Board Evaluation For The Virus Game

Peter Cowling
Modelling Optimisation Scheduling And Intelligent Control (MOSAIC) Research Centre
Department of Computing
University of Bradford
Bradford BD7 1DP
UK
P.I.Cowling@bradford.ac.uk
http://mosaic.ac

Abstract- The Virus Game (or simply Virus) is a turn-based two player perfect information game which is based on the growth and spread of competing viruses. This paper describes a CPU efficient and easy to use architecture for developing and testing AI for Virus and similar games and for running a tournament between AI players. We investigate move generation, board representation and tree search for the Virus Game and discuss a range of parameters for evaluating the likely winner from a given position. We describe the use of our architecture as a tool for teaching AI, and describe some of the AI players developed by students using the architecture. We discuss the relative performance of these players and an effective, generalisable scheme for ranking players based on similar ideas to the Google PageRank method.

1 Introduction

For two player games of perfect information with a reasonably low number of moves in any given position, such as chess, draughts and Othello, strong AI players to date have principally used a combination of fast, limited-depth minimax tree search using alphabeta pruning (Knuth and Moore 1975) and a board evaluation function to approximate the probability of each player winning from a given position. Tree search has been enhanced by techniques such as iterative deepening, using on-chip hardware to conduct the search, maintaining hash tables of previously evaluated positions (Campbell et al 2002) and heuristic pruning techniques (Buro 2002).

Creating a machine to "learn" game strategy has been an important goal of AI research since the pioneering work on checkers/draughts of (Samuel 1959). Research to date has shown little advantage for learning approaches applied to the tree search for two player perfect information games. However, many of the strongest AI players in existence now "learn" board evaluation functions. Logistello (Buro 2002) uses statistical regression to learn parameter weights for over a million piece configurations based on a very large database of self-play games. Blondie24, which has been popularised by the highly readable book (Fogel 2002), evolves weights for an artificial neural network to evaluate positions in the game of draughts. (Kendall and Whitwell 2001) uses an evolutionary scheme to tune the parameter weights for an evaluation function of a chess position. Their evolutionary scheme is notable in that evolution occurs after every match between two AI players, which gives faster convergence, since running a single game can take several CPU seconds. (Abdelbar et al 2003) applies particle swarm optimisation to evolve a position evaluation function for the Egyptian board game Seega. (Daoud et al 2004) evolve a position evaluation function for Ayo (more comonly known as Awari). Their evolutionary scheme is interesting in that a "test set" is chosen and the result of matches against this test set determines the fitness of an individual in the population. The "test set" was chosen at random. Later in this paper we will suggest how this idea may be taken further using a ranking scheme based on the principal eigenvector of the results matrix. In the work of (Ferrer and Martin 1995) the parameters used to measure board features for the ancient Egyptian board game Senet are not given in advance, but are evolved using Genetic Programming. While Senet is not a game of perfect information, this is also an interesting angle of attack for perfect information games.

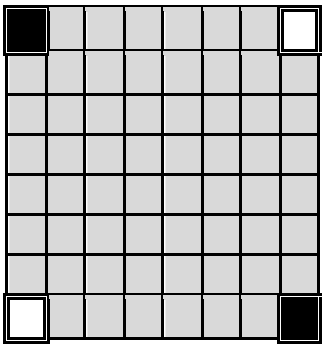

Fig.1. The Virus game starting position.

In this paper we will explore the two-player perfect information game of Virus. The earliest appearance of which we are aware of the Virus Game was in the Trilobyte

game 7th Guest (Matthews 2000). The Virus game is a two player game of perfect information, played on an *nxn* board (in this paper we will use $n = 8$ as in chess, draughts/checkers or Othello/reversi). There are two players, black and white, who play alternately, starting with black. Initially the board is set up as in Fig. 1.

Two types of moves are available at each turn. The first type of play involves *growing* a new piece adjacent to an existing piece of the same colour (Fig. 2). The second type of play involves *moving* a piece to another square a distance exactly 2 squares away (via an empty square) (Fig. 3). Note that squares are considered adjacent if they share an edge or corner. In either case all opposing pieces next to the moved piece change colour.

Play continues until neither player can move, or until one player has no pieces left, when the player with the most pieces wins the game.

Virus has a higher branching factor than all of chess, draughts/checkers or Othello/reversi, but in common with all of those games there are a large number of moves from any given position which would immediately be discounted as ridiculous by a reasonably intelligent player. Hence tree pruning based on alphabeta search is effective for Virus, as we will see later.

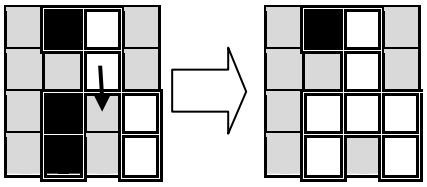

Fig. 2. The white piece grows.

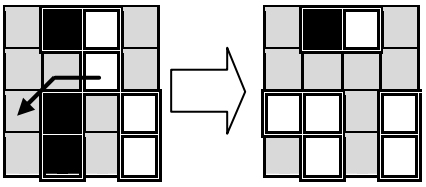

Fig. 3. The white piece moves.

We have used Virus as a testbed since it has very simple rules and yet the tactics and strategy of the game appear somewhat difficult. In particular, since the board changes a very great deal after only a few moves, it is arguably a difficult game for a human player to play well (much the same might be argued for Othello (Buro 1997)). However, after playing the game several times strategic and tactical ideas start to appear, and we will discuss these later. We have developed an Application Programming Interface (API) which greatly simplifies the implementation of AI ideas amd which ensures that they use highly efficient code.

We do not know of any published work about the game, so Virus proved very useful as a tool for teaching AI to undergraduate and Masters students. Students were given a library containing the API, search code and a client for visualising Virus games (Fig. 7), and had to devise and refine a board evaluation function using positional ideas and evolution of parameter weights. The students produced 43 Virus board evaluation functions of varying sophistication and effectiveness. We will describe briefly some of the ideas used in section 5 below.

This paper looks at how we may represent the Virus board in section 2, investigates the nature of the search tree for Virus in section 3, discusses the API for developing AI for Virus (and other board games) as well as the Virus client and server in section 4, discusses a method for tournament ranking and its wider possibilities in section 5, explains several parameters which may be used to evaluate Virus positions in section 6, finishing with conclusions in section 7.

2 Representing the Virus Board

To greatly speed up computations based on the Virus board, we represent the board as a pair (B,W) of 64-bit unsigned integers, where each bit in the first (second) integer is set to one if and only if there is a black (white) piece in the corresponding square. We use the convention that in any board representation, it is always the black player to move (by reversing the colour of all pieces if necessary) since the game is symmetric with respect to black and white. It is possible to represent all positions of the Virus board using $\lceil \log_2(3^{64}) \rceil = 102$ bits, but any such representation would be much more difficult, and slower, to manipulate. We may then use fast bit manipulation routines such as those at (Anderson 2004).

A Virus move is represented by a single 64 bit unsigned integer M where

$$(B,W) \mapsto_M ((W \,\&\, M) \,\mathbf{XOR}\, W, B \,\mathbf{XOR}\, M)$$

Where **&** is the binary AND operator and **XOR** is the binary exclusive-or operator. This move representation can be used for any game with alternating turns where (for black about to move):

1. No new white pieces may be added.
2. Black squares may become empty.
3. Empty squares may become black.
4. White squares may become black or remain white, but may not be emptied.

Hence this representation is immediately applicable to Othello. With the minor modification

$$(B,W) \mapsto_M ((W \,\&\, M) \,\mathbf{XOR}\, W, B \,\mathbf{XOR}\, (M \,\mathbf{XOR}\, W))$$

we can change restriction 4. to

4. White squares may become empty or remain white, but may not become black.

when we can use this representation for games such as draughts.

The power of this representation becomes evident when we consider how succinctly we may write useful board manipulation functions using binary arithmetic. For example, one of the most commonly used functions in move generation and board evaluation is to find all squares adjacent to a set *C* of squares, **Adj**(*C*). Let **N**, **S**, **E**, **W**, **NE**, **NW**, **SE**, **SW** be the 64 bit integers which

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1

NW

1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

SE

Fig.4. **NW** and **SE** unsigned long integers.

consist of all ones except for a row of zeroes in the top row (**N**), bottom row (**S**), left column (**W**) or right column (**E**) as illustrated in Fig. 4, where the least significant bit is in the bottom right hand corner, the bit to the left of the 2^i bit is the 2^{i+1} bit and the bit above the 2^i bit is the 2^{i+8} bit.

Then, if Ú represents the binary OR operator, Ø the NOT operator and $^{\circledR i}$, $^{\neg j}$ the right- and left-shift operators, respectively,

$$\begin{aligned}\textbf{Adj}(C) = \text{Ø}C \;\&\; [&(C \;\&\; \textbf{N})^{\neg 8} \text{ Ú } (C \;\&\; \textbf{NE})^{\neg 7} \text{Ú } (C \;\&\; \textbf{E})^{\circledR 1} \\ &\text{Ú } (C \;\&\; \textbf{SE})^{\circledR 9} \text{Ú } (C \;\&\; \textbf{S})^{\circledR 8} \text{Ú } (C \;\&\; \textbf{SW})^{\circledR 7} \\ &\text{Ú } (C \;\&\; \textbf{W})^{\neg 1} \text{Ú } (C \;\&\; \textbf{NW})^{\neg 9}]\end{aligned}$$

which can be computed using only 25 primitive binary operators, corresponding to 25 machine instructions at chip level. Other operators such as the iterators **FirstSquare**(*C*) which finds the rightmost square in the lowest possible row of *C*, and **NextSquare**(*C*,*d*) which finds the next square in *C* after square *d*, may be calculated as follows:

$$\textbf{FirstSquare}(C) = C \;\; \textbf{XOR} \; (C \;\&\; (C\text{-}1))$$

$$\textbf{NextSquare}(C,d) = \textbf{FirstSquare}(C \;\&\; (\text{Ø}((d^{\neg 1})\text{-}1))$$

Generation of all moves from a given position becomes simple and fast, using this representation, and pseudocode is given in Fig. 5.

Another fundamental function for a Virus board, is to count the number of squares in a subset. A naïve implementation which looks at each bit would be far too slow. In this case we use a **Count** operator which uses 24 machine instructions to count the number of bits in a 64-bit integer as follows:

$S \leftarrow (S \;\&\; 01..01) + (S^{\circledR 1} \;\&\; 01..01)$
$S \leftarrow (S \;\&\; 0011..0011) + (S^{\circledR 2} \;\&\; 0011..0011)$
...
$\textbf{Count}(S) = (S \;\&\; 0..01..1) + (S^{\circledR 32} \;\&\; 0..01..1)$

where after the first step each pair of bits is replaced by a count of the number of bits in the pair, after the second step each group of 4 bits (nybble) is replaced by the number of bits in the nybble, and so on. A slightly faster implementation based on table lookup is possible, but would consume large amounts of memory which could be a problem when using a large number of AI players on a server.

```
// Generate all moves from position (B,W)
Moves = {}

// Empty squares in position (B,W)
E = Ø(B Ú W)

// First 1-step moves (to c1 from any adjacent square)
C1 = Adj(B) & E
c1 = FirstSquare(C1)
while (c1 ≠ 0)
    Moves.Add(c1 Ú (Adj(c1) & W))
    c1 ← NextSquare(C1,c1)

// Now 2-step moves (from c2 to d2)
c2 = FirstSquare(B)
while (c2 ≠ 0)
    D2 = Adj(Adj(c2) & E) & E
    d2 = FirstSquare(D2)
    while (d2 ≠ 0)
        Moves.Add(c2 Ú d2 Ú (Adj(d2) & W))
        d2 ← NextSquare(D2,d2)
    c2 ← NextSquare(B,c2)
```

Fig. 5. Pseudocode for Virus move generation.

We will see further illustration of the efficiency of the compiled code resulting from this representation and the succinctness with which tactical ideas may be expressed when we talk about board parameters later.

3 Searching the Virus Game Tree

We search for the best move from a given position by searching the Virus game tree to a fixed depth (ply) using negamax search. This negamax search is further enhanced by using alphabeta pruning and ordering moves based on the very simple evaluation function (number of black pieces) – (number of white pieces). The difference between the basic minimax search and the search with these two enhancements is illustrated in Fig. 6 where we plot log (|leaf nodes|) against search depth for a mid-game position. The figure clearly illustrates the very large advantages of alphabeta pruning and move ordering. Using regression on these three plots, we estimate that the branching factor per ply is 86.6 for pure minimax search, 14.3 for Minimax with alphabeta pruning and 8.5 for alphabeta pruning with a simple ordering heuristic. More complex move ordering heuristics and transposition tables might also be used to drive the branching factor down a little further.

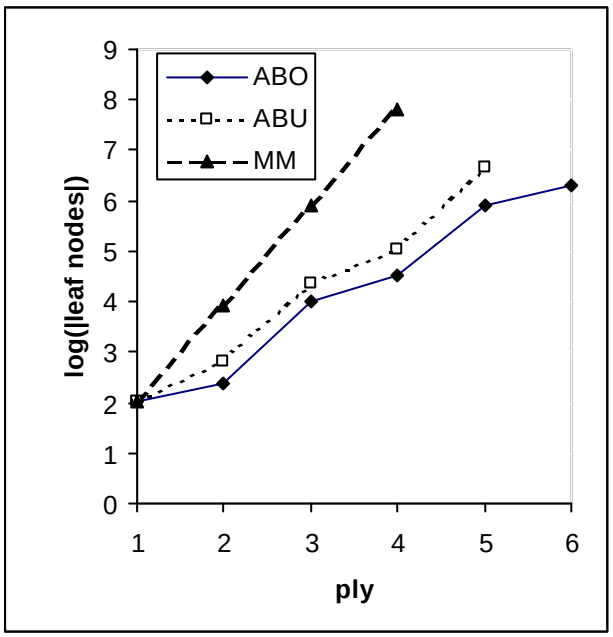

Fig.6. Plot of $\log_{10}$(|leaf nodes|) against ply for Alphabeta ordered (ABO), alphabeta unordered (ABU) and minimax without alphabeta (MM) search strategies.

4 The Virus Client and Server

A graphical Virus client, illustrated in Fig. 7 allows visualisation of games. This client allows the user to watch a game being played as well as offering the possibility to save and load games and to step forwards and backwards through saved games.

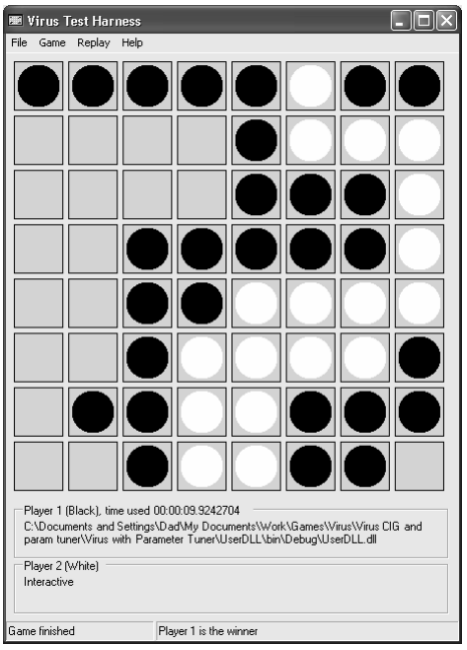

Fig. 7. The Virus graphical interface.

In order to present a user-friendly Application Programming Interface (API) for Virus and yet still benefit from the speed of the binary representation as given in section 2, we use the expressiveness of C# and its ability to hide complex functions and expose a straightforward programming interface. We have built a SquareSet struct which may be visualized as a set of squares by the AI programmer, and which presents highly efficient (and relatively complex) binary manipulation to the user as a suite of simple variables and functions. For example, SquareSet.Count is one of the most common operations on a SquareSet, as described earlier. "Pretending" that this is a variable of the SquareSet hides the complexity behind that function. Another fundamental operator which is hidden by an efficient binary function is the **GetConnectedGroup()** function which gets a connected group of squares. As we can see below using our API this becomes straightforward:

```
GetConnectedGroup(S)
    G ← GetFirstSquare(S)
    while (Adj(G) & S ≠∅)
        G ← G Ú (Adj(G) & S)
    return G
```

Fig. 8. Pseudocode for **GetConnectedGroup()**

The operators on a square set (**&**, Ú, Ø, **XOR**) are documented in terms of squares (rather than bits) to facilitate their understanding in a game context. We define the natural operators + as a synonym for Ú and – where $A - B = A$ **&** (ØB).

A Board consists of two SquareSets (B,W), one for black pieces and one for white. Again it implements a number of fast methods for board manipulation while hiding the details. All access functions accept the player as a parameter (so that separate board evaluation functions do not have to be written for white and black). In particular a Board exposes the sets B_i and W_i. B_0 (W_0) is the set of black (white) squares. B_i (W_i) is the set of empty squares at distance i from a black (white) square. We also define $B_{\le m} = \bigcup_{i=0,1,..m} B_i$ and $W_{\le m} = \bigcup_{i=0,1,..m} W_i$. Then, for example, B_1 is the set of squares which black can move to via a 1-step move, B_2 is the set of 2-step moves for black etc. Finding these sets using our API is equally straighforward. We illustrate the calculation of B_i and $B_{\le i}$ in Fig. 9.

The client and API is generic across all games on an 8×8 board with pieces played on squares and two piece colours (including Othello, draughts and Seega). Only the GetMoves() function needs to be changed in order for the API to work for a different game. Of course the leaf node evaluation would be very different for each of these games.

To use Virus as a teaching tool and to allow competition between Virus AI players submitted via the Internet, Black Marble, a software development house developed a web server which ran games between

different AI players and reported the results and league tables. It was found that allowing 10 CPU seconds per game for each AI (on a powerful twin 3.06 GHz Xeon processor server machine with 2GB RAM) for a 3ply search was the best compromise between AI search, board evaluation and getting a large number of results. A league was played using all of the currently loaded AI players. As new players were uploaded (by ftp) they jumped to the head of the queue and quickly got games, in order to encourage submission to the server. Over 600,000 games were played on the server, submitted by 45 different players over a 4 week period. Following this tournament the AI players had to be ranked. We will discuss an effective ranking mechanism in the next section.

```
// Empty squares in position (B,W)
E = Ø(B Ú W)

// Current distance under consideration
d = 0

B0 ← B
B≤0← B

while (Bd ≠ ∅)
    B≤d+1← B≤d Ú (Adj(Bd) & E)
    Bd+1← B≤d+1 XOR B≤d
    d ← d + 1
```

Fig. 9. Pseudocode for finding B_d and $B_{\leq d}$

5 Tournament Ranking and Results

Suppose that we have played a tournament with n players, where each pair has played both as black/white and as white/black. Then we have an $n \times n$ matrix M of tournament results. A player scores 64 points for a won game, plus the difference in the number of squares, 32 points for a drawn game, and 0 points for a loss. Then M_{ij} is the score for player j when i and j played. If we normalise this matrix so that row sums are one, then we have a stochastic matrix, and the entries in this matrix can be regarded as the transition probabilities in a Markov chain, where the probabilities represent the probability of jumping from a losing player (state) to a winning player (state). If we work out the steady state distribution of this transition matrix then this represents the probability that a given player wins a match averaged over an infinite series of games. This is similar to the method that the Google PageRank algorithm uses for ranking web pages (Brin and Page 2000). Hence the steady state of this Markov chain gives a very fair reflection of the relative performance of each player.

In order to find the steady state we must find a vector $\boldsymbol{x}$ satisfying

$$\boldsymbol{x}M = \boldsymbol{x}$$

i.e. we must find the principal eignevector of matrix M. By the Perron-Frobenius theorem (Grimmett and Stirzaker 1987) we know that such an $\boldsymbol{x}$ exists and is unique. Moreover, we know that for any vector $\boldsymbol{y}$ which is not orthogonal to $\boldsymbol{x}$ we know that

$$\lim_{k\to\infty} \boldsymbol{y}M^k = \boldsymbol{x}$$

so that we may calculate $\boldsymbol{x}$ iteratively starting from, say, $\boldsymbol{1}$ (the vector of all 1s) and iteratively multiplying by M until convergence occurs (typically after only five or six matrix multiplications).

The use of ranking methods based on the principal eigenvector has wide potential in the evolution of board evaluation methods, since it gives us a very precise idea of the relative effectiveness of players, which can be used, for example, in roulette wheel selection for evolutionary algorithms, or for choosing elite populations. An incomplete set of matches between players can be used to get a good idea of their ranking. Hence rather than, for example, choosing a test set as in (Daoud et al 2004) we may find the results of a small percentage of the matches and use the stationary distribution/eigenvector method to "fill in the gaps". The fact that these values are already normalised so that a principal eigenvector value twice as large corresponds to a player who is twice as strong makes them particularly useful. We are currently investigating a range of board evolution methods based upon the eigenrank.

The ideas present in the 45 AI players submitted to the server represent a wide cross-section of tactical (and to a lesser extent strategic ideas). These ideas include:

- Consideration of "degree of safety" for a piece and square
- Mobility
- Pattern matching (notably to count the number of times the disadvantageous pattern consisting of three pieces of one colour and one empty square occurs.
- Assigning positional scores for owning different board squares such as recognising that corner squares are good in the opening and assigning a different score to each square.
- Development of an opening book. Note that similarly to the game of Go (Müller 2002) the opening stages of Virus are very difficult to analyse, even on an 8×8 board.
- Ratio of surface area to volume (using a biological analogy)
- Tuning parameter weights dependent on game stage (as measured by the number of empty squares left).

The sophistication of the ideas submitted to the Virus server gives strong support to the idea of games as an effective way to teach and promote understanding of

advanced AI concepts. In addition, many of the submitting players developed add-on tools and used evolutionary parameter tuning schemes.

6 Evaluation Functions for the Virus Game

There are several parameters for Virus which capture the strategic and tactical ideas behind the game. In this section we will illustrate how succinctly these tactical and strategic ideas may be presented using the notation (and API) described above.

The simplest measure is the number of black and of white counters in the board position:

- $|B_0|$ and $|W_0|$

and indeed a 3-ply search using only $|B_0| - |W_0|$ as an evaluation function is a challenging opponent for a human virus player.

Other ideas capture the common notion of mobility (ie. the number of moves in a given position:

- $|B_1|, |B_2|, |W_1|$ and $|W_2|$

We may further refine these measures by considering only moves which result in captures for one player or another.

It is also interesting to consider the size of the largest move of each type for each player (for $i = 1,2$):

- $\max_{s \in Bi} |\mathbf{Adj}(s) \,\&\, W_0|, \max_{t \in Wi} |\mathbf{Adj}(t) \,\&\, B_0|$

We may also consider the total number of the opponent's pieces which are vulnerable to capture

- $|(\mathrm{Adj}(B_1) \cup \mathrm{Adj}(B_2)) \,\&\, W_0|$,
  $|(\mathrm{Adj}(W_1) \cup \mathrm{Adj}(W_2)) \,\&\, B_0|$

An important strategic idea which emerges after several games is the notion of "encirclement". Here we wish to limit the range of squares which may count for the opponent at game end by encircling the opponent's pieces into a small area of the board, and thus capturing all of the empty space remaining at our leisure at game end (since there is no way for the oponent to move into the space). Hence we have the idea of "totally safe squares". For example, all of the empty squares in Fig. 7 are "totally safe squares" for black since they are surrounded by black pieces which cannot be captured (i.e. the black pieces are not adjacent to any "unsafe" empty squares which would allow them to be captured. The algorithm for calculating this parameter is quite complicated without our API notions, but with the API we get the algorithm as shown in Fig. 10.

While the idea of totally safe squares captures the immediate tactical consequences of enclosure, in order to address the strategic issue of how we can work towards enclosure of the opponent, we can consider the set of empty squares which are k closer to one player than the other, e.g. for the black player:

- $\bigcup_{i=1,2,\ldots} (B_i - W_{\leq i+k})$

Other strategic and tactial ideas for Virus (and indeed other games) can be easily and efficiently implemented using our API. The above set of parameters are arguably a set which, given proper tuning of parameter weights, could give rise to a very strong player.

```
// Find the set T_B of totally safe squares for black from position (B,W)
T_B = ∅

// Squares not reachable from W_0
W¥ = Ø(W_0 Ú W_1 Ú ...)

// The candidate groups (together with their boundary of black squares)
C = W¥ Ú Adj(W¥)

// Empty squares in position (B,W)
E = Ø(B Ú W)

// Go through connected group by connected group checking for safety along the boundary.
while (C ≠ 0)
    G = GetConnectedGroup(C)
    while (G ≠ 0)
        if (Adj(G) & E == ∅)
            T_B ← T_B Ú (G & E)
        C ← C - G
```

Fig. 10. Pseudocode for a fast algorithm to find totally safe squares for the black player.

7 Conclusion

We have presented the board game Virus and a generalisable API for Virus which allows effective AI to be developed quickly and with relatively little experience. We have analysed the search tree for Virus as well as presenting a range of tactical and strategic ideas. We have used these ideas to show how an easy-to-use API can facilitate the development of AI for research and teaching purposes. We have discussed a server architecture for Virus that allows AI players to be submitted across the Internet and a generally applicable ranking method based on treating match results as transition probabilities in a Markov chain.

We are currently working on a generalisable evolutionary scheme for tuning the parameters of the evaluation function for any board game which may be represented using the Virus API.

The Virus API, client and server are publically available for non-commercial use. If you would like to make use of them send an email to P.I.Cowling@bradford.ac.uk.

Acknowledgments

The work described here was partially funded by Microsoft UK Ltd., and I am particularly grateful to Gavin

King of Microsoft for his support and advice. I would like to thank Steve Foster who first introduced me to the Virus Game. I am also grateful to Robert Hogg, Richard Fennell and Nick Sephton of Black Marble Ltd. who created and continue to maintain the client and server for Virus. I would particularly like to thank the 43 students on the "AI for Games" module who acted as guinea pigs for this learning experiment. I am grateful to Professor Simon Shepherd who reminded me of the possibilities of the principal eigenvector and pointed out the link with Google. Finally, Naveed Hussain gave me some additional references and is continuing this work.

Bibliography

Abdelbar, A.M., Ragab, S., Mitri, S., "Applying co-evolutionary particle swarm optimisation to the Egyptian board game Seega", in Proceedings of the First Asia Workshop on Genetic Programming (part of CEC 2003) 9-15.

Anderson, S.A., "Bit Twiddling Hacks", http://graphics.stanford.edu/~seander/bithacks.html.

Brin, S., Page, L., "The anatomy of a large-scale hypertextual Web search engine", Computer Networks and ISDN Systems, vol. 30 (1-7) (1998) 107-117.

Buro, M., "Improved Heuristic Mini-Max Search by Supervised Learning", Artificial Intelligence, Vol. 134 (1-2) (2002) 85-99.

Buro, M., "The Othello match of the year: Takeshi Murakami vs. Logistello", ICCA J. 20 (3) (1997) 189-193.

Campbell, M., Hoane, A.J. Jr., Hsu, F-h., "Deep Blue", Artificial Intelligence 134 (2002) 57-83.

Daoud, M., Kharma, N., Haidar, A., Popoola, J., "Ayo, the awari player, or how better representation trumps deeper search" in Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC 2004) 1001-1006.

Ferrer, G.J., Martin, W.N., "Using genetic programming to evolve board evaluation functions" in Proceedings of the 1995 IEEE Congress on Evolutionary Computation (CEC95) 747-752.

Fogel, D.B., "Blondie24: Playing at the edge of AI", Morgan Kaufmann, 2002.

Grimmett, G.R., Stirzaker, D.R., "Probability and Random Processes", Oxford Science Publications 1987.

Kendall, G., Whitwell, G., "An evolutionary approach for the tuning of a chess evaluation function using population dynamics", in Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC 2003) 995-1002.

Knuth, D.E., Moore, R.W., "An analysis of alpha-beta pruning", Artificial Intelligence 6(4) (1975) 293-326.

Matthews, J., "Virus Game Project", http://www.generation5.org/content/2000/virus.asp.

Müller, M., "Computer Go", Artificial Intelligence, Vol. 134 (2002) 145-179.

Samuel, A., "Some studies in machine learning using the game of checkers", IBM J. Res. Develop. 3 (1959) 210-229.