| < Prev | Recursion Index | Next > |
|---|---|---|

# Connect 4

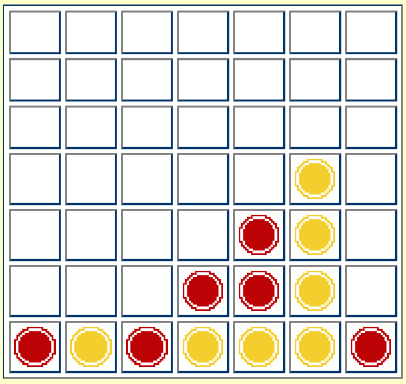
**(also know as: *Plot 4, Go 4 it*)**

*This is a game very similar to Tic Tac Toe, but probably not as famous. It offers many more possibilities of games than the very restricted Tic-Tac-Toe. This introduces new philosophies into our world of Recursion. Fasten your seat-belts for a mind-blazing session into the intricacies of one of the best games I've known!*

## The Game

The actual game is played on a 7x7 grid rack. The rack is placed vertically, with an opening at the top. The players take turns dropping their coins into one of the 7 columns. The dropped coin rests on top of whatever was dropped into the same column earlier. Coins may not be dropped into a full column, ie, a column filled with 7 coins. The objective of the game is to get 4 of the player's coins in a continuous line, either horizontally, vertically or diagonally. If all 49 cells are filled without a winner, the game is drawn.

Eg: yellow wins in the following game.



## The Program

As you may already have realised, the program for this game should be very similar to that for Tic-Tac-Toe, as discussed in the previous section. Except that we don't have to check each cell at every move - just one in each column, i.e., there

are 7 possible moves, not 49, at the start of the game.

You could proceed as we did with tic tac toe, checking all possible moves, and recursing till a depth at which there are no further possible moves.

You *could* try that... to find out the hard way about the **limitations of Recursion**. Let me explain. Let's go back to the Tic Tac Toe program. Assume each call to the CheckWin ( ) function takes *t* seconds (possibly *t* = 0.0001). Consider the first move. There are 9 possibilities (which means 9 calls to the function). For each of these possibilities, there are 8 possibilities for the next move. For each of the 8, there are 7 further possibilities, and so on.
Total time taken for the first move (approxiamately) = (9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1) x *t*
which is, 9! x *t*.
9! = 362,880
9! x *t* = 36 seconds, assuming *t* = 0.0001 sec.s

Okay, forget about that. Now, let's see what happens if we do the same thing over here in Connect 4. Although incorrect technically, it is fair to assume there are 7 possibilities at every move, just for the sake of calculation [It is incorrect since, at some point, the columns start getting filled, and hence possibilities reduce]. Now, suppose the function which checks for a win takes some *t* seconds (*t*=0.0001). What happens in the first move? There are 7 possibilities. For each of these, there are 7 more possibilities... and so on 49 times!

So, the total time taken = (7 raised to 49) x *t*, which is... a big number. Too big.

Let us not check ALL possibilities at every move. Instead, let us just check the moves up to a depth of say, 5. In other words, we'll calculate (for each of our 7 possible moves) the opponent's 7 possible moves, who'll check for each of his 7 possible moves, our 7 possible moves,.... 5 times. In other words, the number of times the CheckWin( ) function is called is:

7 raised to 5 = 16807

Let's assume this takes just 1 second. Now, let's increase the recursive depth by 1. Number of times the CheckWin( ) function is called is now: 7 raised to 6
Time taken (depth 6) = 7 seconds

Let's again increase recursive depth by 1.
Time taken (depth 7) = 49 seconds

and again...
Time taken (depth 8) = 6 min.s

and again...
Time taken (depth 9) = 42 min.s

and so on... A depth of 12 will require a time of more than 9 days. A depth of 16 will require 63 years.... So let's forget about a depth of 49, shall we?

## Limitations of Recursion

I guess it is obvious what the problem is after the evaluation above. In Mathematics, we would call the algorithms *exponential*. The time taken for each successive value of depth increases *exponentially*. This is the limitation of recursion which destroys its usefulness prematurely. It forces the algorithms to recurse to only a small depth. Even the almost exponential increase in computer speed nowadays hardly offsets the exponential increase in the amount of time required for an increased depth.

## Back to Connect 4.

So, we're going to limit the depth of recursion. Or maybe we can provide the user with the choice for various depth levels, corresponding to increasing difficulty levels. It is obvious that the greater the depth the better the move, since the computer can look deeper into a given move, and look further 'ahead'.

The basic algorithm would be:-

```
int Goodness(player, depth)
{
    if CheckWin(-player)
        return -128
    if depth=0
        return 0

    max = -200
    for i = 1 to 7
        if column i is not full
        {
            drop coin into column i
            value = -Goodness(-player,depth-1) / 2
            remove coin from column i
            if value > max
                max = value
        }
    return max
}
```

The depth variable mechanism makes sure the recursive descent does not continue beyond a certain value. Assuming a depth of 5, the function should first be called by :

```
value = -Goodness(-player,5)
```

The rest is very similar to Tic-Tac-Toe.

Note that the time taken for any move is *t* x (7 raised to something). Therefore, it is very important that *t* be as small as possible. *t* represents the time taken for the CheckWin( ) function. This function is going to be executed some 7-raised-to-something times. So, this particular function should be as efficient (fast) as possible. For instance, the CheckWin( ) function should not check every row and column and diagonal for a possible win. Instead, it should check only the row and column and diagonals in which the last coin played resides. Further improvements are possible.

| < 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