A FURTHER COMPUTER ANALYSIS OF SPROUTS

JULIEN LEMOINE - SIMON VIENNOT

Abstract. Sprouts is a two-player topological game, invented in 1967 in the University of Cambrigde by John Conway and Michael Paterson[3]. The game begins with n spots; the first player that cannot play loses, which happens after at most $3n-1$ moves.

The complexity of the n-spot game is very high, so that the best hand-checked proof only shows who the winner is for the 7-spot game[2], and the best computer analysis reaches $n = 11$[1].

We wrote a computer program, using mainly two new ideas. The nimber (also known as Sprague-Grundy number) allows us to compute separately independant subgames; and when the exploration of a part of the game tree seems to be too difficult, we can manually force the program to seek elsewhere. Thanks to these improvements, we reached $n = 26$. All the computed values support the *Sprouts conjecture* [1] : the first player has a winning strategy if and only if n is 3, 4 or 5 modulo 6.

We also used a check algorithm to reduce the number of positions that need to be computed to find who wins a given game. It is now possible to hand-check all the games until $n = 11$ in a reasonable amount of time.

1. Introduction

Sprouts is a two-player pencil-and-paper game invented in 1967 in the University of Cambrigde by John Conway and Michael Paterson[3]. The game begins with n spots; players alternately connect the spots by drawing curves between them, adding a new spot on each curve drawn. This produces a graph that must be *planar* : a new curve can't touch another curve. The first player that cannot play loses.

The rule that induces the ending of this game is that a spot can't be connected to more than 3 curves : if we consider that a spot has 3 "lives" in the beginning of a game, there is a total of $3n$ lives. A move consumes 2 lives and creates a spot with 1 life, so each new move consumes one life, and the game ends in at most $3n-1$ moves.

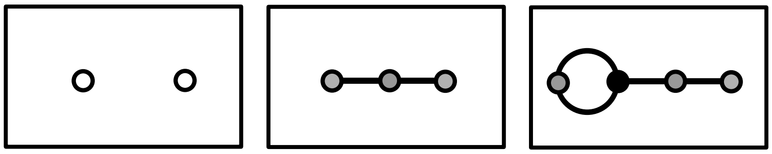

Figure 1. Two moves in a 2-spot game.

Despite the little number of moves of a given game, it is difficult to determine whether the first or the second player wins, provided those players play perfectly.

Date: April 7, 2007.

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[2].

The last published computer analysis of this game 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 n-spot game if and only if n is 3, 4 or 5 modulo 6. In our program, we take up their main ideas, and we improve them, sometimes even with tricks they gave in their article.

The program we present in the following has many uses : it computes which player wins for the n-spot game (from now on, it performed this computation up to 26 spots, and even for 35 spots). All the computed values support the Sprouts conjecture. The program also tries to minimize the number of positions needed to prove who the winner is for a given n-spot game. It generates graphs to illustrate those demonstrations, and it provides a winning strategy for playing by correspondence.

2. Position representation

2.1. **An algebraic definition of the game of Sprouts.** In this paragraph, we give an algebraic definition of the game of Sprouts, which is equivalent to the graphic one. We will punctually draw figures, but only to illustrate some representations.

Definition 2.1. A *representation* is a string of characters verifying the following rules :

- The last character of the string is the *end-of-position* character : "`!`"
- A representation is the union of regions, the *end-of-region* character is "`}`"
- A region is a union of boundaries. The *end-of-boundary* character is "`.`"
- A boundary is a union of vertices.

The set of representations will be denoted by $\mathscr{R}_0$. The *initial* representations are `A.}!` (1-spot game), `A.B.}!` (2-spot game), `A.B.C.}!` (3-spot game)... Players alternately do a move, and the first who cannot play anymore loses. Moves always take place in a single region. There are two different types of moves :

Definition 2.2. $x_1...x_m$ and $y_1...y_n$ are two different boundaries in the same region, with $m \geq 2$ and $n \geq 2$. We suppose that x_i and y_j are vertices that occur two times or less in the whole representation, with $1 \leq i \leq m$ and $1 \leq j \leq n$.

A *two-boundaries* move consists in merging these two boundaries in $x_1...x_i z y_j...y_n y_1...y_j z x_i...x_m$.

The same definition holds if $m = 1$ or if $n = 1$, but in these cases, $x_i...x_m$ and $y_j...y_n$ are empty boundaries.

Definition 2.3. $x_1...x_n$ is a boundary, with $n \geq 2$. We suppose that x_i and x_j are vertices that occur two times or less in the whole representation, with $1 \leq i \leq m$, $1 \leq j \leq n$ and $i \neq j$, or that $i = j$ and x_i occurs only one time in the whole representation.

A *one-boundary* move consists in separating the other boundaries in the same region in 2 sets B_1 and B_2, and in separating the region in two new regions : $x_1...x_i z x_j...x_n.B_1\}$ and $x_i...x_j z.B_2\}$.

The same definition holds if $n = 1$, but in this case, $x_j...x_n$ is an empty boundary.

Notice that we arbitrarily chose in the definitions to denote the new vertex by z, but any available letter would be fine.

Let's give an example with a 3-spot game : the initial representation is `A.B.C.}!`. First, we make a two-boundaries move, which leads to `A.BDCD.}!`, and then a one-boundary move, which separates the representation in two regions : `BDCDBE.}BE.A.}!`. Figure 2 explains what happens :

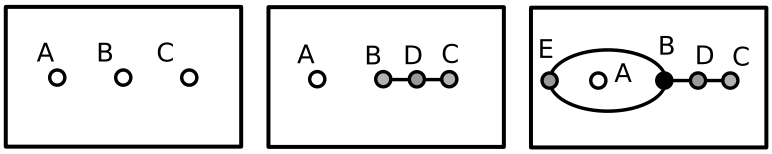

FIGURE 2. Two moves in a 3-spot game.

Remark 2.4. It is possible that a representation from $\mathscr{R}_0$ can't be obtained with the legal moves from an initial position, this is for instance the case of `AB.}AB.}AB.}!`.

These definitions are sufficient to create a first version of a program for computing Sprouts. This program would be rudimentary, but it could determine the Win/Loss of a n-spot game for a few values of n. We'll call such a program a *theoretical* program.

2.2. **Link with classical definition.** We call *position* a graph embedded in the plane, verifying the rules of the game of Sprouts. We can determine a representation of this position if in each region, for each boundary, we write in the order the vertices met while following the boundary. We must respect the orientation : we turn around each boundary clockwise, except around the only boundary that surround the region, where we turn counterclockwise.

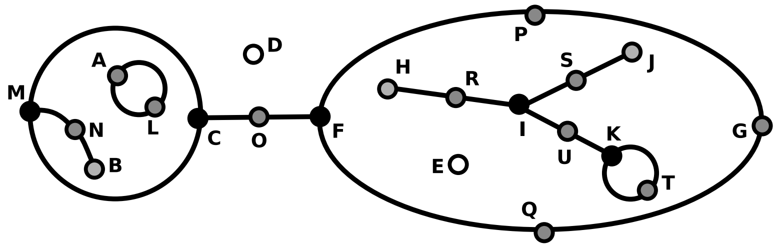

FIGURE 3. After 10 moves in a 11-spot game.

For instance, a representation of figure 3 is :

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

A single position can lead to several representations. For example, the last position in figure 2 can lead to `BDCDBE.}BE.A.}!` or to `A.EB.}DCDBEB.}!`. Conversely, a single representation can lead to several representations : `BDCDBE.}BE.A.}!` can lead to the two positions in figure 4 :

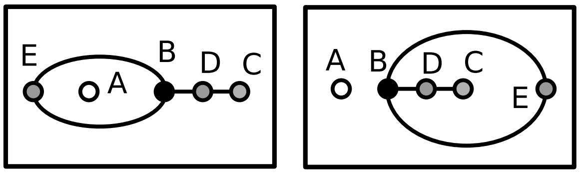

FIGURE 4. Two equivalent positions.

2.3. **The $\mathscr{R}_1$ set.** First of all, we define an equivalence relation on $\mathscr{R}_0$. Two representations are equivalent if they are equal modulo :

- the first vertex chosen in a boundary.
- the boundaries order in a region.
- the regions order.
- renaming the vertices.
- a permutation of the vertices.

With this relation, the set of equivalence classes of representations that can be obtained from the initial representations with legal moves will be denoted by $\mathscr{R}_1$.

We now define the term *canonization* : this is the choice of a single representation in each equivalent class. This choice should be as simple as possible, so we'll choose the minimal representation for the lexicographical order.

Our theoretical program is now more efficient, as we need to store and compute significantly less positions. We'll explain in paragraph 2.6 why using canonization doesn't change the result of the game. But the running time is still quite problematic, as the canonization is a very long procedure. We will do two types of improvements for this, and this is the subject of the following paragraphs.

2.4. **Slimming equivalence classes.** We will describe in this paragraph all the transformations that will decrease the number of representations in the equivalence classes, without changing the number of classes. So, the game tree is invariant under these transformations; their interest is to make the canonization faster.

Here's the list of the transformations we use :

- replace the vertices that only occur once, in a boundary with a single vertex, by the generic vertex "`0`".
- replace the vertices that only occur once, in a boundary with several vertices, by the generic vertex "`1`".
- replace the vertices that occur twice, in a single boundary and consecutively by the generic vertex "`2`".
- for the other vertices that occur twice, use a lower-case letter for those that occur twice in the same boundary, and use an upper-case letter for the others, those that appear in two different regions.
- cut the representation in independant subgames, named *lands*. The end-of-land character is "`]`".

Remark 2.5. The lexicographical order chosen in our program is :

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

Our theoretical program is now more efficient. Nevertheless, it's still rough. `AB.}ABC.}ABC.}]!` is a final representation, and instead of storing it, like our theoretical program currently does, we should merge it with the others final representations. The next paragraphs describes such improvements.

2.5. **Merging equivalence classes.** The following transformations have an influence on the number of representations in the equivalence classes, but they especially reduce the number of equivalence classes. They merge some classes, so they modify the game tree, reducing the number of nodes. And we'll store less representations after these transformations.

- delete the vertices that occur 3 times in the representation.
- delete empty boundaries and dead regions (with 0 or 1 life). After having deleted a dead region, replace the upper-case letters that occur only once by the generic vertex "2".
- identify the representations similar modulo a change of the orientation in one or several regions.
- regions equivalences.
- using $\mathscr{R}_1$ and canonization.

We begin with giving an example of what we mean by "merging the equivalence classes", with the first transformation, deleting the vertices that occur 3 times : before using this transformation, the two representations `1b1ba1ab.}]!` and `11a1a.}]!` are in two different equivalence classes, and they are in the same after the transformation.

Last point of the enumeration, the transformation that consists in using $\mathscr{R}_1$ and the canonization is of the same type : the previous equivalence classes were composed with a single representation, and after this transformation, the classes are the elements of $\mathscr{R}_1$.

What we called "region equivalences" is a very useful trick in our program : when a region has 3 lives or less, we have an equivalent game if we regroup the boundaries. For instance, the region `A.BC.}` becomes `ABC.}`. It considerably reduces the number of stored representations.

Let's come back to figure 3 : after all these simplifications, the representation of this position is now `0.1ab1bc2ca.ABC.}0.2ABC.}]12.AB.}AB.}]!`

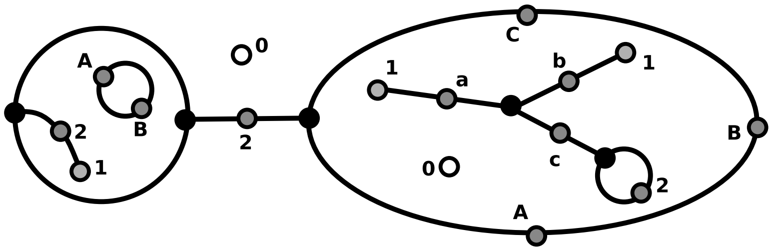

FIGURE 5. After 10 moves in a 11-spot game, simplified form.

Notice that in the outside region, we change the orientation, so that the canonized representation of this region is `0.2ABC.}]`, which is lower than `0.2CBA.}]` for lexicographical order.

We will explain in the next paragraph why merging these classes is authorized, ie why it doesn't change the result of the game, and under which conditions.

2.6. **Merge condition.** We can consider that we have a sequence of equivalence classes, $\mathscr{R}_0$, $\mathscr{R}_1$,...,$\mathscr{R}_n$,... A transformation that merges some equivalence classes will provide a projection application, for instance f : $f : \mathscr{R}_n \to \mathscr{R}_{n+1}$.

Let's call C_n the function that returns the children of an equivalence class of $\mathscr{R}_n$. So $C_n : \mathscr{R}_n \to \mathscr{P}(\mathscr{R}_n)$ where $\mathscr{P}(\mathscr{R}_n)$ is the powerset of $\mathscr{R}_n$. Graphically, if a node on the game tree is given, this function returns the nodes that are linked to it on the lower stage. We also have C_{n+1} for the children of $\mathscr{R}_{n+1}$.

The condition that any projection application must verify to assure that it is equivalent to play the game with $\mathscr{R}_n$ or with $\mathscr{R}_{n+1}$ is : for each element $x \in \mathscr{R}_n$, we have $f(C_n(x)) = C_{n+1}(f(x))$. The consciencious reader ought to check that these conditions are verified for the above transformations.

3. Main algorithm

3.1. **Win/Loss algorithm.** The standard recursive algorithm to compute the Win/Loss of a position can be described as follows :

Algorithm 3.1. *function compute-win-loss(position P)*

- *compute the children of P*
- *for each child, do : if the child is a Loss, return "P is a Win"*
- *return "P is a Loss"*

The core of our main algorithm uses this classical procedure, figure 6 shows how it works with the 2-spot game :

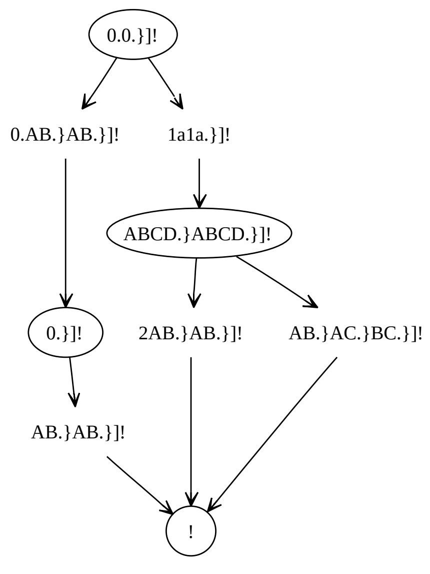

Figure 6. 2-spot game.

Surrounded positions are the losing ones. We see on this graph that a position only needs to have a losing child to be a Win, but all its children have to be a Win if the position is a Loss.

3.2. **Nimber theory.** The nimber will be helpful in our computations as many positions can be broken in several lands, and thanks to the nimber, we can replace the computation of P by the computation of these smaller positions.

Definition 3.2. 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 quickly implies that $[P] = 0$ if P is a Loss, and $P \geq 1$ if P is a Win. A less trivial but important result follows :

Theorem 3.3. *If a position P is made of two independant positions P_1 and P_2, then $[P]$ is the "bitwise exclusive or" of $[P_1]$ and $[P_2]$, which is denoted by $[P_1] \wedge [P_2]$.*

For a demonstration of this result, see for instance [4].

Here's how we use this notion in the program : instead of studying the Win/Loss of a position, we will now study the Win/Loss of a couple (position,nimber). The Win/Loss of ($\emptyset$,n) is "Loss" if $n = 0$, "Win" if $n \geq 1$. The original position P_0 will be replaced by the couple (P_0,0).

We notice that "(P,n) is a Loss" means that $[P] = n$. As we only store losing positions in our program, it now means that we'll only store positions whose nimber is known (and the winning positions that we don't store correspond to positions whose nimber is only known to be different of certain values).

To determine the Win/Loss of a couple, we can still use the algorithm 3.1 with this alternative : the children of a couple (P,n) are the children of the position part, whose form is (child(P),n), and the children of the nimber part, whose form is (P,m), with $m < n$.

3.3. **Nimber computation.** Algorithm 3.1 allows us to determine if a position P has n for nimber, but if not, we may still not know the nimber $[P]$. To determine this nimber, here's our (simple but efficient) method :

Algorithm 3.4. *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.

3.4. **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 $[P_2]$, theorem 3.3 shows that the Win/Loss of $(P_1]P_2]!, n)$ is the same that the Win/Loss of $(P_1]!, n \wedge [P_2])$.

So when our main algorithm meets a position with several lands, it computes the nimber of every land but one with the algorithm 3.4, and merges them with the nimber part of the couple; the remaining land is the one with the biggest number of lives. Consequently, we can store only lands in our database.

3.5. **Main algorithm.** With the previous ideas, algorithm 3.1 is now :

Algorithm 3.5. *function compute-win-loss(position P, nimber n)*

- *for lands whose nimber is already stored in the database, merge their nimber with the nimber part (with bitwise xor).*

- *compute nimber for every unknown land with algorithm 3.4, except for the land with the biggest number of lives, and merge those nimbers with the nimber part.*
- *compute the children of (P,n) (of position part and children part) :*
  - *if any child is losing, return "Win"*
  - *if every child is winning, store (P,n) in the database and return "Loss"*

Figure 7 shows how it works with the 3-spot game :

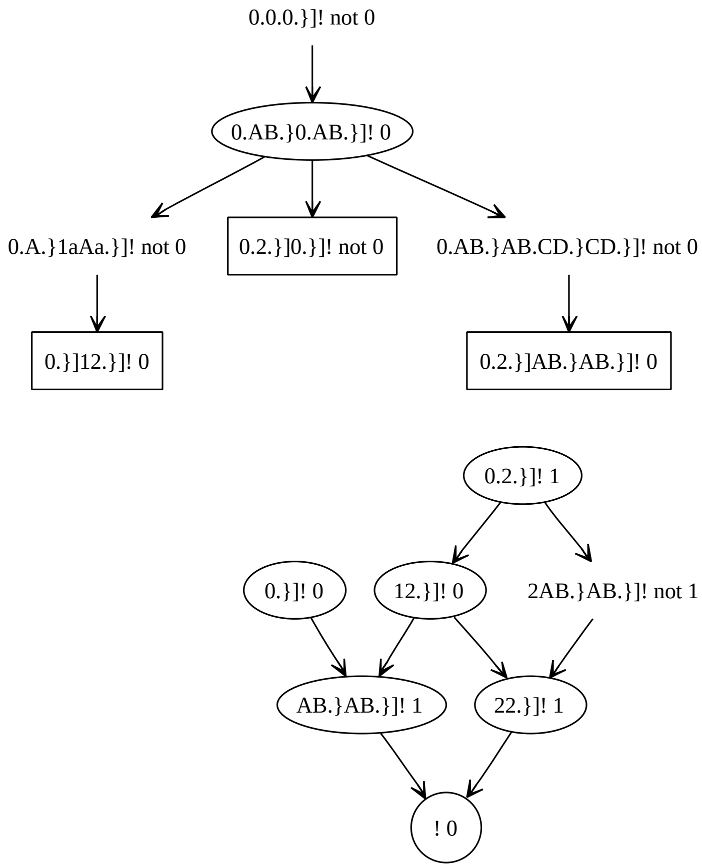

FIGURE 7. 3-spot game.

Couples surrounded by a rectangle are those that are made with 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 ones, only those couples are stored by the program. All their children are computed to find their position's nimber.

The remaining couples are winning ones, we don't know their position's nimber. We only know one (or sometimes several) nimber that is not the right one, and we know this because a child of the position is known to have this nimber.

4. Computation improvement methods

4.1. **Storing the positions.** We must store as few positions as possible, in order not to exceed the memory capacity. We'll explain a few tricks we use to complete this objective.

As in [1], we decided to store only losing couples : as the complexity of the computation increases, losing couples become less frequent, so this choice allows us to reduce the size of the database, in exchange of a bigger running time. The profit is variable, but we can consider that the size of the database is divided by a factor between 5 and 20, which isn't negligible.

So we store only losing couples, ie positions with a single land, and their nimber which is known. It is possible to export databases as text files, so that we can study them manually, or we can resume later an uncomplete computation.

4.2. **Pseudo-canonization.** We can't easily perform the true canonization described in section 2, mainly because choosing the name of the upper-case letters costs too much running time. So we only perform a pseudo-canonization : the same position can have several representations, for example, `0.AB.CD.}0.AB.}CD.}]!` and `0.AB.CD.}0.CD.}AB.}]!` represent the same position.

If a land has k upper-case letters, performing the true canonization consists in choosing the minimal representation for lexicographical order amongst the $k!$ possible representations. Our pseudo-canonization algorithm only rename these upper-case letters from A in the order of their appearance, and then sort the representation. Experience showed that performing this operation twice is most efficient both for running time and memory usage. Ultimately, we only lose a few percents of memory compared to a true canonization, while a true canonization has a time complexity that prevents from computing n-spot games for $n \geq 5$ or 6.

We have developed the complete graph of the n-spot games, for $n \leq 6$ in order to evaluate the performance of our pseudo-canonization. Notice that its performance (comparatively to true canonization) is proportionally better in a real computation that in this complete graph development, because in a real computation we meet positions easier to compute, as they have less upper-case letters. Here's an extract of the database of the complete graph of the 6-spot game :

```
0.ABC.}0.ABDEF.}CD.}EF.}]! 6
0.ABC.}0.ABDEF.}CF.}DE.}]! 6
0.ABC.}0.BCDEF.}AD.}EF.}]! 6
0.ABC.}0.BCDEF.}AF.}DE.}]! 6
0.ABC.}0.BCFED.}AD.}EF.}]! 6
```

These strings represent the same land, a true canonization would have displayed only one representation instead of these five. But if we compute the 6-spot game, we don't need to compute this land.

The following table gives the number of lands met after a complete graph development for the n-spot game.

n	number of lands
2	18
3	146
4	1724
5	26320
6	459194

4.3. **Children exploration.** When a position is a losing one, there is no short way to prove it : we need to compute all the children and prove they all are winning. But when a position is a winning one, we only need to find one losing child, and of course the faster we find it, the better. The strategies used to find this losing child quickly have a very strong influence on the program's efficiency. Here, we need to emphasize a property of sprouts game trees : they are unbalanced, with areas far much complicated than other ones. It means the difficulty of win/loss computation varies a lot for the children of a given position. As a consequence, the best way to minimize the amount of computation done before finding a losing child is to order children in their order of difficulty.

We can't know the real difficulty of a computation before doing it, but it is possible to evaluate it with only the position representation. The rules used to evaluate difficulty and order the children are a critical part of the program, because a bad set of rules can push the computation in complicated parts of the tree. Our current set of rules is (the first are the most important) :

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

We only estimate the number of children as a whole computation of those children would cost a lot of running time. Here's how we do this :

- 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 distribute 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 $(number\ of\ vertices\ of\ B_i) \times (number\ of\ vertices\ of\ B_j)$.

These choices are far than being perfect; but we think that it would be necessary to develop very different rules from the current ones to improve them significantly. It is also important to note these rules are designed to be globally effective. Globally effective rules have sometimes a bad effect on the first levels of some games, so that it is difficult to evaluate the efficiency of the rules.

4.4. **Trace system and children selection.** During the computation, the program displays the list of the currently studied couples : with our algorithm, at each instant, the program is working to determine the win/loss of a list of couples, each one being the child of the previous. For example, here's what the program could display during a 12-spot game computation :

position	nimber
`0.0.0.0.0.0.0.AB.}0.0.0.0.0.0.AB.}]!`	0
`0.0.0.0.0.0.}]0.0.0.A.}0.0.0.A.}]!`	0
`0.0.0.0.0.0.}]!`	1
`0.0.0.0.0.AB.}AB.}]!`	1
`0.0.0.0.0.}]!`	1
`0.0.1a1a.}]!`	1

With a click on the interface, it is possible to choose another child on a given level, and so to study another part of the game tree. For example, we could decide to study the couple (`0.0.AB.}0.0.AB.}]!`,1) in the fourth level. But if the program is performing the algorithm 3.4, the corresponding level is printed in yellow in the interface, and it is impossible to click on this level, as it would be useless (this is the case for the third level on the above example).

We empirically decide 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 as a loss (which means that almost all its children have been computed as winning), it is often efficient to click two levels above, to try to quickly find losing children for its 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 in lands).

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

4.5. **Check computation.** User interactions proved 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 stand-alone check computation, which uses the previous results only to guide itself in the game tree. This check computation allows no interaction to 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 (all the winning children of winning couples are usually useless). When the check computation meet a new couple, it uses the previous computation to know whether it's a losing or a winning one. If it's a losing couple (ie a position whose nimber is known), we compute again all its children to check it. But if it's a winning couple, by using the previous database, we only study one of its losing children.

If a position consists of several lands, we compute those lands separately. The nimber is known for every land (except perhaps one), so, for a given land, we decided to immediately check the couple with the right nimber instead of trying 0, 1, 2... like in the algorithm 3.4.

This check computation reduces the size of the databases, providing hand-checkable proofs for the n-spot games with little values of n. For bigger values of n, it's a way of avoiding to excess the memory size.

During the check computation, we also generate a file, compatible with the graph visualization tool Graphviz[1], so that we'll be able to easily draw graphs that describe how the computation proceeds. See for example figures 6 and 7. For big values of n, the graph would be rather unreadable, so we can choose to plot only the positions with a minimal number of lives. Here's the beginning of the graph for the 5-spot game, with only the positions with 12 lives or more displayed :

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

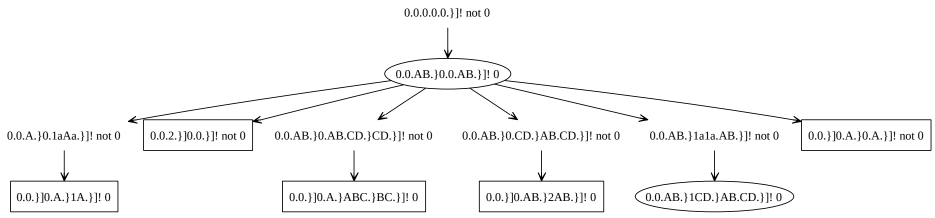

FIGURE 8. A solution for the 5-spot game, positions with 12 lives or more.

Figure 8 shows how cutting positions in lands simplifies the computation. The most spectacular example of this behavior is the case of n=17 : in our computation, after 3 moves from the starting position (which has 51 lives), every position is cut in several lands that have less than 27 lives.

For bigger graphs, we can plot only numbers instead of whole positions. See for instance this graph for the complete 4-spot game (numbers are meaningless, but we can see in the generated file whose position they are corresponding to).

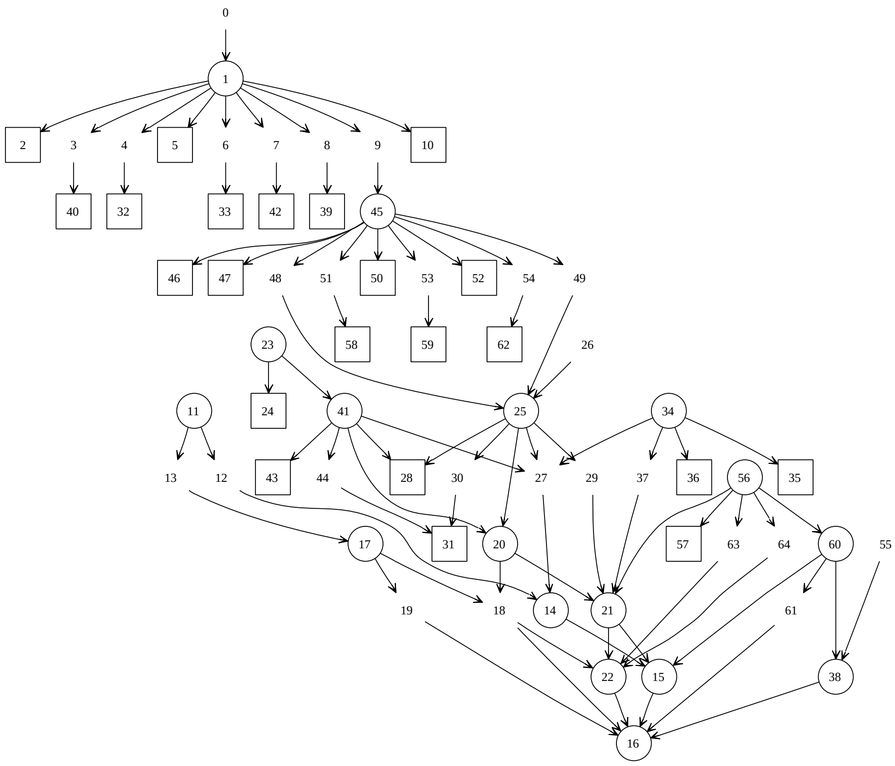

FIGURE 9. A complete solution for 4-spot game.

The reader will find the positions corresponding to those nimbers in the appendix.

5. Results

5.1. **n-spot games already computed.** The following table shows the number of couples stored in the database after a check computation, for the n-spot games computed up to now. All the results are compliant with the conjecture emitted in [1] : first player loses if and only if n is 0, 1 or 2 modulo 6.

n	size	n	size	n	size	n	size	n	size	n	size
0	0	6	64	12	498	18	4058	24	29403	30	?
1	1	7	105	13	597	19	4648	25	29634	31	?
2	3	8	240	14	1986	20	4683	26	25604	32	?
3	6	9	66	15	3763	21	29034	27	?	33	?
4	16	10	142	16	1105	22	6161	28	17431	34	33588
5	38	11	142	17	524	23	2139	29	4017	35	24746

It is rather surprising that the number of stored positions isn't increasing with n. Nevertheless, some patterns seem to occur modulo 6.

More precisely, $n = 15$ and $n = 21$ were really hard to compute, because they are winning and only have one losing child which is the most complicated one (its representation is `0.0.0.(...)0.1a1a.}]!`). $n = 27$, the first still unknown, seems to follow the same path.

At the opposite, $n = 17$, $n = 23$, $n = 29$ and $n = 35$ are really easier to compute. They are winning positions, and the child obtained by linking one dot to itself and separating the remaining dots in two equal sets is losing (this child is often the easiest to compute).

We can also notice that the number of stored positions is almost the same for $n = 15$ and $n = 18, 19$: once the result of $n = 15$ is known, we can quickly deduce the result for those values of n. The same phenomenon occurs for $n = 21$ and $n = 24, 25$.

5.2. **Nimber conjecture.** We observed that the nimber of the winning starting positions, for $n \leq 23$, is 1, so we can propose a stronger conjecture than the "Sprouts conjecture" emitted in [1] :

Conjecture 5.1.
The nimber for the starting position with n spot is 0 *if n is* 0, 1 *or* 2 *modulo* 6, *and* 1 *if n 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 n 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.

We also imagined another extension of the Sprouts conjecture : we supposed that if you added 6 boundaries "`0.`" into a single domain, it wouldn't change the nimber of the position. This result is wrong, as `0.22.}]!` has nimber 0, and `0.0.0.0.0.0.0.22.}]!` has nimber 2. But it works for more than 90% of the positions we tried. The existence of such nearly-true patterns tends to infirm the Sprouts conjecture.

5.3. **Hand-checkable proofs.** Using a computer to determine mathematical results isn't 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 a hand-ckeckable proof of our results.

We printed the result of a check computation for the 9-spot game and manually checked 66 losing positions (we checked that we obtained the same sets of children than our program), and 258 winning positions (we had to check only one child for these positions, very seldom two or more). This took us a few hours.

The table in paragraph 5.1 implies that we could use this method to check the results in a reasonable amount of time for the n-spot game with $n \leq 11$. But the program can also head in the right direction somebody who would like to create a totally manual proof, like the one in [2].

Conclusion

We still could imagine many improvements for this program. First of all, it would be interesting if it could automatically choose which branch of the game tree to explore, as currently, the main obstacle for computing higher n-spot games is neither memory nor computation time, but rather human time for clicking.

Accordingly, distributed computing would probably help too to compute higher n-spot games, and the check computation, by reducing the size of the databases, would be really efficient for this.

Sprouts computation is far from being completed, and it is likely that in the next years, we'll know who wins in the Sprouts game when starting with more than fifty spots.

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

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

[4] Daniel Sleator, *Finite two-person deterministic games, impartial games, nimbers*, 2000, http://www.cs.cmu.edu/afs/cs/academic/class/15859-f01/www/notes/sleator-nim-notes.txt.

Licence

Appendix : Correspondence for Figure 9

n^o	position	nim
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

n^o	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.}]!	$\neq 1$