Boundary Matching for Interactive Sprouts

Cameron Browne

Queensland University of Technology,
Gardens Point, Brisbane, 4000, Australia
c.browne@qut.edu.au

Abstract. The simplicity of the pen-and-paper game Sprouts hides a surprising combinatorial complexity. We describe an optimisation called *boundary matching* that accommodates this complexity to allow move generation for Sprouts games of arbitrary size at interactive speeds.

Keywords: Sprouts, Combinatorial game, Realtime, Optimisation

1 Introduction

Sprouts is a combinatorial pen-and-paper game devised by mathematicians Michael S. Paterson and John H. Conway in the 1960s [1], and popularised in a 1967 *Scientific American* article by Martin Gardner [2]. The game is played on a set of n vertices, on which players take turns drawing a path from one vertex to another (or itself) and adding a new vertex along that path, such that $|v_i|$ the *cardinality*[1] of any vertex v_i never exceeds 3, and no two paths ever touch or intersect (except at vertices). The game is won by the last player able to make a move, so is strictly combinatorial in the mathematical sense.

Figure 1 shows a complete game of n=2 Sprouts between players 1 and 2. Player 2 wins on the fourth move, as player 1 has no moves from this position. A game on n vertices will have at least $2n$ moves and at most $3n - 1$ moves.

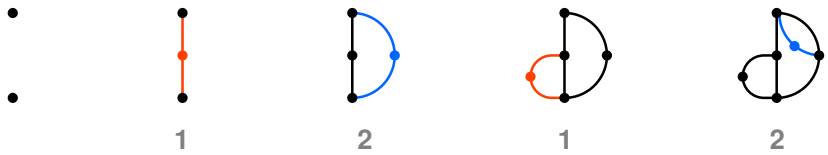

Fig. 1. A game of n=2 Sprouts, won by player 2, who makes the last move.

It is conjectured that a game on n vertices is a win for player 1 if (n modulo 6) is 0, 1 or 2, and a win for player 2 otherwise [3]. This so called *Sprouts Conjecture* [4] has not been proven yet, but has held for all sizes so far, which include $n = \{1–44, 46–47, 53\}$.[2]

[1] Number of paths incident with vertex v_i.
[2] http://sprouts.tuxfamily.org/wiki/doku.php?id=records

1.1 Motivation

There exist computer Sprouts solvers [3, 4] and interactive Sprouts position editors [5], but almost no Sprouts AI players beyond the *3Graph* player [6] which plays a perfect game up to n=8 vertices, and an iOS player [7] that apparently[3] supports up to n=15 vertices.

This relative lack of Sprouts AI players is something of a mystery. The game is well-known, interesting, looks simple, and has an intuitive topological aspect that just cries out for fingers tracing paths on touch screens – so why are there not more Sprouts apps? We identify the following barriers to implementation:

1. *Complexity*: The game's state space complexity grows at a surprising exponential rate, for even small game sizes [8].[4]
2. *Geometry*: It is non-trivial to synchronise free-form user input curves with an underlying algebraic representation of the game.

This paper addresses the first issue by describing an optimisation called *boundary matching*, which accommodates the game's inherent combinatorial complexity to reduce move generation time to interactive speeds even for large game sizes (the issue of geometry will be left for another paper). Section 2 summarises relevant computational representations for Sprouts, Section 3 describes the boundary matching optimsation, Section 4 examines the performance of the new approach, and Section 5 discusses its suitability for the task at hand.

2 Representation

Sprouts positions can be represented internally at three levels: *set representation*, *string representation* and *canonical representation*, as per previous work [3]. We briefly summarise these levels of representation, as necessary background for describing the Boundary Matching algorithm in Section 3.

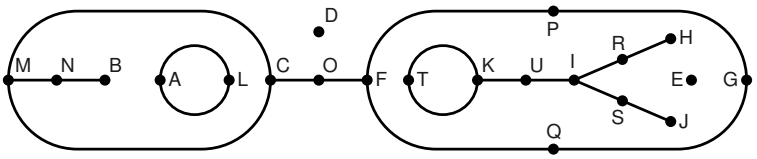

Fig. 2. Example position after ten moves in an n=11 game (from [4]).

[3] The app did not work on any device tested.
[4] Denis Mollison's analysis of the n=6 game famously ran to 47 pages [1, p.602].

2.1 Set Representation

A Sprouts *position* is a planar graph obtained according to the rules of the game. Closed paths resulting from moves divide the position into connected components called *regions*. Each region contains at least one *boundary* that is a connected component of the paths made by players and associated *vertices*. Each vertex may be enclosed within its respective region, or exist on the region border to be shared with adjoining regions.

For example, Figure 2 shows an example position after ten moves in an n=11 game (from [4]), and Figure 3 shows the five regions that make up this position.

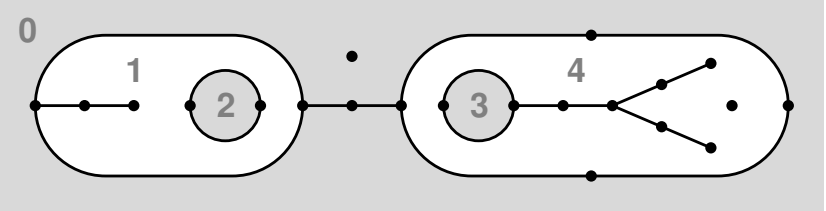

Fig. 3. Region labelling of the position shown in Figure 2.

A vertex v_i is *alive* if $|v_i| < 3$, otherwise $|v_i| = 3$ and the vertex is *dead*. For example, vertices {A, B, D, E, G, H, J, L, N, O, P, Q, R, S, T, U} in Figure 2 are alive, whereas vertices {C, F, I, K, M} are dead and play no further part in the game.

2.2 String Representation

We use a slightly simplified version of previous string representations, that we believe is easier to read. Each vertex v is labelled with a unique uppercase character {'A', ... 'Z'} in these examples. Each boundary B is represented by the list of vertex labels encountered as the boundary is traversed. Each region R is represented by the concatenation of the boundaries it contains, separated by the character ','. A position P is represented by the concatenation of the regions it contains, separated by the character ';'. For example, the position shown in Figure 2 might be described by the string:

```
AL;AL,BNMCMN;D,COFPGQFOCM;E,HRISJSIUKTKUIR,FQGP;KT
```

Subgames: It can be convenient to subdivide positions into independent *subgames*, that describe subsets of regions in which moves cannot possibly affect other subgames. This occurs when all vertices on the border between two such region subsets are dead. Subgames are denoted in the position string by the separating character '/'. For example, the position shown in Figure 2 can be

subdivided as follows:

```
AL;AL,BNMCMN/D,COFPGQFOCM;E,HRISJSIUKTKUIR,FQGP;KT
```

Move Types: Before introducing the canonical representation, it is useful to describe the two possible move types. Each move occurs within a region R:

1. *Double-Boundary Moves*: Moves from a vertex v_i on boundary B_m to a vertex v_j on boundary B_n join the two boundaries within R. For example, the first move in Figure 1 joins two singleton vertices to form a common boundary.
2. *Single-Boundary Moves*: Moves from a vertex v_i on boundary B_m to a vertex v_j on the same boundary (v_i may equal v_j) partition the region R into two new regions R_a and R_b. For example, the second move in Figure 1 creates two regions, one inside and one outside the enclosed area.

See [4, p.4] for the exact computational steps required to perform these moves.

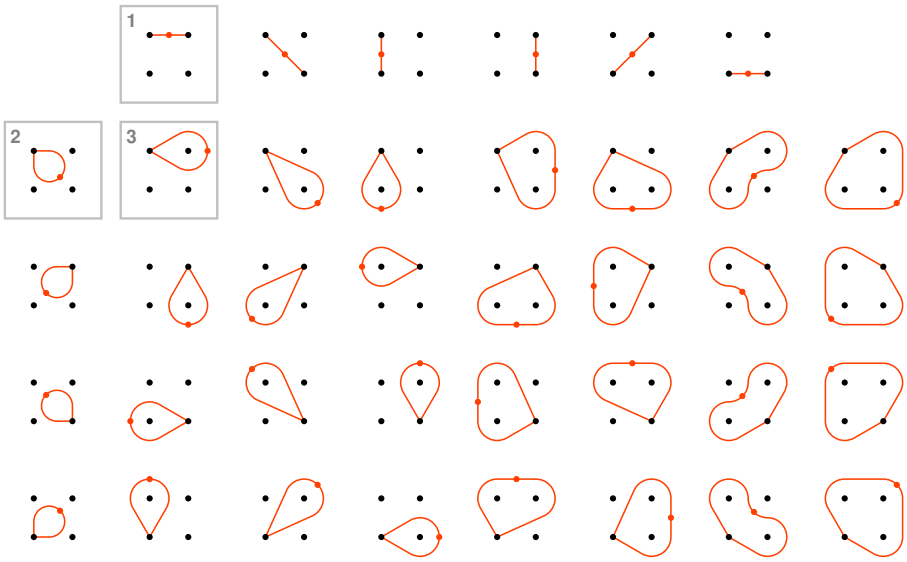

Fig. 4. Opening moves for the n=4 game. Representative invariants are indicated.

Figure 4 shows all possible opening moves for the n=4 game, by way of example. The top row shows the six possible double-boundary moves, and the remaining four rows show the 32 possible single-boundary moves. For any given game size n, the number of opening moves O_n is given by:

$$O_n = \frac{n(n-1)}{2} + n2^{n-1} \tag{1}$$

If O_4=38 is a surprising number of opening moves for only four vertices, consider that O_{10}=5,165 opening moves and O_{20}=10,485,950 opening moves(!) The main problem is the exponential growth factor $n2^{n-1}$, due to the number of ways that the region can be partitioned by single-boundary moves.

Invariants: This problem of combinatorial explosion can be addressed by observing that many of the positions in a game of Sprouts are topologically equivalent to others, and can be reduced to a much smaller set of *invariant* forms. For example, each of the 38 positions shown in Figure 4 are topologically equivalent to one of the three invariant forms indicated, which represent, respectively:

1. All double-boundary moves between one vertex and another.
2. All single-boundary moves that create partitions of {0} and {3} boundaries.
3. All single-boundary moves that create partitions of {1} and {2} boundaries.

We therefore only need to consider these three representative cases when evaluating opening moves for the n=4 game. The number of invariant opening forms O_n^+ for a game of size n is given by:[5]

$$O_n^+ = \left\lceil \frac{n}{2} \right\rceil + 1 \tag{2}$$

For comparison, there are six invariant opening forms for the n=10 game from the 5,165 actual opening moves, and only eleven invariant opening forms for the n=20 game from the 10,485,950 actual opening moves. The following section explains how to derive these invariant forms.

2.3 Canonical Representation

Canonical representation involves reducing the string representation of a given position to its canonical (invariant) form. The steps are briefly described below, using the string representation of Figure 2 as an example:

`AL;AL,BNMCMN/D,COFPGQFOCM;E,HRISJSIUKTKUIR,FQGP;KT`

1. Relabel singleton vertices as '`0`':
   `AL;AL,BNMCMN/0,COFPGQFOCM;0,HRISJSIUKTKUIR,FQGP;KT`
2. Relabel non-singleton vertices that occur exactly once as '`1`':
   `AL;AL,1NMCMN/0,COFPGQFOCM;0,1RIS1SIUKTKUIR,FQGP;KT`
3. Eliminate vertices that occur three times (i.e. dead vertices):
   `AL;AL,1NN/0,OPGQO;0,1RS1SUTUR,QGP;T`
4. Eliminate boundaries with no remaining vertices and regions with < 2 lives:
   `AL;AL,1NN/0,OPGQO;0,1RS1SUTUR,QGP`
5. Relabel vertices that occur twice in a row along a boundary as '`2`':
   `AL;AL,12/0,2PGQ;0,1RS1SUTUR,QGP`

[5] Except for $O_1^+ = 1$.

6. Relabel vertices that occurred twice but now occur once due to step 4 as '2': `AL;AL,12/0,2PGQ;0,1RS1SU2UR,QGP`
7. Relabel vertices that occur twice within a boundary with lower case labels, restarting at '`a`' for each boundary: `AL;AL,12/0,2PGQ;0,1ab1bc2ca,QGP`
8. Relabel vertices that occur in two different regions with upper case labels, restarting at '`A`' for each subgame: `AB;AB,12/0,2ABC;0,1ab1bc2ca,CBA`

The resulting string is then processed to find the lexicographically minimum rotation of each boundary within each region, with characters relabelled as appropriate, and sorted in lexicographical order to give the final canonical form:

```
0,1ab1bc2ca,ABC;0,2ABC/12,AB;AB
```

Note that the boundaries within a region can be reversed without affecting the result, provided that *all* boundaries within the region are reversed.

It would be prohibitively expensive to perform a true canonicalisation[6] that finds the optimal relabelling of uppercase characters for positions in larger games, so we compromise by performing a fast pseudo-canonicalisation at the expense of creating some duplicate canonical forms. See [3] and [4] for details.

3 Boundary Matching

Generating all possible moves for a given position, then reducing these to their canonical forms, is a time consuming process for larger game sizes. Instead, we use a technique called *boundary matching* (BM) to reduce the number of moves that are generated in the first place.

BM works by deriving an invariant form for each boundary *relative to its region*, and moving most of the invariant filtering further up the processing pipeline, before the moves are actually generated. The basic idea is to identify equivalent boundaries within a region, then simply avoid generating duplicate moves from/to boundaries if such moves have already been generated from/to equivalent boundaries in that pass.

3.1 Boundary Equivalence

We describe two regions as *equivalent* if they have the same invariant form, and all live vertices along any member boundary adjoin the same external regions. We describe two boundaries within a region as *equivalent* if they have the same invariant form, and all regions shared by live vertices along each boundary are equivalent.

[6] We prefer to use the full term "canonicalisation" rather than the abbreviation "canonisation"; we do not claim that the algorithms perform miracles.

For example, consider the position shown in Figure 5. in which region 1 contains the boundaries `AK`, `BC`, `DF`, `GH`, `I` and `J`. Singleton vertices `I` and `J` are obviously equivalent boundaries within this region. Boundaries `AK`, `BC` and `GH` are also equivalent within this region, as each adjoin equivalent regions that adjoin back to only region 1. Boundary `DF` does not have any equivalents, as the region it adjoins (region 3) has a different internal boundary structure.

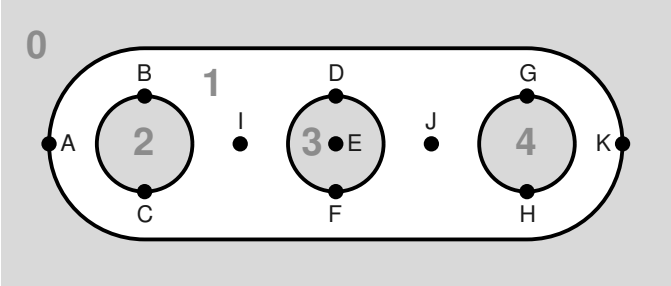

Fig. 5. Within region 1, `I` and `J` are equivalent, and `AK`, `BC` and `GH` are equivalent.

The exact steps for calculating region equivalences are not important here, as a singleton optimisation (described shortly in Section 3.4) simplifies this step.

3.2 Efficient Move Generation

If each equivalent boundary type is assigned an index $t = 1 \ldots T$, then a *boundary profile* of each region is provided by the multiset of component boundary type indices. For example, region 1 in Figure 5 would have the boundary profile {1, 1, 2, 2, 2, 3}, where $T = 3$ equivalent boundary types.

Double-Boundary Moves: It is only necessary to generate double-boundary moves between unique pairs of equivalent types, where the actual boundaries to be used are selected randomly from the appropriate equivalent subset.

Double-boundary moves would be generated between equivalence types {1, 1}, {1, 2}, {1, 3}, {2, 2} and {2, 3} in our example. While this provides some improvement, the real savings come from single-boundary moves.

Single-Boundary Moves: It is only necessary to generate single-boundary moves from a single representative of each equivalent boundary type, chosen at random from the appropriate equivalent subset, to itself. Further, it is only necessary to generate partitions of the remaining boundaries within the region, according to the powerset of the region's boundary profile (excluding the source boundary itself).

For our example profile of {1, 1, 2, 2, 2, 3}, generating single-boundary moves from a representative boundary of equivalence type 1 would generate moves defined by the following partitions: {}, {1}, {2}, {3}, {1,2}, {1,3}, {2,2}, {2,3}, {1,2,2}, {1,2,3}, {1,2,2,2}, {1,2,2,3}, {1,2,2,2,3}, where the actual boundaries used in each partition are again chosen randomly from the equivalence set with that index.

3.3 Algorithms

Given the boundary profiles for a region R, as outlined above, the algorithms for generating representative double-boundary and single-boundary moves within R using BM optimisation are presented in Listings 1 and 2, respectively. In each case, the algorithms avoid duplication by only processing boundaries of equivalence type t once in each role as source and/or destination boundary. The actual moves themselves are generated as per usual (see Section 2.2) from each pairing of live vertices along each selected from/to boundary pair.

Algorithm 1 Double-Boundary Moves with BM

```
1.  for each boundary type t1 = 1 ... T
2.      select boundary b1 of type t1 at random
3.      for each live vertex v(i=1...I) in b1
4.          for each boundary type t2 = t1 ... T
5.              select boundary b2 of type t2 at random, such that b1 ≠ b2
6.              for each live vertex v(j=1 ... J) in b2
7.                  generate double-boundary move from vi to vj
```

Algorithm 2 Single-Boundary Moves with BM

```
1.  for each boundary type t = 1 ... T
2.      select boundary b of type t at random
3.      for each live vertex v(i=1 ... I) in b
4.          for each live vertex v(j=i ... I) in b
5.              if i ≠ j or |vi| + |vj| < 2
6.                  generate single-boundary moves from vi to vj (i may equal j)
7.                      one partition for each powerset entry (excluding t)
```

3.4 Singleton Optimisation

The approach described above provides significant savings in terms of reducing combinatorial complexity, by avoiding effectively duplicate permutations. However, we can further improve the runtime of the technique by realising that *singleton* boundaries comprised of a single vertex are: easy to detect; easier to

Table 1. Opening move generation without canonicalisation.

		Without BM		**With BM**	
n	O_n^+	Moves	s	Moves	s
1	1	1	<0.001	1	<0.001
2	2	4	<0.001	3	<0.001
3	3	15	<0.001	4	<0.001
4	3	38	0.002	5	<0.001
5	4	90	0.004	6	<0.001
6	4	207	0.010	7	<0.001
7	5	469	0.029	8	<0.001
8	5	1.052	0.060	9	<0.001
9	6	2,340	0.071	10	<0.001
10	6	5,165	0.067	11	<0.001
11	7	11,319	0.073	12	<0.001
12	7	24,642	0.172	13	<0.001
13	8	53,326	0.413	14	<0.001
14	8	114,772	0.941	15	<0.001
15	9	245,859	2.203	16	<0.001
16	9	524,387	5.261	17	<0.001
17	10	1,114,124	13.408	18	<0.001
18	10	2,358,850	25.110	19	<0.001
19	11	4,798,038	59.962	20	<0.001
20	11	10,473,050	217.342	21	<0.001

match than other boundary types; the most likely boundary type to match others (on average); and the most common boundary type, at least in the early stages of a game.

It has proven sufficient in our tests to assign all such singleton boundaries to equivalence group 1, and assign each remaining boundary to its own equivalence group without attempting to match it with other boundaries. A typical boundary profile will therefore look something like: {1, 1, 1, 1, 2, 3, 4, ...}, with the occurrences of index 1 reducing as the game progresses and singleton vertices are consumed. This has the same effect as an optimisation used by Lemoine and Viennot in their "Glop" combinatorial game solver, which skips all singleton groups except the last in any position during move generation.[7]

The singleton optimisation avoids the need for potentially costly boundary matching calculations for more complex cases. It appears that the choice to not match more complex boundaries is compensated by the speed benefit of only matching singleton boundaries, at least for the cases tried so far.

4 Performance

The approaches outlined above were implemented in Java 7 for performance testing. For the boundary matching tests, each boundary profile was ordered by

[7] http://sprouts.tuxfamily.org/wiki/doku.php?id=home

index, converted to a string, and a hash map maintained to store the relevant powerset that contains the single-boundary move partitions for each distinct profile. Powersets are calculated on-the-fly as required whenever a previously unseen profile is encountered. Hence it is useful to seed the table by playing out a number of random games whenever the program changes to a new game size n, so that more commonly needed powersets are pre-generated during initialisation rather than during crucial AI thinking time. All timings were made on a single thread of a standard 2 GHz Intel $i7$ machine.

Test #1: Opening Move Generation: The first test concerns the generation of opening moves with and without BM optimisation. Table 1 shows the number of opening moves generated for games of size $n = 1$ to 20, and the time required to generate each legal move set, with and without BM optimisation.

The computational cost of move generation without BM increases exponentially with n, whereas BM demonstrates linear performance that requires less than a millisecond for legal move generation regardless of n. Note that canonicalisation is not applied in either case here; these figures indicate the raw move counts generated by each approach before invariant filtering.

The move counts with BM matching are one less than twice the number of invariant forms in each case, as each partition in the powerset contains a mirror image that is its complement. Unoptimised move generation (i.e. without BM) starts to get too slow for realtime play from around n=12 upwards.

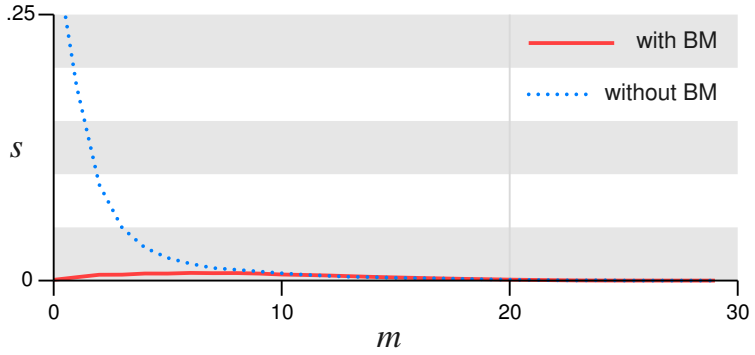

Fig. 6. Convergence of move generation timings over 100 × n=10 games.

Test #2: In-Game Move Generation: The second test concerns the relative performance of the BM optimisation over the course of a game. Table 2 shows the average *branching factor* (BF) for each move m over 100 randomly played n=10 games, and timings required to generate the legal move sets with and without BM. Note that full canonicalisation is performed in this case, to give accurate

Table 2. Branching factors and timings over 100 × n=10 games.

m	Samples	BF	**Without BM** ms	**With BM** ms
0	100	6.00	318.147	0.698
1	100	21.72	183.094	3.141
2	100	25.12	90.098	5.473
3	100	28.30	49.610	5.550
4	100	28.43	30.916	6.632
5	100	25.82	21.304	6.589
6	100	25.74	15.856	7.196
7	100	24.75	11.787	6.968
8	100	22.35	10.233	7.066
9	100	20.98	8.593	6.599
10	100	18.35	6.926	5.704
11	100	16.48	5.524	5.276
12	100	13.89	4.480	4.806
13	100	12.19	3.517	4.115
14	100	10.80	3.246	3.458
15	100	9.15	2.620	2.966
16	100	7.79	2.280	2.465
17	100	7.32	1.986	2.027
18	100	5.85	1.540	1.603
19	100	4.47	1.107	1.273
20	100	3.23	0.817	0.895
21	100	2.37	0.598	0.613
22	100	1.58	0.336	0.365
23	96	0.71	0.133	0.151
24	58	0.26	0.342	0.061
25	15	0.13	0.044	0.017
26	2	0.00	0.010	0.008
27	0	—	—	—
28	0	—	—	—
29	0	—	—	—

branching factors, so timings with and without BM are greater than in Test #1. The ruled line at the m=20 mark indicates the point at which games reach the $2n$ mark (recall that all games must last at least $2n$ moves) and typically enter the end game. These results are shown in Figure 6, in which the solid line indicates timings with BM and the dotted line indicates timings without BM.

Move generation with BM is performed in reasonably constant time throughout the course of each game, despite an increase in branching factor in the early-to-mid game,[8] while move generation with BM takes much longer at the start of each game, dropping quickly until the performance of the two approaches is almost indistinguishable from the mid-game onwards. Similar tests on larger

[8] The *early*, *mid* and *end* games could be described as approximately covering moves $1 \ldots n$-1, $n \ldots 2n$-1 and $2n \ldots 3n$-1, respectively, on average.

game sizes reveal a similar convergent trend in timings throughout the course of games, although the initial discrepancy becomes much greater as n increases.

5 Discussion

Boundary matching provides significant savings for opening move generation in Sprouts, especially for larger game sizes. For games of n=15 and higher, a single (unoptimised) legal move generation can take much longer than the desired AI thinking time of a few seconds, making BM – or some optimisation like it – necessary to achieve realtime response in such cases.

The benefit of BM optimisation quickly diminishes until there is little to choose between optimised and unoptimised performance around the mid-game, but it is the early moves that count. An AI player that relies on the lookup of known positions from pre-calculated win/loss tables is more likely to encounter known positions towards the end game, as the game decomposes into simpler subgames. It is in the opening stages that an AI player, without complete win/loss lookup information for the game size being played, really needs to maximise its lookahead penetration into the game tree.

A caveat with using BM is that the distribution of legal moves produced will not necessarily be the same as that of a random sampling of the search space, which can be a factor if Monte Carlo playouts are involved. For example, the 10,473,050 opening moves of the N=20 game are composed of $n(n-1)/2 = 190$ double-boundary moves and $n2^{n-1} = 10{,}485{,}760$ single-boundary moves, in a ratio of 0.000018. However, the BM optimisation will only produce 1 (invariant) double-boundary move and 20 (invariant with mirror reflection) single-boundary moves, in a ratio of 0.05. Random sampling can be biased to reflect this inequity (but care must be taken that the noise of single-boundary permutations do not entirely drown out double-boundary moves for larger game sizes) or random moves for playouts can be made directly from the state representation, choosing move type, region and from/to boundaries with the appropriate probabilities.

6 Conclusion

One of the immediate challenges facing the implementation of AI Sprouts players is the problem of move generation at interactive speeds for larger game sizes. The BM optimisation offers a solution by allowing fast, near-constant time move generation for arbitrary positions in games of arbitrary size, without any apparent drawbacks apart from the potential for random playout bias. Future work might include the investigation of AI search methods, utilising the BM optimisation, for playing the game at arbitrary sizes with minimal corpus knowledge. It would also be worth investigating whether BM performance might be improved through the inclusion of other simple patterns in addition to singletons.

Acknowledgments. This work was funded by a QUT Vice-Chancellor's Research Fellowship as part of the project *Games Without Frontiers*.

References

1. Berlekamp, E. R., Conway, J. H. and Guy, R. K.: Winning Ways for Your Mathematical Plays (Second Edition, vol. 3). AK Peters, Natick (2001)
2. Gardner, M.: Mathematical Games: Of Sprouts and Brussels Sprouts; Games with a Topological Flavour. Sci. Amer. 217 (1), 112–115 (1967)
3. Applegate, D., Jacobson, G. and Sleator, D.: Computer Analysis of Sprouts. Technical Report CMU-CS-91-144, Carnegie Mellon University Computer Science Technical Report (1991)
4. Lemoine, J. and Viennot, S.: Computer Analysis of Sprouts with Nimbers. Technical Report, arXiv:1008.2320v1 (2011)
5. Department of Mathematics, University of Utah: The Game of Sprouts, `http://www.math.utah.edu/~pa/Sprouts`
6. Reiß, S.: 3Graph, `http://www.reisz.de/3graph_en.htm`
7. Gehrig, D.: Sprouts – A Game of Maths!, `https://itunes.apple.com/au/app/spouts-a-game-of-maths!/id426618463?mt=8`
8. Focardi, R. and Luccio, F. L.: A Modular Approach to Sprouts. Discr. Appl. Math. 144 (3), 303–319 (2004)