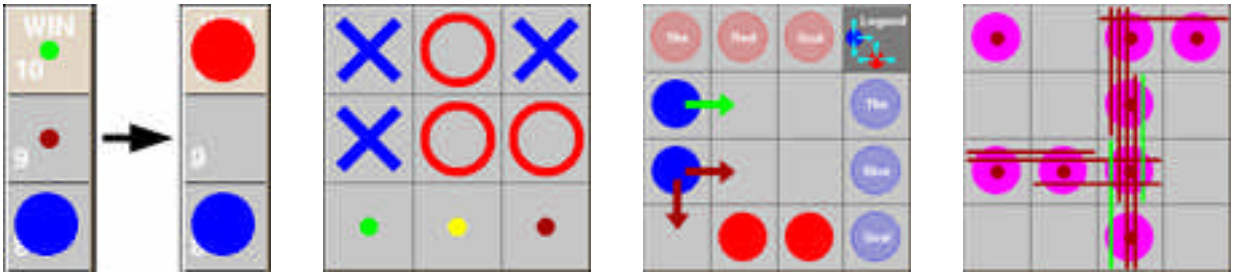


GAMESMAN

A finite, two-person, perfect-information game generator

by

Daniel Dante Garcia

(ddgarcia@cs.berkeley.edu)

B.S. Computer Science and Engineering (Massachusetts Institute of Technology) 1990
B.S. Electrical Engineering (Massachusetts Institute of Technology) 1990

A report submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Elwyn R. Berlekamp, Chair
Professor Brian A. Barsky

Abstract

"Why write a program when you can write a program to write a program?"

– Author unknown

This report introduces GAMESMAN, a system for generating graphical parametrizable game applications. Programmers write game 'modules' for a specific game, which when combined with our libraries, compile together to become stand-alone X-window applications as shown in Figure A.1 below. The modules only need contain information about the rules of the game and how the game ends. If the game is small-enough, it may be solved, and the computer can play the role of an oracle, or "perfect" opponent. This oracle can advise a novice player how to play, and teach the strategy of the game even though none was programmed into the system! If a game is too large to be solved exhaustively, the game programmer can add heuristics to provide an imperfect computer opponent. Finally, the application can provide a useful utility to two human players who are playing each other, since it be a "referee" who constrains the users' moves to be only valid moves, can update the board to respond to the move, and can signal when one of the players has won.

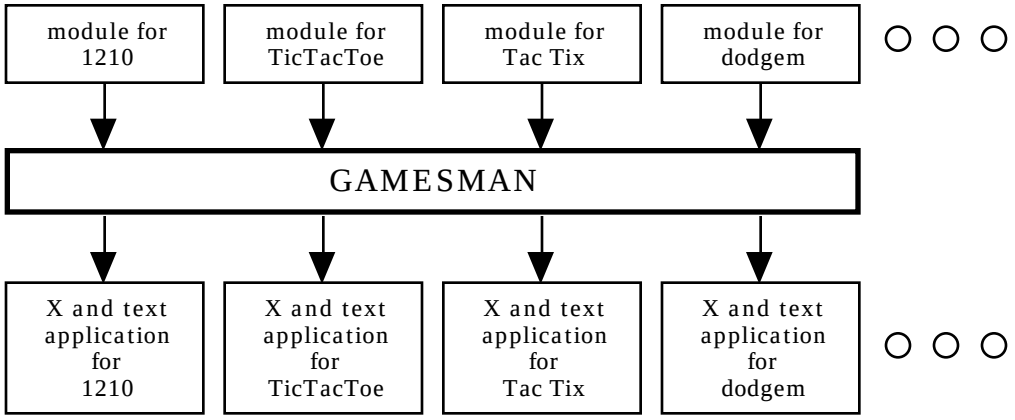


Figure Abstract.1: The Overall Picture of GAMESMAN

Dedication

To my parents, Raymond and Deborah, whose enduring love and simple, generous acts of allowing their 6-year-old son to play when he needed to play and to beat them at chess fostered a love for games that has not abated in the 21 years since.

Table of Contents

List of Figures

1. Introduction

Traditionally, programmers write and tune finite, two-person, perfect-information game-playing programs for one specific game. Their primary goal is entertainment, and little thought is given to analysis, rule modification, or the teaching of game theory concepts. They also do not provide perfect opponents as they typically utilize heuristic look-ahead to chart their strategy.

Game toolkits do exist, but the interactions are either purely text-based or they do not provide a library of user-interaction routines to make the creation of a graphical user interface (GUI) front-end to the games easy. Also, none provide the ability to modify the rules of the game on the fly or analyze a game in progress with hints from an oracle (i.e., perfect player).

In this report we describe the design and implementation of GAMESMAN, a game generator that takes as input the description of a game and produces game-playing applications, to be used for enjoyment, analysis, or education. Once invoked, the application solves the game[1] by exhaustively searching every possible board configuration, or *position*, and storing the values (win, lose, or tie) for every position in a table. After the game is solved, the table is used by the computer to simulate a "perfect" opponent who will never lose if given an initial winning position. It is the application's ability to play games perfectly that is so valuable to game analysts, as they may test any strategic theories against it.

From an implementation point of view, programmers develop modules, which are subroutines and data structures written in the C and Tcl/Tk programming language. These modules describe a game's rules and winning conditions, and compile them with the main solving and interface code of GAMESMAN to produce two applications: one X-based for X displays and one text-based for text-only systems. The modules describe the game's rules, user interactions, and graphic display.

In this paper, we first clarify the lexicon for readers unfamiliar with game theory, mention related work and discuss the constraints placed on the types of games that modules can implement. We then highlight the mechanics of writing a module, as well as provide the module specifications in Appendix A. The next chapter on categorizing games contains a study of the user interactions most board games involve, as well as giving some measure to gauge how large the position table might be. We analyze user-interface issues that arise when creating a library of user-interaction routines for common board games. We follow with a self-evaluation in which we describe the benefits of GAMESMAN, its limitations due to the exhaustive-search method, and suggest some optimizations. Finally, we discuss future enhancements, describe the GAMESMAN interface, and summarize the work.

[1] This is only possible for small games, as we will discuss later.

2. Definition of Terms

This chapter serves to clarify some of the terms related to game theory that are used extensively throughout this paper. When appropriate, examples from the games Tic-Tac-Toe and "1,2,...,4"[2] will be used to illustrate the terms. Readers are encouraged to refer to Appendix B for detailed descriptions of any games described in the text with which they find themselves unfamiliar.

2.1 Position

This is the same as a board configuration – it is a snapshot of the board with pieces on it and a turn designation[3]. It maps to a vertex in the game-tree. Every game has positions, regardless of whether or not it is played on a physical board. Abstract games as well as 1,2,3,...N-dimensional games all have positions, and all can be implemented in GAMESMAN. Figure 2.1 shows a sample position for Tic-Tac-Toe. Note that the outcome of the game would be quite different if it were O's turn. For the abstract game "1,2,...4", any of the integers 1 through 4 are positions.

X's turn

Figure 2.1: A Tic-Tac-Toe position

2.2 Slot

A slot is a coordinate on a board, which in the 2-D case would be an (X, Y) pair. A board, in the literal sense, is a 2-D collection of slots, but it also used to describe the collection of slots for N-dimensional games as well. Keep in mind that abstract games such as "1,2,...4" do not have clear definitions of slots. In the game Tic-Tac-Toe, there are 9 slots, numbered in figure 2.2.

Figure 2.2: The 9 Tic-Tac-Toe slots

2.3 Piece

Pieces rest on slots and are rearranged, added and removed by players on their turn. A single slot can only have one piece on it at one time. In Checkers, a 'king', which to the novice player is two pieces, is computationally a completely

[2] A variation on the popular game, Nim, described in Appendix B.
[3] E.g. the board looks like this and it is X's turn.

different piece with distinct moving abilities. Similar to slots, pieces do not have an equivalent in abstract games. Figure 2.3 shows the two pieces for Tic-Tac-Toe.

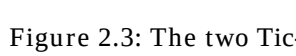


Figure 2.3: The two Tic-Tac-Toe pieces

2.4 Move

"Move only if there is a clear advantage to be gained"

– Sun Tzu
The Art of War [Tzu83, p. 83]

Moves map to edges in the game-tree. It is the action a player performs on his/her turn to change the board's configuration. In Tic-Tac-Toe, a move is the action of placing an X or O on an empty slot. In Chess, a move consists of relocating[4] a piece from its original slot to another slot and perhaps capturing an opponent's piece. Most games' rules dictate that the available moves are a function of position; in the game "1,2,...,4", however, there are exactly two possible moves for *every* position, 1 and 2.

Most games dictate that the players alternate turns every other move. This is not a restriction of our system; players may make multiple moves if that is part of the rules. Combinatorial game theory as described in [Berlekamp82] does not place a restriction on whether players alternate turns even if the rules state that they do, since individual games are considered part of a larger sum of many games. However, throughout this paper we make the assumption that our games are independent entities, played alone.

2.5 Value, or Outcome

Much research [Conway76], [Berlekamp82] has been done to come up with a proper number-theory foundation for game values. Games are either positive, negative, zero or fuzzy, depending on whether the Left, Right, first or second player can always win. Some examples for values of games are *5, {2 | 1 | | -1}, ±6 or ↑*, which are not easily interpreted without prior training in combinatorial game theory. It was our hope that GAMESMAN have appeal and be understood by to the novice game theorist.

To that end, we have simplified the meaning of the value of a game. Every position has a value, which we will consider to be one of { Win, Lose or Tie } for the player whose turn it is to move. This can also be thought as the outcome of the game if played by perfect opponents. Combinatorial game theorists may recognize these

[4] Sometimes the verb "relocate" is used interchangeably with the verb "move" in this context. This can lead to possible confusion due to the overuse of the word "move". In this paper, move will strictly refer to the action a user applies to a position on his/her turn, and the verb "relocate" will refer to the reassignment of a piece to another slot.

as *fuzzy*, *zero* or tieing[5]. This means that if the game was played between perfect opponents, the player whose turn it was would *always* either win, lose or tie. Any move that leads to a winning position for the other player is a *losing* move, and consequently any move that leads to a losing position for the other player is a *winning* move. A tie move is one that leads to a tie position for the other player. If the first, or initial position in a game has value V, then the *game* is said to have value V. For example, if Tic-Tac-Toe began with position A in figure 2.4 rather than a blank board, then Tic-Tac-Toe would be a winning game, since A is a winning position.

Figure 2.4 contains a very detailed Tic-Tac-Toe game tree to help us understand these terms. Note that under every position is a string that represents the position letter, whose turn it is and what the value of that position is. For example, the root position contains "A - X - Win", which means that it is position A, X's turn and a win for X. The arrows have a number and a letter beside them that represent which slot was chosen and what the value of that move was. For example, the upper-left-most arrow contains "7 - Win", which means that player X chose slot 7, and it was a winning move for player X.

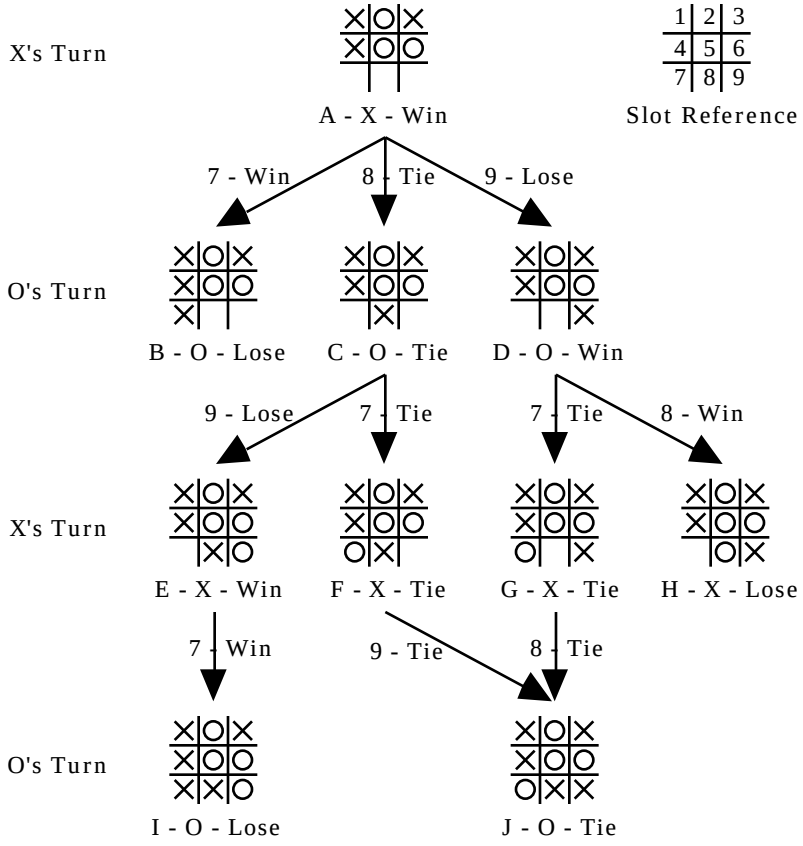


Figure 2.4: A branch in a Tic-Tac-Toe game-tree

[5] Although pure game theorists usually do not consider games which can end in ties, since a tie does not have a meaningful game value.

2.5.1. Win

"For nothing can seem foul to those that win."

– Shakespeare
Henry IV, Part I, Act V, Scene 1

In some references this is referred to as an "N" position, which means the Next player can win. This value is recursively defined by the following rule: *A winning position is one in which there exists a losing child*. This is best illustrated by position A above, which has a winning (D), losing (B), and tieing (C) child, and is considered a winning position due to the existence of B. The move that leads to the losing child, slot 7, is the winning move. The following positions are all winning in the above game-tree: A, D and E.

Just because a player has a winning position doesn't necessarily mean the player *will* win, simply that the player *can* win. In position A above, a winning position, if X chooses the lone losing move to slot 9, X can lose. Sometimes a win is inevitable, since all the children are losing positions, and in these rare cases a winning position indicates that the player *will* win. Position E has one lone slot for X to choose which forces the win. X has no option but to choose slot 7 and win the game. It is important to remember that a winning position in general means that the *potential* for winning against a perfect opponent exists.

2.5.2. Lose

"Dr. Pulaski: To feel the thrill of victory, there has to be the possibility of failure. Where's the victory in winning a battle you can't possibly lose?
Data: Are you suggesting there's some value in losing?
Dr. Pulaski: Yes, yes, that's the great teacher. We humans learn more often from a failure or a mistake than from an easy success."

– Dr. Pulaski and Lieutenant Commander Data
in Star Trek : The Next Generation's *Elementary, My Dear Data*

Similar to a winning position, a losing position is often called "P", which means the Previous player can win. Said another way, it means that the player whose turn it is *will* lose against a perfect opponent. This value is also recursively defined, but by a different rule: *A losing position is one in which there does NOT exist a losing or tieing child*. This means that the children of losing positions are either all winning or it is primitive (and has no children). The losing positions from figure 2.4 (B, I and N) fulfill the latter case and are all primitive losing positions. We describe primitive positions below in section 2.5.4.

If a player has a losing position and is playing against a perfect opponent, the player has already lost the game and might as well concede[6]. This is because a

[6] This brings up the interesting point that no game need ever be played between two perfect opponents since the outcome is decided before the game begins. The first player will achieve the outcome which is exactly the same as the value of the game. I.e. if the game is a win game, the first player will always win and the second will always lose.

perfect opponent will continue to make winning moves until either it has reached a primitive winning position or the other player is left with a primitive losing position. However, if the opponent is imperfect, then a non-primitive losing position does not guarantee a loss, just the *potential* for losing.

2.5.3. Tie

"I wish it could have been a tie"

– Amanda Bonner (Katherine Hepburn),
after defeating her husband in *Adam's Rib*.

A tie position is recursively defined as: *A tying position is one in which there does not exist a losing child, but there does exist a tie child.* Whether or not there are any winning children is irrelevant, as it does not affect the value. Position C in figure 2.4 above is a perfect example of a tying position, since it has no losing child but *does* have a tie child, which is position F. A player with a tie position can either tie[7] or lose against a perfect opponent. Against an imperfect opponent, it is possible to either tie, lose or win. Positions C, F, G and J are tie positions, yet only J is a primitive tie. As mentioned before, a tie move is one which leads to a tie position. If there are no tying terminating criteria (see below), there can never be a primitive tie position, and by induction, no non-primitive tie positions.

2.5.4. Primitive positions and terminating criteria

Primitive positions are the leaves in the game tree and are the positions that fulfill the *terminating criteria* for the game. These criteria are what prevent the game from being infinite, as they force the game to end at some point. In Tic-Tac-Toe, there are two terminating criteria. The first is that the other player has just achieved three-in-a-row of his/her piece, which means that the position is a losing primitive position. The second is that a position has all 9 slots filled, in which case it is a tying primitive position. Note that for most games the order that these are checked is crucial, as position I fulfills both, and would be incorrectly labeled a tying position if the rules were reversed. Positions that are not primitive are either called *non-primitive* or *recursively-defined*, due to the definitions highlighted above. In figure 2.4, the primitive positions are B, I, J and H.

2.5.5. Value-equivalent moves.

Value-equivalent moves are moves that lead to children of equal value. All five moves available to X (slots 2, 4, 5, 7 and 8 in figure 2.5) are winning since all five lead to losing positions. These moves are *value-equivalent* and any may be chosen without risk to the outcome. This is true even if though 2 leads to an immediate primitive losing position (for O) and 4, 5, 7 and 8 lead to non-primitive losing positions (for O).

---

[7] Since this is the best that can be hoped for, we suggest that players consider it a "win" to tie a perfect opponent given a tie game, such as TicTacToe.

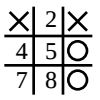


Figure 2.5: A winning Tic-Tac-Toe position with three value-equivalent winning moves

Is it true that moves 2, 4, 5, 7 and 8 are really equivalent? Clearly slot 2 forces an immediate win, 4, 5 and 7 force a two-way win, and slot 8 is only barely a win if followed by slot 7. Therefore we could order these value-equivalent winning moves by some criteria, such as the minimum moves until a forced win. By definition a non-primitive losing position contains nothing but winning children, and consequently nothing but value-equivalent losing moves. It is often useful to order these by maximum-time-until-a-forced-win so that an imperfect opponent will have the most opportunities to choose the wrong move. This is referred to as the *Enough Rope Principle* in [Berlekamp82]. Another strategy is to force an opponent into positions with the most available moves, in hopes that the myriad options will somehow increase the chances for a bad move. Thus, even though for perfect opponents one value-equivalent is no better than another, it may be useful to sort value-equivalent moves by various criteria against imperfect opponents.

2.6 Perfect opponents

"He wins his battles by making no mistakes. Making no mistakes is what establishes the certainty of victory, for it means conquering an enemy that is already defeated."

– Sun Tzu
The Art of War [Tzu83, p. 20]

Perfect opponents are opponents who *always* choose winning moves given winning positions and *always* choose tying moves given tying positions. It doesn't matter what perfect opponents choose given losing positions, since if they are primitive there are no moves available and the game is over and if they are non-primitive all moves are value-equivalent losing ones. It is tempting to define perfect opponents as computers and imperfect opponents as humans, but that would incorrectly label bad programs and very knowledgeable humans.

A simple qualification for a perfect opponent is the ability to recognize a losing and tying position perfectly. The following algorithm describes how to play perfectly given the aforementioned ability. For every turn, if the position is losing, any move may be chosen. If it is tying, all children (moves) are scanned, the losing ones are thrown out, and any remaining moves are chosen, since they will all be tying. If the position is winning, all children are scanned, the losing and tying positions are thrown out, and any remaining move is chosen. GAMESMAN creates a perfect opponent by exhaustively searching games that are small enough to be solved, then builds a table of possible moves. Then when it is the computer's turn, it follows the algorithm listed above when choosing its move, since it knows the value of every single possible position.

2.7 Partisan vs. Impartial games

The definition is best stated from [Berlekamp82, p. 17]: “Any game s.t. exactly the same moves are available to either player are called *impartial*. Games in which the two players may have different options we shall call *partizan*[sic]”. Othello, Chess, Checkers, Go, Dodgem and Tic-Tac-Toe[8] are all partial games, and Tac Tix, Nim and “1,2,...,4” are impartial.

Since a position is both a board configuration *and* a player whose turn it is to play on that board, both types of games can be considered winning, losing or tying. In these cases a winning position for the first player is *by definition* a losing position for the second player, and vice versa. However, if we consider a position to include simply the board configuration and *not* the players turn, partisan games differ from impartial games in that they create two more categories: those that are positive or negative. A positive game means that it wouldn’t matter whether X (Left) went first or last, since it would have a winning position either way. If our definition of a position is both a board *and* whose turn it is, then figure 2.6 with X going first is a winning position and with O going first is a losing position, as expected. However, if our definition of a position is *only* the board configuration, then figure 2.6 is positive – it doesn’t matter who goes first, X will have a winning position on its turn either way.

Figure 2.6: A game with a positive value if a position is only the board configuration, and not the board position and whose turn it is.

2.8 The misère game vs. the standard game

Every game has terminating criteria that constitute the rules of the game. These are called, for convenience, the *standard* rules, and must include winning or losing conditions. Every single game has a *misère* game, which is the game with the words “losing” and “winning” swapped into the terminating criteria for the standard game. For example, whereas the standard game in Tic-Tac-Toe is be explained as: “A player *wins* if he/she achieves three-in-a-row first”, the *misère* game is explained as: “A player *loses* if he/she achieves three-in-a-row first”. Many people confuse the swapping of “win” and “lose” with the logical negation of the rules, but they can be very different. If the rule is: “First person to get *one* piece to the other side *wins*”, the misère game is NOT “First person *not* to get *all* pieces to the other side *wins*”, it’s “First person to get *one* piece to the other side *loses*”. Tying terminating criteria do not change in the misère game.

Interestingly enough, sometimes perfect strategies for the standard game and the misère game differ by only the primitive positions and perhaps a few others. For these games, unless the position in question is very near a leaf in the game-tree, it is played the same as it was in the standard game. Some games have completely

[8] Even though both X and O would have the same *slots* available to them, they would *not* have the same *moves* available to them because the same slot could be a win for one player and a tie for another.

different strategies for the two games. GAMESMAN provides the ability to play either the standard or misère game, and can be used as an analysis tool to determine the differences and similarities of the strategies.

2.9 Solving games

"...you've got them all memorized. The first time anyone opens their mouth, you've got it solved, so there's really no mystery. No mystery, no game. No game, no fun."

– Lieutenant Commander Geordi La Forge
in Star Trek The Next Generation's *Elementary, my Dear Data*

If a game is small enough so that its positions may be stored in memory and its game tree searched in a reasonable amount of time[9], then it can be solved. This process involves performing an exhaustive search of the game-tree and recording every position's value in a table. This table is then used to guide the computer opponent to play the game perfectly. This process is done once at the beginning of the game, and the table is used for many instances of the same game.

There are shortcomings to solving a game. First, there is a (sometimes substantial) delay as the system does the brute-force search. Second, if the user wants to toggle a parameter of the game, e.g., toggling the rules from the standard to the misère, then the current table must be thrown out and the search must begin again. These tables can be stored and recalled[10], but if disk space is small this may not be possible. The advantage of solving games rather than writing static evaluators is that the computer comes up with its own optimal strategy without any programmer intervention. This can then be used to guide the creation of a static evaluator that is used by a MINIMAX routine to determine the computer's moves. Unfortunately, most games cannot be solved, and thus the user must either settle for reasonably small games or for imperfect heuristic-driven opponents, as discussed below.

2.10 Static-evaluation

"Cogley: How many games of chess did you win from the computer, Mr. Spock?
Spock: Five in all.
Cogley: May that be considered unusual?
Spock: Affirmative.
Cogley: Why?
Spock: I personally programmed the computer for chess months ago. I gave the machine an understanding of the game equal to my own. The computer cannot make an error. And, assuming that I do not either, the best that could normally be hoped for would be stalemate after stalemate, and yet I beat the machine 5 times."

– Lieutenant Commander Spock and Defense Attorney Samuel L. Cogley
in Star Trek's *Court Martial*

---

[9] See the "Limitations" section for more details about time and space constraints.
[10] See the "Optimizations" section for further information.

A static evaluator is a black box that takes as input a position and outputs a "goodness" value. This number between 0 and 1 describes the degree to which one player has an advantage. In another sense, it describes the probability that one player will win given that position. As an example, let us consider the static evaluator for Othello, as shown in figure 2.7 below. Static evaluators return a 1.0 for a position that describes a win for the first, or maximizing player and a 0.0 for a position that describes a win for the second, or minimizing player[11]. This only holds true for Partisan games, or games in which different players have different moves available to them. Impartial games differ in that 1 is a winning position and 0 is a losing position always, regardless of whose turn it is. The higher the number, the better it is to have that position on your turn.

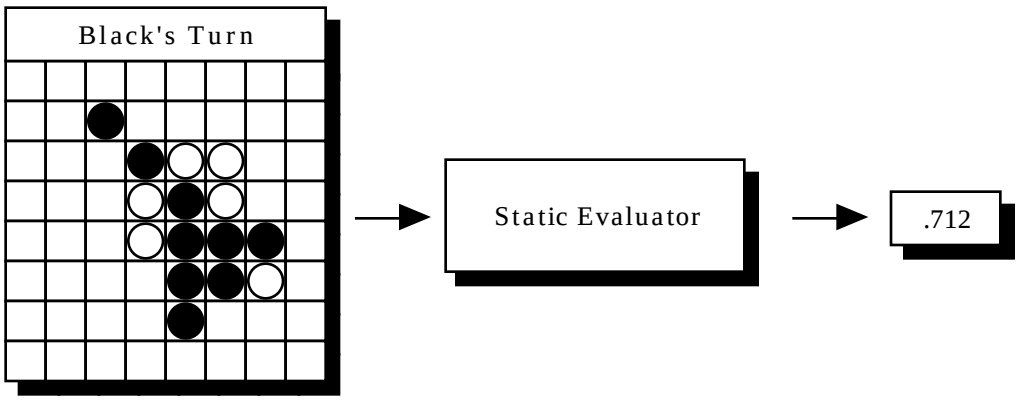


Figure 2.7: The static evaluator for Othello

The final goodness value is often a function of other goodness values, based on smaller sub-goals that more explicitly describe one player's advantage. In order to combine the goodness values, a rather complicated equation based on non-linear weighting functions may have to be created. For Othello, we might construct several evaluators that would only concentrate on certain parts of the game. One evaluator might return the amount of control a player has of crucial board slots, which in Othello's case would be the corners. A second evaluator might return the piece count advantage. Another may simply tally the number of possible areas to attack and compare it to the other player. A final evaluator would grade the control of the inner 4×4 square. An example of these sub-goal evaluators for a sample Othello position is shown in figure 2.8 below.

[11] The concept of maximizing and minimizing is explained in the next section.

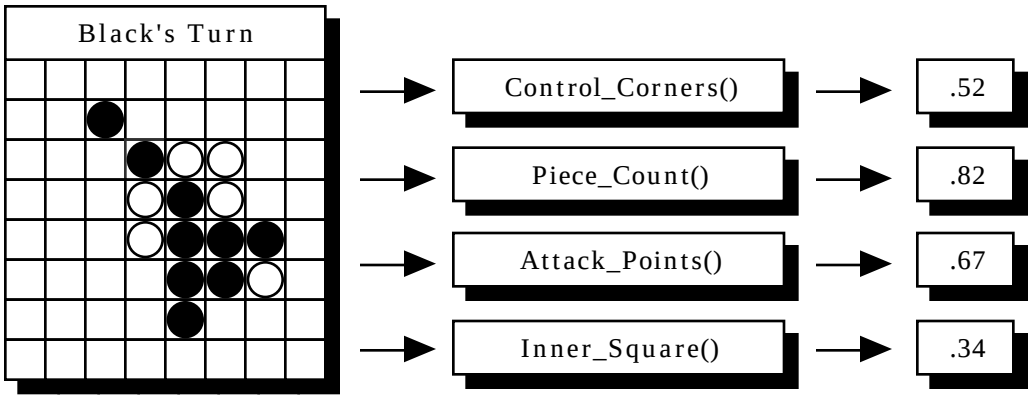


Figure 2.8: Sub-goal static evaluators for Othello

Once the sub-goal evaluators have returned their values, it is the job of the combining function to weigh these values and combine them in some way to arrive at a final goodness value. The sample combining function for Othello is displayed in figure 2.9. How these goodness values guide a computer's choice of moves is described in the MINIMAX section below.

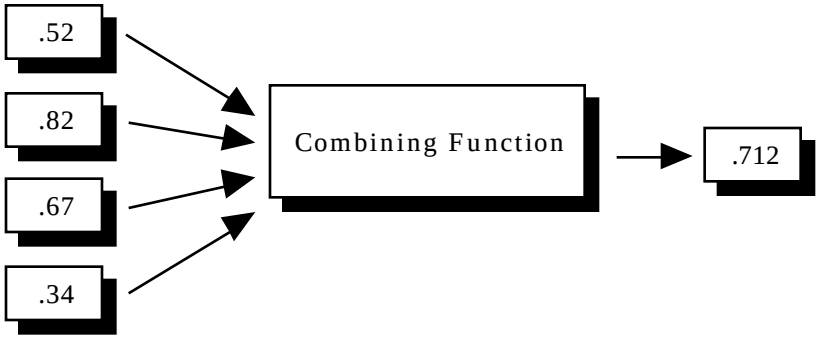


Figure 2.9: The Combining function for Othello's sub-goal evaluators

If a game can be solved, the programmer may wish to attempt to write a static evaluator for it. If the static evaluator ever reaches a point where it can partition the positions into two discrete sets, where every winning position has a goodness value above some value V and every losing position has a goodness value below some value V, then the evaluator is *perfect*. At this stage the programmer can eliminate time-consuming exhaustive search and solely use the perfect static evaluator with a MINIMAX search depth of 1. However, even for small games such as Tic-Tac-Toe, it is often impossible to write a perfect evaluator.

2.11 MINIMAX Heuristic

The heuristic we are utilizing is the MINIMAX [Winston84, pp. 116-120] strategy of look-ahead. Every position, or vertex in our game-tree is assigned a goodness value between 0 and 1, and one player tries to maximize that value while the other player tries to minimize it. Figure 2.10 illustrates the MINIMAX routine graphically on a small example with two levels of look-ahead. The final arrow shows

the optimal move chosen by the maximizing player. The number at the root shows the goodness value of the root position for the maximizing player.

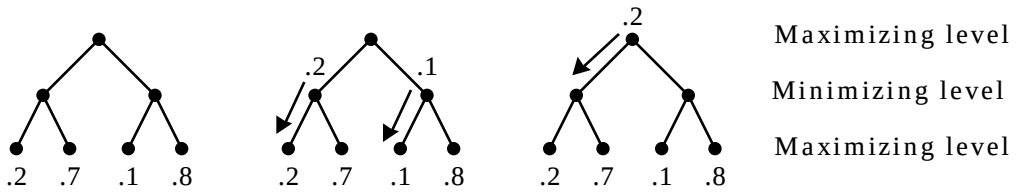


Figure 2.10: MINIMAX run on a small example with two levels of look-ahead.

One obvious parameter that can be changed is the number of plys for the look-ahead. One ply is one move by one player. Figure 2.10 shows This may be set to any integer number greater than or equal to 1, but the larger the look-ahead the slower the system takes to choose a move. The trade-off of increasing the look-ahead depth is that the computer makes better moves, as it is able to look farther into the future and scan the possible positions. In GAMESMAN, the user may change the look-ahead amount at any time.

3. Prior Work

This chapter serves to discuss other existing game toolkits, highlights the benefits and shortcomings of each and then concludes with the niche that GAMESMAN fills.

3.1 David Wolfe's games toolkit

David Wolfe created the games toolkit [Wolfe94] primarily to compute combinatorial game-theoretical values. As of this writing, there are four games written for it: Konane[12], Wyt Queens, Domineering, and Toads & Frogs[13], all described in [Berlekamp82][14]. The program has a text-only interface, although a graphical front end was written for Domineering [Garcia94]. An example of the text-based interface for Domineering is shown below in figure 3.1:

```
unix% games
Type 'help' and 'help help'
> domineering
Enter domineering position followed by extra <cr>
oo
oo

1|-1
> square = $
square = 1|-1
> domineering
Enter domineering position followed by extra <cr>
ooo
ooo

2|-1/2
> rectangle = $
rectangle = 2|-1/2
> rectangle ? square
>
```

Figure 3.1: The interface for the games toolkit. Here we ask for the game-theoretical value of the 2×2 square position and find out it is a 1|-1. Next we calculate the value of the 3×2 rectangle and find out it is 2|-1/2. When we ask which is a better position for the left player, we find that rectangle is better.

The program has the ability to calculate game-theoretic values of the games in the toolkit, and is an invaluable tool when working with them. Its ability to determine the relation between two values (e.g., what the relation between ↑* and * is) is tremendously useful when doing analysis. It can also print out the result in LaTeX form, which is essential when typesetting a paper.

What the toolkit lacks is a user interface. The games it does handle are input via a text-only interface, which has the advantage that it is easily portable to other

---

[12] Programmed by Michael Ernst (mernst@theory.lcs.mit.edu).

[13] Programmed by Jeff Erickson (jeffe@cs.berkeley.edu).

[14] Konane is described in glorious, full-color detail in [Bell83, pp. 132-133].

systems, but the disadvantage that its lack of a graphical interface makes the entry of positions tedious. The toolkit was also never meant to play the games it solved, only to report their values. There is no facility to allow users to play against the computer or other users, and it cannot handle games which end in ties. Also, even though the toolkit can report that a game is a winning game for the first player, it cannot highlight the best move to be chosen.

3.2 Anders Kierulf's Smart Game Board

Anders Kierulf created the Smart Game Board [Kierulf90], a workbench for game-playing programs that runs on the Macintosh™ personal computer. Its purpose was to support via the computer many activities traditionally done on a board and paper. It has an extensive game-tree editor, allowing the user to walk up and down the game tree and investigate different children and their sub-trees. The user can annotate the game tree as they wish, adding textual comments and graphic markers to explain certain points. An example of the graphical interface is shown in figure 3.2 below.

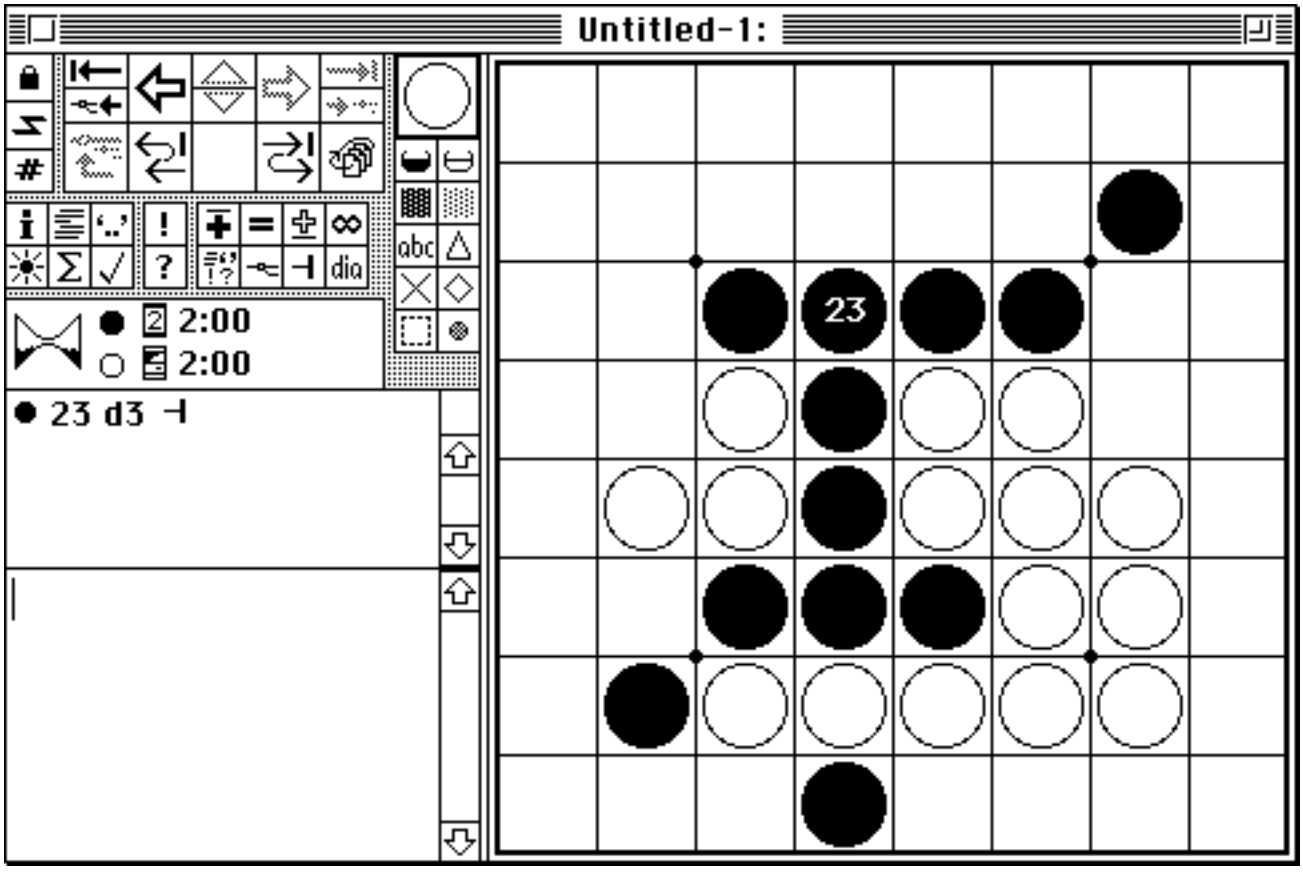


Figure 3.2 : The Smart Game Board Othello module user interface

Several very popular games have been written for it: Go, Othello, Chess, and Nine Men's Morris. The Smart Game Board supports playing algorithms, and several very strong computer opponents have been written for Go and Othello using the toolkit. A particular game need not have a computer opponent written for it; it

functions as a referee for 2-player games if there is no computer opponent. It allows players who wish to play a 2-player game to call another player over a modem and play a remote game.

Probably the most important feature of the Smart Game Board is its ability to record games, make annotations and store them in an archive. It is used in some Go and Othello tournaments to record games and review openings, and used in Go lectures to take notes. It can store all the games of a tournament in a single archive, which can then be transferred to an associate or stored in a publicly readable site.

Overall, it is an excellent toolkit, and comes the closest to providing the functionality that GAMESMAN achieves. However, there are a few missing pieces. It is written in a Modula-2 environment, and writing a game for the toolkit is not a task to be taken lightly. It is also only available on the Macintosh™ computer, and there are no plans to port it to another system. It also does not provide the ability to change the rules of the game at run-time, nor does it have any exhaustive-search capabilities to solve small games. In addition, since a tremendous amount of software engineering has gone into writing the game-playing programs for Go, Othello, etc., they are not free, so the casual player will probably not invest the money to buy the toolkit.

3.3 Rhys Hollow's Gamemaster

Rhys Hollow created the Gamemaster system [Hollow91] to play games over the network between Macintosh™ computers. There are many "rulebooks" written for it: Tic-Tac-Toe, Chess, and Connect-4 among others. It has an excellent graphical user interface, and allows users connected across a network to load and play several different games at once with the same remote opponent. It does not support a computer opponent or any computer intelligence, but it does act as a referee, constraining moves and signaling when the game is over. It also supports animation of the pieces or slots. An example of the user interface with the Othello module is shown in figure 3.3.

The games are written in Pascal, and the programmer simply needs to define the graphical user interaction, the rules and the ending conditions. These rulebooks are not themselves applications – they are documents that are opened by the main Gamemaster application. This means they are smaller than if they were a stand-alone application, and can be easily stored on Macintosh™ archive sites.

The lack of a computer opponent is the biggest shortcoming of this system. It doesn't have a game-solving engine, so even if a game was small enough to exhaustively search, it wouldn't be able to solve it. It also doesn't have any software hooks to allow a user to change the rules of a particular game.

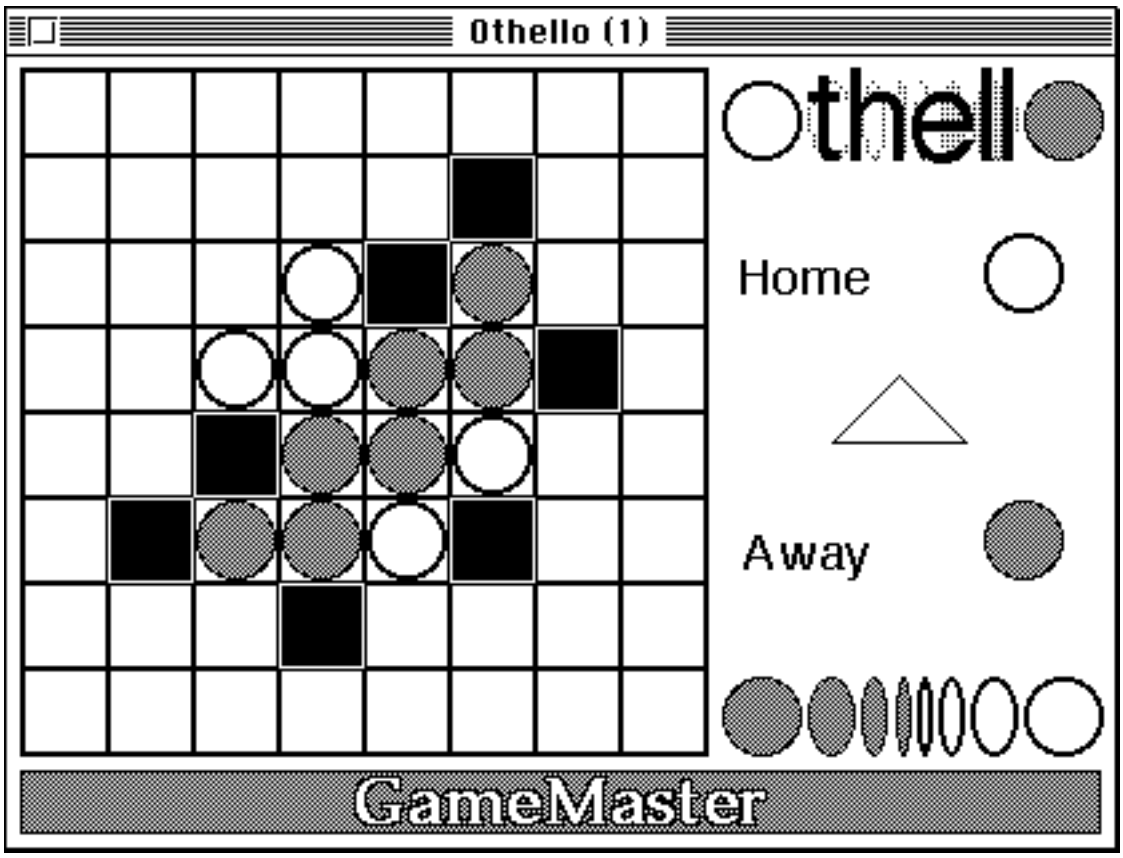


Figure 3.3: The Gamemaster toolkit with the Othello rulebook. The available moves are shown in black, and the pieces animate when being captured. The triangle at the right indicates whose turn it is to play.

3.4 What does GAMESMAN provide?

These three game systems were written for three different groups of users. The games toolkit was written for combinatorial game theoreticians, the Smart Game Board for players very serious about playing one particular large game, and Gamemaster for users on a Macintosh network who wish to play 2-player games against each other. GAMESMAN is for players who want to have the ability to play small, 2-player games in a graphical way like Gamemaster, but with a little of the game-theoretic concepts of games, and a little of the game-tree exploration of the Smart Game Board. GAMESMAN is the only toolkit that allows users to modify the rules of a game, play against a perfect opponent, receive a hint for the best move by a perfect oracle, easily prototype a new game, and quantify the value of various playing strategies against a virtual oracle that knows the complete game tree.

4. What games can be solved by GAMESMAN?

GAMESMAN can solve any small game that is finite, 2-person, and has perfect, or complete information. These terms are discussed in greater detail below.

4.1 Finite

A finite game is one that must terminate after a finite number of turns by each player. However, games in which it is possible to repeat a position are allowable by GAMESMAN, as long as the repeat position is treated in one of three ways. It could be disallowed completely, as the game Go does, considered a tie, or be only reachable a finite number of times. Any of these strategies may be implemented within a module to confront repeat positions.

A simple example of a game that GAMESMAN cannot handle is one in which the goal is to raise a counter to the highest number, and a player may add -1, 0, or 1 to the counter on his turn, which begins at 0. A winning strategy would be to say 1 eternally, which if played by perfect opponents would result in a clearly infinite game, each player taking turns counting the next highest integer. If, on the other hand, an upper limit was set on the total, then the game would be finite, and therefore able to be solved by GAMESMAN.

4.2 2-Person

Three (or more) player combinatorial games differ in their behavior from two-person games, and have been studied by [Propp94], among others. Part of the inherent problem with multi-player games is that a player with a losing position can sometimes arbitrarily decide who wins. In a two-player game, any move from a losing position resulted in a win by the other player. However, there is not such a clear outcome with multi-player games.

A simple example of this is the game “1,2,...,4”, where players take turns saying either the number 1 or 2 on their turn. This number gets added to the total, which starts at zero. The first player to bring the total to 4 or above first wins. It is clear that this game played by two can be perfectly solved. A perfect strategy is to play first and say 1, then say the opposite of whatever the opponent says on the next turn. However, if played by three perfect opponents, the second or third player will always win, depending on what the first player says. The first player has complete control of the outcome, but cannot force a win for himself. Thus, it is arbitrary which number is chosen by the first player, and thus who the winner of the game will be.

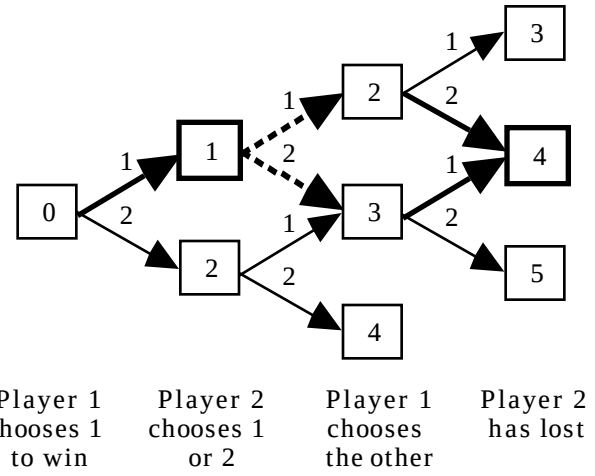


Figure 4.1: A simple example of the game "1,2,...,4" played by two players, Player 1 and 2. The bold arrows are the options played by perfect opponents. Dashed bold arrows are losing moves and solid bold arrows are winning moves. Highlighted boxes are the winning positions.

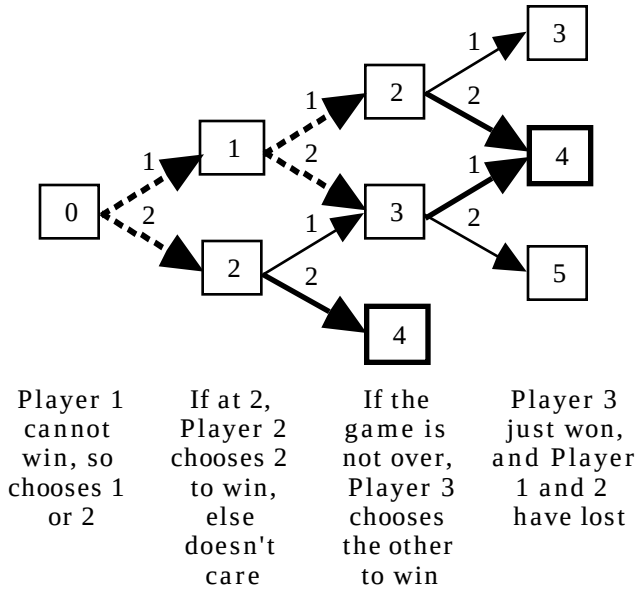


Figure 4.2: An example of why a 3-person game's winner can be arbitrary. This is the same game "1,2,...,4" played by three people, players 1, 2 and 3. The bold arrows are the options played by perfect opponents. Dashed bold arrows are losing moves and solid bold arrows are winning moves. Highlighted boxes are the winning positions. Player 1 has complete control over who wins with his first move.

Only 2-player games are supported by GAMESMAN, in part due to the difficulty described above of characterizing the value of games with multiple players, and in part due to the added complexity that would have been involved.

4.3 Perfect Information

Perfect information games restrict two typical components: hidden knowledge and chance. Without hidden knowledge, both players equally know as much as there

is to know about the game at all times. Without chance, there can be no probability or luck involved in the game or in the outcome when played between perfect opponents. This is the most important component of the categories of games GAMESMAN implements. By eliminating chance, we can guarantee that one player can always win (or at least tie), regardless of luck of the opponent. This obviously excludes any games involving dice or shuffled cards.

5. What are the mechanics of generating a game?

In this chapter, we discuss what a module designer, or programmer, would need to do to generate a game for GAMESMAN. The programmer first writes a module, edits the Makefile entry and then compiles it, which builds stand-alone applications that can solve and play the game.

A module is C and Tcl/Tk code that is compiled along with the main solver and interface code to create the custom game applications. The C code describes the rules of the game (what a move is, what winning conditions are, what the internal representation of a position and move are, etc.) and the Tcl code uses the Tk toolkit to describe how a user will graphically interact with the game.

The easiest way to write a module is to look at the previous modules that have been written and see which one is closest, in terms of number of positions and user interactions, to the game in question. If a close match has been found, it may be more efficient to modify a copy of the other game's already completed module than to write one from scratch.

Once the module has been completely written, the programmer adds an entry to the makefile so that when the module is compiled it can be automatically included with the others in the game database. This is not essential, as it is feasible to compile the module by hand, but the make facility provides a reasonably simple and accepted method of controlling the compilation.

After the makefile entry has been added, all that is needed to build the applications is to compile the module. This is normally executed with a call to `make <module-name>`, which compiles the sources and creates two applications: one small and text-based and the other relatively large and X-window-based. These two applications have roughly the same menus and control structure, just different front-ends. The reason two programs are generated is that the X application will not serve for users whose only interaction is through text-based display. Of course, if there are any errors in the module, the programmer iterates the previous steps until it is bug-free. Discussed below are the three tasks required when writing a module.

5.1 Set global variables

These globals inform the main code (and the interface) about the specifics of the game and the options the particular module supports. Currently, these are: the name of the game, help strings, and whether the game supports symmetries, graphics, a debugger menu, a game-specific menu, and tie games.

5.2 Choose a Representation for Positions and Moves

A crucial component of the module design is deciding upon the representations for the positions and moves of the game. These representations should provide a compact way of uniquely describing every possible position or move that could be encountered.

As mentioned previously, a position is a board, piece configuration and encoding of whose turn it is. Most of the games discussed in this paper are two dimensional board games, but the strategies discussed here can be extrapolated and used for almost any game. If the game is to be solved, there are subtle issues that must be taken into account that are discussed in the next section. Otherwise, the representation of the position just needs to provide a unique mapping from physical board position to the internal board representation.

A move is an edge in the game-tree, and is the action taken that transforms one position to another. The simplest representation of moves is also integers, as we discovered with positions. For a single-piece, a move usually consists of an integer or pair of integers which map to the representations of the slots on the board. With multiple-piece movements available, the move may be a linked list or an array of integers representing the pieces to move and their destinations. Whatever the case, it is not crucial that the representations be compact, as they are not stored anywhere. They are only used internally to specify which position to consider next, so virtually any representation with which the module developer feels comfortable is acceptable.

5.3 Additional issues if the game is to be solved

If the programmer wishes the game to be solved, there needs to be a way to store positions that have already been seen. The main solver routine uses an exhaustive search method to walk down the game tree in which the vertices are positions and edges are moves. An optimization strategy known as memoization is employed to prevent searching previously computed paths multiple times. This involves storing whether a position has been visited or not into a table. We also need to store the value of every position in order to compute the final value of the overall game. This leads to three constraints for our representation: it must be compact, unique for every position, and there must exist some constant-time mapping function (which the game programmer has to write) to access the table of values. We discuss these below and give an example.

5.3.1. Compact

As we will see in the “Limitations” section, the number of games that GAMESMAN can solve is limited by the exponential nature of game trees. Thus it is critical that whatever representation we choose to utilize for the positions be as compact as possible. For example, instead of using a 3×3 matrix of characters (9 bytes) for Tic-Tac-Toe, we could save space if we used a short integer (2 bytes) and found an appropriate mapping function to map to integers.

5.3.2. Unique

We need to store a value for every position, so the representation we use must be unique. If it were not, symmetrically distinct positions might map to the same representation, which would be incorrect if the positions had different values. A thorough discussion of symmetrically equivalent positions is provided in the "Future Enhancements" chapter. There does not need to be a valid position for every representation, just the other way around. The mapping must be one-to-one, but not necessarily invertible. Said another way, the mapping must be an injection, but need not be a bijection.

5.3.3. Constant-Time Mapping Function

The mapping function is a constant-time function that maps a position to its representation. It is used by the solver routine when checking if a position has been visited when storing and retrieving a value. The optimal, most compact map is one-to-one and produces a representation for every position, i.e., it is bijective. This way, the valid positions are maximally "packed" into the array, so that every representation could be a valid position.

5.3.4. Example: Tic-Tac-Toe

Let us use the example of the game Tic-Tac-Toe (described in Appendix B) to illustrate some of these concepts. In the game, a position consists of a 3×3 board with three different pieces that could be on any single slot at a time: X, O and the blank. The rules of the game state that the players take turns adding their piece to the board. No piece is ever removed. These constraints imply that the following equation is maintained: $\|X|-|O\| \leq 1$. This means that neither X nor O has more than one more piece than the other on the board at *any* time. The representation we choose is the positive integers, and our mapping function will map the board positions to the integers. If we think of a position as a ternary number in which the Blank is a 0, the O a 1 and the X is a 2, we come up with a mapping and representation that satisfies our three requirements: compact, unique, and constant-time. Let us analyze this further.

First, our representation is relatively compact, as each position only need occupy 2 bytes, since the number of possible positions (3^9 = 19683) is smaller than the positions that can be represented with 2 bytes (2^{16}=65536).

Second, there exists a unique integer for every position. It is not the case that there is a unique, valid Tic-Tac-Toe position for every integer (consider 3^9-1, or all X's), so this mapping is not optimal. We will find that it becomes increasingly important to find optimal mappings as the size of our games (and therefore the number of positions) grows. For this small game, however, our sub-optimal map will suffice. Third, our mapping function can be computed in constant time – the conversion of an abstract board position to a ternary number takes only a handful of multiplications and additions. Thus, our choice of representation and mapping function is sufficient.

5.4 Write the Subroutines

Once the representations and mapping functions have been decided, the next task is to write the subroutines in C and Tcl/Tk that use and manipulate them. These subroutines are summarized in Appendix A, and range from printing a position to generating all the possible moves to determining whether a position has a primitive value (e.g., three pieces in a row would be a primitive lose for Tic-Tac-Toe). At least ninety percent of the effort that goes into writing a module is spent creating these subroutines.

6. Categorizing Games

Categories exist to assist programmers in determining which, if any, existing modules are similar to their module. If a similar module already exists, it is often easier to modify a copy of that module than it is to create a new one. Thus by utilizing categorization, a programmer can reduce the time and effort it takes to create a module. In this section we discuss categories based on two components of games: the maximum number of positions a game will have and the types of user interactions to expect. These categories are in no way expected to be empirical, as it would be beyond the scope of this project to summarize every finite, two-person perfect-information game ever invented. However, it is hoped that these categories cover a broad spectrum.

6.1 Categorizing games by total number of positions

These categories are based on equations determining the maximum number of positions that can ever be generated by a game, given a fixed starting position and terminating criteria. This number is helpful when optimizing the mapping function, the importance of which was described in the previous chapter. For example, we concluded that a simplistic ternary number mapping function for Tic-Tac-Toe was sub-optimal. It did not make use of the fact that the number of X's and the number of O's differ by at most one for any position, since the players alternate placing a single piece and there is no capture. It is important to note that an optimal mapping function for one game in a category would be optimal for all games in that category. Discussed below are the five major categories we have considered, as well as one for hybrids – games that combine aspects of several categories.

The upper bound on the number of moves gives us a rough estimate of how long a game might take. For example, we can be guaranteed that a Tic-Tac-Toe game played on a 3×3 board will end in 9 moves, but a game of Go with slightly modified rules[15] played on a 19×19 board may last $3^{(19*19)}$, or 1.74×10^{172} moves!

For each of the following categories, we calculate the upper-bound of the number of moves and positions as a function of slots and provide examples for each one. Throughout the calculations, it is assumed that there are only two types of pieces, which we will call X and O. We have not taken multiple pieces into consideration, but the equations would follow the same general form, albeit with a bit more complexity.

6.1.1. Dart-Board without Capture

The games in this category are the simplest of all; a move consists of placing a single piece onto the board. Rearranging or removing existing pieces is not allowed, which is why the category is labeled "dart-board" – the moves are similar to

[15] Our modified rule would allow every position, and hence every ternary number from 0 to $3^{(19*19)}$ to be reached in the same game.

throwing darts across the room at a board. The board itself could be the pattern on the dart-board, and the darts thrown one at a time by alternating players would be likened to the pieces. The starting board here is assumed to be empty[16], and play continues as each player places a piece until all slots on the board are occupied. The game may end before the board is filled, but when calculating the upper bound we assume that it is prolonged as long as possible.

6.1.1.1 Number of moves upper bound

The upper bound on the number of moves for this category is therefore easy to compute. It occurs when all of the pieces have filled the slots on the board, one at a time. Thus, the equation[17] is simply:

$$\text{Moves}(slots) = slots \tag{6.1}$$

6.1.1.2 Number of positions upper bound

The one important property maintained by all games in this category is illustrated by equation 6.2 below, which states that the number of X's minus the number of O's is always -1, 0 or 1.

$$\big|\,|\text{X}| - |\text{O}|\,\big| \le 1 \tag{6.2}$$

From equation 6.1 we know that at move *i* there are *i* pieces on the board. Those pieces are either evenly distributed among X and O when *i* is even and favor X by one when *i* is odd. These *i* pieces can be located in any of the *slots* regions of the board. Thus, the total number of positions is: for every move, the product of the number of ways to put *i* pieces in *slots* regions with the number of ways to rearrange those *i* pieces among X and Os.

Equation 6.3 does not have the characteristic multiplying factor of two to remember whose turn it is since we can assume the position determines this. For example, if we declare that X always goes first, we can then conclude that if there is an even number of pieces, then it must be X's turn, otherwise it's O's turn.

$$\text{Positions}(slots) = \sum_{\substack{i=0 \\ i+=2}}^{i \le slots} \text{even}(i, slots) = \sum_{\substack{i=1 \\ i+=2}}^{i \le slots} \text{odd}(i, slots) \tag{6.3}$$

Where odd(*i, slots*) and even(*i, slots*) are defined as:

---

[16] However, any starting position with any number of X's and O's is a valid starting position. As long as play continues with each player alternating placing one piece and not removing any, the calculations here still hold.

[17] Throughout this section, we will use C function prototype notation to describe these functions. Therefore this should read: the procedure "Moves" takes the variable *slots* as an argument and returns slots. So the upper-bound on the number of moves that a Dartboard game of 9 slots (like TicTacToe) will have is 9 moves.

$$\text{odd}(i, slots) = \binom{slots}{i}\binom{i}{\frac{i-1}{2}} \tag{6.4}$$

$$\text{even}(i, slots) = \binom{slots}{i}\binom{i}{\frac{i}{2}} \tag{6.5}$$

and $\binom{n}{k}$ is defined as always as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{6.6}$$

The even() and odd() functions can be combined into one function even_or_odd() by use of the floor(x) = $\lfloor x \rfloor$ routine:

$$\text{even\_or\_odd}(i, slots) = \binom{slots}{i}\binom{i}{\left\lfloor \frac{i}{2} \right\rfloor} \tag{6.7}$$

which simplifies 6.3 to:

$$\text{Positions}(slots) = \sum_{i=0}^{i \le slots} \text{even\_or\_odd}(i, slots) = \sum_{i=0}^{i \le slots} \binom{slots}{i}\binom{i}{\left\lfloor \frac{i}{2} \right\rfloor} \tag{6.8}$$

6.1.1.3 Examples

Examples of games of this type are Tic-Tac-Toe, Gomuku (five-in-a-row), and Hex, to name a few. In these games, since capture is absent, winning usually consists of a position-based goal being reached (such as n pieces in a row or connecting a chain across the board) so the maximum number of moves is seldom reached.

6.1.2. Dart-Board with Capture

Games in this category are similar to the previous category with the simple exception that on any move, a player adds a piece in the standard dart board manner, but at the same time may capture some number of opponent's pieces and either remove them from the board or convert them to his own. The beginning board position usually either begins empty or with a small number of "pilot" pieces.

6.1.2.1 Number of moves upper bound

These games do not benefit from the previous category's constraint that limited the difference of the number of X's and the number of O's on the board at any one time. On a single move, one piece is added, but many pieces may be added or flipped[18]. For this reason we are not able to come up with a tight upper bound on the number of moves. The only hard upper-limit we can use is that no slot may ever contain anything other than one of three pieces, an X, O or 'blank' piece. Thus the longest game possible would be one that covered every single position. This gives us the equation for the maximum number of moves as:

$$\text{Moves}(slots) = \text{Positions}(slots) \tag{6.9}$$

6.1.2.2 Number of positions upper bound

In a similar light, it is difficult to determine a hard upper bound for the number of positions. Some games whose moves involve multiple piece removals may indeed push this upper bound, so this mapping may be near-optimal for them. For others, it may be necessary to consider particulars of the game, and modify the mapping function to achieve optimal results. For the general case, the equation is:

$$\text{Positions}(slots) = 2 * 3^{slots} \tag{6.10}$$

To see how this was derived, consider that every ternary number using $|slots|$ digits may be generated by the mapping function. For every position, it usually matters whose turn it is, so the number of positions doubles to compensate for storing two copies of every physical board position, one if it is X's turn next and one if it's O's turn next.

6.1.2.3 Examples

Many popular games fall into this category, such as Go, Othello (also known as Reversi), and Pente. Winning for these games usually involves capturing the most pieces by the end of the game, however the "end of the game" is be defined. Sometimes, as in the case of Pente, a win can either be due to a position-based advantage (5 in a row) or by surpassing a certain number of captures (5 pairs in this case).

6.1.3. Rearranger

As the name suggests, the objective in these games is to be the first to switch places with the opponent, or manipulate the pieces into some winning configuration. There are no pieces removed or added at any time to the game, which gives us good, crisp upper bounds on the number of moves and positions.

6.1.3.1 Number of moves upper bound

[18] That is, converted to be of the opponent's piece type. This term derives from the game Othello, which had playing pieces which were white on one side and black on the other. Pieces were captured by being flipped over.

The number of moves is purely a function of the number of positions for these games. We have no constraints available to us to reduce this number, as we have before with the dart-board category. Thus, as for dart-board with capture, the maximum number of moves is the same as the number of positions, since in the worst case every position could be reached in the game.

$$\text{Moves}(Xpieces, Opieces, slots) = \text{Positions}(Xpieces, Opieces, slots) \quad 6.11$$

In this equation, *Xpieces* is the number of pieces X has and *Opieces* is the number of pieces O has.

6.1.3.2 Number of positions upper bound

The upper bound on positions is the product of the number of ways to place the total pieces into the available slots and the number of ways to rearrange X's pieces with the total number of pieces. The result is multiplied by two to compensate for the need to store whose turn is next for any given position.

$$\text{Positions}(Xpieces, Opieces, slots) = 2\binom{slots}{Xpieces + Opieces}\binom{Xpieces + Opieces}{Xpieces} \quad 6.12$$

6.1.3.3 Examples

Some popular examples of rearranger games are Fox and Geese, Chinese Checkers, the 'L' Game by Edward de Bono, Fox and Geese, and Hoppers. The goal for most of these games is to either be the first player to swap places with the opponent or to reach a certain configuration before the opponent. In most of these games, the number of moves in a game is far, far less that the upper-bound of the number of moves.

6.1.4. Impartial Removal

This is the only category we have considered so far that is impartial. Impartial games satisfy the condition that "from any position exactly the same moves are available to either player" [Berlekamp82]. This helps us in several ways. First, we do not have to store twice the number of positions to remember whose turn it is. Second, it simplifies our moves and positions calculations, as typically pieces are only of one type. The games almost always begin with a full or empty board. Impartial Addition games are exactly the same as Impartial removal games, but the players add pieces to the board instead of removing pieces from the board. All calculations for the number of moves and positions hold for these games as well.

6.1.4.1 Number of moves upper bound

A move consists of only removing (or only adding, they are equivalent) some subset of the board's pieces. In terms of the number of total moves, the worst case is when each player removes only one piece at a time. As a result, the upper bound on the moves is simply:

$$\text{Moves}(slots) = slots \tag{6.13}$$

6.1.4.2 Number of positions upper bound

Since there can be any combination of pieces on the board, but all of the pieces are the same, this corresponds to the board representing any binary number with the number of digits equal to the number of slots, since the board begins full. Thus, the total positions are exactly as described in the following equation, which also makes for an optimal mapping function.

$$\text{Positions}(slots) = 2^{slots} \tag{6.14}$$

6.1.4.3 Examples

Examples of impartial removal games are Dots and Boxes, Nim, Tac Tix, Kayles, and other variations. The particular aspects of some of these games allow us to give a lower upper-bound for the number of positions, so it may be useful to fine-tune the mapping function for the specific game for this category.

6.1.5. Partisan Removal

For games of this type, the board begins with a full set of pieces. Each person on his turn moves a piece or pieces around and may remove opponent's pieces. The object is to either capture a special piece, reduce the opponent to a certain number of pieces, or capture the most pieces before a certain stage in the game is reached. In no game are pieces added to the board. This is the crucial difference between this and the seemingly identical “dart-board with capture” category.

6.1.5.1 Number of moves upper bound

We do not have any natural constraint on the number of pieces nor on the moves. It is feasible that a game could continue and cycle through all of the possible positions, and for this reason, as we have seen before, the number of moves is

$$\text{Moves}(Xmin, Xmax, Omin, Omax, slots) = \text{Positions}(Xmin, Xmax, Omin, Omax, slots) \tag{6.15}$$

In this equation, <X | O><min | max> is the minimum or maximum number of pieces that either X or O can have at any point during the game. The variable *slots* is, as always, the number of slots on the board.

6.1.5.2 Number of positions upper bound

The equation for this upper-bound is based on a result we derived earlier for the rearranger category. If we freeze the number of pieces on the board, then the number of positions is exactly the same as the number of positions for the Rearranger category from equation 6.12. We will rename that result as the function RearrangerPositions(). Thus, for our total positions, we simply sum over the number

of pieces X and O could ever have on the board at one time and then call our other position-calculating subroutine.

$$\text{Positions}(Xmin, Xmax, Omin, Omax, slots) = \sum_{i=X\min}^{X\max} \sum_{j=O\min}^{O\max} \text{RearrangerPosition}(i, j, slots) \quad 6.16$$

6.1.5.3 Examples

Many popular games, such as Roundabouts and Mancala are members of this category. Unfortunately, two very popular games, Chess and Checkers (a.k.a. Draughts) are members of this category but cannot be described by these equations since they involve more than one type of piece. This category probably contains more games than any other, as it is the most general. As before, the number of moves in an actual game is far, far less than the upper bound.

6.1.6. Hybrids

Games that do not fit into the previous 5 categories get clumped into this one, the "none-of-the-above" category. These games fall into two partitions: those that are in some way a union of the above categories, with different "phases", and those that are completely different from the games mentioned here. They may be different because they are not played on a board, or because the pieces and moves are totally unique. Nine Men's Morris, as described in Appendix B, is an excellent example of a hybrid.

If the hybrid game has several distinct phases, each of which can be broken down into one of the above categories (as is the case with Nine Men's Morris), then its number of moves and positions is easily calculated. The number of moves and positions is the sum of the individual categories' number of moves and positions, since the game can be thought of as two distinct games played one after another.

6.1.7. Comparisons

To gain an understanding of the relative number of positions for these different categories as well as an understanding of where certain games fit on the chart, we've graphed the number of positions compared to the number of slots in Figure 6.1 below:

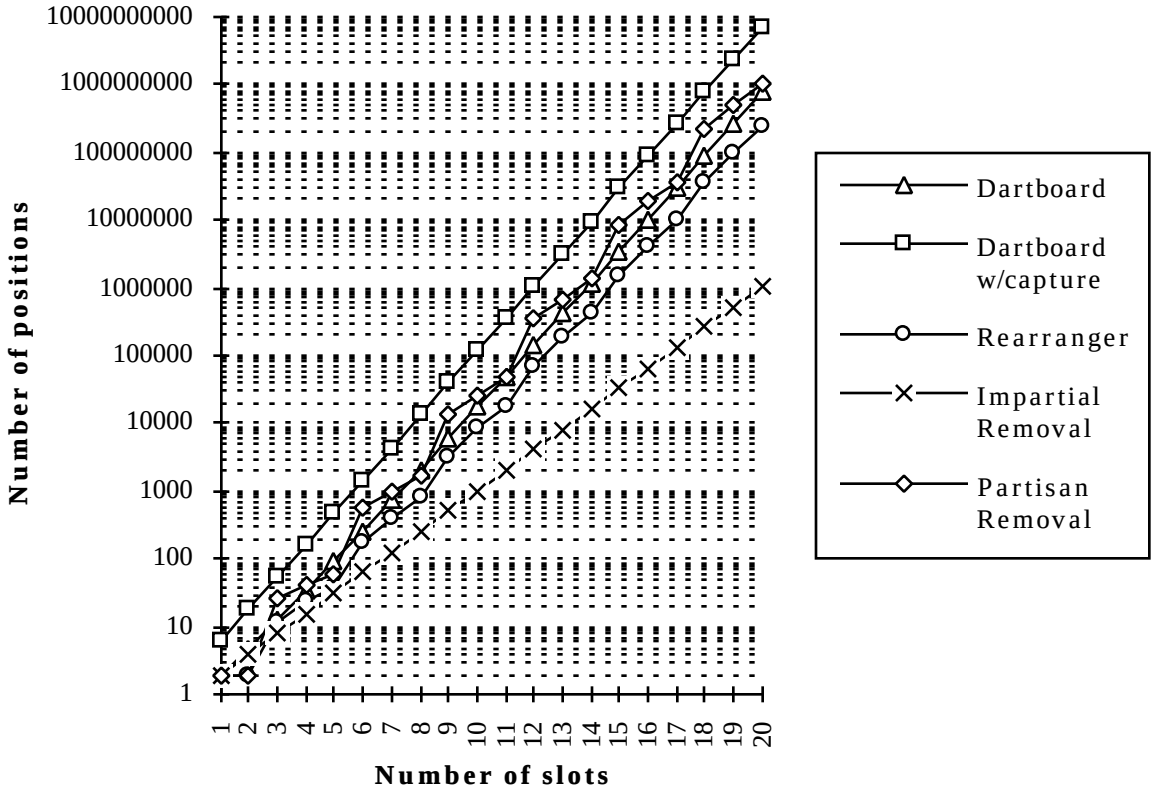


Figure 6.1: The relation of number of positions and the number of slots for different categories. The Rearranger values were calculated assuming the board was a third filled with X pieces, a third with O pieces and a third empty. The Partisan Removal values were calculated assuming the number of X and O pieces each varied between zero and a third of the board.

It is clear from this figure that the exponential nature of the games prevents us from solving games with large slot sizes. The Impartial Removal category, as expected, trailed the other values significantly since it only had one type of piece. Note that the number of slots was only 20 (less than a 5×5 board) when the number of positions for most of the categories was already past a billion. We will discuss this issue in the Limitations chapter.

6.2 Categorizing games by type of interactions

Here we categorize games on the basis of how a user interacts with the board during a move. This is useful so a game programmer can quickly see if there are games of the similar category implemented already, and use the same interface routines. Once again, these categories only address 2-D board games (the board may be N-dimensional, however), but GAMESMAN has the flexibility to use game-specific interface routines to handle games of a completely different type.

These categories describe a class of interactions that would be packaged as a separate interactions library. These libraries would be accessed by the module when

programmers wanted textual or graphical user input. This would eliminate the need for them to write code that was common to many other games. The overall effect is that a module is easier to write due to the reusable library of interface code provided within GAMESMAN.

Described below are the four categories we have chosen. They are: (single- and multiple-piece) (removal/placement and movement) interactions. The following illustration demonstrates how these interactions build on each other.

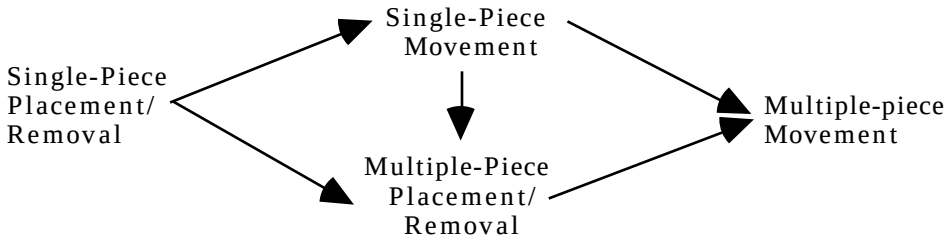


Figure 6.2: The interconnection of move-selection interactions.

For most games, a user's turn consists of a single user's move that would be one of these four categories. Multiple turns occur either when one player is unable to move, has lost a turn, or when certain circumstances arise, such as in Checkers and Mancala. This does not constitute a separate category, since the individual interactions are still the same, albeit executed multiple times. GAMESMAN has facilities to handle multiple turns for any game. It does this by asking the game's rules "Whose turn is it now, at this position?". Every game in GAMESMAN must write the subroutine `WhoseTurn` that answers this question.

6.2.1. Single-piece removal/placement

These interactions are the simplest of all, and are implemented by all dart-board games, as well as partisan removal games in which the piece to remove is chosen by the user. They consist of a user choosing a single slot on the board to either add (placement) or subtract (removal) a piece.

6.2.2. Single-piece movement

Single-piece movements are the most popular interactions with board games, and are required for rearranger and partisan removal games. In this category, the user chooses a single source, or FROM slot, and a single destination, or TO slot, which completely describes the move. Notice that these could be built from a pair of single-piece removals or placements, but we may not want to use same interactions discussed above for the move as there may be a more intuitive approach. This is discussed in detail in the following chapter. The FROM might include the special constant OFF_THE_BOARD, if the game allows a user to remove his own pieces on his move.

6.2.3. Multiple-piece removal/placement

This category builds on the previous two, and is implemented by impartial removal games. It may involve selecting pieces one at time as in the removal/placement case, or a drag metaphor from single-piece movements. In this interaction, there are several pieces to be selected, and the user may use several methods available to choose the subset of pieces desired. If a game allows both multiple and single-piece removal or placement, it falls into this category. That is, single-piece selections are simply multiple-piece sets of cardinality one.

6.2.4. Multiple-piece movement

Multiple-piece movements are not very common, and are the most complicated of interactions. In this interaction, the user wishes to choose a set of pieces and move them to another spot, perhaps separating them and manipulating them as well. The interaction library draws from subroutines used to select multiple pieces, as discussed previously and also to select destinations, as in the single-piece movement case. As before, the destinations may include OFF_THE_BOARD.

6.2.5. Hybrids

This category is reserved for all interactions that were either not described above or which changed over the course of a game. Games in this hybrids category force a module programmer to incorporate game-specific interactions, since they do not fit into the interactions that already exist. Two examples of unique-interaction games are the L-game by Edward de Bono and Nine Men's Morris, both of which are described in Appendix B. The L-game requires the user to rearrange an L-shaped piece to another location, with twisting and flipping, then optionally rearrange an existing square. Nine Men's Morris contains two phases, a dart-board-with-removal phase and a partisan removal phase.

6.3 Examples of the categories of popular games

This is a summary of the categories (number of positions and interactions) of popular games, many of which were discussed in the previous sections. This is meant to give the reader a better understanding of what the categories mean, by seeing several examples.

Game	Number of Position category	Interactions category	Comments
Tic-Tac-Toe Gomuku Hex Connect 4	Dart-board	Single-piece placement	No pieces removed from the board. Tic-Tac-Toe is implemented as a module in GAMESMAN.
Othello / Reversi	Dart-board with removal	Single-piece placement	Captured pieces are not removed, just automatically converted.
Go Pente	Dart-board with removal	Single-piece placement	Captured pieces automatically removed.

Dodgem Chinese Checkers Fox and Geese Hoppers	Rearranger	Single-piece movement	No pieces removed from the board. Dodgem's "goal" areas can be considered another slot. Dodgem is implemented as a module in GAMESMAN.
The L Game	Rearranger	Hybrid	Two instances of single-piece movement, one for L piece and one for square.
1,2,...,10	Rearranger	Single-piece movement or Single-piece placement	Since there is only one counter, the interaction could also be Single-piece placement (this is how it is implemented in GAMESMAN)
Tac Tix Nim	Impartial removal	Multiple-piece removal or Single-piece removal	Both of these games can be implemented as multiple-piece removal or single-piece removal (which is how Tac Tix is implemented in GAMESMAN)
Dots and Boxes	Impartial removal	Single-piece placement	Counters need to be added to keep track of the boxes captured by each player.
Roundabouts	Partisan removal	Single-piece movement	Captured pieces automatically removed.
Mancala (Awari)	Partisan removal	Single-piece movement	Multiple turns sometimes allowed.
Chess Checkers / Draughts	Partisan removal	Single-piece movement (except in Chess when user chooses pawn promotion piece)	For these games, there are more than one type of piece per player, so not true partisan removal. Captured pieces automatically removed.
Nine Men's Morris	Hybrid (Dart-board with removal, then Partisan removal, then Rearranger)	Hybrid (Single-piece placement, then partisan removal)	Captured pieces are chosen by user in both phases.

Figure 6.3: Examples of the categories of popular games.

7. User Interface Issues

In this section we discuss issues and decisions made during the design of GAMESMAN's user interface and supporting libraries. We also describe issues a game programmer should understand when writing a game library: how to allow the user to interactively choose a move and how to display all possible moves.

7.1 Customization: giving users the options

We stress the importance of allowing the user to make all of the choices as to which interaction methods and displays are to be used. Whenever we discuss several options in the following section, attempts have been made to indicate which we thought were the most intuitive choices, and these would be set to the default values.

The user should be able to customize the interface via a preferences box, and the settings should be remembered for future gaming sessions. The settings should also be unique for different games. That is, even if two games are in the same category, their particulars may lead the user to choose different modes of interaction and display. In the case of human vs. human games where GAMESMAN only served as referee and helper, each player would be able to uniquely select his/her preferences.

The user-customization is not limited to the interface. Steps have been taken to parametrize every component of GAMESMAN and its modules. This benefits the user, as he is placed in a position of complete control. Any idiosyncrasy of the game that the user finds bothersome may be disabled or switched for another. For example, the user may not have any interest in the value of a game. In this case, all references to the value may be suppressed.

The disadvantage of parametrization is that the code becomes permeated by the conditional statements corresponding to the user's configuration settings. This is part of the cost associated with writing any good interface, and usually does not generate a significant increase in the length or complexity of the code.

7.2 Interactively choosing a move

Selecting a move is common to all games, regardless of type. It is crucial that the method for move selection be intuitive, because that is what the user will be doing most of the time. However, the various interaction categories warrant the need for separate (but hopefully relatively consistent) interaction models.

For every case below, if the selection was invalid, it is not chosen and a message indicating either why the selection was incorrect or simply what options *can* be selected is printed. It is important that the user be informed in *some* intelligent manner, as there is nothing more frustrating to a user than a beep indicating an invalid section with no indication what was being done wrong. Ideally

the computer would literally tell the user in English (or whatever language was chosen) what was wrong with a move.

One idea common to the Macintosh™ user interface is not to allow the user to choose invalid moves at all! For example, the system could disable (visually shown as graying-out) slots which cannot be chosen as determined by the constraints of the game. Clicking on a disabled slot would have the same effect as choosing a disabled menu – nothing would happen, save perhaps a message mentioning that the slot could not be selected.

As selecting moves is an integral component of *every* game, it warrants extra attention. Below are several interaction categories and corresponding suggestions for choosing a move.

7.2.1. Single-piece removal/placement

This is the easiest interaction a user would need to have with a game; a single click suffices to choose the single desired slot. In piece removal mode, the slot would contain the piece to be removed, and in placement mode, the slot would be empty. Whichever mode is being used, the interaction remains the same. If the user clicks on a slot that is invalid, nothing happens, except for an optional helpful message telling the use to click on an empty slot (or full slot if the game is single-piece removal). Figure 7.1 illustrates the way a user would correctly interact with a game of this type.

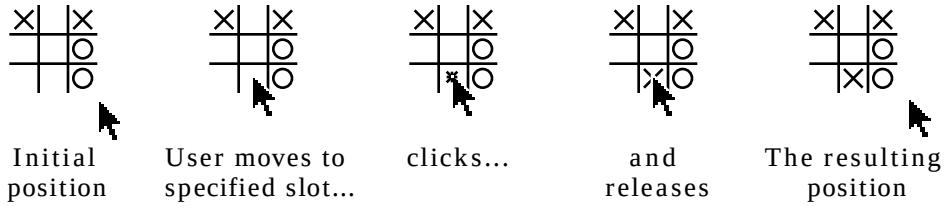


Figure 7.1: A graphical example of single-piece placement.

To help guide a user to valid slots, we propose to highlight valid slots when the cursor is placed over it. Figure 7.2 shows this technique.

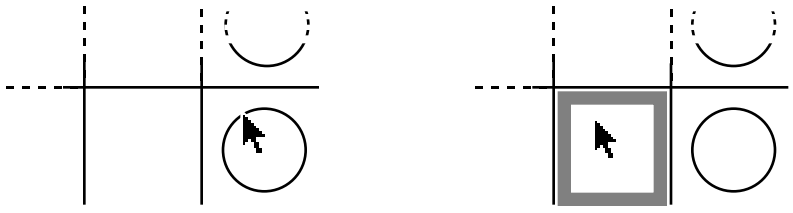


Figure 7.2: Highlighting a valid slot when the cursor is over it. Note that the cursor is in an invalid region in the image on the left, so the display does not highlight anything. In the image on the right, the cursor is over a valid slot, which is highlighted.

7.2.2. Single-piece movement

In this interaction category, the move consists of a pair of slots: a FROM, or source slot, and a TO, or destination slot. The user clicks once on the FROM slot, which "picks-up" the piece in question. Then, with the mouse button held down, the user drags the piece over to its destination, the TO slot, and releases the mouse button. The piece plops down in the TO slot[19] and the move is immediately confirmed. This follows quite closely with what a user does in a physical board game, so it seems perfectly intuitive. Therefore, this option should be the default whenever possible when implementing games involving single-piece movements.

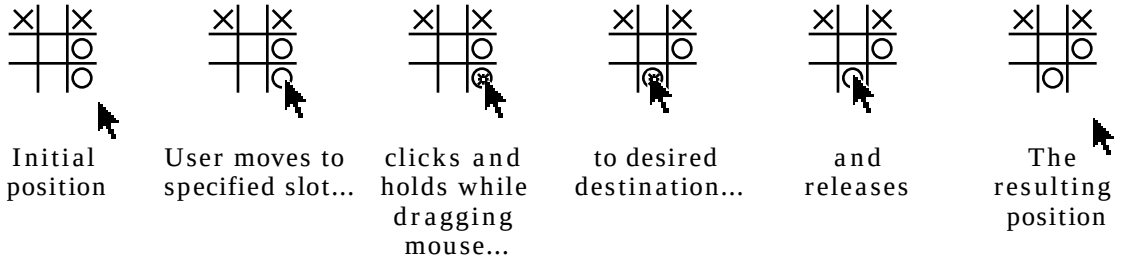


Figure 7.3: A graphical example of the interface for single-piece movement.

7.2.2.1 Outline vs. Opaque drags

The user should be given a choice, on any drag-style movements, whether to drag the outlines of the pieces or the opaque pieces themselves. In the real world, game pieces are opaque, therefore opaque drags are preferred. However, outline drags are the standard employed by the Macintosh™ finder, well-renowned as having excellent user-interface standards. Outline drags are also are useful when the machine on which GAMESMAN is running does not have a fast-enough graphics engine.

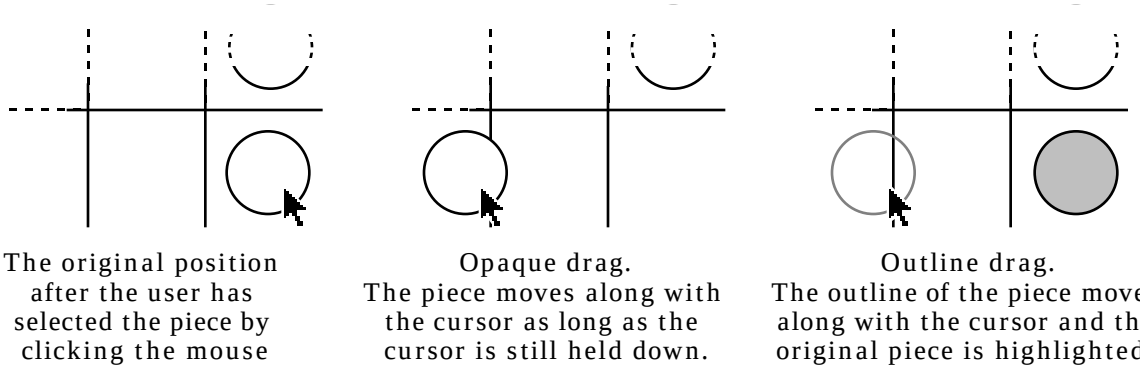


Figure 7.4: Outline vs. Opaque drags of the cursor

7.2.2.2 Computer-assisted movements

[19] When a piece is being dragging to its destination, the cursor may be anywhere in the FROM slot (as opposed to being over the PIECE in that slot if the piece is to be captured) when the mouse button is released and the selection will be accepted.

An option which many users may appreciate is computer-assisted movements. When selected, pre-defined constraints on a piece's possible TO slots allow the computer to 'guess' the destination of a selected piece. Consider a game in which pieces may only move to adjacent empty slots. Once the user clicked the mouse, the system would place the piece in the slot determined by the octant into which the vector defined by the current cursor position and original click position fell. This would mean that the cursor need only move one pixel in order to determine a new slot, which is a bit faster way of specifying a move. It is very important to also highlight the destination slot as described earlier to inform the user what the system thinks the TO slot is.

The downfall of this method is that there is no clear undo for the user. Without computer-assisted movement, if the user didn't like the move in question, they could always move the cursor back to the original slot, release, and the move would be canceled. However, with computer-assisted movement, the cursor would have to return to the original, exact *pixel* to cancel the move. This problem can be allayed with the addition of a circular "safety-region" that, when the cursor returned to, would effectively cancel the move. This safety region would be centered on the position of the initial click. The figures below illustrate the octant definition for 8-way moves and the quadrant definition for 4-way moves with the safety region in the center.

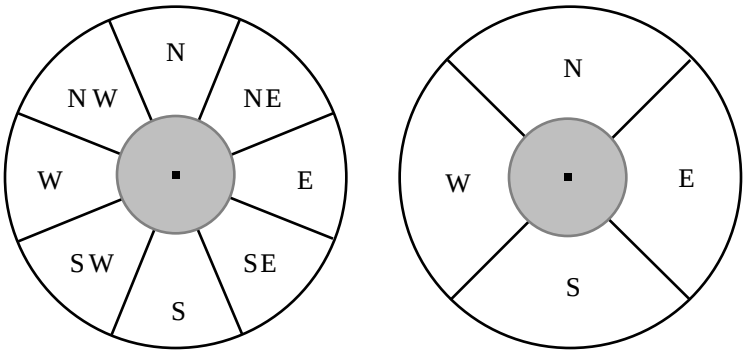


Figure 7.5: Computer-Aided Movement octant and quadrant with the safety region in the center which the cursor has to be outside of to register as a move.

7.2.3. Multiple-piece removal/placement

In this case, several slots are selected at once. As with the single-piece case, the interaction is the same regardless which mode (removal or placement) is chosen. So, without loss of generality, this discussion will focus on removal mode.

As the selections may be disjoint, the final selection may at times be a sequence of smaller selections. In this case, it is important to include the ability to correct a mistake without having to restart the selection process. For example, if the user has 5 separate clusters selected and accidentally chooses too many while attempting to select the 6th, he shouldn't have to re-select all 6 clusters to correct it – he should be able to somehow remove the unwanted pieces from his selected set.

To modify a selection that has been chosen, we could use the Macintosh™ Finder operation of shift-clicking to add or subtract pieces to or from our set. The shift key acts like a boolean NOT for set items – if the new selection was in our set before the shift-click selection, it isn't afterwards. If it was *not* in our set before, it *is* afterwards.

7.2.3.1 Single toggle click

The user could click once on each of the pieces to be removed, which would highlight that piece as 'pending removal'. The user would then invoke a separate click (e.g., the middle button on a three-button mouse or an "OK" button on the screen or the delete key on the keyboard) to confirm the choices (effectively an "I am done with my choices" button).

The advantages are that if a user is not familiar with click-dragging, this method seems the most intuitive. It is the same metaphor used when selecting which toppings are to be put on an ice cream – the "I'll take this one, this one, this one, and that's all" metaphor.

The disadvantages are that if the desired selection is a large M×N rectangle, M×N + 1 clicks are needed: one for every piece and one to confirm. If M and N are large, this is a horribly inefficient method of interaction. For small games in which the number of items to select at a time is small, the toggle-click may be a reasonable mode of interaction.

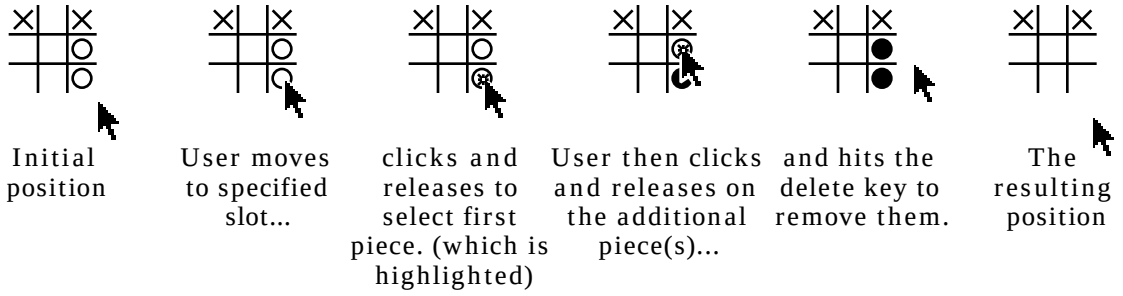


Figure 7.6: A graphical example of multiple-piece removal using the single toggle-click approach.

7.2.3.2 Click-drag selection rectangle

With this option, the user click-drags over the desired pieces, as in the Macintosh finder. A gray non-filled selection rectangle would be drawn with corners at the current and previously clicked cursor locations. Everything within or overlapped by the rectangle would then be selected. This is optimal for contiguous rectangular selections. If there is little or no drag, the piece under the clicked point is chosen. This means the system is a super-set of the previous single toggle click method, and as we will see, also a super-set of the selection line method sans the ability to make diagonal selections.

There are two times we could highlight the pieces selected: when the user releases the mouse (i.e., finished selecting) or as soon as the selection rectangle

drags over the pieces. The second option is preferable because the feedback for a properly chosen set and an improperly chosen set should be provided while the item is still being chosen.[20]

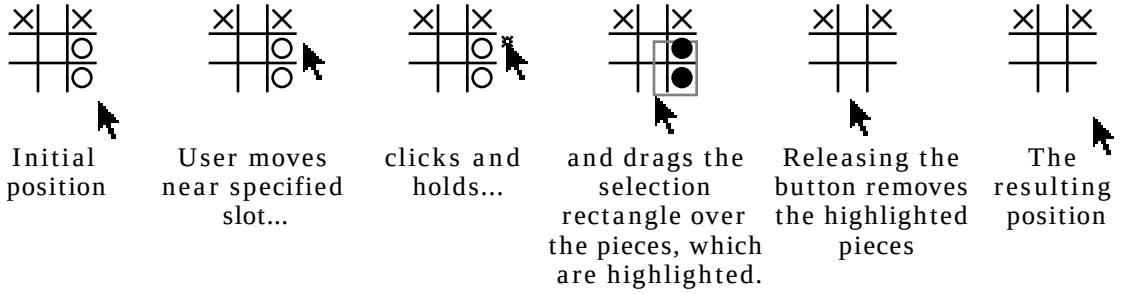


Figure 7.7: A graphical example of multiple-piece selection using the click-drag selection rectangle.

For diagonal pieces, we must revert to the slow process of clicking once for every piece in the diagonal. If we click-dragged from the upper to the lower item on the diagonal, all of the pieces within the rectangle defined by the diagonal endpoints would be selected, which is not what we want. The next section provides a method tuned precisely for linear selections along rows, columns and diagonals.

7.2.3.3 Click-drag selection line

This method is very much the same as the previous, except instead of a rectangle, a line is drawn with endpoints at the current and clicked cursor locations. Any pieces the line intersects are selected. As in the previous case, a quick click with little or no drag selects the piece directly under the cursor, and shift-clicking selects or un-selects disjoint sets.

This method is useful if the only selections that can be made are contiguous pieces in a single row, column or diagonal. It is the diagonal selection that separates this from the selection rectangle method. If the input is contiguous, the interaction is perfectly intuitive for selecting a line of pieces. However, if the selections are rectangles, it takes several 'sweeps' of lines to complete the selection. In this case, the previous selection rectangle is better.

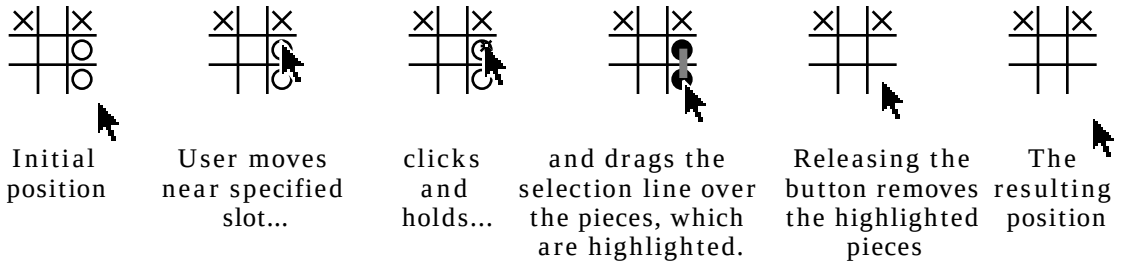


Figure 7.8: A graphical example of multiple-piece selection using the click-drag selection line.

[20] Apple's Finder version 6.0.x chose the former for file selection, but the new System 7 implements the latter, presumably because they learned their lesson.

7.2.4. Multiple-piece movement

There are two methods we can use to select and move several pieces at once. We can either select the pieces as a group and then move the group or make several single-piece movements and confirm that we're done. In the following discussion, any single-piece movement may be considered to be a multiple-piece movement of just one piece.

7.2.4.1 Group select and move

The selection and movement process can be thought of as separate actions. The multiple-piece selection process is covered in the previous "Multiple-piece removal/placement" section. So, in the following discussion we will assume we have successfully selected the pieces we desire to move, and they are somehow highlighted. Fortunately, the Macintosh™ Finder and most drawing programs have already tackled this question and have come up with a relatively good method that we can borrow. Once a group is selected, a click-drag on any member in the selected group moves the group as a whole.

One advantage of this option is fewer clicks are needed, which makes for faster move selection. Another advantage is that the user can divide his move into two tasks: choosing the pieces to move and then choosing the destination slots for them. This method is optimal for moves that have the TO slots at exactly the same orientation and spacing as the FROM slots, because the selected pieces can just be dragged over to the TO slots and dropped.

The disadvantage of this option is that a move is constrained because the TO slots must be the same spacing and orientation as the FROM slots. If a move corresponded of sliding and rotating and perhaps separating a series of linked pieces, this interaction method would not serve. In that case, the following "individual select and move" method is best. As an example, consider the move in chess when a king and a rook exchange places. If both king and rook were selected, a separate button would be required to perform the switch. However, the move is easily executed with individual select and move.

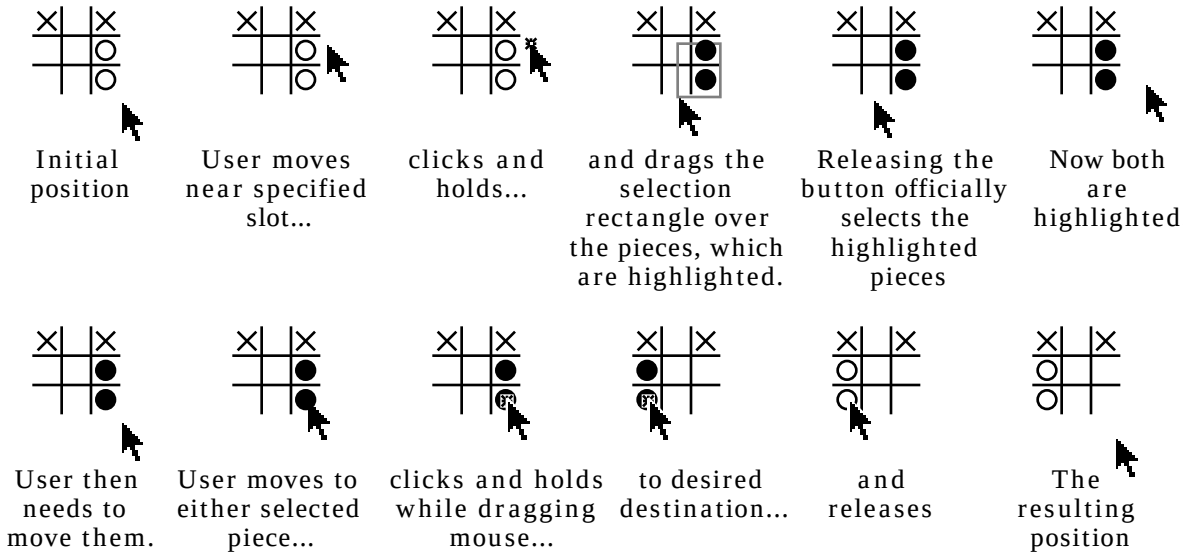


Figure 7.9: A graphical example of multiple-piece movement using the group-select-and-move method with the click-drag selection rectangle method to initially select the set of pieces to move.

7.2.4.2 Individual select and move

This consists of doing the same thing we did for single pieces as described in the "Single-piece movement" section above, but for every piece in the group we wish to move, with an additional click to confirm the completion of a move.

We now need to have a method to inform the system that we are done with our turn, i.e., done moving single pieces. This can be solved easily by either using one of the mouse buttons as a 'confirm' button or having a virtual button labeled 'done' or 'OK' that we would move our pointer to and click on to signal the completion of a move. While the latter may be more intuitive, the former is quicker. Once again, we can leave this option to the user.

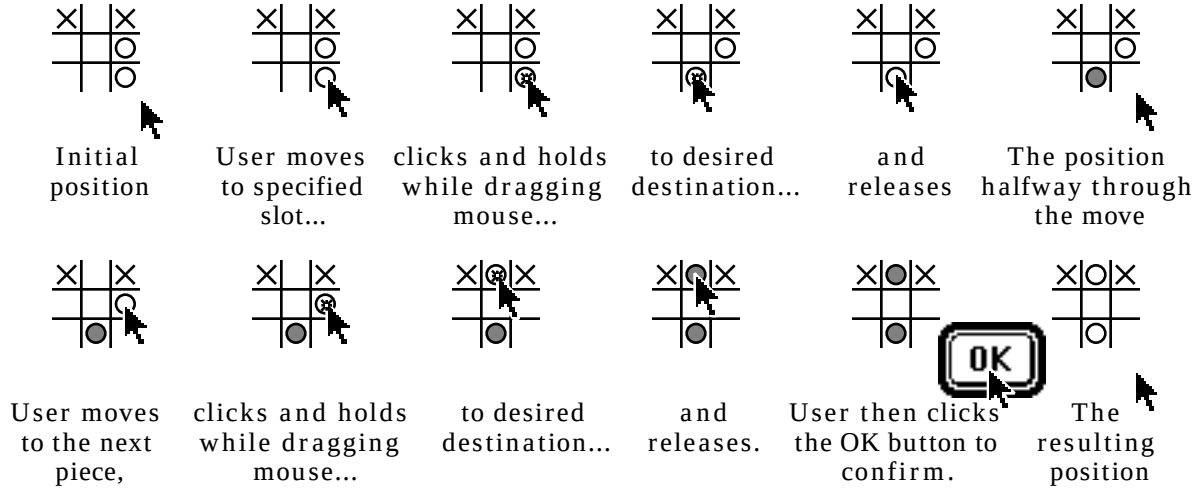


Figure 7.10: A graphical example of multiple-piece movement using the individual-select-and-move method with an OK box as confirmation of the completion of the move.

The advantage of this option is that it allows for the most generic manipulation. It can handle for moves that consist of rotation, removal, restructuring, flipping, and even separation of a series of selected pieces. The importance of this flexibility must not be overlooked, for it is reason enough to choose this option over others even against the myriad problems discussed below.

The disadvantages here are three-fold. The most obvious is the sheer number of clicks and drags needed to move a large selection to a new location. This cannot be overcome as it is inherently part of this option to require the user to choose every piece to be moved, one by one. This is what gives this option its power.

A second problem with this option is that the TO slot for some selected pieces may be the FROM slot for others. For example, imagine shifting a horizontal line of pieces one space to the left. Using the individual selection method, there is only one way to do this: move the left-most piece, then the one next to that, and so on. This will inevitably lead to confusion for users who choose the wrong order for selection.

The third problem is that the user cannot divide the move into the two discrete tasks of selection and moving. Since the user must operate in the mode of choosing a piece, moving it, choosing another, etc., the board at the part-way stage of a move may be confusing. If the user's concentration is disturbed midway through the move, he may forget which pieces were moved so far and which were to be moved. His move would have been corrupted, and if he accidentally hit the confirm button, would be incorrectly entered in as his option. Since a group-select-and-move is clearly divided into two tasks, it does not suffer from the problem of requiring a user to remember the midway-state of a move.

7.2.5. Hybrid example : Nine Men's Morris

Hybrids pose a unique problem. Ideally their interactions are a union of the previous methods of interaction, so it would be possible to use a combination of the special purpose library routines. However, as mentioned earlier, it is a virtually impossible task to attempt to categorize the interactions of all games, past, present and future. Therefore, when a game's interactions cannot be summarized as the union of the above categories, it is perfectly feasible for the module to include game-specific interactions.

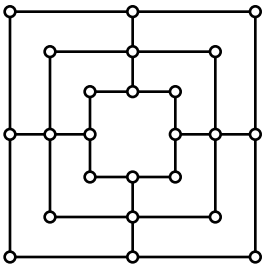


Figure 7.11: The Nine Men's Morris board and the 24 slots.

Nine Men's Morris is an example of a game that falls into the category of Hybrids. It has interactions that are a nice combination of the above categories. The

game's rules are described in greater detail in Appendix B, but a quick summary of the interaction is that it can be categorized into three phases as described below.

7.2.5.1 Phase I : Single-piece placement

The first phase involves players taking turns placing a game piece onto an empty slot. If the new piece becomes part of three pieces in a line (called a 'mill'), then the player who placed the piece may remove any opponent's piece not part of a mill. The interactions are single-piece placement at this phase, with perhaps an additional click on an opponent's piece to remove. For the removal click, all invalid slots could be grayed-out, leaving only those pieces that are valid. Invalid slots include any opponent's piece in a mill, any of the removing players' pieces, and any empty slots.

7.2.5.2 Phase II : Single-piece movement

After 9 pieces have been placed by each player, the game switches to the second phase, where each player moves a piece to an empty adjacent slot with the hopes of creating a mill. When a mill is formed, as in phase one, that player may remove any opponent's piece, *even* if it is in a mill (n.b., in phase I this is illegal).

The interactions here are single-piece movements, so we can use any of the interactions discussed above. If a mill if formed, removal is the same as in phase I. Computer-aided movements would be of a great help here, as the only valid TO slots are either north, south, east or west of the FROM slot.

7.2.5.3 Phase III : Rearranger

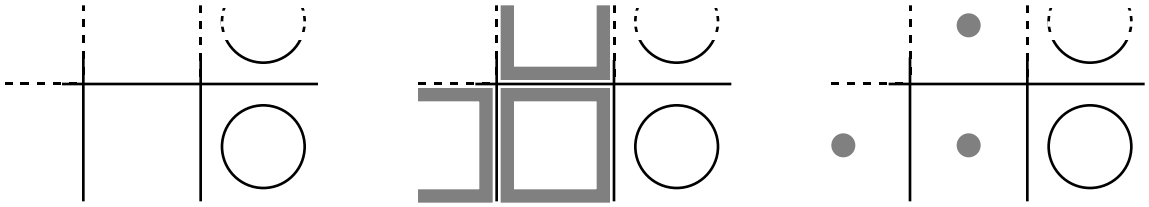
If a player has been reduced to three pieces, that player is given the extra freedom of being able to move a piece to *any* free spot on the board on his turn. In phase two, the TO slot had to be adjacent, whereas now the TO slot need only be empty. The same interactions used in phase two can be utilized, but computer-aided movements must be modified, since any empty slot on the board is a valid TO slot.

7.3 Displaying all possible moves

In this section we analyze the graphic that represents which moves are available to the user on a turn. The desire here is to find a single, static display that represents every option a user has for a certain position. Several displays are suggested for each category of game interactions mentioned previously, as well as options that could be chosen by the user to enhance the display.

7.3.1. Single-piece removals/placements

The possible moves are just the empty slots themselves, so it is quite easy to create a satisfying display showing the valid slots. In this case we have to find a method to indicate available slots. In order to highlight a particular slot, we could show a small colored/patterned dot (or "bullet") on the center of the slot, of have an outline of the slot that is available.



Original position | Slot outlines used to indicate the available moves | Slot bullets used to indicate the available moves

Figure 7.12: The two ways of highlighting available slots for single-piece removal/placement.

7.3.2. Single-Piece movements

In this category, a move consists of simply a pair of slots, the TO and FROM slots, which indicates the source and destination of a piece. Discussed below are three options that take three distinct approaches to the problem.

7.3.2.1 Arrows

In this option we draw arrows from the FROM slots to the TO slots. The only shortcoming occurs when arrows cross and overlap or are collinear. One solution to this is to have a 3-D view of the board with the arrows 'arcing' over one another, and the option to rotate the board around to view the separate moves. This would be hard to do in real time on a system that didn't have dedicated graphics hardware. Another solution to the crossing-arrows problem is to have different patterns and/or colors for different arrows. This may be useful for a few arrows, but the possibility still exists that the display may be tremendously crowded with colored and patterned arrows and may just confuse the user. In this case, it is probably best to cycle the display, showing moves (or arrows) one at a time.

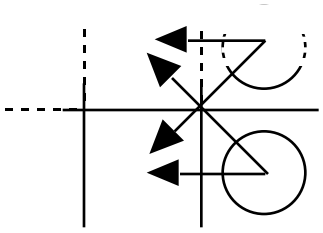
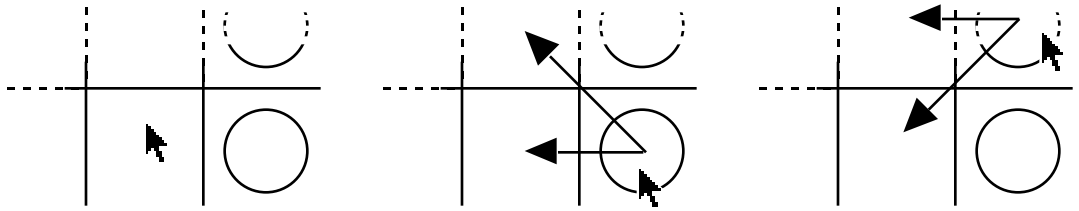


Figure 7.13: Arrows to indicate single-piece movement.

7.3.2.2 Cursor-initiated highlighting

An alternative to indicate single-piece movements is to display moves of a certain piece when the cursor passed over it. Valid FROM slots would be highlighted in some way (as suggested in single-piece removal/placements) and when the cursor passed over a particular FROM slot, the valid TO slots would be indicated, either by arrows that would branch from that slot terminating in valid TO slots, or by highlighting the TO slots in some other way. For the arrow case, this

would solve the problem of them overlapping, but not the problem of collinearity. A small horizontal offset and color difference given to the arrows terminating along the same ray would solve that problem.



The cursor over an invalid slot; no arrows are shown.

The cursor passes into the lower-right slot and its FROM slots are highlighted.

The cursor passes into the upper-right slot and its FROM slots are highlighted.

Figure 7.14: Cursor-initiated highlighting to alleviate single-piece movement arrow clutter.

7.3.3. Multiple-piece removal/placement

This category of game involves selecting slots on a board and either removing pieces from them or adding piece to them. It is very common for games of this type to have the constraint that pieces or slots, when chosen, be contiguous. A good example of a game in this category is Tac Tix, as described in Appendix B. In this game, a move consists of removing contiguous pieces in any row or column. Any orthogonal[21] board shape can be a Tac Tix starting position, but the standard game is played on a filled N×N board. The simplest method is to represent a move with a line for every valid subset of pieces that may be removed. This becomes confusing when there are overlapping lines, so the desire is to find a system that does not allow the lines to overlap.

The following diagrams below show how the lines could be drawn for 1×N boards, for values of N between 1 and 5. A small circle indicates that only that piece is to be removed. For the general N×N case, the 1×N board is simply replicated for the other N-1 rows and N columns. They should be replicated using an inward spiral, winding toward the inside so that rotational symmetry is maintained by the display for all rows and columns.

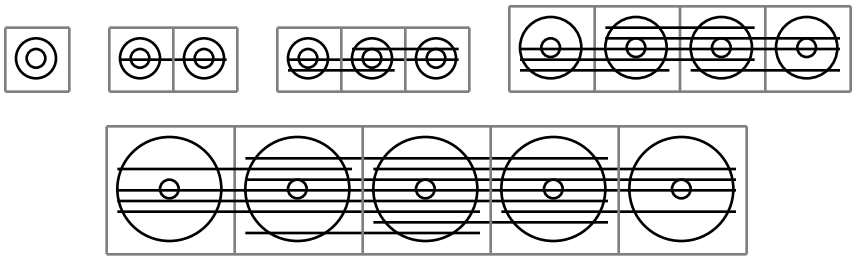


Figure 7.15: Multiple-piece removal/placement possible moves for 1×[1-5] Tac Tix boards.

[21] Although we restrict the game to be orthogonal in this case, the interface suggestions also hold for 2-D nim games of other topologies, e.g. triangluar or hexagonal.

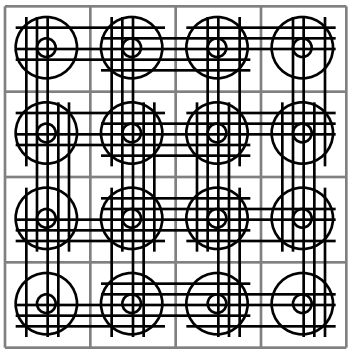


Figure 7.16: Multiple-piece removal/placement possible moves for the 4×4 board.

7.3.4. Multiple-piece movements

Multiple-piece movements are, as always, more complicated than single-piece movements. An attempt to find a static display that shows all the possible moves may be an impossible task. The image may be indecipherably covered with arrows or colors, indicating a multitude of possible moves.

Arrows would seem to be the most logical method of move displays. However, since the board will probably be *packed* with lines and arrowheads, it is probably best to make the cyclic display mode (displaying the moves, one by one, as described in the next section) the default for this category, and switch to using a static display only when the number of pieces or moves is small.

7.3.5. Improving the possible moves display

The preferences discussed here serve to enhance a static display. The first allows the user to reduce the moves that need to be shown at once, which eases the clutter and confusion that plague some display options. The second would be worthwhile when a single static display is too confusing or clustered.

7.3.5.1 Displaying value-equivalent moves

As previously discussed, moves can have several values (win, lose, tie) as determined by the value of the position to which they lead. All possible subsets (i.e., { {}, {W}, {L}, {T}, {W,L}, {W,T}, {L,T}, {W,L,T} }) of these moves should be able to be viewed. There could be three virtual toggle buttons labeled winning, losing and tieing. Whenever any one of these was toggled on, the moves of that particular set would be visible in appropriate colors.

The color scheme could be taken from traffic signals. Winning moves could be green (for “go”, the directive, not the game), tieing moves could be yellow (for “caution”), and losing moves could be red (for “stop”). However, red is also the color of the right player’s pieces, so perhaps a dark red would work here. For example, if a user wanted to see only winning moves, he would toggle the tieing and losing buttons to the OFF position, winning moves to the ON position, and only winning moves would appear. A ‘hint’ is the same as a winning move, so viewing all possible

hints is analogous to viewing all winning moves. In the case of a losing position (there are *no* winning moves) a hint would just display any move.

7.3.5.2 Cycle the available moves one at a time

When there are multiple moves, we could display one at a time and cycle to the next with a mouse-click or after a user-definable time-out. Hopefully, the static display that displays *all* the possible moves simultaneously is enough, but for the cases when the static display is too dense, this should be offered as an option. The user can choose among the following three display modes: static, cyclic, and a static one that cycles through all the individual moves and returns back to the static one once again.

8. Self-Evaluation

In this chapter we discuss the benefits of GAMESMAN as well as outline the limitations of solving a game exhaustively. Finally, we provide several optimizations to combat the limiting factors.

8.1 Benefits

The approach to game generation GAMESMAN takes offers many benefits to game designers, analyzers and players. Several of these are highlighted below.

8.1.1. Analysis tool

GAMESMAN allows the user to walk the position tree by playing a game interactively against the computer, and determine the value-equivalent moves and value for each position. This provides a tool for formulating what aspects of the game are crucial for winning and what are trivial and can be ignored. For example, someone playing chess might determine that every position in which he had lost his queen but his opponent had not was a losing position for him. A conclusion could be reached that the queen was of significant importance.

8.1.2. Consistent interface

Every application generated by GAMESMAN has exactly the same top-level interface. The only thing that varies is the specifics of playing the game, such as how to choose a move, how to display the possible moves, etc., which themselves are consistent among games in the same interface category, as mentioned in the "Categorizing games" chapter.

8.1.3. Facility to design, prototype and test a new game

Since writing a module can be accomplished so easily, it is possible to invent a new game and have a completed application up and running in order to analyze the properties of the game in about six hours. This is much faster than it would have taken to create a dedicated application from scratch.

8.1.4. Database to introduce and teach new games

New modules (and therefore applications that play new games) can be written easily. Hence, large database of unique, exotic games can be created. These games can be as old as chess or as new as something created last night by an inventive game designer. In any case, this database allows the non-initiated to be introduced to many different games, as well as understanding the basics of game theory. The "value moves" feature allows the user to see what the good (winning) moves are at any position and can help guide the user to formulate what a good strategy might be.

8.1.5. Hooks to incorporate game parametrization

As described earlier, there are software hooks in place that allow a game designer to parametrize the games. This means that when a user runs the game application, there are options that can be set to modify the game's rules. This 'breathes new life' into games that perhaps have been completely analyzed, solved and understood.

For example, Tic-Tac-Toe is considered trivial and is usually mastered by 10-year-old children. However, the strategy required for the Misère game is not immediately obvious. Even better, it is completely different from the strategy for the standard version. The misère game can be considered a completely different game, whose analysis can provide hours of enjoyment and insight for a Tic-Tac-Toe master.

It is suggested that module designers parametrize as *much* as possible of the game. This allows a user to examine and twiddle particular subtleties of the game. One particularly interesting component that can be parametrized is dimension. It is quite possible that a game's rules would not change drastically if played on a three-dimensional cube instead of the standard two-dimensional board. Several games map very nicely to three-dimensions, among these are Othello, Hoppers, and Nim to name a few.

8.1.6. Perfect opponent

In the process of performing an exhaustive search on the entire game tree, the computer builds a table containing the values for every position. The computer provides a 'perfect' opponent in the sense that it will never make an improper move, always referring to the tables when choosing a move, and forcing the opponent to take the worst possible position.

This 'perfect opponent' benefits both the analyzer who wishes to derive conclusions regarding the strategies of the game and also the enthusiast who wishes to hone his abilities.

8.1.7. Strategies can be evaluated for small games

Small games that can be solved can also have strategies written for them in the form of the evaluation functions. The quality of these strategies can be verified by the perfect opponent that can report what percentage of the positions the strategy reports a correct answer for (i.e., whether it is a winning and losing position) and what percentage it gets incorrect. Then the programmer can manually tune the strategy to improve the accuracy.

8.1.8. Fun

Most of the motivation for this project was a love of playing and analyzing games. Every component of GAMESMAN is fun – designing a new game, implementing its module, testing it, playing it to analyze the subtleties and deduce strategies, and watching someone else enjoy and learn from the application.

8.2 Limitations

In this section we discuss what inherent constraints limit the size and types of games GAMESMAN can solve. When solving a game using exhaustive-search, the computer builds a table of all of the positions ever encountered in a process called memoization. This not only saves repeat calculations, but is necessary to prevent infinite loops when move-trees are cyclic. Move-trees (and their corresponding position tables) are inherently exponential in nature, and thus severely limit the sizes of games we may consider. The constraints that limit us are space and time, and are discussed in detail below, with Tac Tix as an example.

8.2.1. Space

In the context of GAMESMAN, space refers to a computer's memory, or storage. The aforementioned position table needs to be stored in primary (or silicon) memory since it is frequently referenced and since secondary (or disk) memory is prohibitively slow. Therefore, one constraint that limits the size of games is the computer's primary memory. Virtual memory is a common method to boost the amount of primary memory a user appears to have; however, this space increase comes at the price of a very costly time increase.

8.2.2. Time

Let's assume memory isn't the constraining factor. A game's size may be limited by the *time* it takes to build the position table rather than the *space* it takes to store it. It seems that a good upper-bound on how long we can run a GAMESMAN program to solve a game for a dedicated machine may be a couple of months, and may be limited to a couple of days for non-dedicated machines. Since the exponential nature at which a game grows can usually be calculated with small examples, it may be possible to estimate how long it may take a particular game to be solved.

The first step is to figure out to which category the game belongs[22] and deduce how many positions that game will require, which is a function of board size. The second step is to determine how fast a small example is solved. These can be used to extrapolate and determine how long it will take to solve a larger game. The next section shows this calculation for Tac Tix.

Two games with the same number of positions may take astonishingly different amounts of time to solve. To investigate this in detail, let us consider the exhaustive search algorithm. The search algorithm is $O(|V| + |E|)$, where V represents the vertices, or possible positions, and E represents the edges, or possible moves. It is clear to see that for two games with the same V can have very different Es. The minimum E is a Minimum Spanning Tree, in which E is exactly $|V| - 1$, and the maximum E is for a fully connected tree, in which E is exactly $|V| * (|V| - 1)$, or approximately $|V|^2$. Thus, a very restrictive game with few moves per

[22] This may not be possible for abstract (i.e. non-board) games.

position (i.e., a sparsely connected tree) can be approximately |V| times faster than a less restrictive game with an equal number of overall positions.

8.2.3. An Example : Tac Tix on an Alpha workstation

Tac Tix is a classic Impartial Removal game. Although it may be played on any M×N board, we shall restrict this discussion to N×N boards only. The test case is a 4×4 board run on a dedicated single-user Alpha 21064 150Mhz workstation, and the time to solve is approximately 15 seconds. Our goal is to find out how large a board we can solve in 3 days (for typical workstations) and 3 months (for dedicated workstations). We would like to find a relation telling us how many slots we can solve given T time.

The number of positions is shown in equation 6.14 and is repeated below in equation 8.1.

$$\text{Positions}(slots) = 2^{slots} \tag{8.1}$$

which we invert to calculate the maximum number of slots, MaxSlots, given a fixed number of positions:

$$\text{MaxSlots}(positions) = \lg_2(positions) \tag{8.2}$$

Next we need a relation between positions and time. We know the speed of the machine from our calculation of the 4×4 board. We can then calculate how many positions per second the machine can compute:

$$\text{Speed} = \frac{2^{4*4}\ positions}{15 seconds} \approx 4370 \frac{positions}{second} \tag{8.3}$$

We can convert this to a function to compute for us the number of positions possible given a certain time. This assumes the game tree scales linearly, which from empirical evidence for this particular game is true. We can create a function that we'll call HowManyPositions() to return the number of positions reached given T seconds:

$$\text{HowManyPositions(T)} = \text{T}\ seconds\ * \text{Speed}\ \frac{positions}{second} = (4370 * \text{T})\ positions \tag{8.4}$$

Now we have all the relations we need. We take the amount of time given to us, T, pass it to the function HowManyPositions to find out how many positions that will be and pass that information to MaxSlots to calculate the maximum number of slots we could calculate given that many positions as follows:

The maximum number of slots in time T = MaxSlots(HowManyPositions(T))

$$= \lg_2(4370 * \text{T}) \approx 12 + \lg_2(\text{T})\ \text{slots} \tag{8.5}$$

Now, if we calculate the number of seconds in 3 days we get

$$3 days * 24 \frac{hours}{day} * 60 \frac{minutes}{hour} * 60 \frac{seconds}{minute} = 259{,}200 seconds \qquad 8.6$$

which when plugged into 8.5 gives us

$$12 + \lg_2(259200) = 12 + 18 = 30 \text{ slots.} \qquad 8.7$$

The number of seconds for 3 months is approx. 30 times that of 3 days, so plugging that in we get

$$10 + \lg_2(7776000) = 12 + 22 = 34 \text{ slots.} \qquad 8.8$$

Thirty slots corresponds to just a bit more than a 5×5 (= 25 slot) board. Thus, with 3 days, we can only barely solve a 5×5 board, and with 3 *months* we cannot even solve a 6×6 board. In fact, since the number of positions doubles with each added slot and we can solve 34 slots in 3 months, it would take a dedicated computer $2^{36-34} = 4$ 3-month cycles, or *an entire year* to solve the 6×6 board! This is why GAMESMAN can only solve small games and why alternate methods are necessary for larger games.

8.3 Optimizations

This section mentions several techniques to attempt to reduce the large time frames needed to solve even modest-sized games. Parallel and distributed computers will benefit every game, as will maintaining a database of solved position tables. However, the other optimizations are game-specific and may not be applicable to games in general

8.3.1. Parallel and distributed computing

Dividing the task of searching a game tree among several processors will certainly help defray the time expense, but doesn't give us any space savings. However, this optimization, as with most optimizations mentioned here, gives us only a linear time gain at best. This gain is a function of the number of processors assigned to the task and the amount of multiple writes and reads that are allowed to the shared memory containing the position table. With distributed computing, the maximum network transmission rate also enters into the speed equation, as a slow network may erode the gains provided by multiple machines. Whatever the final increase happens to be, it will certainly be faster than a single processor, and thus is worth attempting if time becomes the constraining factor for a certain game.

8.3.2. Stored position table

We saw in equation 7.1 that the number of positions for Tac Tix is 2^{slots}. For a 25 slot game (a 5×5 board), Tac Tix has 2^{25} total positions. An obvious idea is to store the table once it had been created. Since 2^{25} is only about 32 megabytes, it could easily be stored on even the smallest hard drive. Then, when other users wished to play a 5×5 Tac Tix, the table could be referenced from disk rather than solved from scratch once again. This saves lots of time at a fairly large space cost.

However, this only works for moderate-sized games, as there *are* limits to disk sizes. For example, the innocent little 6×6 Tac Tix game requires a whopping 64 gigabytes to store, which would overflow the average workstation's disks many times over.

8.3.3. Symmetry

Symmetry is the idea that some positions, although they might look quite different, are just simple rotations and reflections of each other. Depending on the game, these may or may not be symmetrically equivalent. If they are, then the move-trees rooted at the corresponding positions are themselves equivalent, and need not all be computed. For example, the eight Tic-Tac-Toe positions from figure 8.1 are symmetrically equivalent, and thus do not all need to be solved. This saves us both time and space equally, as we do not have to search these redundant positions nor do we have to store them in our table.

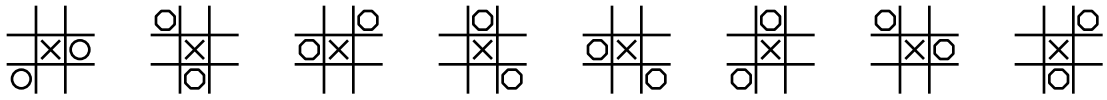


Figure 8.1: 8-way symmetrically equivalent positions for Tic-Tac-Toe

A game's rules determine the number of equivalencies, which remains constant throughout the game. A generic 2-D board game has 8-way equivalency shown above in figure 8.1: 0°, 90°, 180° and 270° rotations of the original board and 0°, 90°, 180° and 270° rotations of the reflection of the original board. Examples of games with 8-way equivalency are Tic-Tac-Toe, Othello, Nine Men's Morris and Tac Tix. Some games have directionality, i.e., a sense of "your side" and "my side". This limits these games to simply 2-way equivalency, the 0° and 0° reflection symmetries. Examples of popular games with 2-way equivalency are Chess and Checkers.

Symmetries are a way to partition all of the positions into sets. Only one canonical member of a symmetrical set of positions needs to be stored. Within the same game, different positions may have a different number of equivalencies. This is demonstrated with 1, 4 and 8-way symmetrically equivalent Tic-Tac-Toe positions in figure 8.2.

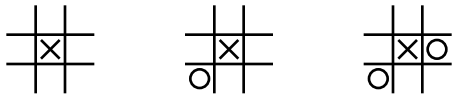


Figure 8.2: 1, 4, and 8-way symmetrically equivalent Tic-Tac-Toe positions

Since not every position in a certain game has the maximum amount of equivalent positions (8 in the case of Tic-Tac-Toe), the savings in time and space may not always be maximal. However, as the game's board size grows, so does the

percentage of positions that *do* have maximal equivalencies, and thus our improvement approaches the most we could get from our symmetry optimization.

8.3.4. Component-equivalent positions

This game-specific optimization is similar to symmetry in that different positions are in fact equivalent, component-wise. This means separate pieces of the board can be thought of as sub-components, and rearranging the orientation of these components would not change the position. This is best shown with an example, using Tac Tix. In the game, components that cannot be connected orthogonally can be considered truly distinct, and can be rearranged without affecting the position. Figure 8.3 demonstrates 3×3 Tac Tix component-equivalent positions. Note that the single piece and the "L" piece are different components, so they may be rearranged within the 3×3 board as long as they stay orthogonally disjoint.

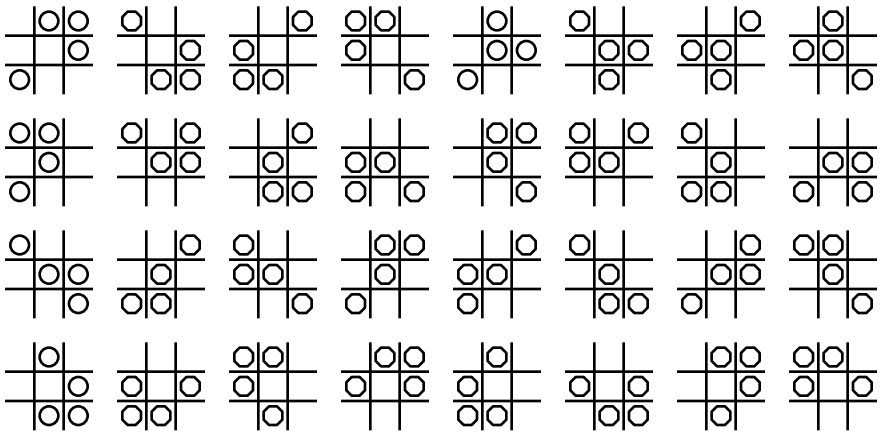


Figure 8.3: 32 Component-equivalent 3×3 Tac Tix positions

Much larger space and time gains can be achieved with component-equivalent positions than with symmetry only, since symmetries are a subset of all component-equivalent sets, as is witnessed in figure 8.5. However, unfortunately very few games can be decomposed into position-independent components to take advantage of this technique.

8.3.5. Delayed evaluation

Delayed evaluation is an alternative to exhaustive-search. The idea is that in order to determine the value of a game, the *entire* tree need not be searched. We exploit the definition of a winning position to accomplish this. Since a winning position means there *exists* a losing child, as soon as we figure out that we have found a losing child, we need not search our other children. We can return to our parent that we are winning! The other children are left unevaluated, or undecided. If during the course of a game we encounter an undecided position, we resume the delayed-evaluation search from there. Chances are we'll never encounter one, but if

we do, restarting the search from there may result in a massive delay[23]. This is the biggest shortcoming of this optimization – users may not want GAMESMAN solving a large game tree in the middle of play.

The worst case is no improvement over exhaustive search because there may be only one losing child for every winning position and we may always happen to search it last. The best case may be tremendous time and space savings, with the exact figure depending on how many losing children each winning position has and how well we can find them in order to search them first. We would need to have some notion of what a good position was in order to find losing positions with any consistency. This would require the system know what it thought a winning and losing position was, which up to now was not necessary. This requires adding intelligence to GAMESMAN, and is the final optimization discussed below.

8.3.6. Intelligence and Heuristics

The advantage of keeping intelligence out of a module is obvious. The programmer need not know anything about the strategies of the particular game, yet after exhaustive searching the game tree the computer can play as a perfect opponent. However, even with every single optimization mentioned above, the size of the game tree (and corresponding position table) is still daunting, and forbids any large game from being solved. How, then, do computers play chess? The programmer adds intelligence to the system and discards the theoretical beauty of exhaustive search and the resulting definitive solution.

Using heuristics and optimized versions of MINIMAX search, a computer opponent is programmed to look ahead at possible moves, and determine which one is the best given counter-moves and counter-counter-moves, etc. Fortunately, the MINIMAX algorithm is well-known, straightforward and requires only that the programmer write a static-evaluator. This is a black-box that takes a position and determines its “goodness” value, typically a number between 0 and 1. A goodness value of 0 means a win for the minimizing opponent, 1 means a win for the minimizing opponent. The computer, on its turn, attempts to find a move that maximizes the number, and the opponent is assumed to make the best move to minimize the number, hence the name for the algorithm. In essence, all the intelligence is encoded within the static evaluator.

This is the accepted method for writing a computer opponent for a board game. The speed at which the opponent responds with a move depends on how many plys ahead it is searching. The more moves available, the slower the machine will be, as it has to search all of them. Optimizations to this algorithm have it vary the depth of its search depending on how good or bad a future position may be.

When programmers add modules to GAMESMAN, they have several options for the computer opponent. They may choose to forego adding intelligence to the system and simply have the system exhaustively search the game tree in order to come up with the definitive solution and perfect opponent. If the game is too large,

[23] Reminscent of the typical coffee-break-length delay introduced by garbage collection.

they may choose to provide a static evaluator and let the internal MINIMAX algorithm choose the move based on the goodness values returned from the evaluator. They also may opt to dismiss computer opponents altogether, and simply program their module as a 2 player game, with the computer acting as referee. This would be useful if the primary desire was to play inter-network games with users at remote sites. The fourth and final option is only viable for small games that may be exhaustively searched. In this option, the programmer first codes the module sans intelligence and has the computer solve it. Then the programmer slowly creates a static evaluator, all the time checking its correctness with the "answers" provided by the solved position table. If all winning positions have a goodness value greater than V_w and all losing positions have a goodness value less than V_l AND if V_w is greater than V_l, then the programmer has written an evaluator to, in effect, *perfectly* determine the value of a position without search. At this point (which is seldom reached for obvious reasons) the programmer can feel confident that the static evaluator provides a *perfect* opponent. This is represented graphically in figure 8.4:

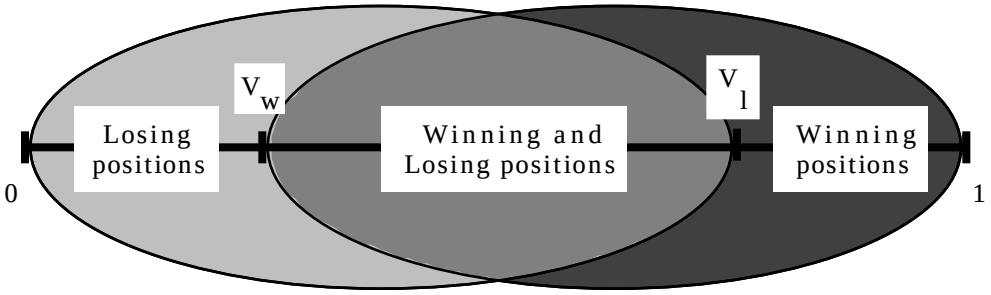


Figure 8.4: The average static evaluator. V_w is the lowest value of a winning position returned from the static evaluator and V_l is the highest value for a losing position. Since V_w is less than V_l, these regions overlap on the static evaluator number line, and the two sets of games are not partitioned at all. The smaller the intersection region, the better the static evaluator.

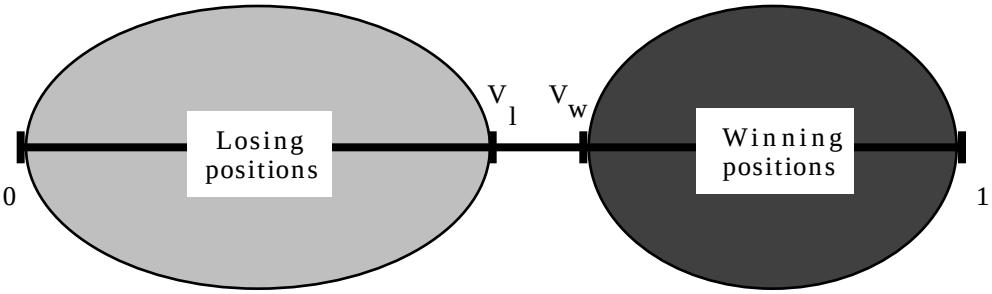


Figure 8.5: A perfect static evaluator. V_w and V_l are defined as before, but in this ideal case, $V_l < V_w$ and two sets are partitioned perfectly.

It is very unlikely that a static evaluator, no matter how complete, will be perfect. However, for games whose position table is too large to either fit in memory or be solved within reasonable time bounds, it is the only recourse for the programmer who wishes to have a computer opponent.

9. Future Enhancements

This chapter contains ideas for future enhancements to GAMESMAN that were beyond the scope of this Master's project, but may someday be added as time and energy permit.

9.1 Cross-network games

A user at a remote site could request a connection with another user, and either user would choose "Connect with Remote Player" on their local version of the GAMESMAN application. Once a connection was established, the two users could play in two-player mode against each other across the network. Aside from the slight transmission delays, the applications on each end would function the same as a single application in two-player mode, i.e., alternately prompting the users for their moves. These moves would then register on the opponent's screen. Either user would have the option of quitting the game and closing the connection at any time.

If other users wished to watch a cross-network game, they could connect with either opponent and request 'onlooker-only' access to the game. If granted by either opponent, the game would boot at the onlooker's site, but no input would be accepted – this would be an output-only window to the game. This feature would be useful if three users wished to participate in a GAMESMAN round-robin tournament. The players could alternately choose which two were playing the game at that time and which person would watch. This would allow the idle player to glean strategies from the other two by simply watching how the game progresses. This feature would be similar in function to what already exists at game servers, like the Internet Go Server (IGS) which allows anyone to watch a game in progress between two opponents.

9.2 Object-Oriented Graphical Programming

NeXT offers a graphical programming environment called NeXTStep, which allows a programmer to create a user interface graphically and iconically, saving many lines of code and reducing implementation time considerably. With this feature, instead of writing Tcl code to define how the player would interact with the system, the module programmer could graphically define it. For example, to create a Tic-Tac-Toe module, the programmer could choose a 3×3 blank initial board with 'X' and 'O' pieces, dart-board-style piece placement, a terminating condition of three-in-a-row, and a tie position of a filled board, all without typing a line of C code. As of this writing, designers at Sun™ Microsystems are investigating a GUI builder in Tcl.

9.3 Graphics & Animation

A popular game for the IBM-PCs is "Battle Chess". It is a standard chess game, except the pieces are animated figures (e.g., the queen is a woman, not a

chess piece) and when a capture is made, the capturing piece is shown defeating the captured piece in a short, animated battle scene on the chess-board. When pieces are moved, they walk to the destination. In this spirit, moved pieces in GAMESMAN should do more than just disappear from the FROM slot and reappear in the TO slot. They could roll, spring, energize, slide, bounce, blob, dig, walk, run, stagger, etc. If a capture is made, similar animations could be played.

9.4 Graphical Move History

This feature supports both multiple levels of UNDO and REDO, allowing the user to walk up and down the move tree and take a branch that had been previously skipped. This is a feature already found in the Smart Game Board system. For example, if the user chose a safe move that proved to be a win, it would be possible to back up the game tree and choose an unsafe move to see if it was still possible to win from there. It would also be useful to save this move history for later analysis.

If the ability to load and save games was added, then the system could be used to help solve endgame problems. For example, a problem involving only three or four pieces on a very large board could be loaded in and solved. So even though the complete game was far too large to solve, a small sub-tree could be broken off and solved independently.

9.5 Different computer strategies

This feature allows the user to change how the computer chooses its value-equivalently moves. The current setting is random that means the computer is free to choose *any* move that would produce a value-equivalent result – if the position is a winning position, any winning move is chosen. Other strategies are to force the quickest win, or force the user into a position with the most options (the Enough Rope Principle mentioned in [Berlekamp82]), or least options, etc.

9.6 Implement optimizations

Many of the optimizations mentioned in the last chapter are themselves future modifications, and thus should be considered for inclusion here. These are parallelization, symmetry, best-first value-equivalent directed search, and move-equivalent positions.

9.7 Write more modules

Programmers benefit from a large database of modules because there are more examples to reference. Users benefit from the wide variety of new, perhaps exotic games that would be available to learn and analyze.

9.8 Port to different platforms & distribute

GAMESMAN can only benefit from wide circulation. It is free and completely public domain and thus should be available on as many different platforms as

possible. Currently it is available on any machine that runs X-windows, but it could be ported to the Macintosh, NeXT, IBM Windows, and Silicon Graphics environments as well. Its existence could be announced on the net, and a moderated publicly-writeable area could be set up for modules to be added.

10. The GAMESMAN User Interface

GAMESMAN has two different interfaces: a text-based interface used when interacting with the system without X-window graphical capabilities and the X-based user interface that is what most users will see. To demonstrate the textual interface, we give a simple interaction with the system using the Tic-Tac-Toe module. To demonstrate the graphical interface, we include screen shots of all of the user interface components as well as the boards for each of the four modules.

10.1 The GAMESMAN textual user interface

Everything in **bold** is typed by the user. Comments are sprinkled throughout the output below. Unfortunately, it is hard to determine where the dividing line between the module and GAMESMAN lies; this would give an idea how much of the interaction changes for different modules. For the most part, the only component of the following interaction that is module-specific is the output of the help request, and the input and output of the move and position for Tic-Tac-Toe.

```
ddgarcia% tictactoe


Welcome to GAMESMAN, version 1.0 (05-15-95), written by Dan Garcia.

        (G)ame-independent
        (A)utomatic
        (M)ove-tree
        (E)xhaustive
        (S)earch,
        (M)aniuplation
        (A)nd
        (N)avigation

This program will determine the value of Tic-Tac-Toe, and allow you to play
the computer or another human. Have fun!

        ----- Hit <return> to continue -----


        ----- GAMESMAN, version 1.0 (05-15-95) with Tic-Tac-Toe module -----

        s)      (S)TART THE GAME

        Evaluation Options:

        o)      (O)bjective toggle from STANDARD to REVERSE

        h)      (H)elp.


        q)      (Q)uit.


Select an option: h
```

Had the user chosen "O" above, the game would have internally set itself up to prepare to solve the misère game of Tic-Tac-Toe.

```
        ----- HELP for Tic-Tac-Toe module -----

        Tic-Tac-Toe Help:

        1)      What do I do on MY TURN?
        2)      How do tell the computer WHICH MOVE I want?
        3)      What is the STANDARD OBJECTIVE of Tic-Tac-Toe?
        4)      Is a TIE possible?
        5)      What does the VALUE of this game mean?
        6)      Show SAMPLE Tic-Tac-Toe game.

        GAMESMAN Help:

        7)      What is a game VALUE?
        8)      What is EVALUATION?

        Generic Options Help:

        9)      What are PREDICTIONS?
        0)      What are HINTS?


        b)      (B)ack = Return to previous activity.

Select an option: 1

        ----- What do I do on MY TURN? -----

You place one of your pieces on one of the empty board positions.

        ----- Hit <return> to continue -----


        ----- HELP for Tic-Tac-Toe module -----

        Tic-Tac-Toe Help:

        1)      What do I do on MY TURN?
        2)      How do tell the computer WHICH MOVE I want?
        3)      What is the STANDARD OBJECTIVE of Tic-Tac-Toe?
        4)      Is a TIE possible?
        5)      What does the VALUE of this game mean?
        6)      Show SAMPLE Tic-Tac-Toe game.

        GAMESMAN Help:

        7)      What is a game VALUE?
        8)      What is EVALUATION?

        Generic Options Help:

        9)      What are PREDICTIONS?
        0)      What are HINTS?


        b)      (B)ack = Return to previous activity.

Select an option: 2
```

```
        ----- How do I tell the computer WHICH MOVE I want? -----

On your turn, use the LEGEND to determine which number to choose (between
1 and 9, with 1 at the upper left and 9 at the lower right) to correspond
to the empty board position you desire and hit return. If at any point
you have made a mistake, you can type u and hit return and the system will
revert back to your most recent position.

        ----- Hit <return> to continue -----


        ----- HELP for Tic-Tac-Toe module -----

        Tic-Tac-Toe Help:

        1)      What do I do on MY TURN?
        2)      How do tell the computer WHICH MOVE I want?
        3)      What is the STANDARD OBJECTIVE of Tic-Tac-Toe?
        4)      Is a TIE possible?
        5)      What does the VALUE of this game mean?
        6)      Show SAMPLE Tic-Tac-Toe game.

        GAMESMAN Help:

        7)      What is a game VALUE?
        8)      What is EVALUATION?

        Generic Options Help:

        9)      What are PREDICTIONS?
        0)      What are HINTS?


        b)      (B)ack = Return to previous activity.


Select an option: b


        ----- GAMESMAN, version 1.0 (05-15-95) with Tic-Tac-Toe module -----

        s)      (S)TART THE GAME

        Evaluation Options:

        o)      (O)bjective toggle from STANDARD to REVERSE

        h)      (H)elp.


        q)      (Q)uit.


Select an option: s

Initializing insides of Tic-Tac-Toe...done in 0.023436 seconds!
Evaluating the value of Tic-Tac-Toe...done in 1.097714 seconds!

The Game Tic-Tac-Toe has value: Tie

        ----- Hit <return> to continue -----
```

The system reports on the value of the game as well as the time it took (in real seconds, not CPU seconds) to initialize and solve it. This is useful when benchmarking systems and roughly determining how long it would take to solve a larger game.

```
        ----- GAMESMAN version 1.0 (05-15-95) with Tic-Tac-Toe module -----

        Player Name Options:

        1)      Change the name of player 1 (currently Dan)
        2)      Change the name of player 2 (currently Computer)
        3)      Swap player 1 (plays FIRST) with player 2 (plays SECOND)

        Generic Options:

        4)      Toggle from    PREDICTIONS to NO PREDICTIONS
        5)      Toggle from NO HINTS       to    HINTS

        Playing Options:

        6)      Toggle opponent from a COMPUTER to a HUMAN
        7)      Toggle from going FIRST (can tie/lose) to SECOND (can tie/lose)
        e)      (E)valuator toggle from EXHAUSTIVE SEARCH to HEURISTIC
        l)      Change (L)ookahead used for Heuristic (currently 2)

        a)      (A)nalyze the game

        p)      (P)LAY GAME.

        h)      (H)elp.


        q)      (Q)uit.

Select an option: p


Ok, Dan and Computer, let us begin.

Type '?' if you need assistance...


         ( 1 2 3 )           : - - -
LEGEND:  ( 4 5 6 )  TOTAL:   : - - -
         ( 7 8 9 )           : - - - (Dan should Tie)

     Dan's move [(u)ndo/1-9] :  ?

Text Input Commands:
--------------------
?           : Brings up this list of Text Input Commands available
s (or S)    : The computer will list all (S)afe (Value-Equivalent) moves
u (or U)    : (U)ndo last move (not possible at beginning position)
r (or R)    : (R)eprint the position
h (or H)    : (H)elp
a (or A)    : (A)bort the game
q (or Q)    : (Q)uit
Valid Moves : [ 1 2 3 4 5 6 7 8 9 ]
```

Choosing help would have brought up the same help window from earlier in the session; help is available at any stage in the game for all modules.

```
     Dan's move [(u)ndo/1-9] :  1

         ( 1 2 3 )           : X - -
LEGEND:  ( 4 5 6 )  TOTAL:   : - - -
         ( 7 8 9 )           : - - - (Computer should Tie)

Computer's move              :  5

         ( 1 2 3 )           : X - -
LEGEND:  ( 4 5 6 )  TOTAL:   : - O -
         ( 7 8 9 )           : - - - (Dan should Tie)

     Dan's move [(u)ndo/1-9] :  s

Here are some 'safe' moves   : [ 9 8 7 6 4 3 2 ]
```

A “safe” move is one that is value-equivalent. Since this is a tie position, the system prints out all tying moves at this point. It turns out they all are!

```
     Dan's move [(u)ndo/1-9] :  8

         ( 1 2 3 )           : X - -
LEGEND:  ( 4 5 6 )  TOTAL:   : - O -
         ( 7 8 9 )           : - X - (Computer should Tie)

Computer's move              :  6

         ( 1 2 3 )           : X - -
LEGEND:  ( 4 5 6 )  TOTAL:   : - O O
         ( 7 8 9 )           : - X - (Dan should Tie)

     Dan's move [(u)ndo/1-9] :  s

Here are some 'safe' moves   : [ 4 ]

     Dan's move [(u)ndo/1-9] :  2
```

We are clearly living life on the edge. The only value-equivalent move was 4 and we chose 2. We are certain to lose now. Note the prediction changes from “Computer should Tie” to “Computer *will* win”. It knows the end is near.

```
         ( 1 2 3 )           : X X -
LEGEND:  ( 4 5 6 )  TOTAL:   : - O O
         ( 7 8 9 )           : - X - (Computer will Win)

Computer's move              :  3
```

Note the curious strategy the computer uses when faced with a winning position. It does not go for the quick win at position 4, but instead is perfectly happy waiting a move to win. It knew it had a sure win either way, so it didn’t matter to the computer when it won. Careful study of the position below indicates that it can afford to choose 3 to block, because in doing so forces a win in an additional two directions. If its move did not force any additional win directions Dan (“O”) could have chosen 4 to block and would have returned the game to a tie position.

```
         ( 1 2 3 )           : X X O
LEGEND:  ( 4 5 6 )  TOTAL:   : - O O
         ( 7 8 9 )           : - X - (Dan will Lose)
```

```
     Dan's move [(u)ndo/1-9] :  s

Here are some 'safe' moves   : [ 4 7 9 ]

     Dan's move [(u)ndo/1-9] :  4

         ( 1 2 3 )           : X X O
LEGEND:  ( 4 5 6 )  TOTAL:   : X O O
         ( 7 8 9 )           : - X - (Computer will Win)

Computer's move              :  7

         ( 1 2 3 )           : X X O
LEGEND:  ( 4 5 6 )  TOTAL:   : X O O
         ( 7 8 9 )           : O X - (Dan will Lose)


Computer wins. Nice try, Dan.


        ----- Hit <return> to continue -----


        ----- GAMESMAN, version 1.0 (05-15-95) with Tic-Tac-Toe module -----

        Player Name Options:

        1)      Change the name of player 1 (currently Dan)
        2)      Change the name of player 2 (currently Computer)
        3)      Swap player 1 (plays FIRST) with player 2 (plays SECOND)

        Generic Options:

        4)      Toggle from    PREDICTIONS to NO PREDICTIONS
        5)      Toggle from NO HINTS       to    HINTS

        Playing Options:

        6)      Toggle opponent from a COMPUTER to a HUMAN
        7)      Toggle from going FIRST (can tie/lose) to SECOND (can tie/lose)
        e)      (E)valuator toggle from EXHAUSTIVE SEARCH to HEURISTIC
        l)      Change (L)ookahead used for Heuristic (currently 2)

        a)      (A)nalyze the game

        p)      (P)LAY GAME.

        h)      (H)elp.


        q)      (Q)uit.

Select an option: q

Thanks for playing Tic-Tac-Toe!
```

10.2 The GAMESMAN graphical user interface

This section contains pictures of the interface with all of the different modules (“1,2,...,10”, “Tic-Tac-Toe”, “Tac Tix” and “Dodgem”) and explains how to interact with the system. Great care was taken to insure that the graphic design was visually pleasing and the interactions intuitive.

Figure 10.1: The main GAMESMAN interface control window with the Tic-Tac-Toe module loaded in. The play buttons are disabled because the user has not solved the game by clicking on the “Start” button. The “Modify the starting position” button is also disabled because this module does not yet allow for the starting position (in this case, the familiar blank board) to be modified.

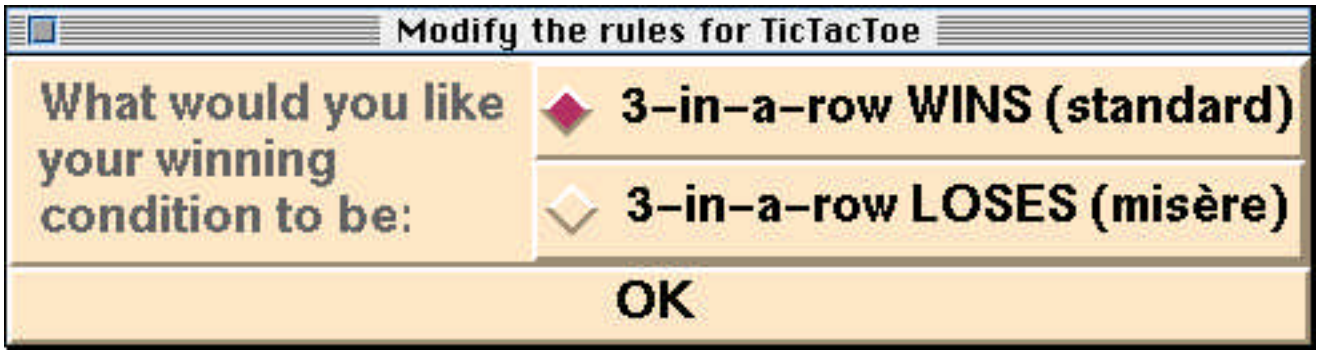


Figure 10.2: The “Modify the rules for <module-name>“ window. This allows the user to change the rules of the game. Here we allow the user to choose between the standard and the misère game.

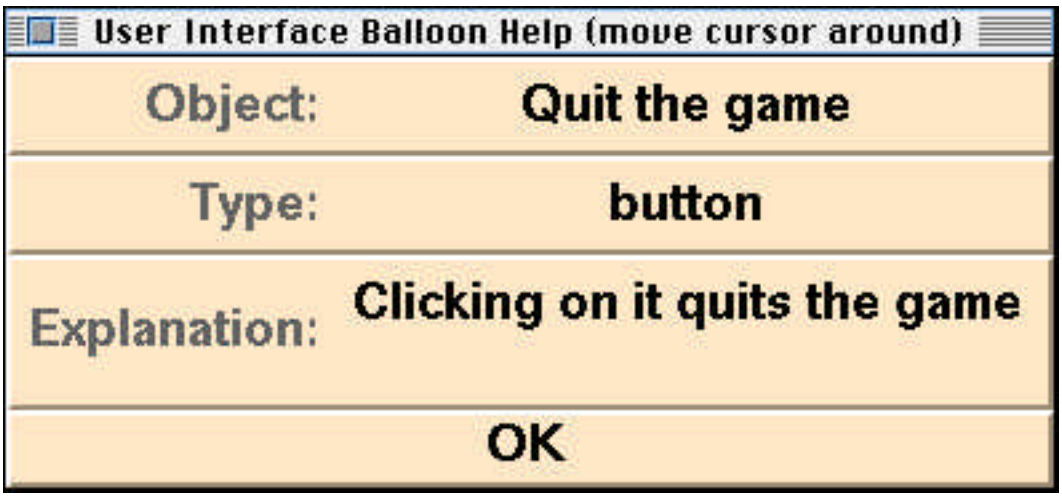


Figure 10.3: The User interface balloon help window. When this window is open, whatever object the user's cursor is over gets explained in this window. Here the cursor was over the "Quit" button. This serves the purpose of providing an on-line interface manual.

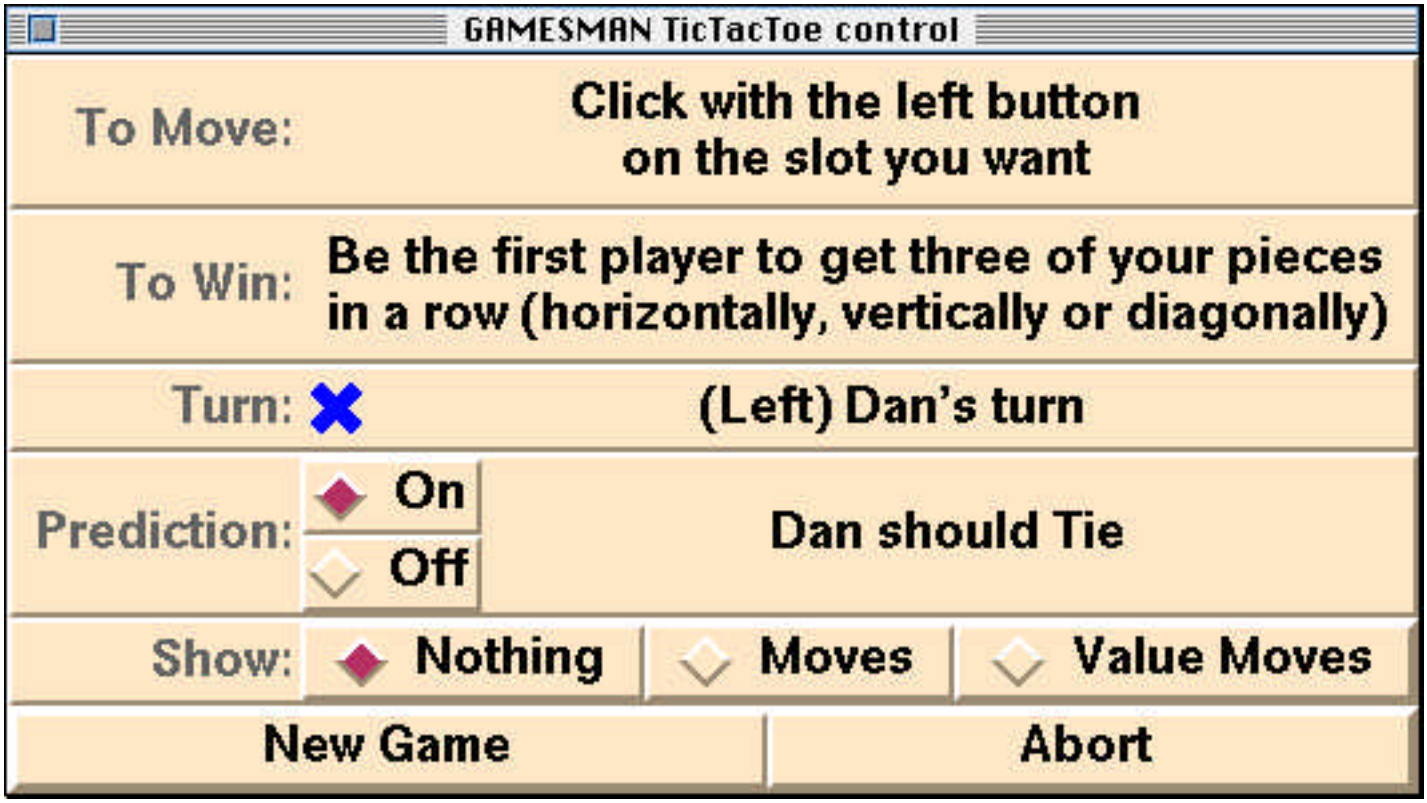


Figure 10.4: The GAMESMAN Tic-Tac-Toe control window. This window tells the user how to move and win in this game, whose turn it is, its prediction of the outcome of the game, and allows the user to augment the game board with a visual display of the available moves, possibly color-coded by value. It also allows the user to restart this game (the "New Game") button and abort the game altogether.

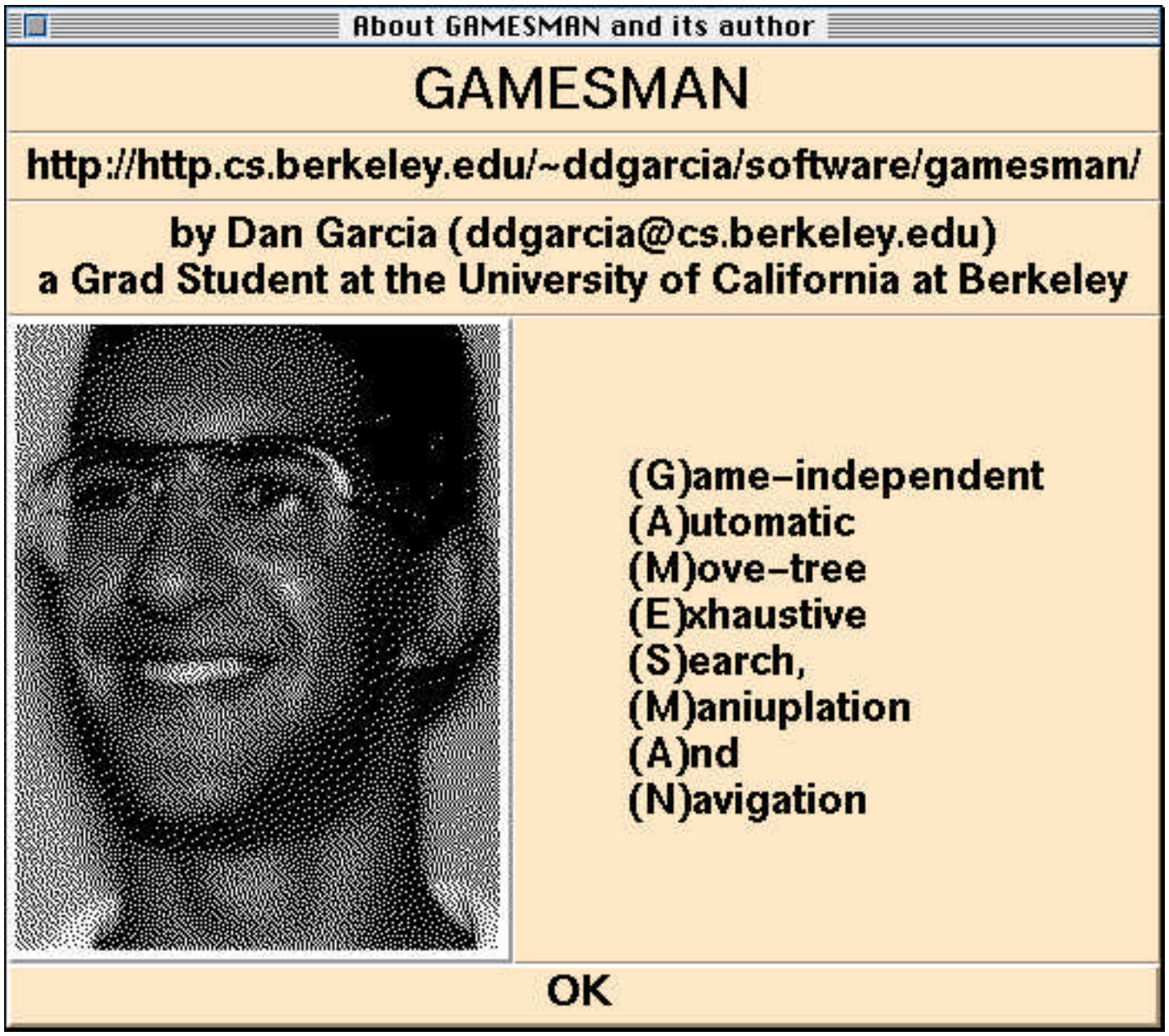


Figure 10.5: The about-the-author window. This window is brought up when the user clicks on the Krusty-the-clown icon in the lower left of the main GAMESMAN window shown in figure 10.1. Here we advertise the GAMESMAN World Wide Web home page where update information (and this document) can be found, list the author's name, email and status, show what he looks like, and explain the GAMESMAN acronym.

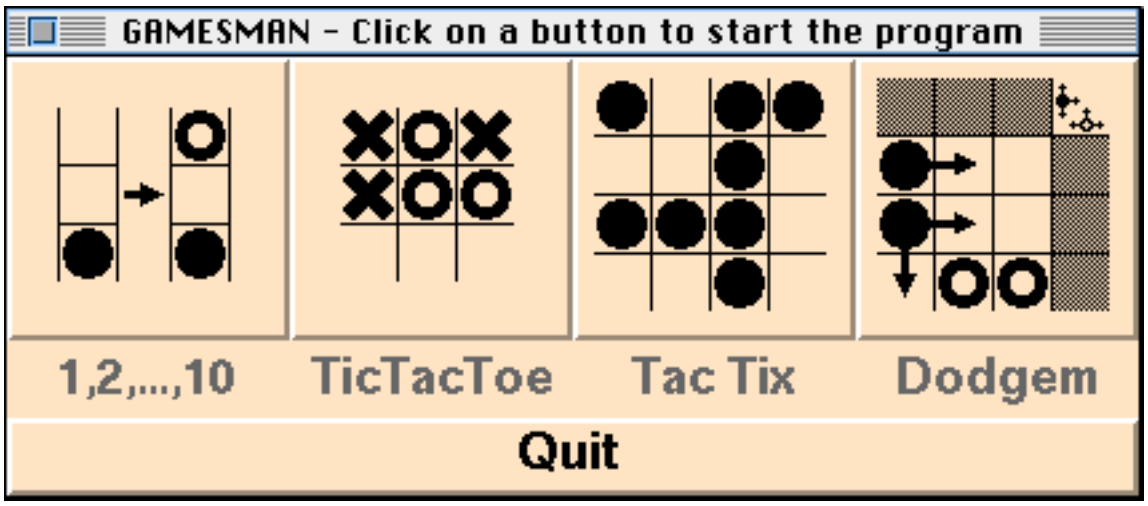


Figure 10.6: The GAMESMAN front-end which can be used to bring up the various X-window GAMESMAN programs. If more modules are written, this user-interface will provide a simple and graphical way to run them.

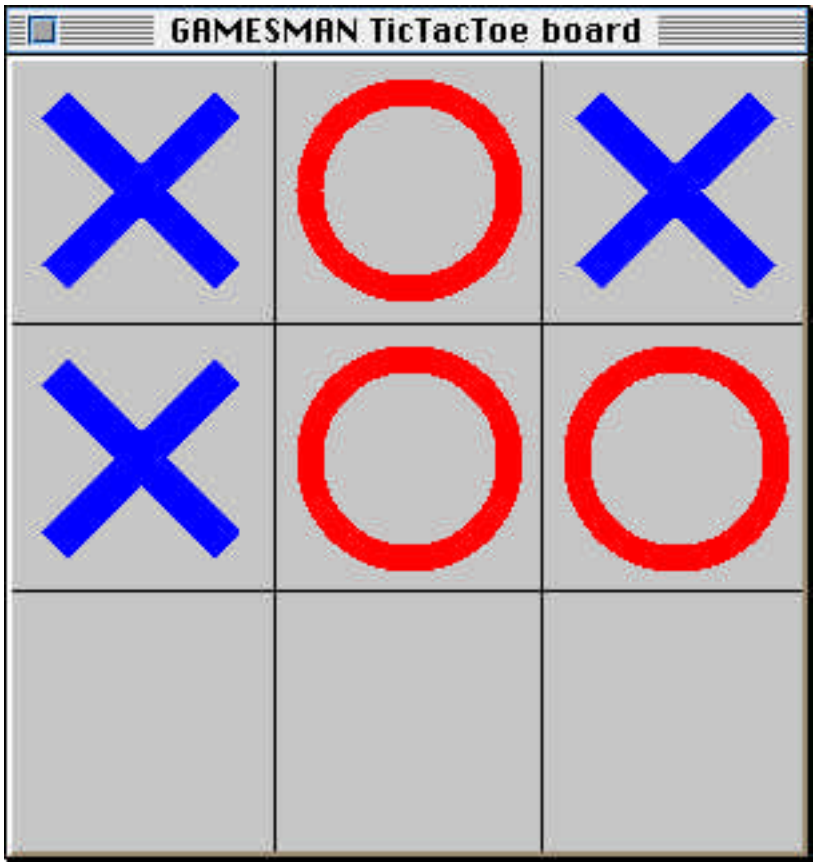


Figure 10.7: The GAMESMAN Tic-Tac-Toe board window with no moves shown. It is X's turn to move. The blank slots are "alive" in the sense that they respond (by inverting to black) when the cursor is over them. The other, filled slots do not react when the cursor is over them. This provides valuable feedback to the user, who may be unfamiliar with the game and how to make a move.

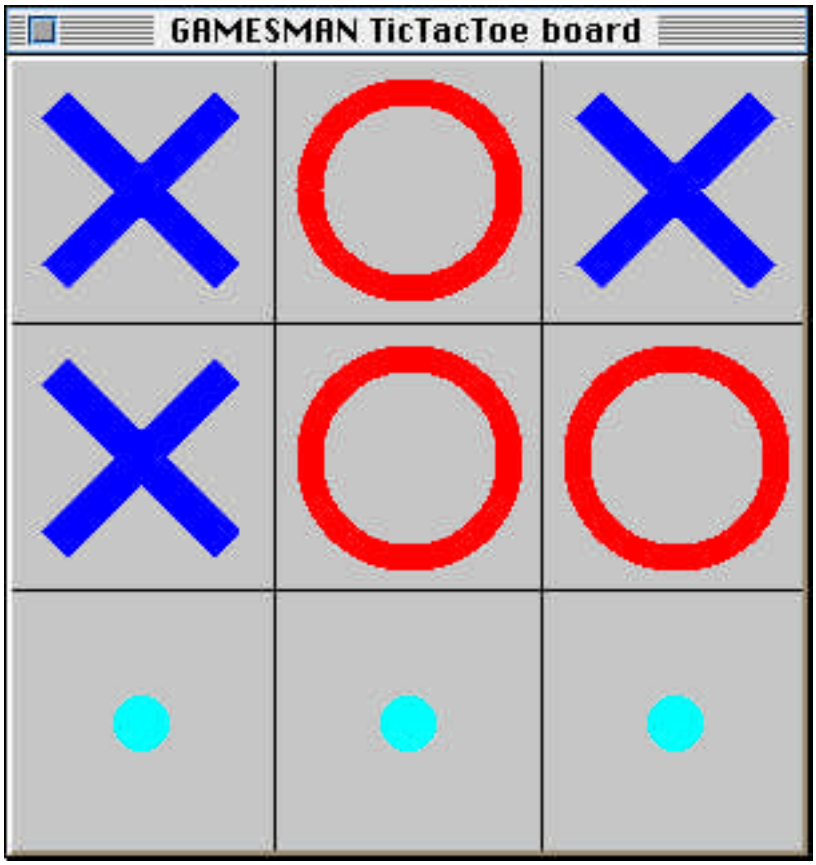


Figure 10.8: The GAMESMAN Tic-Tac-Toe board window with the available moves shown as cyan-colored circles. This also helps the first-time user of the system.

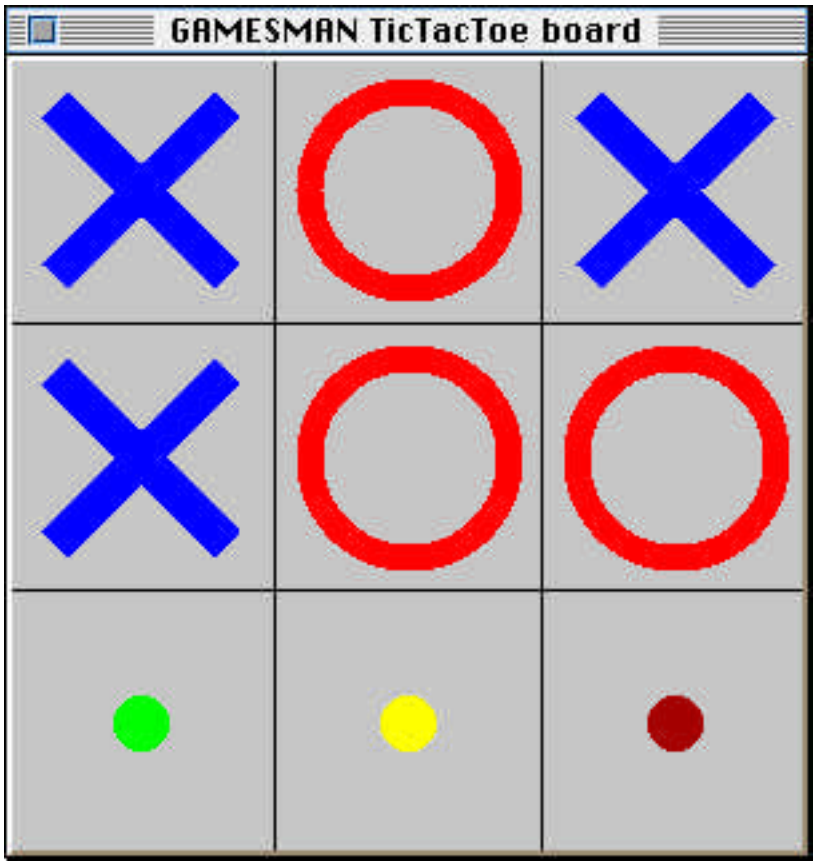


Figure 10.9: The same GAMESMAN Tic-Tac-Toe board window with value moves shown. They are color-coded as shown in figure 10.10 below. By reading the display we see that X can win by moving on the left, tie by blocking O in the middle and lose by moving on the right, allowing O the chance to win. This is the same position from figure 2.4 we analyzed in section 2.5

Figure 10.10: The color-coded explanation of value moves. The color choice coincides with that of a stoplight: green = go = win, yellow = caution = tie, dark red (red could not be used since it is reserved for right) = stop = lose.

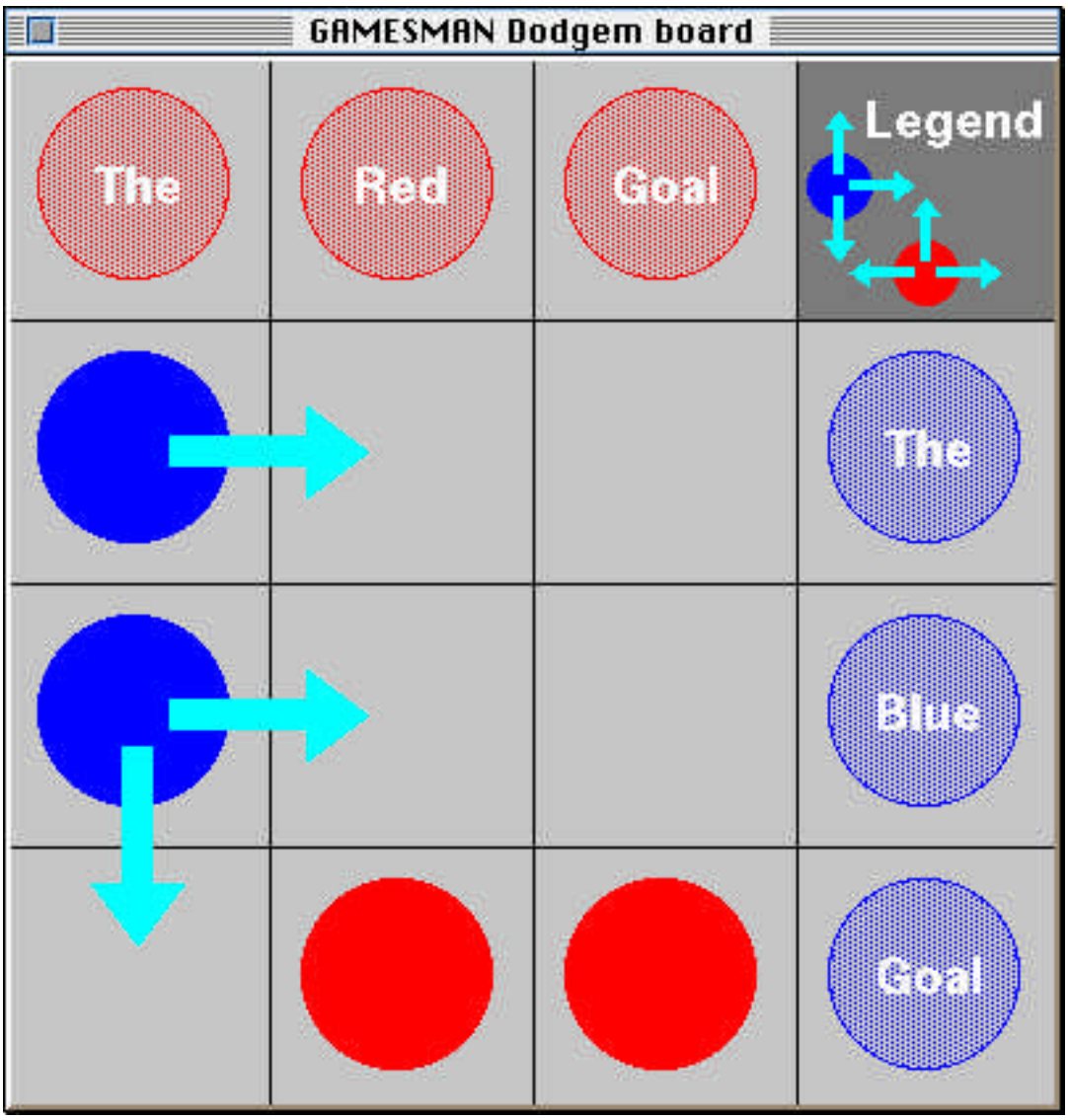


Figure 10.11: The GAMESMAN Dodgem board. The user clicks on the cyan arrows to make a move; the pieces then animate and move to the next slot. When a piece moves into its own goal, an animation plays which makes the piece shrink and disappear. The legend in the upper-right shows the moves available to both players.

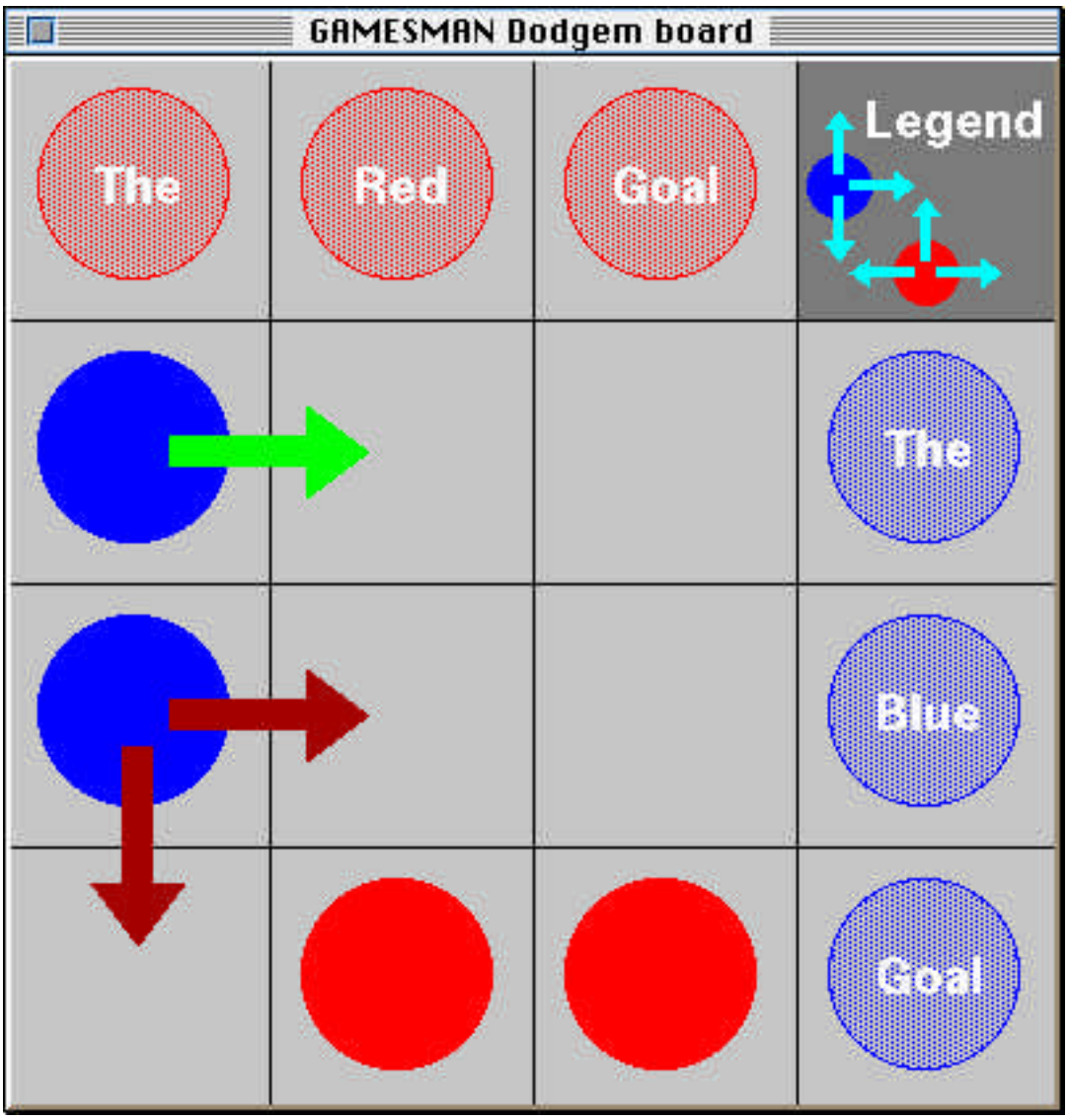


Figure 10.12: The same GAMESMAN Dodgem board as in figure 10.11 with value moves turned on. Here it is clear that the only winning move for Left (the blue pieces) is to move the upper piece to the right.

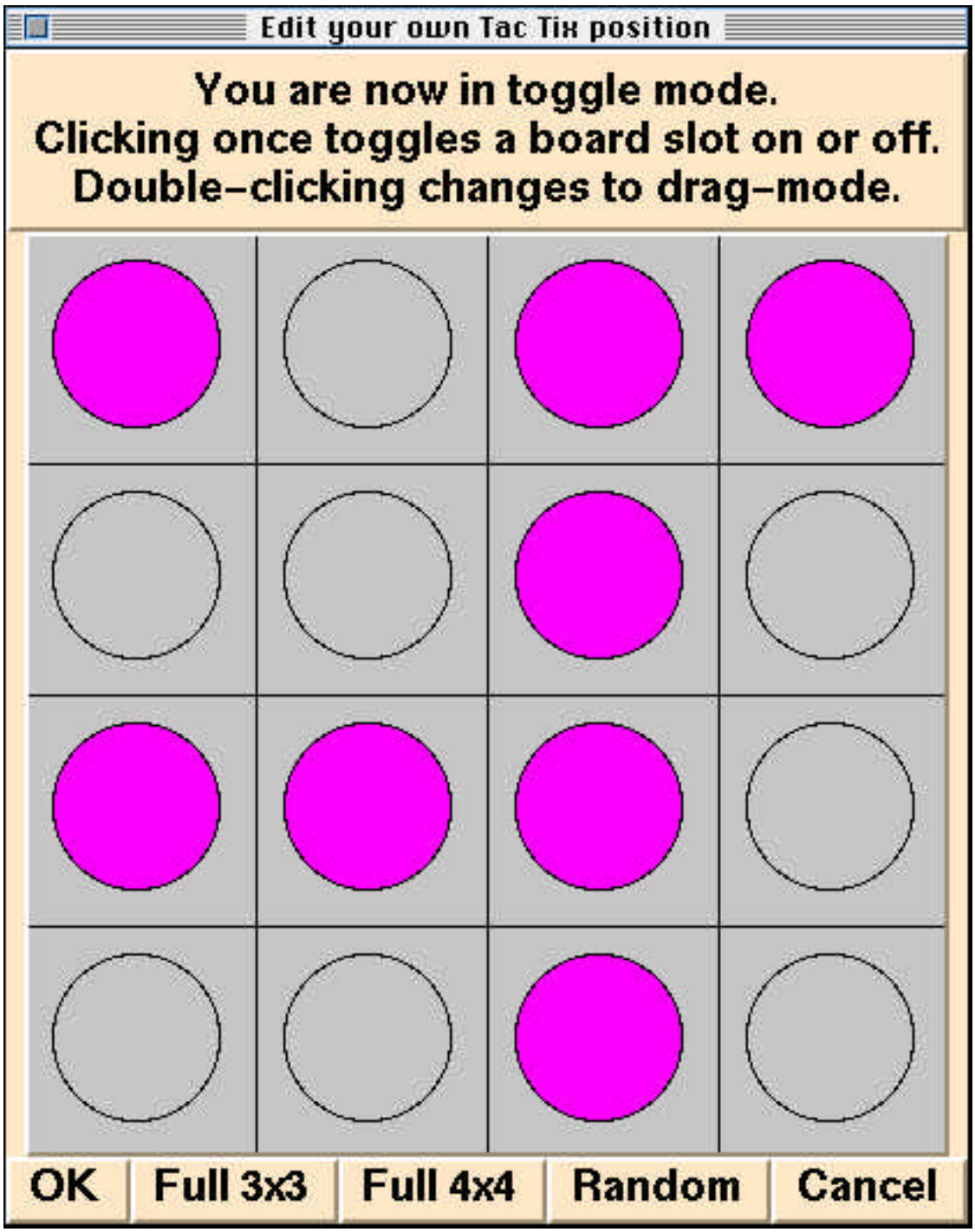


Figure 10.13: The GAMESMAN Tac Tix "Edit the initial position" window. Here the user clicks on the slots to toggle them on and off. We have chosen this position because it contains a single piece, and a horizontal or vertical line of pieces of lengths 2, 3 and 4. This is so we can illustrate the available moves as shown in figures 10.14 and 10.15.

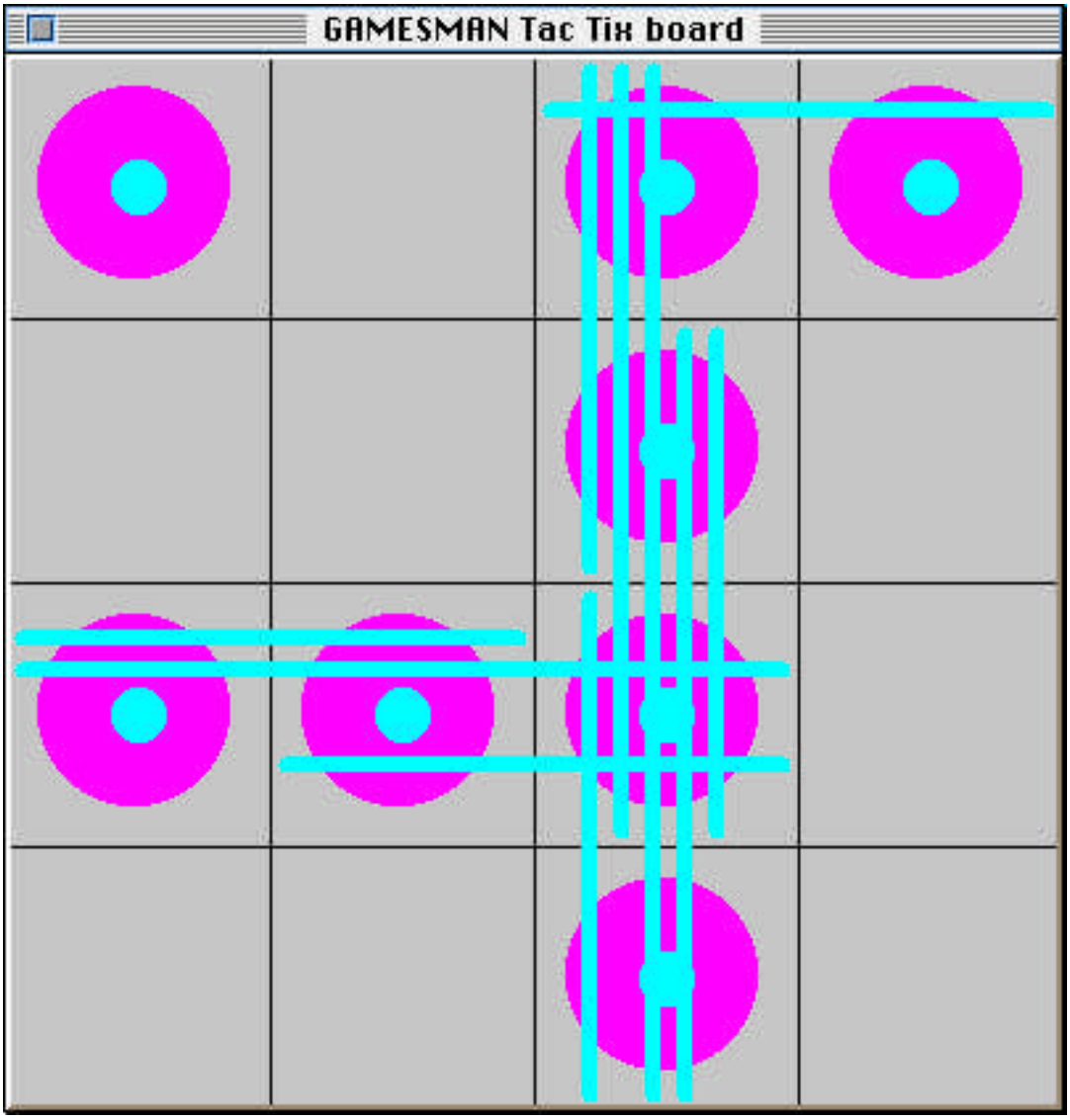


Figure 10.14: The GAMESMAN Tac Tix board with the available moves shown. Any cyan line removes the pieces it is overlapping. The cyan circles remove only the single pieces under them. The color of the pieces (magenta = red + blue) was chosen because this is an impartial game so the moves available to left (blue) are the same as those available to right (red). Moving in this game consists of clicking on the cyan move. Placing the cursor over a move highlights the pieces about to be removed so the user has visual feedback what the pieces the move will affect.

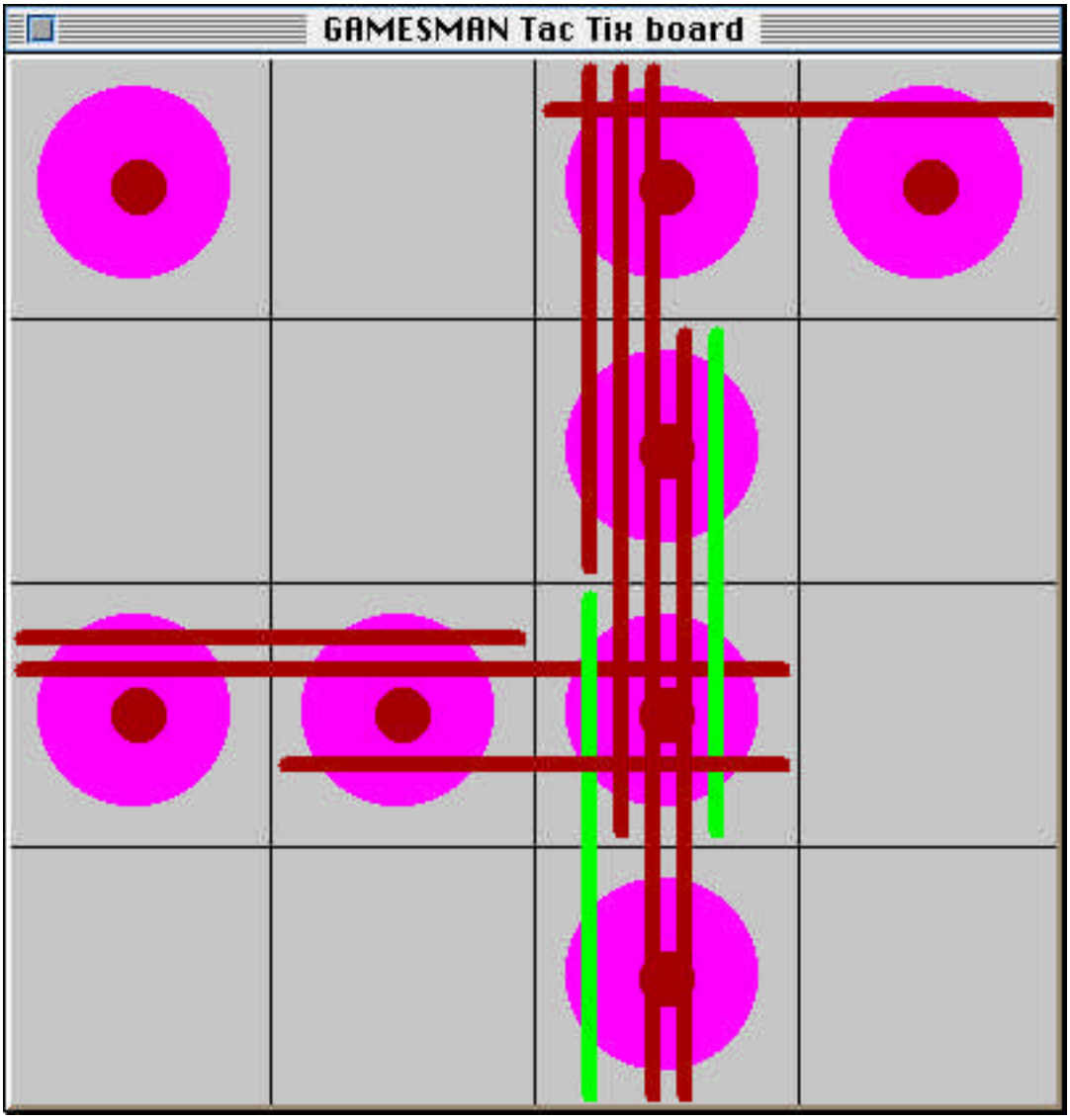


Figure 10.15: The same GAMESMAN Tac Tix board as figure 10.14 with value moves turned on. It is clear from this position that the person whose turn it is has only two winning moves: removing two vertical pieces in the center or bottom. For readers familiar with Nim sums, these are winning moves because they reduce the board to (* + * + *2 + *2 = 0) and (* + *2 + *3 = 0) respectively.

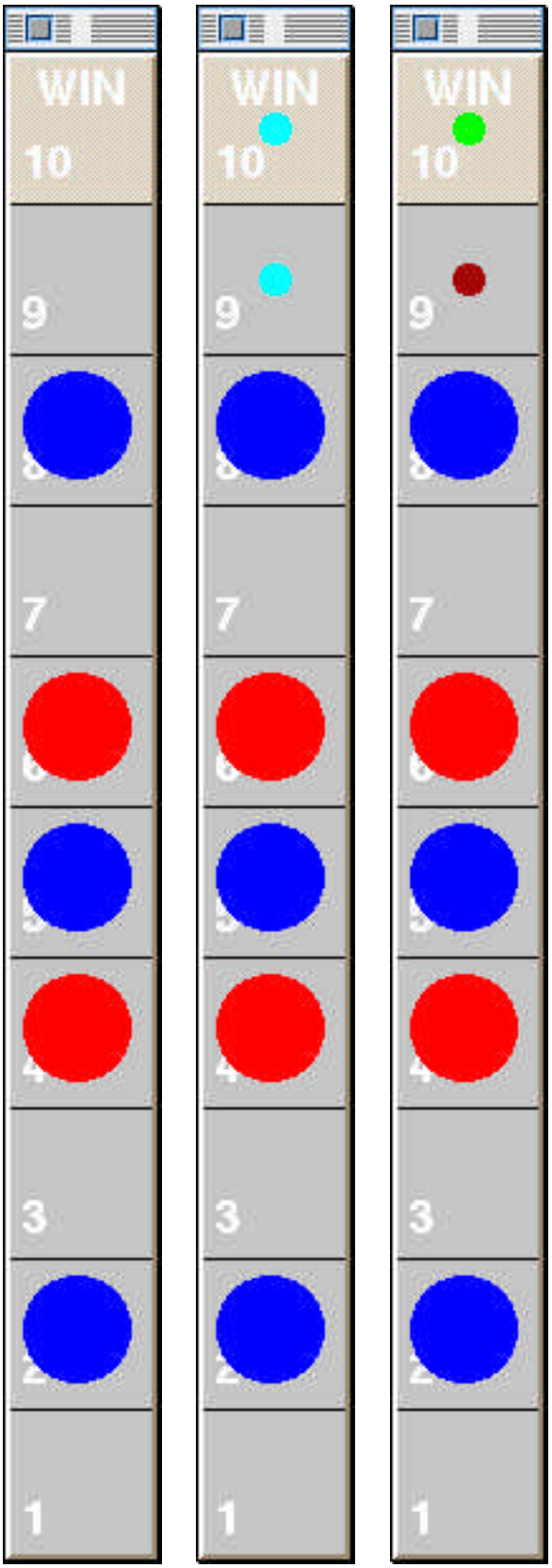


Figure 10.16: The GAMESMAN “1,2,...10” boards with (no, valid, value) moves.

11. Summary

GAMESMAN is a package that allows programmers to easily develop, test, analyze and eventually play and share finite, two-person perfect-information games. It saves the developer the onerous task of creating a game from scratch by only requiring that an internal representation, subroutines and optional mapping function be written in the form of C code. A library of interactions is provided which eliminates the need to write user interface code from scratch. GAMESMAN also provides hooks for user parametrization, which allows users to change the rules of a particular game at runtime. Once a game has been written and compiled, it can be added to a growing database of games to be learned from, shared, and enjoyed.

This report outlines the design of a complete game-generation system. There exists a working implementation that does not contain everything from the full design, but does have almost all of the essential functionality. This provides a stable framework on which to begin adding modules. Programmers have several options when deciding how the computer will choose its moves. If the game is small enough to be solved, then a flag can be set which tells the system to exhaustively search the move-tree and generate computer moves from the table of positions it creates. If the game is too large to be solved, or if the programmer wishes to attempt to encode the table, then a static evaluator can be written to drive a look-ahead MINIMAX computer strategy. If the game is too big to solve and the developer does *not* wish to write a static evaluator, then the game can be restricted to 2-player mode only, and no computer opponent will be available. The computer in this case will act as a referee, only allowing valid moves and signaling when the game is over.

GAMESMAN cannot even begin to solve large games, such as Chess, Checkers, or Go. This is due to the massive number of possible board positions that either cannot be stored, would take centuries to compute, or both. However, smaller games, whose board positions *can* be stored and computed in reasonable time *can* be solved. The advantage of this is the computer provides a perfect opponent without the developer having to program any intelligence or strategies into the system. Also, since the computer is truly a *perfect* opponent, strategies can be deduced from its play. This provides a tool that allows game analysts to conclude and prove theories about particular games. Even if a game cannot be solved, GAMESMAN provides a relatively easy way to implement a game for static evaluation or simply 2-player mode. In summary, GAMESMAN is a system to facilitate game design, implementation, testing, parametrization, distribution, analysis and enjoyment.

Acknowledgments

Sincere thanks go to my reviewers, Brian and Elwyn, who were open-minded enough to support this project.

Special thanks go to Mark Bomi Moss for his insight and constructive comments at the early stages of this project. Professor Carlo Sequin proposed the crucial L-game move-selection strategy as well as discussed the symmetries of a cube. Noam Tene volunteered optimization ideas and suggested blinking slots and cursor-initiated highlighting, as did Armando Fox. Noam also suggested delayed evaluation. Chris Borton mentioned the value of graying-out invalid moves. Barry Gysbers listened with a keen ear and proposed several future enhancements. Dan Wexler provided the needed impetus to investigate displaying all possible moves graphically. Zijiang Yang provided a few meals down the stretch which were sorely needed. David Wolfe provided great last-minute help. Extra thanks to Narguesse Bakhsh who endured testing early versions of the implementation and provided emotional support throughout.

My thanks to all these individuals, and thanks and apologies to anyone who should have been credited and was not.

Appendices

Appendix A Module Specifications in C and Tcl/Tk

If a programmer decided to implement a new game, there are two components to write: C code describing the internals of the game (what the internal representation of a position and move is, what a primitive position is, etc.) and Tcl/Tk code describing how the user should graphically interact with the board. In this appendix we provide the prototypes for all of the routines that need to be filled in to implement a module.

A.1 Module specifications in C

```
/*****************************************************************
**
** NAME:        InitializeDatabases
**
** DESCRIPTION: Initialize the Global database table to store the values
**              and the next move the computer should choose.
**
*****************************************************************/

InitializeDatabases()

/*****************************************************************
**
** NAME:        InitializeGraphics
**
** DESCRIPTION: Initialize the graphics vars.
**              Empty if kSupportsGraphics = FALSE
**
*****************************************************************/

InitializeGraphics()

/*****************************************************************
**
** NAME:        DebugMenu
**
** DESCRIPTION: Menu used to debug internal problems. Does nothing if
**              kDebugMenu == FALSE
**
*****************************************************************/

DebugMenu()

/*****************************************************************
**
** NAME:        GameSpecificMenu
**
** DESCRIPTION: Menu used to change game-specific parameters, such as
**              the side of the board in an nxn Nim board, etc. Does
**              nothing if kGameSpecificMenu == FALSE. Not even called.
**
*****************************************************************/

GameSpecificMenu()

/*****************************************************************
**
** NAME:        DoMove
**
```

```
** DESCRIPTION: Apply the move to the position.
**
** INPUTS:      POSITION thePosition : The old position
**              MOVE     theMove     : The move to apply.
**
** OUTPUTS:     (POSITION) : The new position that results after the move.
**
**********************************************************************/

POSITION DoMove(thePosition, theMove)

/**********************************************************************
**
** NAME:        GetInitialPosition
**
** DESCRIPTION: Return the position that the user manually entered
**
** OUTPUTS:     (POSITION) : The position that the user entered.
**
**********************************************************************/

POSITION GetInitialPosition()

/**********************************************************************
**
** NAME:        StoreComputersMove
**
** DESCRIPTION: Store the next move of the position
**              into the Global Database.
**
** INPUTS:      POSITION thePosition : The position in question.
**              MOVE     nextMove    : Where to go to from thePosition
**
**********************************************************************/

StoreComputersMove(thePosition, nextMove)

/**********************************************************************
**
** NAME:        GetComputersMove
**
** DESCRIPTION: Get the next move for the computer from the Global Database
**              GlassBox: It currently randomly picks a value-equivalent
**              move - this is something that could be changed.
**
** INPUTS:      POSITION thePosition : The position in question.
**
** OUTPUTS:     (MOVE) : the next move that the computer will take
**
**********************************************************************/

MOVE GetComputersMove(thePosition)

/**********************************************************************
**
** NAME:        GetRawValueFromDatabase
**
** DESCRIPTION: Get a pointer to the value of the position from Database.
**
** INPUTS:      POSITION position : The position to return the value of.
**
** OUTPUTS:     (VALUE *) a pointer to the actual value.
**
**********************************************************************/
```

```
VALUE *GetRawValueFromDatabase(position)

/*********************************************************************
**
** NAME:        PrintComputersMove
**
** DESCRIPTION: Nicely format the computers move.
**
** INPUTS:      MOVE   *computersMove : The computer's move.
**              STRING  computersName : The computer's name.
**
*********************************************************************/

PrintComputersMove(computersMove,computersName)

/*********************************************************************
**
** NAME:        Primitive
**
** DESCRIPTION: Return the value of a position if it fulfills certain
**              'primitive' constraints. Some examples of this is having
**              three-in-a-row with Tic-Tac-Toe. Tic-Tac-Toe has two
**              primitives it can immediately check for, when the board
**              is filled but nobody has one = primitive tie. Three in
**              a row is a primitive lose, because the player who faces
**              this board has just lost. I.e. the player before him
**              created the board and won. Otherwise undecided.
**
** INPUTS:      POSITION position : The position to inspect.
**
** OUTPUTS:     (VALUE) an enum which is oneof: (win,lose,tie,undecided)
**
*********************************************************************/

VALUE Primitive(position)

/*********************************************************************
**
** NAME:        PrintPosition
**
** DESCRIPTION: Print the position in a pretty format, including the
**              prediction of the game's outcome.
**
** INPUTS:      POSITION position   : The position to pretty print.
**              STRING   playerName : The name of the player.
**              BOOLEAN  usersTurn  : TRUE <==> it's a user's turn.
**
*********************************************************************/

PrintPosition(position,playerName,usersTurn)

/*********************************************************************
**
** NAME:        GenerateMoves
**
** DESCRIPTION: Create a linked list of every move that can be reached
**              from this position. Return a pointer to the head of the
**              linked list.
**
** INPUTS:      POSITION position : The parent position to branch off of.
**
** OUTPUTS:     (MOVELIST *), a pointer that points to the first item
**              in the linked list of moves that can be generated.
```

```
**
*********************************************************************/

MOVELIST *GenerateMoves(position)

/*********************************************************************
**
** NAME:        GetAndPrintPlayersMove
**
** DESCRIPTION: This finds out if the player wanted an undo or not.
**              If so, fill *theMove with NIL.
**              Otherwise get the new theMove and fill the pointer up.
**
** INPUTS:      POSITION  thePosition : The position the user is at.
**              MOVE *theMove         : The move to fill with user's move.
**              STRING playerName     : The name of the player whose
**                                      turn it is
**
*********************************************************************/

GetAndPrintPlayersMove(thePosition, theMove, playerName)

/*********************************************************************
**
** NAME:        PrintMove
**
** DESCRIPTION: Print the move in a nice format.
**
** INPUTS:      MOVE *theMove         : The move to print.
**
*********************************************************************/

PrintMove(theMove)
```

A.2 Module specifications in Tcl/Tk

```
##############################################################################
##
## GS_InitGameSpecific
##
## This initializes the game-specific variables.
##
##############################################################################

proc GS_InitGameSpecific {}

##############################################################################
##
## GS_EmbellishSlot
##
## This is where we embellish a slot if its necessary
##
##############################################################################

proc GS_EmbellishSlot { w slotX slotY slot }

##############################################################################
##
## GS_ConvertInteractionToMove
##
## This converts the user's interaction to a move to be passed to the C code.
##
##############################################################################
```

```
proc GS_ConvertInteractionToMove { theMove }

##############################################################################
##
## GS_ConvertMoveToInteraction
##
## This converts the C code encoding of a move to the Interaction's encoding
##
##############################################################################

proc GS_ConvertMoveToInteraction { theMove }

##############################################################################
##
## GS_PostProcessBoard
##
## This allows us to post-process the board in case we need to add something
##
##############################################################################

proc GS_PostProcessBoard { w }

##############################################################################
##
## GS_ConvertToAbsoluteMove
##
## Sometimes the move handed back by our C code is a relative move. We need
## to convert this to an absolute move to indicate on the board.
##
##############################################################################

proc GS_ConvertToAbsoluteMove { theMove }

##############################################################################
##
## GS_HandleEnablesDisables
##
## At this point a move has been registered and we have to handle the
## enabling and disabling of slots
##
##############################################################################

proc GS_HandleEnablesDisables { w theSlot theMove }

##############################################################################
##
## GS_EnableSlotEmbellishments
##
## If there are any slot embellishments that need to be enabled, now is the
## time to do it.
##
##############################################################################

proc GS_EnableSlotEmbellishments { w theSlot }

##############################################################################
##
## GS_DisbleSlotEmbellishments
##
## If there are any slot embellishments that need to be disabled, now is the
## time to do it.
##
##############################################################################
```

```
proc GS_DisableSlotEmbellishments { w theSlot }

##############################################################################
##
## GS_NewGame
##
## "New Game" has just been clicked. We need to reset the slots
##
##############################################################################

proc GS_NewGame { w }
```

Appendix B Description and Rules of Games

This section provides three important functions. First, it describes the rules, winning conditions and references for further inspection for all of the games described in the text. Second, it introduces many wonderful games, some of which have existed for centuries. Lastly, it provides references to some truly wonderful books that introduce and analyze literally thousands of games. We begin with the four games implemented in GAMESMAN: "1,2,...,10", Tac Tix, Tic-Tac-Toe and Dodgem. The remaining games are listed alphabetically to make locating a particular game easy.

There are a few reference books that are worth noting. The two-volume set of [Berlekamp82] should be on *every* game enthusiast's shelf for three reasons: it presents more games than any other, it analyzes them, and explains the combinatorial game theory upon which the analysis is based. Robbie Bell has done an excellent job in [Bell79] and [Bell83] and presenting obscure games from different countries as well as providing mathematical insight in [Bell88]. [Brandreth81], [Diagram92], [Pritchard82], [Grunfeld75] and [Pentagram90] also wonderful resources for exploring the exotic games of the world.

B.1 1,2,...,4 / 1,2,...,10 / One Line Nim

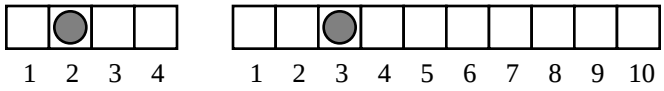


Figure B.1: The games 1,2,...,4 and 1,2,...,10 played with a counter.

The game 1,2,...,4 is very simple. Players alternate saying either 1 or 2. The total amount that has been said is kept in a counter, which starts at 0. The person who first raises the counter's total to 4 wins. It is a win game. The winning strategy is to say 1, then say the opposite of whatever the opponent says. The game "1,2,...,10" is the same game as "1,2,...,4" except the goal is to reach 10 instead of 4. It is also a win game. The winning strategy is to say 1 first, then say the opposite of whatever the opponent says. The game is called "1,2,..,4" by GAMESMAN because the options available to both players are only 1 and 2 and the goal is to reach 4. This can be generalized easily: the game "(x_i, x_{i+1},..., x_n), ... ,k" is such that each player can choose (x_i, x_{i+1},..., x_n) on their turn with the goal of reaching k first.

The game can also be thought of in another way: imagine there is a counter on a board and each player on his turn can move the counter 1 or 2 spaces to the right. The player who moves the counter to the slot labeled 4 (or 10) wins.

Variants of the game are explained (as "1,2,3,...,n" and mislabeled as "Nim") in [Anderson89, pp. 80-82], (as "1,2,3,...,15" and called "One Line Nim") in [Brandreth81, pp. 216-217] and [Pentagram90, pp. 116,136], (as "1,2,3,4,5,...,37" and called "The 37 Puzzle Game") in [Dudeny67, pp. 186-187], (as "1,2,3,...,<odd>" and called "The Pebble Game") in [Dudeny70, p. 117] and [Kaplan68, p. 22], (as "1-10,...,100" in both standard and misère form) in [Frochlichstein62, pp. 25-26], (as

"1,2,3,...,11", "1,2,3,...,30", and "1-p,...,n") in [Kordemsky72, p. 120] and analyzed in [Beasley89, pp. 108-109].

B.2 Tac Tix / Nimbi

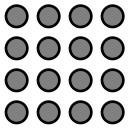


Figure B.2: The beginning 4×4 board for Tac Tix.

Invented by Piet Hein, this is a very simple game with an interesting nim-based analysis. Each player takes as many pieces from any row or column as long as the pieces are *contiguous* (i.e. there is no gap between them). The player who removes the last piece is the winner. It is described in [Gardner59, pp. 157-161], [Grunfeld75, p. 268], [Holt78, p. 67] and [Brandreth81, p. 218].

B.3 Tic-Tac-Toe / Noughts and Crosses

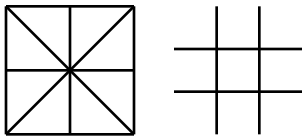


Figure B.3: The initial boards for Tic-Tac-Toe and Noughts & Crosses. In Tic-Tac-Toe the pieces are usually ○ and ● and they are placed on the intersection of the lattice points, but in Noughts & Crosses the pieces are X and O and are placed in the interior regions.

This game is played on a 3×3 board. Players alternate placing X or O. First player to reach 3-in-a-row (horizontally or diagonally) with their counters wins. If nobody achieves this by the time the game is over, the game is a tie.

Historically, these are the rules for Noughts & Crosses. Tic-Tac-Toe is different in that both players only have 3 pieces, and when all are played (if 3-in-a-row has not yet been reached) the pieces are slid to adjacent slots until one player achieves 3-in-a-row with his pieces [Bell83, p. 145].

The games are described in [Gardner59, pp. 37-38], [Abraham61, p. 174], [Bell79, pp. 91-92], [Bell88, pp. 5-6], [Binmore92, pp. 27-28], [Diagram92, p. 286], [Frochlichstein62, pp. 26-27], [Gardner77, pp. 210-211], [Gardner83, pp. 94-97], [Gardner92, pp. 202-203], [Gilgallon88, pp. 66-67], [Holt78, pp. 73-74], [Horne70, pp. 7-8], [Maguire90, pp. 96-99], [Mason63, pp. 139-140], [Mott-Smith54, pp. 126-129], [Pappas91, p. 240], [Pentagram90, pp. 113,122], [Silverman71, pp. 69-70], [Singleton74, p. 24] and [Dudeny67, p. 185]. The strategy is discussed in [Anderson89, pp. 83-85], and solved in [Berlekamp82, pp. 667-672]. [Dewdney89] contains an interesting application in which M.I.T. students built a Tinkertoy computer to play the game perfectly.

B.4 Dodgem

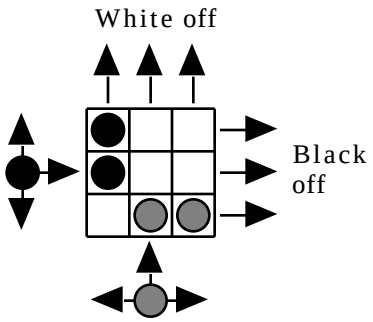


Figure B.4: The initial Dodgem board with white (○) against black (●)

This simple game is described in [Gardner88, pp. 156-158] and solved in [Berlekamp82, pp. 685-686]. The rules are taken from [Berlekamp82, p. 685]:

> "Colin Vout invented this excellent little game with two black cars and two white ones on a 3×3 board, starting as shown above. The players alternately move one of their cars one square in one of the three permitted directions (E, N or S for Black; N, E or W for White) and the first player to get both of his cars off the board wins. Black's cars may only leave the board across its right edge and White cars only leave across the top edge. Only one car is permitted on a square, and you lose if you prevent your opponent from moving."

B.5 Checkers / Draughts

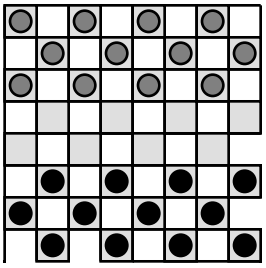


Figure B.5: The initial Checkers board with red (○) against black (●)

Checkers is described with and without variants in [Bell83, pp. 26-29], [Brandreth81, pp. 141-144], [Diagram92, pp. 34-46], [Morehead83, pp. 220-224], [Pentagram90, p. 22], [Pritchard82, pp. 53-59], [Provenzo81, pp. 195-207], [Schmittberger92, pp. 169-184], [Wiswell73] and [Cassidy91, pp. 6-7]. The following rules are taken from [Cassidy91, p. 6]:

> "**The Play:** The object of the game is to capture all of your opponent's pieces, or block them so they cannot be moved. Checkers are always moved diagonally, one square at a time, towards the other player's side of the board. You can capture an enemy checker by hopping over it.
>
> **Capturing:** Capturing, just like moving, is always done on the diagonal. You have to jump from the square directly next to your target and land on the square just beyond it (diagonally!). Your landing square has to be vacant. If you have a capture available on a turn, you have to take it. If you have more than one, it's your choice.

Multiple Captures: It is legal, in fact required, to capture more than one piece on a single move so long as the jumping checker has vacant landing spots available to it that will also serve as legal take-off points for another jump(s).

Kings: If you can get a checker to the last row of the board, that checker becomes a king. Turn it over. Now it can move, or capture, going in either direction – forwards or back, but always on the diagonal."

B.6 Chess

"Nat: You play chess?
Death: No, I don't.
Nat: I once saw a picture of you playing chess.
Death: Couldn't be me, because I don't play chess. Gin rummy, maybe."

– Woody Allen
Death Knocks from *Getting Even*

The rules have been omitted here as they are a bit lengthy and readily available. It is described with and without variants in [Bell83, pp. 18-21], [Brandreth81, pp. 154-164], [Grunfeld75, pp. 63-69], [Morehead83, pp. 213-219], [Pritchard82, pp. 38-48], [Schmittberger92, pp. 19-20, 185-220] and [Costello91, p. 19-34]. An excellent introduction is provided in [Diagram92, pp. 48-63]. [Fischer66] contains not only a wonderful introduction, but also provides a tremendous tutorial in the form of "quizzes". Endgame puzzles are described in [Brandreth85, pp. 48-49] and [Kendall62, pp. 61-65].

B.7 Connect-Four

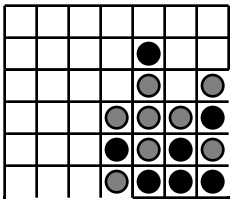


Figure B.6: A Connect-Four game in which white (○) has beaten black (●)

It is described in [Pentagram90, p. 92], and has been solved by Victor Allis in [Allis88] and found to be a win for white. The rules, below are taken from that reference:

"Connect-Four is a game for two persons. Both players have 21 identical men. In the standard form of the game, one set of men is yellow and the other set is red. The game is played on a vertical, rectangular board consisting of 7 vertical columns of 6 squares each. If a man is put in one of the columns, it will fall down to the lowest unoccupied square in the column. As soon as a column contains 6 men, no other man can be put in the column. Putting a man in one of the columns is called: a move.

The players make their moves in turn." ... "White makes the first move. Both players will try to get four connected men, either horizontally, vertically or diagonally. The first player who achieves one such group of four connected men, wins the game. If all 42 men are played and no player has achieved this goal, the game is drawn."

B.8 Dots and Boxes

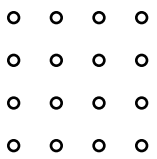


Figure B.7: The initial board for a game of Dots and Boxes played on a 4×4 lattice board. Games are usually played on $n \times n$ boards (where n is even) so that the number of available boxes will be odd and a tie will be impossible.

It is described in [Loyd59, pp. 88-89], [Diagram92, p. 288], [Holt78, p. 69], [Maguire90, p. 35], [Mason63, pp. 140-141], [Pentagram90, p. 63], [Sole88, p. 119] and [Gilgallon88, p. 68]. A version played on a triangular board is suggested in [Pentagram90, p. 13] and [Schmittberger92, p. 134]. Here are the rules from [Berlekamp82, pp. 507-550], where it is analyzed almost exhaustively:

> "Two players start from a rectangular array of dots and take turns to join two horizontal or vertically adjacent dots. If a player completes the fourth side of a unit square (box), he initials that box and must then draw another line (so that completing a box is a complimenting move). When all the boxes have been completed the game ends and whoever has initialed more boxes is declared the winner. A person who *can* complete a box is not obliged to do so if he has something else he prefers to do."

B.9 Fox and Geese / Wolves and Sheep / Asalto

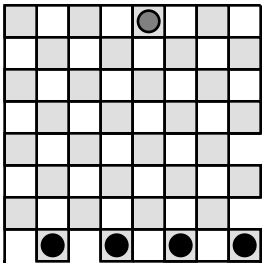


Figure B.8: The initial board for Fox and Geese.

Fox and Geese is described in [Diagram92, pp. 46-47], [Pentagram90, p. 131], [Kraitchik53, pp. 309-310], and [Silverman71, pp. 67-68]. [Berlekamp82, pp. 635-646] has analyzed it exhaustively and provided the following rules:

> "The game of Fox and Geese is played on an ordinary checkerboard between the *Fox*, who has just one piece (○ above) and the *Geese*, who have four pieces (● above). The players use squares of only one color (as in Checkers), and the Geese are initially placed in the squares marked above. The fox is usually placed as shown above, but since the Geese seem to have the better chances, it is perhaps wiser to allow the Fox to choose his own starting square (provided this has the correct color), and then let the Geese have first move.
>
> The Geese move diagonally one place forward – like ordinary checkers they may not retreat. The fox also moves diagonally one place, but like a King in Checkers, he may move in any one of the four diagonal directions. There is not taking or jumping. The Geese aim to trap the Fox so that he has no legal move, while

conversely the Fox tries to break through the barrier of Geese so that he can stay alive indefinitely. We can therefore say simply that the first player unable to move is the loser, the usual normal play convention."

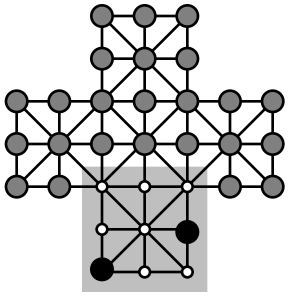


Figure B.9: The initial board for Wolves-and-Sheep / Asalto (known as Fox and Geese in most references, however).

However, Fox and Geese sometimes refers to a different game, the one shown in the figure above. [Berlekamp82, p. 631] renames this game "Wolves-and-Sheep" to eliminate confusion. It is described in [Agostino85, p. 67], [Bolt85, p. 100], [Brandreth81, pp. 127-129], [Costello91, pp. 10-13], [Grunfeld75, pp. 94-95], [Pentagram90, p. 41], [Pentagram90, p. 153] and [Provenzo81, pp. 174-178]. [Cassidy91, pp. 10-11] calls it "Dalmatian Pirates and the Volga Bulgars" and "Officers and Sepoys". The following rules are taken from [Cassidy91, p. 10]:

"**To Start:** The Pirates (○ above) are 24 in number and the Bulgars (● above), an embattled 2. The Pirates locate their pieces in the 24 spaces outside the fortress (shown by the ▒ square); the Bulgars locate their 2 on any of the 9 spaces inside the fortress. The Pirates play first.

The Play: On a turn, players move one piece along a line to an adjacent empty space. The Pirates can only move toward the fortress (or, should I say, they can't move *away* from the fortress, since it is legal to go from one point to another if both points are equidistant from the fortress). Once the Pirate is on any of the 9 points, it can move in any direction it wants either in the Fortress, or elsewhere. The Bulgars can always move in any direction, anywhere on the board.

Capturing: Capturing is done in any direction by hopping over the captured piece. Only the Bulgars can capture, and only in the following way: the Pirate has to be adjacent to the Bulgar, and the landing spot has to be open. As in checkers, multiple jumps are possible, although not required, so long as the Pirate pieces are all separated by open landing spots. If the Bulgar has the opportunity to jump, he or she must take it. If more than one are available – their choice.

Winning: The Pirates can win in either of two ways: (1) Occupy all 9 spaces inside the fortress. (2) Trap both bulgars so that they can't move. The Bulgars win by capturing enough Pirates to make their task impossible."

B.10 Go

The rules have been omitted here as they are a bit lengthy and easily available. The game is described in [Agostino85, p. 107], [Brandreth81, pp. 165-168], [Cassidy91, pp. 16-19], [Freeman79, pp. 133-142], [Grunfeld75, pp. 42-51], [Jackson75, pp. 176-185], [Pritchard82, pp. 73-82], [Schmittberger92, pp. 58-66] and

briefly mentioned in [Costello91, p. 10] and [Kraitchik53, pp. 279-280]. It is also covered with and without variants in [Bell83, pp. 124-126] and [Diagram92, pp. 158-167]. [Berlekamp94] contains an application of combinatorial game theory to Go endgames.

B.11 Gomuku / Go-Bang / Renju and Pente / Ninuki Renju

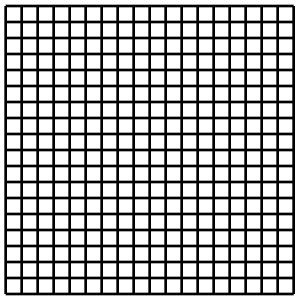


Figure B.10: A 19 × 19 Gomuku board

Gomuku is mentioned in [Berlekamp82, pp. 676] and described in [Bell83, pp. 127], [Brandreth81, p. 168], [Gardner83, pp. 97-102], [Gilgallon88, pp. 68], [Horne70, p. 11], [Jackson75, pp. 164-165], [Koch92, pp. 33-34], [Pentagram90, p. 158], [Pritchard82, 140-144], [Provenzo81, pp. 148-152], [Schmittberger92, p. 52] and [Diagram92, p. 166]. The following rules are taken from [Diagram92, p. 166]:

> "This is a straightforward game played on a Go board. It originated in Japan and is sometimes called Go-bang or Spoil five." ... "Each player has a set of 100 stones; one set black and the other white.
>
> **Objective:** Players aim to position five stones so that they form a straight line (horizontally, vertically or diagonally).
>
> **Play:** The board is empty at the start of the game, and black has the opening move. The players take it in turns to place a stone on any point (line intersection). Once a stone has been placed it may not be moved again until the end of the game. If all the stones are used up before either player has succeeded in forming a 'five', the game may either be declared drawn, or the players may take it in turns to move one stone one point in a horizontal or vertical direction until a 'five' is formed."

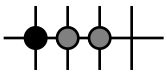


Figure B.11: The conditions for a black capture of red's pieces in Pente / Ninuki Renju

Pente / Ninuki Renju differ from Gomuku only in that an additional winning condition is the capture of 5 "pairs" of pieces. This is possible if there are two adjacent opponent pieces flanked on one side by the capturing player's piece and free on the other. This is shown in the figure above, where black can capture red's two pieces (which are removed from the board) by placing a piece on the empty point on the right.

B.12 Hex

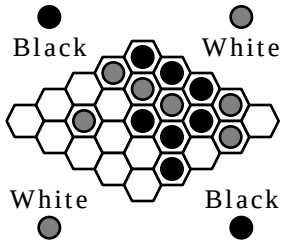


Figure B.12: Hex: a win for black

Hex is described in [Gardner59, pp. 73-83], [Beasley89, pp. 142-143], [Berlekamp82, pp. 680], [Bolt90, pp. 12-13], [Brandreth81, pp. 125-126], [Cassidy91, pp. 20-21], [Gardner88, pp. 158-159], [Holt78, pp. 86-87], [Pritchard82, pp. 86-89], [Silverman71, pp. 79-80] and [Diagram92, pp. 172-173]. It is analyzed in [Binmore92, pp. 37-41]. The rules below are from [Beasley89, pp. 142-143]:

> "It is played on a board such as that shown in the figure above, the actual size of the board being a matter for agreement between the players. Each player takes opposite sides, and his move is to place a man in any unoccupied cell, his objective being to form a continuous chain between his two sides. Thus Figure B.12 shows a win for black."

B.13 Hoppers / Halma / Chinese Checkers

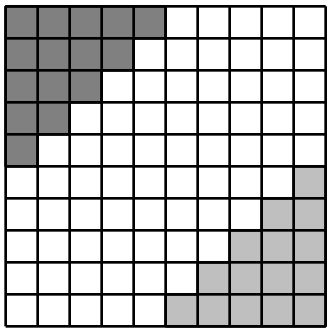


Figure B.13: The initial empty board for Hoppers. Pieces are placed in the centers of the corner camp squares.

Mentioned with and without variants in [Brandreth81, pp. 132-133], [Pentagram90, p. 91], [Provenzo81, pp. 91-108] and [Cassidy91, pp. 24-25], from which the following rules for Hoppers are taken:

> "**To Start:** Each player places 15 pieces of his or her color in the spaces of one of the corner camps.
>
> **The Play:** On a turn, a player can either "step" or "hop". A "step" is moving a piece to an adjacent vacant square in any direction, including diagonally. A "hop" is jumping over an adjacent piece, in any direction, including diagonally, into a vacant space. If more hops are available for the moving piece the player can take them or not – their choice. Hopping and stepping are done within the Corner camp exactly the same as outside it. *The pieces hopped over can be either friendly or enemy, and they are not removed from the board.*

Winning: The first player to fully occupy the enemy's Corner Camp is the winner.

Note: To stop an obnoxious opponent from simply leaving a piece in their corner camp forever so as to block the enemy from fully occupying it, the following rules can be used: A Corner Camp is considered full, even if one or more of the pieces in it belong to the player who started there."

Chinese Checkers is mentioned in [Diagram92, pp. 64-65], [Schmittberger92, p. 8] and [Freeman79, pp. 146-148] and is almost exactly the same as Hoppers, but played on a triangular, star-shaped board.

B.14 L Game, The

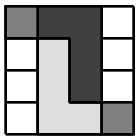


Figure B.14: The initial board for the L game

The L game is mentioned in [Pritchard82, pp. 107-112]. The rules are simple. Here they are, from [Berlekamp82, pp. 364-365 and pp. 388-389], where the game is also described and heavily analyzed:

"It is played on a 4×4 board. Each player has his own L-shaped piece which may be turned over, and there are 2 neutral 1×1 squares. A move has two parts:

You *must* lift up your own L-piece and put it back on the board in *another* position. You *may*, if you wish, change the position of *one* of the two neutral pieces. If you can't move, because there's only the one place for your L-piece, you lose."

B.15 Mancala / Awari / Wari

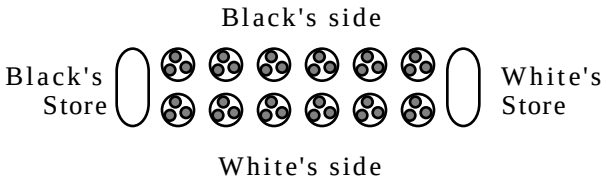


Figure B.15: The beginning position for the game of Mancala.

Mancala is described with varying rules in [Bell79, pp. 113-114], [Bell88, pp. 22-29], [Brandreth81, pp. 140-141], [Costello91, pp. 6-9], [Freeman79, pp. 119-120], [Holt78, pp. 80-81], [Jackson75, pp. 160-161], [Pappas91, pp. 196-198], [Pentagram90, p. 46], [Pritchard82, pp. 113-117] and [Cassidy91, pp. 32-33]. The following rules are taken from [Cassidy91, pp. 32-33]:

"**To Start:** Each player puts 3 pieces in each of the 6 spaces along his/her side of the board. In Mancala, the color of the pieces doesn't matter. Both players can use the same color, or a mix of colors. It's irrelevant.

The Play: Let's say it's your turn. Pick up all the pieces (or piece, if it's later in the game and there's only one) from any one of the six spaces on your side. Then, moving to the right (counter-clockwise), put one piece in each space you come to (your spaces, or your opponent's spaces; use them both). If you hit your Store, put a single piece in it. If you hit your opponent's Store, skip it.

Free Turn: If your last piece ends up in your own Store, you get a free turn.

Capturing: If you last piece ends up in an empty space on your side of the board, you have captured all the pieces in the space directly opposite. Collect them and put them in your Store along with the single piece of yours that made the capture. That ends your turn.

How the Game Ends: When all six of your spaces are empty, the round is over. However, it is usually not in your best interest to be the player with the empty spaces, because your opponent can then place all of the pieces left in *his* six spaces in *his* Store.

Scoring: Count the number of pieces in your Store, that becomes your score. If you're playing a single round, whoever has the most pieces in their Store at the end of the round wins."

B.16 Nim

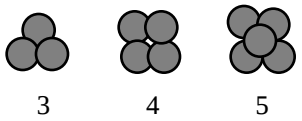


Figure B.16: A nim game with piles of size 3, 4 and 5.

Nim forms one of the foundations of combinatorial game theory. The Sprague-Grundy theory [Guy89, p. 26] states that any impartial game is equivalent to a nim heap with reversible moves[24]. References to Nim can be found in [Gardner77, pp. 212-213], [Bolt90, pp. 12-13], [Brandreth81, p. 216], [Grunfeld75, p. 268], [Holt78, pp. 66-67], [Kraitchik53, pp. 86-88], [Pentagram90, p. 97], [Schmittberger92, p. 131], [Silverman71, pp. 131-132], [Singleton74, pp. 24-29] and [Costello91, pp. 10-11]. It is analyzed a bit in [Northrop44, pp. 36-40], [Gardner59, pp. 151-157], [Gardner61, p. 63], [O'Beirne65, pp. 151-167], [Agostino85, p. 147] and [Binmore89, pp. 35-37]. Exhaustive analysis can be found in [Beasley89, pp. 99-108], [Guy89] and [Berlekamp82]. The following rules are taken from [Beasley89, pp. 99]:

"Players divide a number of counters or other objects into piles, and each player in his turn may remove any number of counters from any one pile. ... A player's objective is to take the last counter."

B.17 Nine Men's Morris

[24] A reversible move is a move that does not change the outcome, only delays it.

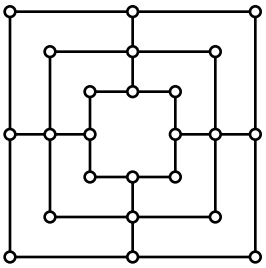


Figure B.17: The starting board for Nine Men's Morris.

This was already introduced in section 7.2.5, but it is repeated here for completeness. It was mentioned briefly in [Berlekamp82, p. 673], and described with and without variants in [Bell83, pp. 142-145], [Brandreth81, pp. 122-123], [Costello91, pp. 6-7], [Diagram92, pp. 210-213], [Dudeny70, pp. 58-59], [Grunfeld75, pp. 59-61], [Holt78, p. 75], [Mott-Smith54, pp. 132-135], [Pentagram90, p. 83], [Provenzo81, pp. 29-40] and [Cassidy91, pp. 34-35]. The following rules are taken from [Cassidy91, p. 34]:

> "**To Start:** Players start with 9 pieces each, and the board starts empty.
>
> **Setting Up:** Taking turns, players put all their pieces, one at a time, on 18 vacant points on the board. Having accomplished that, players continue taking turns. A turn consists of moving a piece along a line to an adjacent empty point.
>
> **Making a string:** A string is a complete line-up of 3 pieces – same color – filling all 3 points of a line. Any line counts. Players can make a string either during the set-up phase or during play. Once a player has managed to build a string, he or she is immediately allowed to grab any enemy piece. The only limitation: an enemy piece from an enemy string may not be removed – unless no other piece is available. It's legal to break up one of your own strings by moving one of its pieces out and then – if your opponent is napping and doesn't block – putting it back in place on a later move, thereby reforming the same string all over. If you succeed, you can claim another enemy piece.
>
> **Winning:** A player wins by getting his or her opponent down to 2 pieces. A player also wins if the opponent is blocked so that no move can be made. A player who has only 3 pieces remaining is allowed, on a turn, to move any one of his pieces to any vacant space. This is known as "flying" and it's designed to give the underdog a fighting chance."

B.18 Othello / Reversi

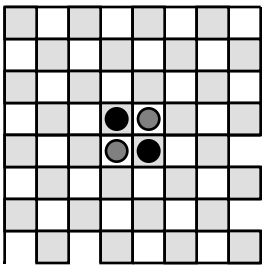


Figure B.18: The starting board for Othello / Reversi.

This game, a variant of Tic-Tac-Toe, is mentioned in [Gardner66, 75-78], [Brandreth81, pp. 152-153], [Freeman79, pp. 143-146], [Pentagram90, p. 109], [Pritchard82, 145-150], [Provenzo81, pp. 214-218] and [Diagram92, pp. 302-303]. The following rules are taken from [Diagram92, pp. 302-303]:

> "**Objective:** Play ends when there is a piece on every square; the winner is the player who has the most pieces with his face up a the end of the game.
>
> **Turns** alternate. In his turn each player attempts to place one piece on the board with his color or symbol face up." ...
>
> "**Taking:** After the first four pieces are placed, each player attempts to make one move in each turn. Only taking moves are permitted, and if a player is unable to make a taking move he loses a turn. (Both players are, however, limited to a maximum of 32 plays.) A taking move is made by positioning a piece so that: (a) it is in a square next to a square containing an opposition piece; and (b) it traps at least one opposition piece in a line in any direction between itself and another of the taker's pieces." ...
>
> "When a piece is taken it is turned over to show the other player's symbol or color. A piece may be turned over many times during a game as it passes from one player to the other. Pieces are never removed from the board when they are taken.
>
> **Multiple takes** By positioning a single piece a player may simultaneously take several in more than one line ." ... "Note that pieces may not be taken if a line is completed only when a piece is turned over."

B.19 Roundabouts / Surakarta

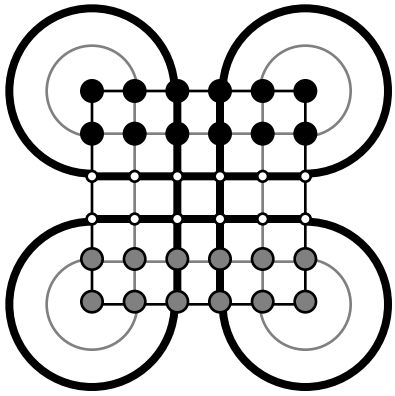


Figure B.19: The starting board for Roundabouts.

It is described with and without variants in [Bell83, pp. 96-97] and [Cassidy91, pp. 12-13]. The following rules are taken from [Cassidy91, pp. 12-13]:

> "**To Start:** Let's make you player number one. It so happens you like the color black. So take 12 black pieces and put them on the black dots on your side of the board. Your opponent puts 12 white pieces on the white dots on the other side of the board.
>
> **The Play:** On every turn, you either move, or capture.

Moving: You can move to any unoccupied adjacent point, in any direction (including diagonally). Remember, moving is done only to *adjacent* points. (A point is the intersection of two lines).

Capturing: Capturing is a lot more exciting than moving because capturing has to be done via a loop, or race-track. There are two racetracks on the board, the inner one (red) and the outer track (yellow). Note that all the points on the board, except the far corners, are on one track or the other, and you can launch yourself from any of them. You can go as far as you like along a racetrack except that you can't pass over another's piece, either your own or your opponent's. To capture an enemy piece, you have to land on its point, bumping it off and putting it in your prisoner pile. However, to get there you have to travel via a loop or loops." ...

"**Winning:** "Let's say you've captured all the enemy pieces. Count the number of your *own* pieces left on the board, and that becomes your score. Play 2 games (alternate who goes first) and add up your scores to find the winner of the entire match".

Bibliography

[Abraham61] R. M. Abraham. Easy-to-do Entertainments and Diversions with Coins, Cards, String, Paper and Matches. Dover Publications, Inc., 1961.

[Agostino85] Franco Agostino and Nicola Alberto DeCarlo. *Intelligence Games*. Simon and Schuster, 1985.

[Allis88] Victor Allis. *A Knowledge-based Approach of Connect-Four*. Master's thesis, Vrije Universiteit, October 1988.

[Anderson89] Harry Anderson and Turk Pipkin. Games you Can't Lose. Simon and Schuster, 1989.

[Beasley89] John D. Beasley. *The Mathematics of Games*. Oxford University Press, 1989.

[Bell79] R. C. Bell. *Board and Table Games from Many Civilizations*. Oxford University Press, 1979.

[Bell83] R. C. Bell. *The Boardgame Book*. Exeter Books, 1983.

[Bell88] Robbie Bell and Michael Cornelius. *Board Games Round the World: A Resource Book for Mathematical Investigations*. Cambridge University Press, 1988.

[Berlekamp82] Elwyn Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways*. Academic Press Inc., 1982.

[Berlekamp94] Elwyn Berlekamp and David Wolfe. *Mathematical Go Endgames : Nightmares for the Professional Go Player*. Ishi Press International, 1994.

[Binmore92] Ken Binmore. *Fun and Games : A Text on Game Theory*. D. C. Heath and Company, 1992.

[Bolt85] Brian Bolt. *More Mathematical Activities*. Cambridge University Press, 1985.

[Bolt90] Brain Bolt. *The Amazing Mathematical Amusement Arcade*. Cambridge University Press, 1990.

[Brandreth81] Gyles Brandreth. *The World's Best Indoor Games*. Pantheon Books, 1981.

[Brandreth85] Gyles Brandreth. *Classic Puzzles*. Harper & Row Publishers, 1985.

[Cassidy91] John Cassidy. *The Book of Classic Board Games*. Klutz Press, 1991.

[Conway76] J. H. Conway. *On Numbers and Games*. Academic Press, London and New York, 1976.

[Costello91] Matthew J. Costello. *The Greatest Games of All Time*. John Wiley & Sons, 1991.

[Dewdney89] A. K. Dewdney. Computer Recreations: A Tinkertoy computer that plays tic-tac-toe. In *Scientific American*, pages 120-123, October, 1989.

[Diagram92] Diagram Visual Information, Ltd. *Family Fun & Games*. Sterling Publishing Company Inc., 1992.

[Dudeny67] Henry Ernest Dudeney. *536 Puzzles and Curious Problems*. Charles Scribner's Sons, 1967.

[Dudeny70] Henry Ernest Dudeney. *Amusements in Mathematics*. Dover Publications, Inc., 1970.

[Fischer66] Bobby Fischer, Stuart Margulies, and Donn Mosenfelder. *Bobby Fischer Teaches Chess*. Bantam Books, 1966.

[Freeman79] Jon Freeman. *The Playboy Winner's Guide to Board Games*. Playboy Press, 1979.

[Frochlichstein62] Jack Frochlichstein. *Mathematical Fun, Games and Puzzles*. Dover Publications, Inc., 1962.

[Garcia94] Dan Garcia. Xdom : A Graphical, X-based Front-End for Domineering. In *Proceedings of the Workshop on Combinatorial Games*, July 1994. Program available at http://http.cs.berkeley.edu/~ddgarcia/software/xdom/ or send e-mail to ddgarcia@cs.Berkeley.EDU.

[Gardner59] Martin Gardner. *The Scientific American book of Mathematical Puzzles and Diversions*. Simon and Schuster, 1959.

[Gardner61] Martin Gardner. *Entertaining Mathematical Puzzles*. Dover Publications, Inc., 1961.

[Gardner66] Martin Gardner. *New Mathematical Diversions from Scientific American*. The University of Chicago Press, 1966.

[Gardner77] Martin Gardner. *Mathematical Carnival*. Vintage Books, 1977.

[Gardner83] Martin Gardner. *Wheels, Life and Other Mathematical Amusements*. W. H. Freeman and Co., 1983.

[Gardner88] Martin Gardner. *Time Travel and Other Mathematical Bewilderments*. W. H. Freeman and Co., 1988.

[Gardner92] Martin Gardner. *Fractal Music, Hypercards and More...* W. H. Freeman and Co., 1992.

[Gilgallon88] Barbara Gilgallon and Sue Seddon. *Travel Games*. Ward Lock Limited, 1988.

[Grunfeld75] Frederic V. Grunfeld, UNICEF. *Games of the World*. Plenary Publications International, Inc., 1975.

[Guy89] Richard K. Guy. *Fair Game*. COMAP, Inc., 1989.

[Hollow91] Rhys Hollow. *Gamemaster*. Shareware software, 1991. Program available at ftp://mac.archive.umich.edu/archive/mac/game/board/gamemaster1.0.cpt.hqx or send email to gurhs@uniwa.uwa.oz.au

[Holt78] Michael Holt. *Math Puzzles & Games, Volume 1*. Dorset Press, 1978.

[Horne70] Sylvia Horne. *Patterns and Puzzles in Mathematics*. Franklin Publications, 1970.

[Jackson75] John Jackson. *A Player's Guide to Table Games*. Stackpole Books, 1975.

[Kaplan68] Philip Kaplan. *Puzzle Me This*. Warner Books, Inc., 1968.

[Kendall62] P. M. H. Kendall and G. M. Thomas. *Mathematical Puzzles for the Connoisseur*. Charles Griffin & Co. Ltd., 1962.

[Kierulf90] Anders Kierulf. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies*. Ph.D. thesis, Swiss Federal Institute of Technology (ETH), Zürich, 1990. Nr. 9135.

[Koch92] Karl-Heinz Koch. *Pencil & Paper Games*. Sterling Publishing Co., Inc., 1992.

[Kordemsky72] Boris Kordemsky. *The Moscow Puzzles*. Dover Publications, Inc. 1972.

[Kraitchik53] Maurice Kraitchik. *Mathematical Recreations*. Dover Publications, Inc., 1953.

[Loyd59] Sam Loyd. *Mathematical Puzzles of Sam Loyd*. Dover Publications, Inc., 1959.

[Maguire90] Jack Maguire. *Hopscotch, Hangman, Hot Potato, & Hahaha*. Fireside, 1990.

[Mason63] Bernard S. Mason and Elmer D. Mitchell. *Party Games*. Harper and Row, Publishers., 1963.

[Morehead83] Albert H. Morehead and Geoffrey Mott-Smith. *Play According to Hoyle : Hoyle's Rules of Games*. Penguin Books, 1983.

[Mott-Smith54] Geoffrey Mott-Smith. *Mathematical Puzzles for Beginners & Enthusiasts*. Dover Publications, Inc., 1954.

[Northrop44] Eugene P. Northrop. *Riddles in Mathematics*. D. Van Nostrand Company, Inc., 1944.

[O'Beirne65] T. H. O'Beirne. *Puzzles & Paradoxes*. Oxford University Press, 1965.

[Pappas91] Theoni Pappas. *More Joy of Mathematics*. Wide World Publishing/Tetra, 1991.

[Pentagram90] Pentagram. *Pentagames*. Simon and Schuster, Inc., 1990.

[Pritchard82] David Pritchard. *Brain Games*. Penguin Books Ltd., 1982.

[Propp94] Jim Propp. Three-Person Impartial Games. In *Theoretical Computer Science*, 1994.

[Provenzo81] Asterie Baker Provenzo and Eugene F. Provenzo, Jr. *Favorite Board Games You Can Make and Play*. Dover Publications, Inc., 1981.

[Schmittberger92] R. Wayne Schmittberger. *New Rules for Classic Games*. John Wiley and Sons, Inc., 1992.

[Silverman71] David L. Silverman. *Your Move : Logic, Math and Word Puzzles for Enthusiasts*. Dover Publications, 1971.

[Singleton74] Robert R. Singleton and William F. Tyndall. *Games and Programs: Mathematics for Modeling*. W. H. Freeman and Company, 1974.

[Sole88] Tim Sole. The Ticket to Heaven and other Superior Puzzles. Penguin Books, 1988.

[Tzu83] Sun Tzu. *The Art of War*. Dell Publishing, 1983.

[Winston84] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley Publishing Company, Inc., 1984.

[Wiswell73] Tom Wiswell. *The Science of Checkers and Draughts*. A. S. Barnes and Co., Inc., 1973.

[Wolfe94] David Wolfe. The games toolkit. In *Proceedings of the Workshop on Combinatorial Games*, July 1994. Program available at http://http.cs.berkeley.edu/~wolfe/, or send e-mail to wolfe@cs.Berkeley.EDU.