Computer Analysis of Sprouts

David Applegate[1] Guy Jacobson[2]
Daniel Sleator[1]

May 1991
CMU–CS–91–144

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[2]AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

Supported by the National Science Foundation under grant CCR-8658139.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the NSF or the U.S. government.

Keywords: Computer game playing, planar graphs, hashing, game theory

Abstract

Sprouts is a two-player pencil-and-paper game with a topological flavor. It was invented in 1967 by Michael Paterson and John Conway, and was popularized by Martin Gardner in the Mathematical Games column of Scientific American magazine [6].

We have written a computer program to analyze the n-spot game of Sprouts for general n. Our program uses a number of standard techniques to expedite adversary searches such as cutting off the search as soon as the value can be determined, and hashing previously evaluated positions. But the truly innovative feature is our representation of game positions, which provides enough information to generate moves and has the property that many different planar graphs collapse into the same representation. This has an enormous impact on the speed of the search.

The complexity of n-spot Sprouts grows extremely rapidly with n. According to Gardner [7, page 7], Conway estimated that analysis of the eight-spot game was beyond the reach of present-day computers. Before our program, even the value of the seven-spot game was unknown; we have calculated the value of all games up to and including eleven spots. Our calculation supports the *Sprouts Conjecture*: The first player loses if n is 0, 1 or 2 modulo 6 and wins otherwise.

1. Introduction

Sprouts is a popular[1] two-person pencil-and player game. It was invented in Cambridge in 1967 by Michael Paterson, then a graduate student, and John Horton Conway, a professor of Mathematics.

Sprouts is perhaps the most well-known and widely played of the topological pencil-and-paper games discussed by Berlekamp, Conway, and Guy in *Winning Ways* [3]. This popularity is largely due the exposure provided by Martin Gardner in his *Mathematical Games* column in the July 1967 issue of *Scientific American*. The game is listed by David Pritchard, editor of *The Gamer* magazine, in his compilation of the world's best games for two [10]. Sprouts also occupies a prominent place in Piers Anthony's science fiction novel *Macroscope* [1], where the hero's dormant genius is awakened by his exceptional skill at the game.

The initial position of the game consists of a number of disconnected points called spots. Players alternate connecting the spots by drawing curves between them, adding a new spot on each curve drawn. Each curve must be drawn in the plane without touching itself or any other curve or spot (except at the end points). A single existing spot may serve as both endpoints of a curve. Furthermore, a spot may have a maximum of three parts of curves connecting to it. (Its degree is bounded by three.) A player who cannot make a legal move loses. Figure 1 shows a sample game of two-spot Sprouts, with the first player winning. Since draws are not possible, either the first player to move or the second player can always force a win, regardless of the opponent's strategy. Which of the players has this winning strategy depends on the number of initial spots.

It might seem at first that a game of Sprouts could go on forever, as new spots are created each move. Conway gave a simple argument showing that this cannot happen. Each spot has a certain number of *lives* that represent potential connections. The original spots in the initial positions each have three lives, and the spots formed subsequently have only one, since they come into existence with two connections. When a spot has no remaining lives, it is *dead*, and cannot participate in subsequent moves. Each move destroys two lives (one for each endpoint connection) and creates one life (in the new spot), so the total number of lives in the game decreases by one each turn. The number of lives in an n-spot game is initially $3n$, and the number of lives cannot decrease below one, since after any move there is always at least one life (the one created by the move). This gives a upper bound of $3n - 1$ moves in the n-spot game. Conway and Denis Mollison also show that the n-spot game must last at least $2n$ moves [3].

Sprouts is an example of what Conway [4, page 122] calls an *impartial* game: the same set of moves is available to both players, and the last player to make a legal move is the victor. Games in this class have variants where the condition of victory is inverted: the winner is the player who cannot make a legal move. This is called the losing or *misère* version of the game.

[1] at least in academic circles.

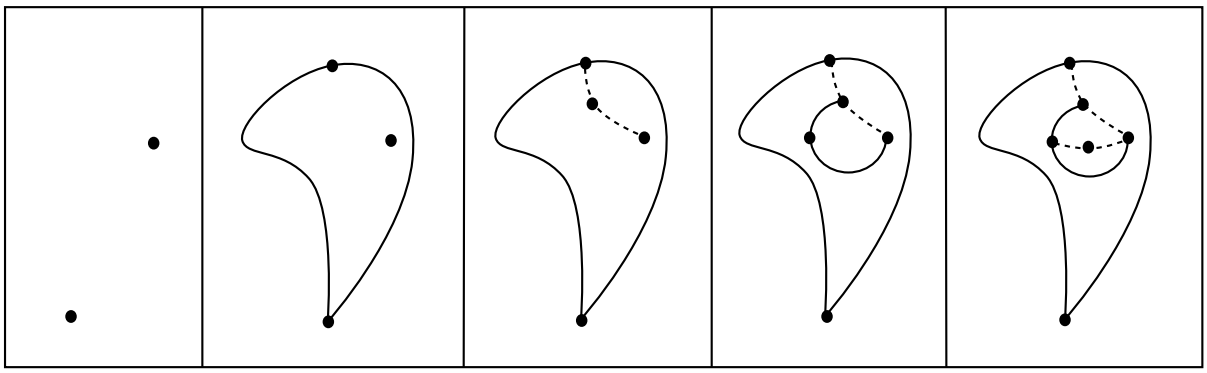

Figure 1: A sample game of two-spot Sprouts

number of spots	1	2	3	4	5	6	7	8	9	10	11
normal play	L	L	W	W	W	L	L*	L*	W*	W*	W*
misère play	W	L	L	L	W*	W*	L*	L*	L*		

W denotes that the game is a win for the first player;
L denotes a loss for the first player.
A "*" indicates new results obtained by our program.

Table 1: Currently known outcomes of n-spot Sprouts

1.1. The values of Sprouts positions

While Sprouts has very simple rules, positions can become fantastically complicated as n grows. Each additional spot adds between two and three turns to the length of the game, and also increases the number of moves available at each turn significantly. Games with small numbers of spots can be (and have been) completely solved by hand, but as the number of spots increases, the complexity of the problem overwhelms human powers of analysis. The first proof that the first player loses in the six-spot game, performed by Mollison[2], ran to 47 pages. Table 1 shows all the currently known values of the game. Notice that even the five-spot misère game had eluded human analysis.

Conway averred that the analysis of seven-spot Sprouts would require a sophisticated computer program. We have written such a program. Conway further claimed that the analysis of eight-spot Sprouts was far beyond the reach of present-day computers [6]. That may have been true when he said it in 1967, but it is true no longer. Using our program we have found the values of all normal games up to eleven spots, and all misère games up to nine spots, as shown in the table.

The n-spot Sprouts positions evaluated so far fall into a remarkably simple pattern,

[2] to win a 10-shilling bet!

characterized by the following conjecture:

> **Sprouts Conjecture:** The first player has a winning strategy in n-spot Sprouts if and only if n is 3, 4, or 5 modulo 6.

The data for misère Sprouts fit a similar pattern.

> **Misère Sprouts Conjecture:** The first player has a winning strategy in n-spot misère Sprouts if and only if n is 0 or 1 modulo 5.

These patterns have persisted over too many values of n to be explained away by "the strong law of small numbers" [8]. It remains an interesting challenge to prove these conjectures.

1.2. The Sprouts program

As far as we know, our program is the only successful automated Sprouts searcher in existence. A fair amount of sophistication was required to achieve sufficient time- and space-efficiency to solve the larger games. Most of this paper is devoted to describing our program. Our program is successful for several reasons:

¶ We developed a very terse representation for Sprouts positions. Our representation strives to keep only enough information for move generation. Many seemingly different Sprouts positions are really equivalent. The combination of this low-information representation and hashing (whereby the results of previous searches are cached) proved to be extremely powerful.

¶ Many Sprouts positions that occur during the search are the *sum* of two or more non-interacting games. Sometimes it is possible to infer the value of the sum of two games given the values of the subgames. Our program makes use of these sum identities when evaluating normal Sprouts. These ideas are not nearly as useful in analyzing misère Sprouts. This is the principal reason that we were able to extend the analysis of the normal game farther than the misère game.

¶ We used standard techniques to speed adversary search, such as cutting off the search as soon as the value is known, caching the results of previous searches in a hash table, and searching the successors of a position in order from lowest degree to highest.

¶ The size of the hash table turned out to be a major limitation of the program, and we devised and implemented two methods to save space without losing too much time efficiency. We discovered that saving only the losing positions reduced the space requirement by a large factor. To reduce the space still further we used a data compression technique.

The most innovative feature of our program is the representation we use for Sprouts positions, which is described in section 2. The searching and hashing methods we used, and the mathematics of game sums and how we took advantage of this in analyzing the normal game are described in section 3. Section 4 contains a summary of the time and space spent on the searches. The final section contains a discussion of our results, and ideas about how the program might be improved.

2. The position representation

A Sprouts position is a planar graph (possibly disconnected) along with information about how this graph is embedded in the plane[3]. It is easy to use this observation to devise a computer representation of the game. The resulting representation is in one-to-one correspondence with pencil-and-paper Sprouts positions (taking isomorphism into account). This idea leads to a correct program for evaluating Sprouts positions. In this section we develop a computer representation that is vastly more efficient than such a graph representation.

Positions whose graphs are isomorphic (and have the same planar embedding) are "the same," and must have the same value. It turns out that there is a high degree of folding realized from this topological symmetry, and this is evident from figure 2 (adapted from *Winning Ways* [3]). This figure shows a complete tree of the two-spot game with isomorphic positions identified.

In a computer game tree search the phenomenon of folding is exploited by saving the values of positions that have been analyzed in a hash table. When a position that has already been evaluated is encountered again, its value is simply recalled from the hash table, and a costly search is avoided. The amount of time saved depends on how astute the program is in in recognizing when a position is the same as one previously encountered. The more frequently it can say "I have seen this position before" or even "I have seen a position whose value must be the same as this one," the fewer positions have to be examined, and the speedier the search will be. Thus, the crucial property of a representation is the extent to which seemingly different positions are folded into the same representation.

To devise a more efficient representation it is necessary to ask what properties of a representation are essential for correctly analyzing the game. Being able to reconstruct the picture of the state of the game is *not* essential. What is essential is the *move generation property*. A representation scheme is said to satisfy the move generation property if given a representation r (in the scheme) of a position s, it is possible (without knowing s) to generate the set of representations (in the scheme) of the successors of position s.

Such a representation scheme in effect defines another game, a game that can be played without reference to the actual positions, only using the representations. *The move generation principle* states that the value (whether it is a win or a loss for the first player) of the original position in the original game is the same as the value of the representation of that

[3]This is discussed at some length by Draeger et al. [5].

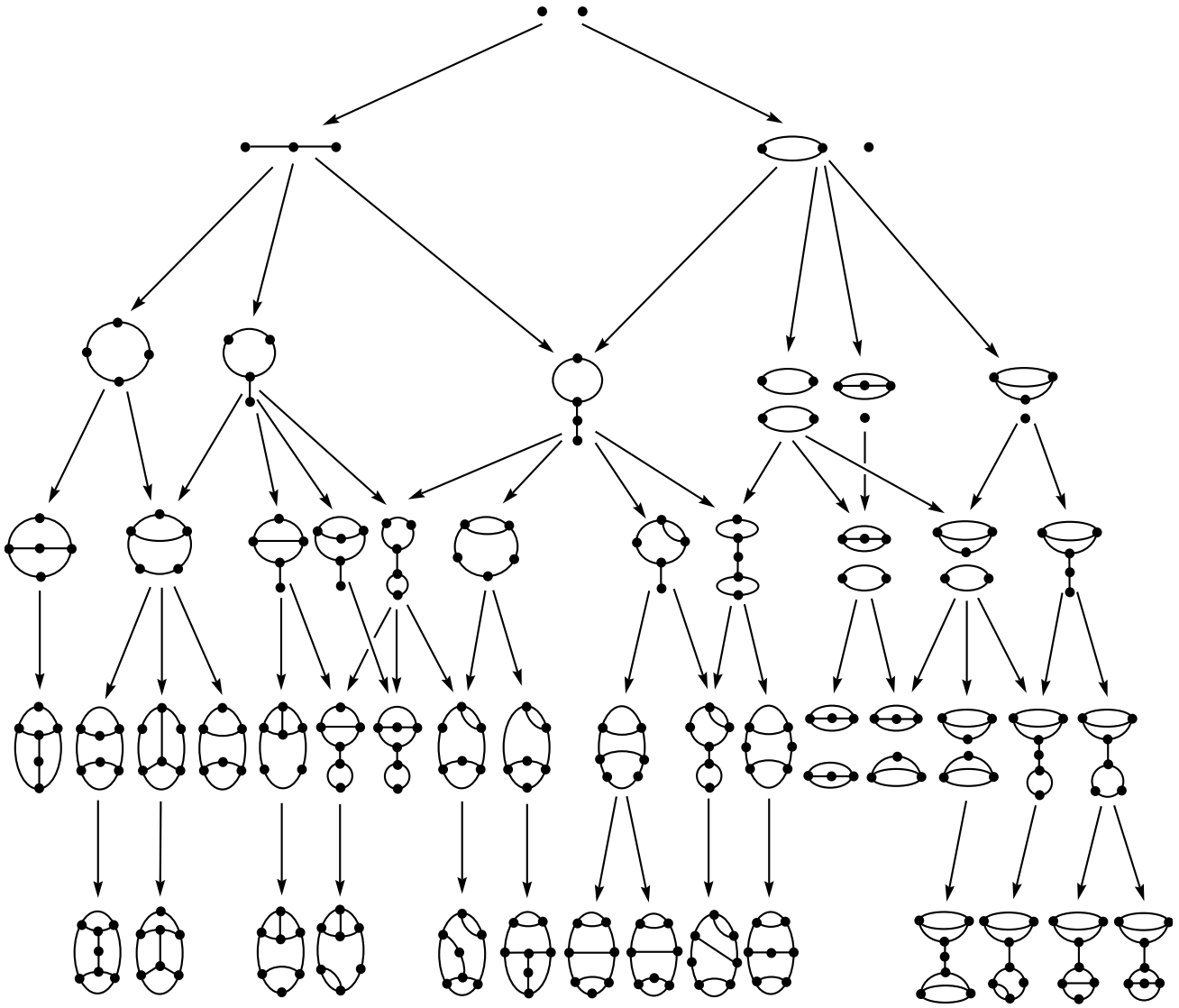

Figure 2: The complete game tree for two-spot Sprouts

position in the new game. The proof of this principle is a straightforward induction. Notice that the number of successors of a position may differ from the number of successors of the representation of the position.

We shall develop our representation in three stages. We call the three representations the *set representation* the *string representation* and the *canonical representation*. At each stage we show how to take a position in the previous representation, and generate its new representation. At each stage the new representation satisfies the move generation property with respect to the previous representation. We shall omit rigorous proofs that the move generation property is satisfied, because such proofs are tedious and unilluminating.

Our canonical representation is remarkably successful at folding positions. For example using the planar graph representation of figure 2 required that 51 positions be evaluated. Using our canonical representation only 20 different positions need to be considered, as shown in figure 3.

Because move generation is so crucial, it is useful to begin with a discussion of it.

2.1. Move generation

Let us begin with some definitions. The curves of a Sprouts position divide the plane into a set of connected components called *regions*. The subgraph induced by spots and lines on the boundary of a region form a number of connected components, which we will call *boundaries*. A region is said to contain the boundaries that touch it. So each region contains one or more boundaries, and these boundaries may share spots with boundaries in other regions.

A move may connect two spots that lie on the same boundary (these we call *one-boundary* moves), or it may connect two spots that lie on different boundaries (these are the *two-boundary* moves). Figure 4 illustrates these two different types of moves.

In a two-boundary move a spot x on a boundary B_1 is connected to a spot y on a different boundary B_2, and a new spot z is placed on the curve. This unites B_1 and B_2 into a single new boundary B. No new regions are formed, and no other boundaries are affected by the move. To identify a two-boundary move it sufficient to specify the starting and ending spots within their respective boundaries. (It is not sufficient merely to specify the starting and ending spots, because there may be two different ways for a move to connect to a spot of degree two. In this situation the spot occurs twice in the boundary.) Notice that the path by which the curve weaves among the other boundaries of the region is irrelevant. All these possible paths all lead to equivalent Sprouts positions.

In a one-boundary move, spots x and y, both on the same boundary B in region R are connected by a curve (and a new spot z is placed on that curve). This forms a cycle which partitions R into two new regions, which we shall call R_i and R_o, (the $i$nside and the $o$utside). Boundary B is split into two boundaries, B_i, lying in R_i, and B_o, lying in R_o. The boundaries in R besides B are not affected, but they end up in either R_i or R_o, depending on how the curve from x to y is drawn. A full description of a one-boundary

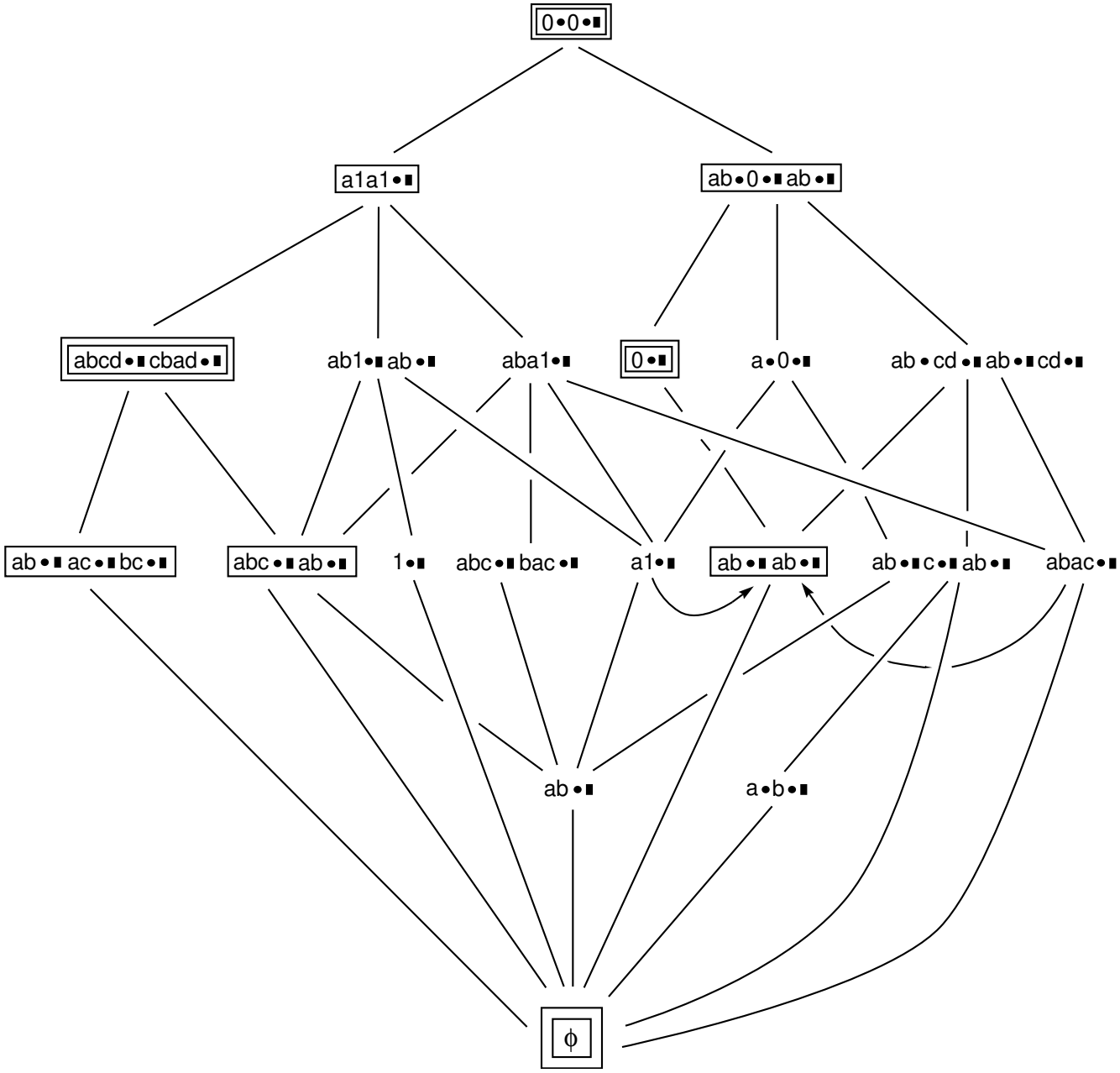

Figure 3: The entire game tree for 2-spot Sprouts using our representation, and assuming ideal canonization. Our search algorithm, using move ordering by branching degree, would evaluate only the nodes shown in boxes. The nodes in single boxes are wins, while those shown in double boxes are losses. The symbol "ϕ" represents the empty position.

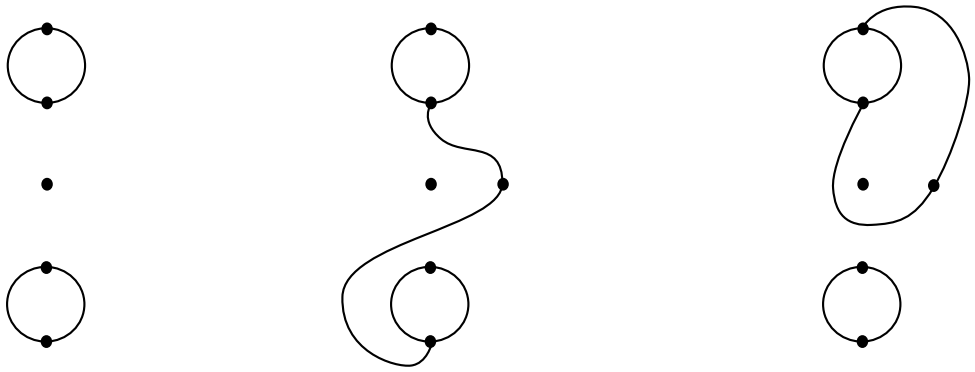

Figure 4: A position, a two boundary move, and a one-boundary move.

move therefore requires more than just an identification of x and y, and their positions in their boundaries. Such a description must also specify which boundaries in R are to lie in R_i after the move, with the others (besides B itself) to lie in R_o. If a one-boundary move is made in a region with k boundaries, there are 2^{k-1} variants of the move starting and ending at the same boundary spots.

2.2. The set representation

We shall explain the set representation by giving a procedure to take a Sprouts position and generate the set representation for it. To get started we assume that each spot has a unique name. The representation for a boundary b of the Sprouts position is a circular list of spot names encountered as we proceed around the boundary. The direction chosen is the one that would be used by a miniature robot within the region walking forward around the boundary with its right (or left) hand on the boundary (the handedness of the robot must be constant within each region, but is free to vary between regions). The representation of a region is a multiset which is the union of the representations of the boundaries of the region. The representation of a position is the multiset which is the union of the representations of the regions in the position. (The degree of each spot can be inferred from this.) Figure 5 shows a position and its set representation.

It is easy to see how move generation is done using this representation. We shall not prove that the representation satisfies the move generation property. The following observations are the underlying reasons that it works.

¶ The plane is partitioned into a set of disjoint regions by the curves of any Sprouts position.

¶ Each move takes place entirely within one of the regions. This is so because the curve drawn is simple and may not cross any existing curves.

¶ As the game proceeds this partition is successively refined.

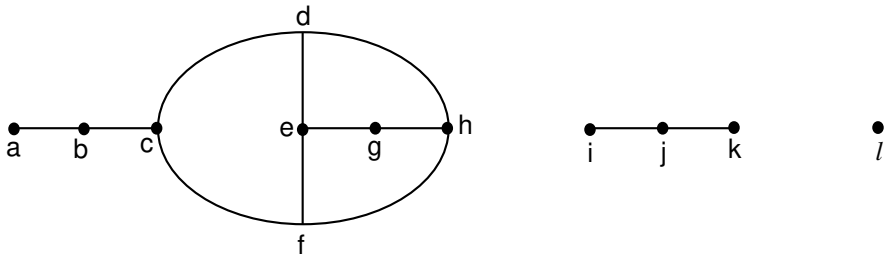

Figure 5: The set representation of this Sprouts position is: {{(abcdhfcb) (ijkj) (l)} {(cfed)} {(degh)} {(efhg)}}

¶ The only way that two regions can interact with each other is if there is a spot that is common to the boundaries of both. Other information about the way these regions are embedded with respect to one another is irrelevant with respect to generating the moves.

From now on we will not distinguish between the use of the word "region" to mean the set representation of the region, or region itself. The same applies to "boundary".

2.3. The string representation

Now we shall take the set representation of the previous section and translate it into a string of symbols. To represent a position as a string, we concatenate the representations of its regions (in arbitrary order). A region is represented by the representations of its boundaries (in arbitrary order) terminated with the end-of-region symbol (∎). A boundary is represented by the names of the spots encountered while walking around the boundary (starting from an arbitrary spot), terminated with the end-of-boundary symbol (•).

A few observations based on the move generation principle will be helpful:

¶ Spots of degree three can be thrown away. In other words, we simply skip such spots when listing the contents of a boundary. Although these spots are important topologically, they are dead and cannot participate in any further moves.

¶ Regions with fewer than two lives, and boundaries with no live spots can also be thrown away.

¶ Spots of degree zero or one do not appear on more than one boundary, so they do not need unique names. We can simply name them all `0` and `1`, respectively.

¶ If a spot of degree two has no live spots between its two occurrences on one boundary, it only needs to be listed once.

The spots of degree two are the only ones that require distinct names. We will conventionally name these with letters of the alphabet (`a`, `b`, etc.). Any spot labeled with a letter has degree two.

For example, the Sprouts position of figure 5 can be transformed by the following sequence of steps into "`1b•1j1j•0•■`" as follows. First we write it as a string and get:

```
abcdhfcb•ijkj•l•■cfed•■degh•■efhg•■
```

Eliminating spots of degree three we get:

```
abb•ijkj•l•■•■g•■g•■
```

Throwing away regions with fewer than two lives and boundaries with no live spots gives:

```
abb•ijkj•l•■
```

Renaming spots gives:

```
1bb•1j1j•0•■
```

Finally, eliminating the second adjacent occurrence of spots of degree two gives:

```
1b•1j1j•0•■
```

2.4. Canonization

In moving from the set to the string representation, we stripped out some useless information and linearized our structure. But there were a number of arbitrary choices made in the process:

- the names for spots of degree two. How did we decide which spot gets to be `a` and which is `b`?
- the order of regions.
- the order of boundaries within a region.
- the spot in each boundary where we start listing the spot names.
- the direction (clockwise or counterclockwise) for listing the spot names in each boundary. This choice must be consistent within each region, but can vary from region to region.

Because of these arbitrary choices a given Sprouts position has a set of possible string representations. Let there be a linear ordering defined over the strings (for example, lexicographic order). Then each set has a maximum element under this ordering; call this the *canonical* element of the set. The process of taking a string representation S and finding the canonical element of the set to which S belongs is called *canonization*. The resulting canonical representation satisfies the move generation property.

One way to perform canonization is to try all possibilities for each of the arbitrary choices made in conversion of a position to a string, and find the maximum under some ordering. This would be much too time-consuming. Instead, we perform the following pseudocanonization:

1. give each spot of degree two the temporary name `2`.
2. rotate each boundary until it is lexicographically least among all of its rotations.
3. sort the boundaries within each region.
4. sort the regions in the position.
5. rename the spots of degree two in alphabetical order from `a` as they appear (making sure that a spot gets its correct name the second time it appears).

The canonical representations of the first four positions of figure 1 are respectively: `0●0●■`, `ab●■ab●0■`, `ab1b●■`, `abc●■ab●■`. The fifth position (and any position with no moves) is represented by the empty string.

This procedure is not true canonization, since two positions that have the same canonical string may get different representations. (This comes about, for example, when there are ties in the sorting process, and there are spots of degree two that occur in different boundaries. Doing true canonization is an instance of a graph isomorphism problem.) Nonetheless, it does map the set of possible string representations of a position into a much smaller set. As long as we do not map two inequivalent positions into the same string, by the move generation principle, our program will function correctly.

3. Computing the value

In this section we describe the searching and hashing techniques that we used. For the benefit of the reader not familiar with the literature on such techniques, our description is self-contained.

3.1. Search algorithm

Since there are no draws possible in Sprouts, the value of each position is either a win or a loss for the player to move. In particular, a position P is a win for the player to move

if any one its successors (those positions that can be reached by making one move from P) are a loss for the player to move. The basic procedure (ignoring hashing) in the search is eval(P), which determines if position P is a win ("W") or a loss ("L"):

```
function eval(P)
        S ←successors(P)
        if S is empty then return "L"[4]
        for each position P' ∈ S do
                if eval(P') is "L" then return "W"
        return "L"
end eval
```

The function successors(P) takes a string representation of a position P and generates a set of (pseudocanonized) strings that represent the successors of P. This is accomplished by straightforward character-string surgery, with postprocessing to strip dead spots and regions and perform canonization. We omit the gory details of this surgery, noting that a large fraction of the program's text and execution time is spent therein.

The only important aspect of the search not yet specified is the ordering of successors tried in the for loop. In many game-playing programs, a heuristic evaluation function (that estimates the likelihood that a move is a win) is used to determine this ordering. By searching the likely winning moves first, we will soon find a successor that is a losing position, and thus get by evaluating fewer successors . (Of course, if the position P is a loss, then all its successors will be wins, and we will have to look at all of them no matter how we choose to order the set S.)

Good move ordering can drastically reduce search time. In many games it is easy to take a quick look at the position and arrive at a crude estimate of the prospects of a win. In chess, for example, we can use advantages in material to perform such an estimation. Finding a good heristic estimator in Sprouts seems to be quite hard. Until the game is near completion, it is hard to see why one move is better than another. David Pritchard, a game expert, says that until the final stages of the game, Sprouts is "...a mass of paths without a signpost in sight" [10].

Thus, rather than using a best-first ordering of the moves, we used a cheapest-first ordering. When we are looking for an object whose location is unknown, it is wise to look first in those locations where the object is likely to be found. But if we have no good way to estimate the *a priori* likelyhoods, we do well to look first in those locations which are easiest to search. (When you lose your car keys on a dark night it is best to search for them near a street lamp, because that is the only place where you have any hope of finding them!)

We estimated the work involved in evaluating a given position P' by counting the number of successors of P'. The justification for this heuristic (called *ordering by branching degrees* by Pearl [9, page 324]) is that those successors that have many successors themselves will

[4]In a misère game, if S is empty then return "W"

naturally take longer to search than those with fewer successors. The branching degree (number of different successors) can vary considerably from position to position. Figure 3 shows that what happens when our searching algorithm is run using this move ordering heuristic on two-spot Sprouts. Notice that only nine positions have to be evaluated.

Actually counting the true branching degree would require an expensive full move generation of each of the successors. Because of canonization, many different moves lead to the same successor position, so computing the actual degree would be costly. We compromise and count the number of different moves, rather than the number of different successor positions.

3.2. The hash table

So far we have ignored the hash table in describing the search. This was done to simplify the description; as evidenced by figure 2, the hash table is absolutely essential to the efficiency of the program. The hash table must be relatively efficient to access and it must be small enough to fit into main memory of the available computer.

In an early version of our program, the hash table stored all positions that had been evaluated and kept a value (win or loss) with each one. Before each recursive call to eval(P'), it looked in the hash table to see if the value of successor position P' was already known to us. A little experimentation showed that better performance is achieved if the table is first scanned to see if any successor P' was already known to be a losing position. If such a successor P' was found, the search terminates immediately. This savings more than pays for the cost of doing a complete move generation before any recursive calls to eval.

As we developed the program further, it became clear that the bottleneck in solving games with more spots was memory, not running time. As the number of positions grew into the millions, it became impossible to store the hash table in main memory, and secondary memory was too slow. We reduced the memory required for the hash table through two orthogonal methods: reducing the number of positions stored, and reducing the space required to store a position.

Examining the positions in the hash table, we discovered that the great majority were wins. In the larger games, the ratio of wins to losses was on the order of ten to one[5]. By storing only losing positions in the table, we achieve an order of magnitude savings in space.

---

[5]The following calculation indicates why there is a close relationship between the branching factor of the game (the number of successors of a position) and the fraction of positions that are losses. Generate a "random" game with branching factor b by the following process: Start with one position that is a loss. In the general step, create a new position P, and choose its b successors at random among the positions already generated. If all of the successors are wins, then P is a loss, otherwise it is a win. Let $\alpha(t)$ be the fraction of the total number of positions generated up to time t that are losses. With probability one, $\alpha(t)$ converges to a constant, which we call α. For large t the probability that a new position is a loss is α. This must also be the probability that all the successors of the newly generated position are wins, which is $(1-\alpha)^p$. So α satisfies the equation $\alpha = (1-\alpha)^b$. A value of $\alpha = .1$ corresponds to a branching factor of about 22. This roughly corresponds to the situation in nine-spot Sprouts.

Since the winning positions are exactly those that have a losing position as a successor, we are guaranteed to find the value for a winning position P that we have seen before on the very next level of the search, while we are scanning the hash table for successors of P. In other words, the price we have to pay for not storing winning positions is the hash table is an extra move generation every time we encounter a winning position we have seen before. And this price is partially compensated by the lower overhead (in execution time) of maintaining a smaller hash table. With a hash table that stores only losing positions, our search looks like this:

```
function eval(P)
        S ←successors(P)
        if S is empty then return "L"[6]
        for each position P' ∈ S do
                if P' is in the hash table return "W"
        for each position P' ∈ S do
                if eval(P') is "L" then return "W"
        put P into the hash table;
        return "L"
end eval
```

The other way to reduce the storage required by the hash table is to make the representation of positions even more succinct. The obvious way to proceed is use classical techniques of data compression: find a mapping between strings of symbols used in the representation and short bit patterns.

We used a macro-compression or dictionary encoding technique [2], mapping strings over the source alphabet into strings over a larger alphabet, called the macro alphabet. Each symbol in the macro alphabet stands for a particular sequence of symbols in the source alphabet. The source alphabet consists of the spot symbols `0`, `1`, and `a`...[7] and the punctuation symbols •(end of boundary), ∎(end of region) and a special symbol \$ marking the end of the string. The macro alphabet size was fixed at 256, a number chosen so that each symbol would fit snugly into an 8-bit byte.

To insure that every source string has a representation as a string of macro symbols, there is a macro symbol for each individual symbol in the source alphabet. The other macro symbols stand for sequences that occur frequently in position strings; we used a large file of random positions from the seven-spot game as training data to chose them. This choice of sequences was done automatically by a program that expanded the macro alphabet from an initial set of symbols corresponding only to the singleton strings in the source alphabet to its final size[8]. In practice, the translation from the source alphabet to the macro alphabet reduces the length of position representation by a factor of about four.

[6] In a misère game, if S is empty then return "W"

[7] We never need more than $2n$ letters in an n-spot game, since that is the maximum number of spots of degree two.

[8] More specifically, the process works as follows. Express the entire database using the current set of

The translation from source strings to macro strings is accomplished very efficiently by a finite-state transducer. The macro symbol that matches the longest initial sequence of the string to be translated is output, and that sequence is removed from the beginning of the string. This process is iterated until the source string is all used up, and the sequence of macro symbols output is its translation. Since each macro symbol represents a particular sequence of source symbols, the process of converting a string of macro symbols into a string of source symbols is merely one of substitution.

The hash table is just a large collection of (compressed) positions which have been proved to be losses for the player to move. The information in the hash table can be viewed as a collection of Sprouts facts that describe all the knowledge that the search has unearthed so far. Because this information is true regardless of the number of initial spots, we were able to use the table from one search to pre-load the hash table for larger searches.[9]

3.3. Factorable positions

It frequently happens in Sprouts that a position consists of the *sum* of two completely independent subgames. It is possible to infer properties of the sum from information about the subgames. In this section we carry out a very cursory discussion of sums of games, and how we used them in our Sprouts program. For a thorough description of this fascinating branch of game theory the reader is referred to *Winning Ways* [3], chapters 1, 2 and 3.

Roughly speaking a position P is the sum of positions A and B if every move in position P is either a move from A or from B, and making a move in A does not effect position B, and vice-versa. This would happen in Sprouts if, for example, position P consisted of positions A and B separated by a cycle of dead vertices.

It is easy to prove that the sum of two lost positions is also a lost position. The proof is by induction and goes as follows. As above, let P be the sum of A and B. If either or both of A and B are the empty game, then the result clearly holds because adding an empty game with another game does not change the outcome. This is the base case. In the general case, the first player chooses to move in either A or B. The game resulting after the move is the sum of a win and a loss. The next player moves in the same game so as to convert it back into a loss. This leaves the first player with a game that is the sum of two losses. This completes the proof by induction. This also shows that the sum of a loss with a win is a win. These results can be expressed succinctly in the following equation:

$$val(L \oplus V) = val(V),$$

---

symbols. Look for the most frequently occurring pair of adjacent symbols in the database, and generate a new symbol to represent that pair. Add this symbol to the set. Iterate this process until 256 symbols have been defined. Then, look for the least frequently used symbol in the database, as well as the most common pair. Redefine the underused symbol to be the heavily used pair. Iterate until the compressed database quits shrinking.

[9]Curiously, we found that in at least one instance the amount of time by which the larger search was reduced was *greater* than the total time taken by the smaller search!

where L represents any position that is a loss, V represents any position, "$\oplus$" is the sum operator for games, and $val(P)$ is the value of a position P (whether it is a win or a loss).

Our representation makes it easy to determine if a position is the sum of independent games. This happens exactly when there is a way of partitioning the regions into several collections such that no spot of degree two appears in both collections. These collections can be identified by finding the connected components of the *region graph*, which has a vertex corresponding to each region, and an edge for each spot of degree two that appears in two regions. Once such a partitioning is found, each subgame can be evaluated (or looked up) separately. Each one that turns out to be a loss can be deleted without changing the value of the position.

Unfortunately, if all of the subpositions turn out to be wins there is no immediate way to use this analysis to restrict the search. Even though the majority of positions are wins adding this feature significantly improved the performance of our program. We did not have sufficient patience to find out how long it would have taken for the program without game splitting to solve the ten and eleven spot games.

The sum formula does not apply to misère games. This is the principle reason that we were unable to solve ten and eleven spot misère Sprouts.

4. Search statistics

The searches we performed are summarized in Tables 2 and 3. The hash table was always initialized with the one resulting at the end of the prior search. The program used in these runs had all of the features described above. (Of course, game splitting was not used in the misère searches.)

Number of spots	Value of game	Cpu seconds (on a DEC 5000)	# of positions in hash table	size of hash table (in bytes)
1	L	< 0.1	1	101
2	L	< 0.1	4	606
3	W	< 0.1	7	606
4	W	0.2	33	1515
5	W	1.1	114	2828
6	L	5.9	338	4070
7	L	75.8	1843	16794
8	L	1813.7	24842	264756
9	W	8.9	24897	264756
10	W	842.8	33252	354721
11	W	10107.6	116299	1467576

Table 2: Statistics on the searches for normal play using game splitting

Number of spots	Value of game	Cpu seconds (on a DEC 5000)	# of positions in hash table	size of hash table (in bytes)
1	W	< 0.1	1	202
2	L	< 0.1	5	303
3	L	< 0.1	11	606
4	L	0.1	37	1010
5	W	1.1	219	2307
6	W	18.9	1805	15403
7	L	44.0	4970	43618
8	L	343.7	23728	202364
9	L	30579.5	1024629	13417664

Table 3: Statistics on the searches for misère play

5. Extensions to our work

The Grundy number (also known as the *nim sum* or *nimber*) of a position is an integral label that contains more information about the structure of position than merely whether it is a win or a loss. Nimbers are defined as follows: the nimber of a terminal position is zero, and in general, the nimber of a position P is the least non-negative integer that is *not* a nimber of the successors P. This way of propagating nimbers from the successors of a node back to the node is called the *mex rule*, which stands for **m**inimum **ex**cluded value.

It is easy to see that any lost position has a nimber of zero. The nimber of the sum of two positions is the bit-wise exclusive or (the result of binary addition without carry) of the nimbers of the two positions. (This way of adding nimbers is known as *nim addition*.) This beautiful non-obvious fact shows that it is easy to compute the nimber of the sum of several games knowing the nimbers of the subgames. (One trivial consequence of this nimber theorem is the fact that the sum of two identical games is a loss; the exclusive or gives zero.)

Unfortunately, although computing the sum of games with nimbers is trivial, it is in general more difficult to evaluate the nimber of a game than merely determining whether it is a win or a loss. When trying to compute the nimber of a position, the program cannot stop the search just because a successor was found to be a loss. The search must continue until the nimbers of all the successors have been computed.

Nevertheless, these ideas can probably be used to speed up the program as follows. In a preprocessing step, the nimber of each of a large set of positions is computed and stored in a table. In the main search, a position will consist of two parts: the standard part (represented as described above) and the nimber part, which is just an integer. The game defined by the position is just the sum of the game defined by the given Sprouts position and the nimber. A move takes place either in the nimber part (which leads to a position with the same Sprouts part and a smaller nimber part), or in the Sprouts part (in which moves are generated just

as usual). If a move causes the Sprouts part to split, one of which is in the nimber table, then its value is looked up and combined with the current nimber (via exclusive or).

If a position is ever reached that is the sum of several components, each of which is in the nimber table, then the nimber of this position can be immediately computed, and no further searching is necessary. To evaluate the same position, the standard algorithm would have to continue searching (if any of the components are wins) even if the values of all of the components had already been computed.

Another idea for improving the performance of the program is to improve the canonization. The conceptually simplest way of doing true canonization is as follows: If the position has k spots of degree two, for each of the $k!$ labelings with `a`, `b`, ..., of these spots run our pseudocanonization procedure (using the true names, not `2`). Among all of these representations, choose the lexicographically least one. There are much more efficient methods for doing true canonization. Many techniques developed for determining graph isomorphism can be applied to this problem. We did an experiment to determine the advantage of doing true canonization. It turned out that true canonization will cut the number of positions evaluated by about a factor of two.

We are still left with the nagging problem of resolving a bet between two of the authors. Sleator believes in the Sprouts Conjecture and the Misère Sprouts Conjecture. Applegate doesn't believe these conjectures, and bet[10] Sleator that one of them would fail on some game up to 10 spots. The only remaining case required to resolve the bet is the 10-spot misère game. This problem seems to lie just beyond our program, our computational resources, and our ingenuity.

References

[1] Piers Anthony. *Macroscope*. Avon, New York, 1969.

[2] Timothy Bell, John Cleary, and Ian Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[3] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways*, volume 2: Games in Particular, chapter 17, pages 564–568. Academic Press, London, 1982.

[4] J. H. Conway. *On Numbers and Games*. Academic Press, London, 1976.

[5] Joachim Draeger, Stefan Hahndel, Gerhard Köstler, and Peter Rossmanith. Sprouts: Formalisierung eines topologischen spiels. Technical Report TUM-I9015, Technische Universität München, März 1990.

[6] Martin Gardner. Mathematical games: Of sprouts and brussels sprouts; games with a topological flavor. *Scientific American*, 217(1):112–115, July 1967.

[10] a six-pack of beer of the winner's choice

[7] Martin Gardner. *Mathematical Carnival*, chapter 1, pages 3–11. Alfred A. Knopf, New York, 1975.

[8] Richard K. Guy. The strong law of small numbers. *The American Mathematical Monthly*, 95(8):697–712, October 1988.

[9] Judea Pearl. *Heuristics*. Addison-Wesley, Reading, 1974.

[10] David Pritchard. *Brain Games*, pages 169–171. Penguin Books, Harmondsworth, 1982.