber (also known as Sprague-Grundy number) allows us to compute separately independent subgames; and when the exploration of a part of the game tree seems to be too difficult, we can manually force the program to search elsewhere. Thanks to these improvements, we reached up to $p = 32$. The outcome of the 33-spot game is still unknown, but the biggest computed value is the 47-spot game ! All the computed values support the *Sprouts conjecture*: the first player has a winning strategy if and only if p is 3, 4 or 5 modulo 6.

We have also used a check algorithm to reduce the number of positions needed to prove which player is the winner. It is now possible to hand-check all the games until $p = 11$ in a reasonable amount of time.

1. Introduction

Sprouts is a two-player pencil-and-paper game invented in 1967 in the University of Cambridge by John Conway and Michael Paterson[5]. The game starts with p spots and players alternately connect the spots by drawing curves between them, adding a new spot on each curve drawn. A new curve cannot cross or touch any existing one, leading necessarily to a *planar* graph. The first player who cannot play loses[1].

The last rule states that a spot cannot be connected to more than 3 curves, and it induces the end of the game in a finite number of moves. If we consider that a spot has 3 *lives* at the beginning of a game, there is a total number of $3p$ lives. Each move then consumes 2 lives and creates a spot with 1 life, globally decreasing by one the total number of lives. It follows that the game ends in at most $3p-1$ moves.

Figure 1. A sample game of 2-spot Sprouts (the second player wins).

[1]The *misère* version of the game corresponds to the opposite rule, i.e. a player unable to move wins. The analysis of the misère version is more difficult, and is the object of another article[6].

Despite the small number of moves of a given game, it is difficult to determine whether the winning strategy is for the first or the second player. The best published and complete hand-checked proof is due to Focardi and Luccio, and shows who the winner is for the 7-spot game[4].

The first published computer analysis of Sprouts is due to Applegate, Jacobson and Sleator [1]: they computed in 1991 which player wins up to the 11-spot game. They noted a pattern in their results and proposed the *Sprouts conjecture*: the first player has a winning strategy in the p-spot game if and only if p is 3, 4 or 5 modulo 6. We have written a new program, using an improved version of their original representation of Sprouts positions, and obtained significant new results with the help of two main ideas: the theoretical concept of *nimber* and the manual exploration of the game tree.

Our program enabled us to compute which player wins the p-spot game up to 32 spots, and also some values up to 47 spots. All the computed values support the Sprouts conjecture. We then performed a second computation to validate the results, in which our program tries to minimize the number of positions needed to prove the value of a given p-spot game. It is then possible to generate graphs corresponding to these minimized databases, which provides a hand-checkable winning strategy for the p-spot game.

2. Game tree

We will call *position* a graph embedded in the plane, obtained with the rules of Sprouts.

From any given Sprouts position, there is necessarily a winning strategy, either for the player to move, or for the next player. If the winning strategy is for the player to move, we will say that the *outcome* of the position is a *win* and in the other case, that it is a *loss*. The main goal of our program is then to compute the outcome of Sprouts positions, and particularly the starting positions with p spots.

We call *game tree of a position* the tree where the *root* is the given position, the nodes are the positions that can be reached by playing moves from the root, and where an edge links two positions if the *child* is obtained from the *parent* in only one move. Figure 2 is an example of such a tree.

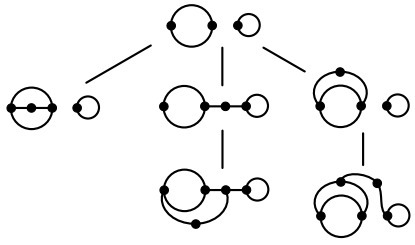


Figure 2. Game tree of a given Sprouts position.

As we will see in section 6, the outcome of a position is computed recursively by developing its game tree.

3. Positions representation

As a pencil-and-paper game, Sprouts is not suitable for programming. We describe here a representation of the game with strings that could be used with a computer to develop game trees, and therefore deduce winning strategies.

3.1. **Regions and boundaries.** In a given position, a *region* is a connected component of the plane and inside a given region, a *boundary*, is a connected component of the curves drawn by the players. An isolated spot is considered as a boundary in itself.

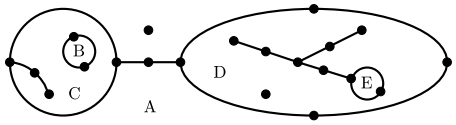


Figure 3. A Sprouts position obtained after 10 moves in a 11-spot game.

For example, figure 3 contains 5 regions and 9 boundaries: there are 3 boundaries in region D, 2 boundaries in regions A and C and 1 boundary in regions B and E.

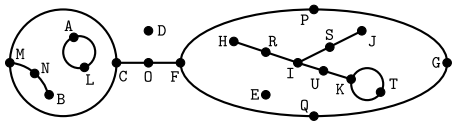
3.2. **String representation.** In order to perform computations with a computer, we need a way to represent the positions of Sprouts by strings of characters. We follow the representation described in [1] in 1991[2]. Basically, each spot (*vertex* in the graph theory) will be denoted by a letter, and the graph is described as follows:

- The complete graph is represented by the list of strings of the regions that it contains and is terminated by an *end-of-position* character: "`!`"
- A region is represented by the list of strings of the boundaries that it contains and is terminated by an *end-of-region* character: "`}`"
- A boundary is represented by the list of its vertices and is terminated by an *end-of-boundary* character: "`.`"

For a given boundary, we write the vertices in the order they appear while following one side of the boundary. Inside a given region, the same orientation must be respected.

In our example, we turn around all boundaries clockwise, except for the (unique) boundary that surrounds a region, around which we turn counterclockwise. A string representing the above position is then:

`AL.}AL.BNMCMN.}D.COFPGQFOCM.}E.HRISJSIUKTKUIR.FQGP.}KT.}!`



The boundary `AL.` appears twice, because it is inside the same region as the 4 vertices `B;N;M;C`, and it also surrounds a region itself. Notice how the string of the boundary containing `B;N;M;C` is obtained: starting at `B`, we follow the bottom of the boundary, meeting first `N`, then `M`, and `C`. Then, we continue to follow the top

[2]The equivalence between graphic and string representations is easy to see, but hard to demonstrate. This is the goal of a full 400 pages report[3].

of the boundary, meeting `M` and `N` once again (but on the opposite side), stopping when we are back at the starting point.

Remark. The strings of the starting positions are `A.}!` (for the 1-spot game), `A.B.}!` (2-spot game), `A.B.C.}!` (3-spot game)...

3.3. **Equivalent positions and strings.** A single position can correspond to several strings. For example, the first position in figure 4 can lead to `BDCDBE.}BE.A.}!` or to `A.EB.}DCDBEB.}!`. In the same way, a single string can represent several positions: `BDCDBE.}BE.A.}!` can represent the two positions of figure 4.

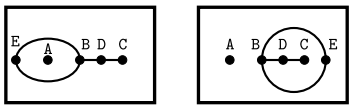


FIGURE 4. Two equivalent positions.

More precisely, a single string represents an infinite number of positions, as we can draw lines of multiple shapes as long as the topology is respected. Conversely, a single position corresponds to only a finite number of strings if we only allow the strings to use the first letters of the alphabet.

3.4. **Moves.** The main interest of the string representation described above is that moves can be easily defined and computed from it. Moves always take place in a single region but they are of two different types, depending on whether we link one boundary to itself or two different boundaries. The program we describe below will thus produce finite game trees.

Move 1. A *two-boundaries* move consists in connecting two spots of different boundaries.

Let $\mathtt{x}_1...\mathtt{x}_m$ and $\mathtt{y}_1...\mathtt{y}_n$ be two different boundaries in the same region, with $m \geq 2$ and $n \geq 2$. We suppose that $\mathtt{x}_i$ and $\mathtt{y}_j$ are vertices that occur two times or less in the whole string, with $1 \leq i \leq m$ and $1 \leq j \leq n$.

Then the two-boundaries move consists in merging these two boundaries in $\mathtt{x}_1...\mathtt{x}_i\mathtt{z}\mathtt{y}_j...\mathtt{y}_n\mathtt{y}_1...\mathtt{y}_j\mathtt{z}\mathtt{x}_i...\mathtt{x}_m$ where $\mathtt{z}$ is the new created vertex.

The same definition holds if $m = 1$ or if $n = 1$, but in these cases, $\mathtt{x}_i...\mathtt{x}_m$ and $\mathtt{y}_j...\mathtt{y}_n$ are empty boundaries.

Move 2. A *one-boundary* move consists in connecting two spots of the same boundary.

Let $\mathtt{x}_1...\mathtt{x}_n$ be a boundary, with $n \geq 2$. We suppose that $\mathtt{x}_i$ and $\mathtt{x}_j$ are vertices that occur two times or less in the whole string, with $1 \leq i \leq m$, $1 \leq j \leq n$ and $i \neq j$, or that $i = j$ and $\mathtt{x}_i$ occurs only once in the whole string.

Then the one-boundary move consists in separating the other boundaries of the same region in 2 sets $\mathtt{B}_1$ and $\mathtt{B}_2$, and the original region is divided into two new regions: $\mathtt{x}_1...\mathtt{x}_i\mathtt{z}\mathtt{x}_j...\mathtt{x}_n.\mathtt{B}_1\}$ and $\mathtt{x}_i...\mathtt{x}_j\mathtt{z}.\mathtt{B}_2\}$.

The same definition holds if $n = 1$, but in this case, $\mathtt{x}_j...\mathtt{x}_n$ is an empty boundary.

Let us give an example of the two types of moves with a 3-spot game. The initial string is `A.B.C.}!`. First, we make a two-boundaries move, which leads to `A.BDCD.}!`, and then we make a one-boundary move, which separates the position into two regions: `BDCDBE.}BE.A.}!`. Figure 5 shows three corresponding positions.

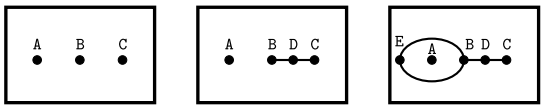


FIGURE 5. Two moves in a 3-spot game.

These definitions are sufficient to create a first version of a program for computing Sprouts. This program could determine the outcome of the p-spot game for a few values of p, but there would be too many equivalent strings, and the memory would saturate quite quickly.

4. Strings simplification

We will now explain several methods to simplify the strings, with two main goals. Firstly, if we go back to figure 2, the middle and the right children of the root can be represented by the same string (we will know at the end of this section that it is `22.}]!`). These kind of simplifications will merge some equivalent branches of the game tree and thus decrease the complexity of the computation.

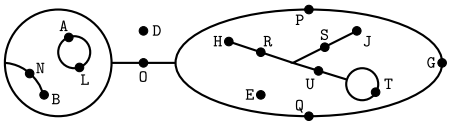
Secondly, the simplified strings will be more suited to perform the *canonization* step described in section 5.

4.1. Deletion of the dead parts.

First of all we delete the dead vertices (those which occur 3 times in the string). Then, we delete the empty boundaries (boundaries whose vertices are all dead) and finally, we delete the dead regions (with 0 or 1 life).

Using our previous example, we would therefore delete 5 dead vertices. Then, no boundary would be empty, but the region `KT.}!` would be dead. The new string and position would be:

`AL.}AL.BNN.}D.OPGQO.}E.HRSJSUTUR.QGP.}!`

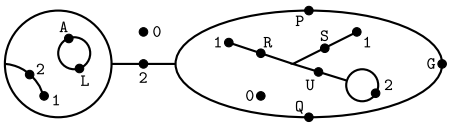


4.2. Generic vertices.

We replace the vertices that only occur once and in a boundary with a single vertex, by the generic vertex "`0`". We also replace the vertices that only occur once but in a boundary with several vertices, by the generic vertex "`1`".

The generic vertex "`2`" can designate the vertices that occur twice in a row along a single boundary, just as `N` or `O` in our example. However, it can also designate the vertices that used to occur twice, and that now only occur once, because of a dead region deletion (such as `T` in our example).

`AL.}AL.12.}0.2PGQ.}0.1RS1SU2UR.QGP.}!`



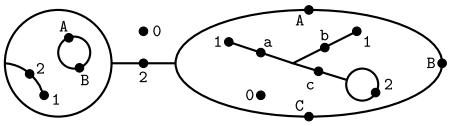
4.3. **Lands.** We cut the position into independent parts, named *lands*. A position can be cut into 2 lands when those lands have no letter in common. The end-of-land character is "`]`". Our example has 2 lands, and its string becomes:

`AL.}AL.12.}]0.2PGQ.}0.1RS1SU2UR.QGP.}]!`

4.4. **Renaming letters.** At this point, letters designate vertices that occur twice in the string, but we can distinguish between two different types: vertices that occur twice in the same boundary (which we will designate with lower-case letters) and vertices that appear in two different regions (which we will designate with upper-case letters).

We rename these vertices from "`a`" and "`A`", in the order of their appearance. We start again from "`a`" when we meet a new boundary and we start again from "`A`" when we meet a new land.

`AB.}AB.12.}]0.2ABC.}0.1ab1bc2ca.CBA.}]!`



4.5. **Regions equivalences.** When a region has 3 lives or less, we have an equivalent game if we merge the boundaries. For instance, the region `A.BC.}` becomes `ABC.}`, or `2.2.}` becomes `22.}`. These equivalences are a useful trick in our program, reducing considerably the number of stored strings.

5. Strings canonization

We already know that several strings can represent the same position, e.g. the previous position could also be represented by:

`BA2C.0.}0.2ca1ab1bc.CBA.}]AB.21.}AB.}]!`

As in [1], we call *canonization* the choice of a single string amongst all the equivalent strings. By merging equivalent branches in a game tree, canonization decreases efficiently both memory consumption and running time. On the other hand, canonization itself takes a lot of running time, and we need to take this into consideration when writing the program.

5.1. **Canonization.** First, we define an equivalence on the set of strings. Two strings are equivalent if they are equal modulo:

- the first vertex chosen in a boundary.
- the boundaries order in a region.
- the regions order in a land.
- the lands order in the position.
- the orientation chosen for each region.
- a renaming of the vertices.

We can now define the term *canonization*: this is the choice of a single string in each equivalent class. This choice should be as simple as possible, therefore, we will choose the minimal string for the following lexicographical order:

`0 < 1 < 2 < a < b < ... < A < B < ... < . < } < ] < !`

A little reflexion shows then that the canonized string of our previous position is `0.1ab1bc2ca.ABC.}0.2ABC.}]12.AB.}AB.}]!`.

Remark. We are not allowed to change the orientation of a single boundary. We can only change the orientation of a complete region, i.e. the orientation of all the boundaries in the region.

For instance, `122a2a.22.2AB.}2A.}2C.}BC.}]1122.}]!` represents a losing position, while `122a2a.22.2BA.}2A.}2C.}BC.}]1122.}]!` represents a winning position[3].

5.2. **Pseudo-canonization.** We cannot actually perform the canonization, mainly because choosing the name of the upper-case letters requires too much running time. In particular, if a land has k upper-case letters, performing the true canonization consists in choosing the minimal string for lexicographical order amongst the $k!$ possible strings.

Therefore, we only perform a pseudo-canonization: the same position can be represented by several strings, for instance the strings `0.AB.CD.}0.AB.}CD.}]!` and `0.AB.CD.}0.CD.}AB.}]!` represent the same position. Our pseudo-canonization algorithm renames the upper-case letters from "`A`" in the order of their appearance and then sorts the string. Experience shows that performing this operation twice is the most efficient for both running time and memory usage. Ultimately, we only lose a few percentages of memory compared to a true canonization, whose time complexity prevents from computing p-spot games for $p \geq 5$ or 6.

We have developed the complete game tree of the p-spot games, for $p \leq 6$, in order to evaluate the performance of our pseudo-canonization. Here is an extract of the complete game tree of the 4-spot game:

```
ABCDEF.}ABCDEG.}FG.}]!
ABCDEF.}ABCDGF.}EG.}]!
ABCDEF.}ABCGEF.}DG.}]!
ABCDEF.}ABGDEF.}CG.}]!
ABCDEF.}BCDEFG.}AG.}]!
```

Since these strings represent the same position, a true canonization would have displayed only one string instead of these five. However, this position is not needed to compute the outcome of the 4-spot game. In fact, the performance of the pseudo-canonization (comparatively to the true canonization) is better in a real computation that in a complete game tree development, because in a real computation we meet strings easier to compute, with less upper-case letters.

The following table gives the number of pseudo-canonized strings stored after a complete game tree development for the p-spot game.

n	number of strings
2	18
3	157
4	1796
5	24784
6	393103

This table could be useful for evaluating the performance of our canonization, in comparison with the canonization of other programs.

[3]More obvious examples may exist, but this is the simplest that we know. However, to check this example, a long computation is necessary.

6. Main algorithm

6.1. **Outcome computation algorithm.** The standard recursive algorithm to compute the outcome of a position[4] can be described as follows:

Algorithm 1. *function compute-win-loss(position P)*

- *compute the children of P*
- *for each child, do: if compute-win-loss(child)=Loss, return Win*
- *return Loss*[5]

The core of our main algorithm uses this classical depth-first procedure. We also store the outcomes of the computed positions in a database (a *transposition table*), in order not to compute several times the same position. Figure 6 shows how this algorithm works with the 2-spot game.

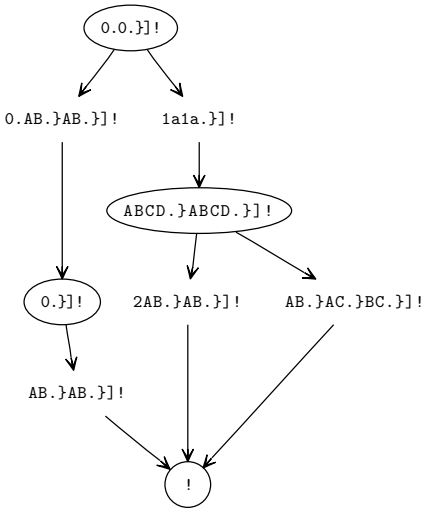


Figure 6. 2-spot game.

Surrounded positions are the losing ones. As figure 6 shows, a position is losing if all its children are winning. But on the contrary, only one losing child is sufficient to prove that a position is winning, which implies that only a part of the complete game tree is sufficient to determine the outcome of the game (only 9 of the 18 positions of the complete game tree are needed here). We will detail this point again in section 7.2.

6.2. **Sum of independent games.** Sprouts positions can frequently be separated into independent games, the *lands*. It would then be convenient to compute these lands independently, and deduce the outcome of the complete position from the outcome of the lands.

As explained in [1], this can partially be done, because the sum of two losing games is losing, and the sum of a losing and a winning game is winning. However, the weakness of this method is that the sum of two winning games can either

[4]In the following, we use the term "position" even if we speak of the string that represents it.
[5]This step is reached only if no child is losing.

be winning or losing. We had implemented it in our program at first, and rather quickly, a large number of positions with several lands was saturating our databases, preventing us from improving the results of [1] of more than 2 or 3 spots.

As described below, the concept of *nimber* can solve this problem, allowing us to compute the lands separately.

6.3. **Nimber theory.** A more detailed view of this theory can be found, amongst others, in [2].

Definition. *The nimber of a position P is denoted by $|P|$, and is defined as the smallest non-negative integer that is not a nimber of a child of P.*

This definition implies that $|P| = 0$ if P is losing, and $|P| \geq 1$ if P is winning. We can also see that, if we know the complete game tree of P, we can recursively compute $|P|$, but in fact it is not necessary to expand the complete game tree to compute the nimber of a position, as the algorithm 2 shows below.

The main result of the nimber theory can be stated as follows:

Theorem. *If a position P is made of two independent positions P_1 and P_2, then $|P|$ is the "bitwise exclusive or" of $|P_1|$ and $|P_2|$, denoted by $|P_1| \wedge |P_2|$.*

For example, $|\texttt{1AB.}\}\texttt{AB.}\}\texttt{]!}| = 3$ and $|\texttt{22.}\}\texttt{]!}| = 1$, so
$|\texttt{1AB.}\}\texttt{AB.}\}\texttt{]22.}\}\texttt{]!}| = 3 \wedge 1 = 2$.

6.4. **Couples.** In our program, instead of computing the outcome of a position, we compute the outcome of a *couple*: (position+nimber), which represents the sum of two independent games. The position part consists of the game of Sprouts for the given position. The nimber part is the game of Nim for the given nimber value. The outcome of $(\emptyset{+}n)$ is a loss if $n = 0$, a win if $n \geq 1$. The original position P_0 is replaced by the couple $(P_0{+}0)$.

We see that "$(P{+}n)$ is losing" means that $|P| = n$. As we only store losing positions in our program, it means that we will only store positions whose nimber is known (and the winning positions that we do not store correspond to positions whose nimber is only known to be different of certain values).

To determine the outcome of a couple, we can still use the algorithm 1, by extending to couples the definition of children: the children of a couple $(P{+}n)$ are the children of the position part, whose form is $(\text{child}(P){+}n)$, and the children of the nimber part, whose form is $(P{+}m)$, with $m < n$.

6.5. **Computation of the nimber of a position.** With algorithm 1 applied to a couple $(P{+}n)$, we already are able to determine if n is the nimber of the position P. If we need to compute the nimber $|P|$ of this position, we use the following (simple but efficient) method:

Algorithm 2. *function nimber-of(position P)*

- $n := 0$, $found := false$
- *while* $found = false$ *do:*
  if compute-win-loss($P{+}n$)=Loss, then $found := true$, *else* $n := n + 1$
- *return* n

It merely consists in trying 0, 1, 2... until we find the right nimber. This algorithm will only be used on single lands, as explained below.

6.6. **Positions with several lands.** If a position is made of two lands, as in $(P_1]P_2]! + n)$, and if we know the nimber $|P_2|$, the above theorem shows that the outcome of $(P_1]P_2]! + n)$ will be the same as the outcome of $(P_1]! + n \wedge |P_2|)$.

So when our main algorithm meets a position with several lands, it computes with the algorithm 2 the nimbers of all lands except one and merges the results with the nimber part of the couple. Therefore, we always compute the lands separately, and an indirect consequence is that we store only single lands in our database.

6.7. **Main algorithm.** With the previous ideas, algorithm 1 is now:

Algorithm 3. *function compute-win-loss(couple(P+n))*

- *for the lands of P whose nimber is already stored in the database, merge their nimber with the nimber part (with bitwise xor).*
- *compute the nimber of all the unknown lands of P with algorithm 2 (except for one land), and merge those nimbers with the nimber part (P is then a single land).*
- *for each child*[6] *of (P+n), do:*
  if compute-win-loss(child)=Loss, return Win
- *store (P+n) in the database and return Loss*

Figure 7 shows how this algorithm works with the 3-spot game.

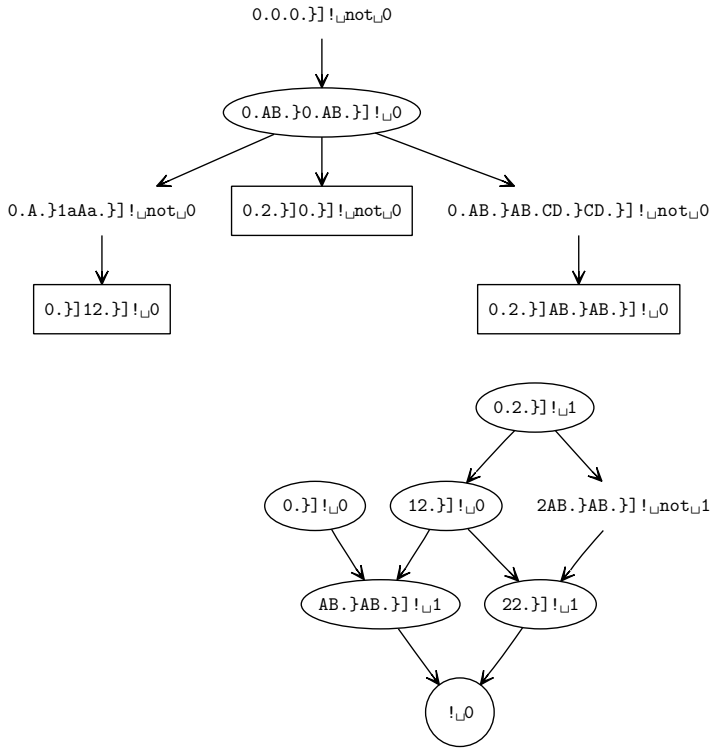


FIGURE 7. 3-spot game.

[6]Children of position part, and children of nimber part.

Couples surrounded by a rectangle are those made of several lands. Their nimber is computed from the nimber of their lands, which can be found lower in the graph.

Couples surrounded by an ellipse are losing. Only those couples are stored by the program. All their children are computed to find their position's nimber.

The remaining couples are winning. The nimber of their position part is not known. We only know the nimber to be different of one (or sometimes several) value, because this value is already the nimber of some child of the position.

We see on this example the efficiency of this algorithm. There are 157 positions in the complete game tree of the 3-spot game, but the program only needs 14 nodes to compute the outcome.

7. Computation improvement

7.1. **Positions storage.** As in [1], we store only losing couples in order to reduce the memory consumption: as the complexity of the computation increases, losing couples become less frequent, so this choice allows us to reduce the size of the transposition table. The profit is variable, but we can take into consideration that the size of the database is divided by a factor between 5 and 20, which is not negligible. In exchange, we need a little more running time when computing again a winning couple that has already been computed before. Since the couple is not stored in the database, its winning outcome will be known only after a computation of its children, one of which will be found to be losing in the database.

The main difference of our database compared to [1] is that we store losing couples (positions whose nimber is known), and not losing positions. Moreover, as explained before, we store only positions with a single land.

It is also possible in our program to export databases as text files, allowing us to analyze them manually, or resume later an incomplete computation.

7.2. **Children ordering.** When a position (or a couple) is losing, there is no short way to prove it: we need to compute all the children and prove that they are all winning. On the contrary, when a position is winning, we only need to find one losing child, so that there are several ways to search the game tree and compute the outcome of the starting position. These ways are not of equivalent difficulty, because Sprouts game trees are unbalanced, with some areas far more complicated than others.

Consequently we sort the children by computational difficulty, in order to find a losing child as quickly as possible. Thus, we lose the least possible time possible in unnecessary computations of winning children and also find first the losing children whose outcomes are easy to compute.

When ordering the children, we try to evaluate their difficulty only from their string. The rules used to evaluate the difficulty and order the children are an important part of the program, because a bad set of rules could push the computation into complicated parts of the game tree. Our current set of rules is as follows:

- priority to couples with minimal (number of lives + nimber).
- priority to positions with a lot of lands.
- priority to positions with a little estimated number of children.

Since it would require too much running time to compute the exact value, the number of children is only estimated as follows:

- When linking a boundary to itself, the possible number of children is $(number\ of\ vertices)^2 \times (number\ of\ partitions)$, where $(number\ of\ partitions)$ is the number of ways to partition the other boundaries into two sets.
- When linking two different boundaries, the possible number of children is the sum, for any couple (B_i, B_j) of boundaries, of $(number\ of\ vertices\ of\ B_i) \times (number\ of\ vertices\ of\ B_j)$.

We must also make a choice when considering a position composed of several lands. We decided to compute first the nimbers of the lands with the smallest number of lives.

These rules significantly improve the computation compared to a random exploration of the game tree, but they are still far from providing an optimal exploration of the game tree. In fact, we believe that whatever the rules used to order the children, their efficiency is eventually limited and more global search algorithms are needed.

7.3. **Manual exploration of the game tree.** We have implemented an interface to follow and interact in real-time with the computation process. When the computation seems to be stuck in some part of the game tree, we can manually decide to explore elsewhere.

During the computation, the program displays the list of currently studied couples. At a given instant, the program is computing the outcome of a list of couples, each one being the child of the previous one. For example, here is what the program could display during a 12-spot game computation:

level	position part	nimber part
1	`0.0.0.0.0.0.AB.}0.0.0.0.0.0.AB.}]!`	0
2	`0.0.0.0.0.}]0.0.0.A.}0.0.0.A.}]!`	0
3	`0.0.0.0.0.}]!`	1
4	`0.0.0.0.AB.}AB.}]!`	1
5	`0.0.0.0.}]!`	1
6	`0.0.1a1a.}]!`	1

With a click in the interface, it is then possible to choose another child on a given level, and so to compute another part of the game tree. For example, we could decide to compute the couple (`0.0.A.}0.0.A.}]!`+1) in the fifth level.

When the position is composed of several lands, the program computes the nimber of one of these lands with algorithm 2 (here, it has already computed that the nimber of `0.0.0.0.0.}]!` is not 0, and is now trying with 1). In this case too, we can click in the interface to compute first the nimber of another land. In our example, we could decide to compute the nimber of `0.0.0.A.}0.0.0.A.}]!` in the third level.

We empirically decide where and when we should click: if the program spends too much time on a branch of the game tree, we decide to change it. If a couple on a level is almost computed to be losing (which means that almost all its children have been computed to be winning), it is often efficient to click two levels below, to try to quickly find losing children of the remaining unknown children. We can also have a look at the position to try to choose easier couples (positions are easier when they tend to be quickly cut into lands).

Being able to manually choose which part of the game tree to explore was far more efficient that any automatic selection that we imagined. For example, whereas a computation for the 12-spot game ends after having stored more than $100,000$ couples without human intervention, by clicking, somebody with some experience can end this game in less than $2,000$ couples.

7.4. **Check computation.** User interactions proved to be powerful, but they have a drawback: it is impossible to reproduce exactly the same computation twice. For this reason, when a computation succeeds, we perform a standalone check computation, which uses the previous results to guide itself in the game tree. This check computation does not authorize any interaction from the user and is of course reproducible for a given database of previous results.

The check computation also reduces the number of couples needed to prove the result. Indeed, during the first computation, we compute many useless couples (the winning children of winning couples are usually useless). When we meet a new couple in the check computation, we look for its value in the database of previous results, and if it is losing (i.e. a position whose nimber is known), we compute again all its children to check it. But if it is winning, we look in the previous database which child is losing, and we compute only this one.

The result of a check computation is then a *solution tree*, providing a winning strategy without any unnecessary information. These solution trees are small enough to provide hand-checkable proofs for p-spot games with little values of p. For bigger values of p, it is mainly a way of reducing the size of the files storing the winning strategies.

During the check computation, we can generate a file compatible with the graph visualization tool Graphviz[7], which enables us to draw graphs describing how the computation proceeds (e.g. figures 6 and 7). For bigger values of p, the complete graph would be rather unreadable, but we can choose to plot only the positions with a minimal number of lives. For example, figure 8 shows the upper part of the graph for the 5-spot game, with only the positions with 12 lives or more.

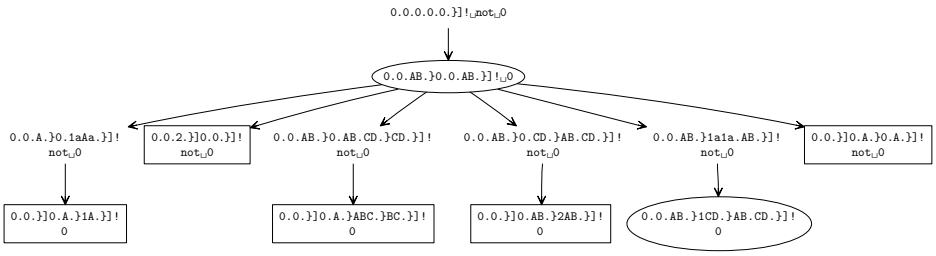


FIGURE 8. A solution for the 5-spot game, positions with 12 lives or more.

Here, we can see how cutting positions into lands simplifies the computation: to complete it, we only need to compute the nimber of 6 lands of less than 7 lives, and the outcome of a position of 12 lives. The most spectacular example is the case of p=17: in our computation, every position is cut into several lands of less than 27 lives after only 3 moves from the starting position (which has 51 lives).

[7]http://www.graphviz.org/

It is also possible to plot only reference numbers instead of the whole positions, which is useful for bigger graphs. See, for instance, figure 9, which shows the complete 4-spot game. The numbers have no special meaning, the positions which they refer to are listed in the appendix, with their computed values.

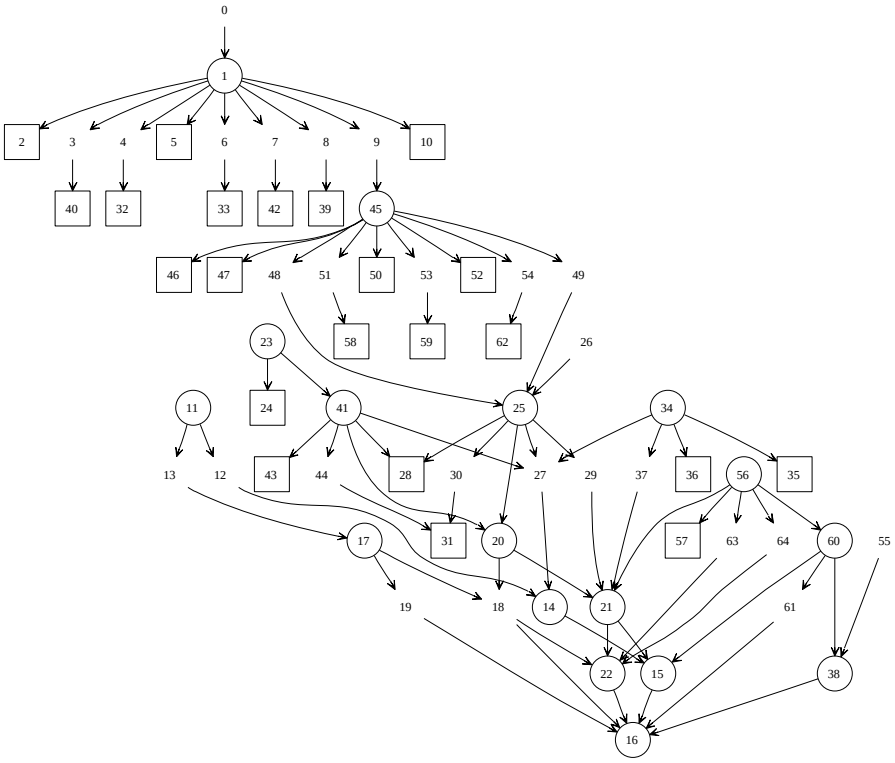


FIGURE 9. A complete solution for the 4-spot game.

8. RESULTS

8.1. ***p*-spot games already computed.** The following table shows the number of couples stored in the minimal databases for all the p-spot games computed up to now. We have computed all p-spot games up to 32 spots. The 33-spot game is the first unknown one, but the biggest computed one is the 47-spot game. The results so far all support the conjecture emitted in [1]: the first player loses if and only if p is 0, 1 or 2 modulo 6.

p	size
0	0
1	1
2	3
3	6
4	16
5	38
6	64

p	size
7	103
8	205
9	63
10	140
11	140
12	475
13	577

p	size
14	1580
15	3252
16	1068
17	471
18	3233
19	3630
20	4051

p	size
21	9270
22	5706
23	2837
24	9316
25	9229
26	18567
27	59117

p	size
28	14813
29	3414
30	58363
31	58365
32	58204
33	?
34	21107

p	size
35	18812
...	?
40	45782
41	48890
...	?
47	54542
...	?

It is rather surprising that the number of stored couples does not increase strictly with p. In fact, some patterns seem to occur modulo 6.

We found that $p = 15$, 21 and 27 were very difficult to compute, because they are winning, and only have one losing child, which is the most complicated one (its string is `0.0.0.(...)0.1a1a.}]!`). $p = 33$, the first value that we do not yet know, seems to follow the same path.

Conversely, $p = 17$, 23, 29, 35, 41 and 47 were much easier to compute. They are winning positions, and the losing child is obtained by linking one spot to itself and separating the remaining spots in two equal sets (this child is often the easiest to compute).

We also noticed that the number of stored couples is almost equal for $p = 15$ and $p = 18, 19$: once the result of $p = 15$ is known, we only needed a little more computation to deduce the result for $p = 18, 19$. The same phenomenon occurs for $p = 21$ and $p = 24, 25$, or for $p = 27$ and $p = 30, 31$.

We tried to minimize the number of couples for $p \leq 25$ (and $p = 29$), but we did not take the time to do the same work for the other values, therefore the reader needs to be aware that it is probably possible to reduce significantly the number of couples for those values.

8.2. **Nimber conjectures.** We observed that the nimber of the winning starting positions, for $p \leq 32$, is 1, so we propose a stronger conjecture than the "Sprouts conjecture" emitted in [1]:

Conjecture. *The nimber for the starting position with p spots is* 0 *if p is* 0, 1 *or* 2 *modulo* 6, *and* 1 *if p is* 3, 4 *or* 5 *modulo* 6.

It is rather easy to imagine other conjectures around the nimber. For example, if we compute the nimber of the position: `222(...)222.}]!`, with p generic vertices "2", we obtain (starting from `22.}]!`): 1;0;2;1;0;3;1;0;5;1;0;3;1;0;3;1;0;3;1;0;3;1;0;3;1;0.

We also imagined another extension of the Sprouts conjecture: the nimber of the position is left unchanged by an addition of 6 boundaries "`0.`" into a given region. This conjecture is false, since `0.22.}]!` has nimber 0, and `0.0.0.0.0.0.0.22.}]!` has nimber 2. But it does work for more than 90% of the positions that we tried. The existence of such nearly-true patterns tends to infirm the Sprouts conjecture.

8.3. **Hand-checkable proofs.** Using a computer to determine mathematical results is not completely convincing, especially because there could be a programming error, considering the size of the program. So, the most skeptic readers could use the program to generate files that would provide them with hand-checkable proofs of the simplest results. For example, figure 9 provides a proof that the first player wins in the 4-spot game.

We printed the result of a check computation for the 9-spot game and manually checked 66 losing couples (positions of known nimber), and 258 winning couples (positions whose nimber is known to be different of certain values). For the losing couples, we checked that we obtained the same sets of children as our program, and for the winning ones, we had to check only one child, very seldom two or more. This took us a few hours.

The table in paragraph 8.1 implies that we could use this method to check in a reasonable amount of time the value of the p-spot game with $p \leq 11$. The program can also point in the right direction someone who would like to create a totally manual proof, like the one in [4].

Conclusion

The main obstacle for computing higher p-spot games is neither memory nor computation time, but rather human time for the manual exploration of the game tree. This human intervention is an embryo of a best-first search, which is probably more suited for the game of Sprouts than the depth-first search currently performed by our program. Therefore, implementing classical best-first search algorithms, such as the PN-search, would probably lead to better results.

Distributed computing is also another solution to compute higher p-spot games, and the check computation, by reducing the size of the databases, would be really efficient for this. It is likely that in the coming years, we will know who wins in the game of Sprouts when starting with more than fifty spots.

The program that we used for the computations is available with its source code on our web site `http://sprouts.tuxfamily.org/` under a GNU licence, together with several resulting databases.

Acknowledgements

We wish to thank Jean-Paul Delahaye for his decisive contribution to the continuation of our work.

References

[1] D. Applegate, G. Jacobson, and D. Sleator, *Computer Analysis of Sprouts*, Tech. Report CMU-CS-91-144, Carnegie Mellon University Computer Science Technical Report, 1991.

[2] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy, *Winning ways for your mathematical plays (second edition, volume 1)*, A K Peters, 2001.

[3] Joachim Draeger, Stefan Hahndel, Gerhard Köstler, and Peter Rossmanith, *Sprouts – Formalisierung eines topologischen Spiels*, Tech. Report TUM-I9015, Technische Universität München, Institut für Informatik, March 1990.

[4] Riccardo Focardi and Flaminia L. Luccio, *A modular approach to Sprouts*, Discrete Applied Mathematics **144** (2004), no. 3, 303–319.

[5] Martin Gardner, *Mathematical games : of sprouts and Brussels sprouts, games with a topological flavor*, Scientific American **217** (July 1967), 112–115.

[6] Julien Lemoine and Simon Viennot, *Analysis of misère Sprouts game with reduced canonical trees*, `http://arxiv.org/abs/0908.4407`, 2009.

Appendix: Correspondence for Figure 9

#	position	nimber
0	0.0.0.0.}]!	$\neq 0$
1	0.0.AB.}0.AB.}]!	0
2	0.0.2.}]0.}]!	$\neq 0$
3	0.0.AB.}AB.CD.}CD.}]!	$\neq 0$
4	0.0.A.}1aAa.}]!	$\neq 0$
5	0.0.}]0.2.}]!	$\neq 0$
6	0.1aAa.}0.A.}]!	$\neq 0$
7	0.AB.CD.}0.AB.}CD.}]!	$\neq 0$
8	0.AB.}0.CD.}AB.CD.}]!	$\neq 0$
9	0.AB.}1a1a.AB.}]!	$\neq 0$
10	0.A.}0.A.}]0.}]!	$\neq 0$
11	0.0.}]!	0
12	0.AB.}AB.}]!	$\neq 0$
13	1a1a.}]!	$\neq 0$
14	0.}]!	0
15	AB.}AB.}]!	1
16	!	0
17	ABCD.}ABCD.}]!	0
18	2AB.}AB.}]!	$\neq 0$ 1
19	AB.}AC.}BC.}]!	$\neq 0$
20	0.2.}]!	1
21	12.}]!	0
22	22.}]!	1
23	0.A.}0.A.}]!	1
24	0.}]12.}]!	0
25	0.AB.}2AB.}]!	0
26	0.0.2.}]!	$\neq 0$
27	0.A.}2A.}]!	$\neq 0$
28	0.}]22.}]!	$\neq 0$
29	1aAa.}2A.}]!	$\neq 0$
30	2AB.}AB.CD.}CD.}]!	$\neq 0$
31	22.}]AB.}AB.}]!	0
32	0.0.}]12.}]!	0

#	position	nim.
33	0.A.}1A.}]0.}]!	0
34	0.A.}1A.}]!	0
35	0.}]AB.}AB.}]!	$\neq 0$
36	12.}]1.}]!	$\neq 0$
37	1A.}ABC.}BC.}]!	$\neq 0$
38	1.}]!	1
39	0.AB.}2AB.}]0.}]!	0
40	0.A.}0.A.}]AB.}AB.}]!	0
41	0.A.}ABC.}BC.}]!	0
42	0.A.}ABC.}BC.}]0.}]!	0
43	12.}]AB.}AB.}]!	$\neq 0$
44	ABC.}ADE.}BC.}DE.}]!	$\neq 0$
45	0.AB.}1CD.}AB.CD.}]!	0
46	0.2.}]1AB.}AB.}]!	$\neq 0$
47	0.AB.}2AB.}]1.}]!	$\neq 0$
48	0.AB.}2CD.}AB.CD.}]!	$\neq 0$
49	0.AB.}ABC.}CDE.}DE.}]!	$\neq 0$
50	0.AB.}AB.}]12.}]!	$\neq 0$
51	0.A.}1B.}aAaB.}]!	$\neq 0$
52	0.}]1AB.}2AB.}]!	$\neq 0$
53	1aAa.}1BC.}ABC.}]!	$\neq 0$
54	1AB.}AB.CD.} CD.EF.}EF.}]!	$\neq 0$
55	1AB.}AB.}]!	$\neq 1$
56	1AB.}2AB.}]!	1
57	1.}]22.}]!	0
58	0.}]1.}]AB.}AB.}]!	0
59	12.}]1A.}2A.}]!	0
60	1A.}2A.}]!	0
61	2A.}2A.}]!	$\neq 0$
62	1AB.}2AB.}]AB.}AB.}]!	0
63	2AB.}2AB.}]!	$\neq 1$
64	2A.}ABC.}BC.}]!	$\neq 1$