< Prev	Recursion Index	Next >

Tic Tac Toe

(also know as: *Noughts and Crosses*, *X and Zero*)

This is one of the favorites when it comes to recursion. All the more pleasurable if you can program it yourself without reading this article. It isn't all that difficult if you give it a few days of thought. So give it a shot before you read the solutions I've provided.

The game is itself quite simple, and very *very* famous. If you are one of the - less fortunate - who never played pencil-paper games in the middle of your 3rd grade Math class, or equivalently, if you're 72 and have forgotten how to play your favorite game of tic tac toe, read the intro to the game. Otherwise, take a sheet from somewhere, start scribbling and work on it - before you go on to read the rest of this article. Good Luck!

The Game

A 3x3 grid. 2 players. One player plays 'X', the other plays '0'. The players take turns. Each marks a free location with his/her symbol (X or 0). The first player to get 3 of his symbols in a straight line, either vertically, horizontally or diagonally, wins. If all 9 locations are filled without any winning line, the game is drawn. E.g.:

0		X
0	X	
X	0	X

The player marking 'X' in the above game has won, since he managed to get 3 X's in a diagonal straight line.

The Program

Before we start off on the program, we should have a general idea of what we want it to do. So, let me make a few suggestions...
The program should support the following game modes:-

- Human vs Human

- Human vs Computer
- Computer vs Computer

The first mode is trivial. Let's deal with the second mode. Once we are able to program the second mode, i.e., once we are able to make the Computer play a game, the third mode should become a trivial problem, and can be easily accomplished. In this article, I am not going to discuss anything about the interface and such (I leave this juicy stuff to you), instead, I concentrate only on the logic.

First, let's look at it broadly. There could be two approaches the computer could take to play a game-
1) plan through the entire game at the start of the game itself, or
2) decide upon the best move at every turn.
Planning the entire game is not worth from many points of view. Besides, there's nothing wrong with selecting the best move at every turn 'just in case' it messed up somewhere along the way and needs to start over. So, the computer should decide upon the move looking only at the future, and ignoring the past. To come to think of it, this is how we humans think - *what has happened has happened, I'll just try my best from now on*.

Okay, so the Computer is going to select the best move at every one of its turns. If we have to program it to '*select*', then obviously, the Computer has to check out each possible move, assigning '*goodness values*' to each of them, and finally select the one with the maximum goodness value. Let's assume that there already exists a function Goodness( ) which returns the goodness value of a particular move [We'll get to this function later]. So, the following function should be able to decide upon the best move:

A - the 1 Dimensional 9-length array (as in Magic Squares) which represents the 2-D 3x3 grid.

```
int SelectBestMove(player) // selects a move
                           // for player 'player'
{
    max = -100
    for i = 1 to 9
        if A[i] is free
        {
            mark A[i]
            value = Goodness( )
            unmark A[i]
            if value > max
            {
                max = value
                best_locn = i
            }
        }
    return best_locn
}
```

Note: This is only the skeleton, just to show the logic. You'll have to add some beef to check for special conditions and prevent errors.

So, it is enough if we call the above function every time the computer has to play. Now, the question is, how do we go about checking the Goodness( ) value ??? For starters, the function Goodness( ) can check if the player *wins* by playing that move. So we can have a Goodness( ) which simply searches for three in a line. If the function finds such a winning line, it returns a high goodness value of, say, 128. If it does not, it simply returns 0.

This would work, but is definitely not good enough. It is equivalent to a child playing tic-tac-toe who simply searches for a place to get 3 in a row, without any deeper thinking, or future planning.

Let's try and scrutinize the way we humans play the game. First, we try to win in the current move (The above suggestion for Goodness( ) already does this). If we can't win in the current move, we try to make it as hard as possible for the opponent to win in the next move. Whatever we play, the opponent is then definitely going to select the best move for himself - in a way, he's going to compare the goodness values of all of *his* possible moves, and select the best one. If you think about it, the *better* this best-goodness-value that he obtains is, the *worse* it is for us. Right? So, in a way, we've got to select a move which gives him the worst best-goodness-value. Think about it for a while.

You may not realize it, but in the above paragraph, we have effectively given the recursive definition for the solution of this problem. See: For each possible move we can make, we first check if that move would make us win. (If yes, return 128). Then, we check the *best-goodness-value* that the opponent gets. The better this value is, the worse ours, and vice versa. The value of his *best-goodness-value* would correspond to a value of *our-goodness-value*. The function should return whatever this value of our-goodness-value is. One simple way of calculating our-goodness-value from the opponent's best-goodness-value is to take our-goodness-value as the *negative* of his best-goodness-value. So, the higher his is, the lower ours is. Ultimately, it's just a comparison of all goodness values, so the negative sign won't cause any problems. In fact, this way, if his best-goodness-value is, say, -128, it means that our-goodness-value is +128, which is equivalent to saying, "*I am sure to win in the next move, if I move here.*" Since we are using the negative-value idea, our recursive definition of Goodness( ) becomes:

```
Goodness( )
1) Check if opponent wins. If yes, return: -128.
2) For each possible next move,
```

```
                get -Goodness(opponent)
3) Finally, select the best (highest) of
                these values and return it.
```

As you may see, the above works very well, despite the modifications. Suppose, in step 2, we select a move which is supposed to make us win. We call Goodness(opponent). In the new instance of Goodness, the opponent checks (in step 1) if we win. We do, so the function returns -128. Since we are taking the negative of this value, we get +128, which is right! You can check similarly for other possibilities. The modification we made above requires a modification in our SelectBestMove( ) function that we defined earlier. The call to Goodness( ) should be changed from "`value = Goodness ()`" to "`value = -Goodness()`". We'll see about the parameters next.

In order to toggle between the two players as we go recursively deeper into a given move, we'll use the 'negative' concept again. Let player = 1 denote Player1, and player = -1 denote Player2. Then, our recursive function call will be:
value = -Goodness(-player). I hope that doesn't confuse you. Let's see the algorithm for Goodness( ):-

```
int Goodness(player)
{
    if CheckWin(-player)
        return -128

    max = -200
    for i = 1 to 9
        if A[i] is free
        {
            mark A[i]
            value = -Goodness(-player)
            unmark A[i]
            if value > max
                max = value
        }
    return max
}
```

As in the previous code fragment, you'll need to fine-tune several aspects of this one, but it is generally trivial, and depends on your data structures and such. Also note that the function above does not recurse infinitely. Why? Because there will come a time (around the 9th recursion) when none of A[i] is free, which means it never enters the loop, and thus, simply returns. (It will have to return 0 - for draw)

Okay, that takes care of most of it. But the program will still act kind of funny. Can you guess why? The answer is not obvious, but you may realize what's happening after you struggle with it for a few days. Let's see what the code above does in the following case: Suppose it is my turn, and there is a move I can make which will get me 3 in a row. Also, there is another move I could make which will definitely make me

win in my next move. The above code treats both these as equivalent, since both have a goodness value of +128. So how do we go about telling the program "The sooner the better"? The answer naturally has to do with the goodness value. Somehow, the 'closer' wins have to have greater precedence. This means that the distant values have to be reduced to a smaller fraction by the time it reaches the main function, SelectBestMove( ). One simple way to do this is: replace the statement

```
value = -Goodness(-player)
```

by

```
value = -Goodness(-player) / 2
```

So, effectively, the priority of deeper instances keeps on diminishing. The effect is that the computer selects the move such that it wins as soon as possible.

One last mention.

Better Methods?

There are some other possibilities we haven't yet looked into, which arguably, could be better than the logic above. You see, we have used the 'max' method, in which the computer selects the best possible move, which is the one with the maximum goodness value. This is fine and correct. However, the program also assumes that the opponent selects the best move. As a result, the computer may not appear to play smartly against a *normal* or *below-average* opponent.

One option available, though not guaranteed to improve all situations, is to take the *average* of the opponent's goodness value as the return value. This kind of generalizes the unpredictability of the opponent's moves. Maybe.

Another option is the **minmax** method. Here, we don't use the negative of the opponent's best goodness value. Instead, we use his worst goodness value. Makes a lot of difference. I haven't tried it out. If you've tried it out successfully, tell me about it.

Exercises

First, get your tic-tac-toe program working. There's nothing like playing against your own game. Your own baby.

1. Write a program using Recursion to play the following game: There are 21 stones. Two players take turns picking the stones. Each player should take at least 1 stone, and a

maximum of 4 stones at each turn. The player who picks the last stone is the loser.

< Prev	Recursion Index	Next >

erw_nerve@email.com *July 2000*

recursion index ☐ introduction ☐ magic squares ☐ tic tac toe ☐ connect 4
optimizing recursion trees ☐ sorting ☐ equation generator ☐ other problems

home ☐ jokes ☐ programming ☐ about me ☐ resume ☐ guest book

Comments:

This page is located at http://personal.vsnl.com/erwin/tictactoe.htm

PANIC

Display thank-you page Return to this page Submit

Recommend this page to someone?

e-mail

e-mail