

# A GENERAL ALGORITHM FOR TIC-TAC-TOE BOARD

## EVALUATION\*

*Aaron Gordon  
Department of Computer Science and Information Systems  
Fort Lewis College  
Durango, CO, 80301  
970-247-7436  
gordon\_a@fortlewis.edu*

### ABSTRACT

This paper introduces a general algorithm for evaluating a tic-tac-toe board. The algorithm is useful for finding both a winning and a blocking move in tic-tac-toe. For students, this algorithm is an example of the elegance of a general solution as opposed to a brute-force solution.

### INTRODUCTION

Tic-tac-toe is a simple game often used as a programming assignment for computer-science students or as an in-class example of how to develop software [1, 2, 3, 4]. Every tic-tac-toe program should include a way of representing the board and evaluating the board for a win. Often, this evaluation is done by brute force, checking all eight possibilities (on the traditional 3x3 tic-tac-toe board).

Strategy can be included in an automated tic-tac-toe player; strategy most often includes ways of finding an available winning move and finding an opponent's potential winning move that can be blocked. Finding these moves can also involve brute force with 2 or 3 possibilities checked for each open space on the board.

We often tell our students that a general solution to a problem is usually better than a brute force solution. This paper shows a general algorithm that can be used to test for a win, a winning move and a block. These tests can be used with a variety of strategies for playing the game. The algorithm involves keeping extra tables to represent the state

---

\* Copyright © 2005 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

of the game. Every potential move can be evaluated as a winning move with one table lookup and evaluated as a blocking move with one more lookup.

### ALGORITHM OVERVIEW

A tic-tac-toe board can be represented as a perfect square where every cell is represented by a unique integer in the range 1 to 9 (see Figure 1). Notice, that every row, column, and diagonal sums to 15, and there is no way to get a sum of 15 from three cells unless they share a row, column or diagonal.

With this representation, a potential move is evaluated as a winning move by checking whether it produces a sum of 15 with exactly two other cells occupied by the same player. A program can check for a block by checking whether the potential move produces a sum of 15 with exactly two other cells occupied by the opposing player.

8	1	6
3	5	7
4	9	2

Figure 1 - a 3x3 magic square

### THE SPECIFICS

For each player, two Boolean arrays are kept. Array taken keeps track of those cells occupied by the player (there are separate arrays for each player). Array taken has nine cells numbered 1..9. Each array element is initialized to false. When a player makes a move the array element corresponding to a cell on the board (having the same number) is set to true.

Array pairs keeps track of two-in-a-row combinations for the player. This information is represented by recording all sums consisting of two cells occupied by the player. This array needs to be indexed from 3 to 17 and with each element initialized to false. When a player makes a move, all resulting sums are recorded in the player's pairs array, as in:

```
for (int k=1, k<=9; k++) if (taken[k]) pairs[move + k] = true;
```

Many of the recorded pairs do not share the same row, column or diagonal (such as 8 and 9). Recording this information does not hurt our solution and is more efficient than incurring the overhead of testing each pair for collinearity.

### Testing for a win

For a move to be a winning move, its cell number must sum to 15 with exactly two other cells occupied by the player. To test if a specific move is a win, a player need only check to see if that specific move sums to 15 with exactly two previous moves the player

made. This check can be done with one table lookup; just check if `pairs[15-move_number]` is true. To test for a block, a player only needs to do a similar check on the opponent's pairs array. Below is a sample program that uses this algorithm to test for a win.

## SIGNIFICANCE

One goal of computer-science education is to teach students to find and use generalizations wherever possible. We say a general solution is more elegant than one where individual cases have to be specified. The algorithm detailed here is a general algorithm applied to a problem where a general solution is not obvious. Here, the program uniformly updates the pairs array as shown in the for-loop above. The test for a win does not involve looking at other squares in line with the tested square; it just involves one table lookup.

This solution can also be extended to tic-tac-toe games on larger boards. For a 4x4 tic-tac-toe game (where the goal is to get 4 in a row), three levels of arrays would be needed. It can be a good exercise for students to evaluate the algorithm for time and space needs as they investigate variations of the game with larger boards.

## SAMPLE PROGRAM

A tic-tac-toe player can employ any of a number of strategies. Most strategies involve determining if a move is a win, and many strategies involve determining if a move is a block. The program shown here involves two players utilizing the random-move strategy. The players use the above algorithm to determine the winner.

```
/* Here is the definition of the board
 8  1  6
 3  5  7
 4  9  2

Here is a tic-tac-toe game played between 2 random players Sam
and Pat. Wins are checked for using the generalized tic-tac-toe
algorithm developed by Aaron Gordon */

public class TttNums {
    private static boolean [] board; //the tic-tac-toe board
    private String name;             //the name of the specific player
    private boolean trying;          //false if this player has won
    private boolean [] taken;        //indicates squares occupied by
        this player
    private boolean [] pairs;        //array marks two in a row for
        this player
    static TttNums [] player;       //the players

    public static void main(String [] arg) {
        player = new TttNums[2];
        player[0] = new TttNums("Sam");
        player[1] = new TttNums("Pat");
        int who = 1; //player numbers are 1 and 0
        board = new boolean[10]; //only 1-9 are used
        for (int k=0; k<9 && player[0].isTrying() &&
            player[1].isTrying(); k++) {
```

```

        player[who].move();
        printBoard();
        who = 1 - who;           //now it's the other player's turn
    } //for
} //main

public static void printBoard() {
    printSquare(8); printSquare(1); printSquare(6);
    System.out.println("\n-----");
    printSquare(3); printSquare(5); printSquare(7);
    System.out.println("\n-----");
    printSquare(4); printSquare(9); printSquare(2);
    System.out.println("\n\n");
} //printBoard

public static void printSquare(int spot) { //prints one square for
    board
    int who = -1;
    if (player[0].isTaken(spot))           who = 0;
    else if (player[1].isTaken(spot))      who = 1;
    if (who < 0) System.out.print("|      "); //square
        unoccupied
    else System.out.print("|  " + player[who].getName() + "  ");
} //printSquare

public String getName() {
    return name;
} //getName

public boolean isTaken(int spot) {
    return taken[spot];
}

public boolean isTrying() { //player stops trying after winning
    return trying;
} //isTrying

public TttNums(String name) { //constructor
    taken          = new boolean[10];
    pairs          = new boolean[16];
    trying         = true;
    this.name      = name;
} //constructor

// game playing strategy goes here - this version makes random moves
public void move() { //make a random move
    int spot;
    do {
        spot = (int)(Math.random() * 9.0 + 1.0);
    } while (board[spot]);
    System.out.println(name + ": move to " + spot);
    if (pairs[15-spot]) { //check for a win
        System.out.println(name + "!!!WIN!!!");
        trying = false;
    } else {
        for (int j=1; j<10; j++) { //update pairs array
            if (taken[j] && j + spot < 15) {
                pairs[j+spot] = true;
                System.out.println(">>>>>> " + name +
"    setting " + (j+spot));
            } //if

```

```
        } //for j
    } //else
    taken[spot] = board[spot] = true;
} //move
} //class
```

## REFERENCES

- [1] Hayes, B. S., Noughts and crosses meets 1970s UI?  
<http://www.sims.berkeley.edu/academics/courses/is255/f04/assignments/Assignment.TicTacToe.ADL.rtf>, October, 2004
- [2] Wachsmuth, B. G., Tic-tac-toe game,  
<http://pirate.shu.edu/~wachsmut/Teaching/CSAS1111/Assigns-CPP/assign7.html>,  
retrieved July 5, 2005
- [3] Massey, B., Tic-tac-toe board evaluation,  
<http://www.cs.pdx.edu/~bart/cs541-fall2001/homework/2-game.html>, retrieved  
July 5, 2005
- [4] Appel, A. W., Game player programs,  
<http://www.cs.princeton.edu/courses/archive/spring05/cos217/asgts/gameplayer/>,  
retrieved July 5, 2005