Solving Tetris in C @ Things Of Interest

Exactly one of the following statements is true.

1. In Tetris, you can always get at least one line before losing.
2. In Tetris, a sufficiently evil AI can always force you to lose without getting a single line, by providing bad pieces.

Which one?

Prove it.

The setup

I created HATETRIS last year because I wanted Tetris players to suffer. It was something I created for my own amusement and as a programming challenge and to investigate the use of JavaScript as a gaming platform.

HATETRIS explores only to a depth of 1 piece. It looks at the immediate future and chooses the single worst piece for the present situation, ignoring the possibility of the player using the current piece to set up a situation where the 2nd piece forces a line. After the first iteration of the game, I started using a bit mask to express each row of the playing field ("well"), which made testing the insertion and removal of possible pieces faster, to the extent that a new piece appeared more or less instantly instead of appearing after a second or two. This improvement allowed the algorithm to be improved to look a second piece into the future, while taking more time. This made the game sluggish, though, so I never left this modification in place.

The other reason I didn't leave HATETRIS to look at depth 2 was because the game was still too easy. Even looking two pieces into the future it was startlingly easy to contrive situations whereby no matter which pair of pieces was supplied, the player could always force a line. Consistent scores of 2 or 3 lines did not satisfy me. HATETRIS+2 wasn't difficult enough. I noted that I intended to return to the project just as soon as I could create a game which *never* permitted the player to get lines. *Ever*. I determined to create the evil AI from case 2.

About a year passed and I finally got around to thinking about the problem again. After some additional thought I decided that the chances were that case 1 was actually true. **The player can always get a line.** But how to prove it?

The problem

"Tetris", the mathematical structure, has to be well-defined before it can be solved.

The "standard" Tetris well has a width of 10 and a height of 20. The piece enters the well from above. In "standard" Tetris the piece spawns in rows 21 and 22 which are hidden off the top of the screen; also, the player loses when a piece spawns in collision with an existing structure. My version is a little different: rows 21, 22, 23 and 24 are clearly visible at the top of the screen, providing room for a piece to be moved laterally and rotated freely before entering the playing field; and the player loses when any landed structure protrudes into this box, even for an instant. HATETRIS is unforgiving: if the player makes a line and loses simultaneously, the player loses.

Tetris always has 7 pieces, I, J, L, O, S, T and Z, but the various orientations of each piece are also subject to variation. Exactly what happens when a piece is rotated? In the naive case, the piece could rotate around its centre of gravity, but that would result in pieces positioned off-centre in many cases, and require floating point numbers. In HATETRIS, pieces rotate around the central point of a 4x4 bounding box, which can be done with integers, but this is unlike any other Tetris game and additionally results in the I, S and Z pieces having not 2 but 4 orientations each. So, for the sake of simplicity and making the process faster, I opted instead for the Original Rotation System. This cannot be procedurally generated, but generating the orientations by hand as bitmasks was trivial.

More recent versions of Tetris implement advanced functionality such as wall and floor kicks, which I omitted for the sake of optimisation. If there is obstruction or the piece would end up partially outside the well, a move or rotation is simply forbidden. As in HATETRIS, it is assumed for the sake of argument that there is no gravity (or, to put it another way, the player is at liberty to move the piece as far horizontally and rotate it as many times as she or he desires before it falls another row). To move down, the player presses "down". A piece locks when the player moves down, but meets an obstruction. Thus, there is effectively infinite "lock delay". Given the "free area" above the playing field, described above, there is also effectively infinite "entry delay".

1. If the player can always force one line, we say that the player wins.
2. If the AI can always force the player to lose with zero lines, we say that the AI wins.

Since 1 line is all that is needed, we can thankfully omit any procedures for restructuring the playing field after a line is made.

Ruling out some cases

The question is still meaningful for Tetris wells as narrow as 4 units wide. Any narrower and the I piece can no longer spawn or rotate. A standard Tetris well has a width of 10 and wider wells are of less interest to me.

If the 4-row "free space" at the top is kept, and we only count the lines below this point, the question is also meaningful for wells as shallow as 0 units, although in the cases of height 0, 1 and 2 the solutions are trivial: the AI always wins. (Height 0: the player cannot land a single piece without losing. Height 1: the AI supplies an O block. Height 2: the AI supplies a series of S blocks.)

One quick observation will save us all a great deal of time. By supplying an endless stream of O blocks, the AI can always win any Tetris well with an odd-numbered width: 5, 7 and 9. So, the widths of interest are 4, 6, 8 and 10 only.

If the player can force a line without losing in a well of depth *H*, the player can do the same in any deeper well by following exactly the same procedure. So, the real question is this: given a well of width *W*, what is the minimum required depth for the player to win? Alternatively, prove that the AI can always win no matter the depth - or at least to depth 20, if possible.

C (slight detour)

I took this as an opportunity to become familiar with a programming language which I knew very little about, even when I used it in university for my computing projects.

There is nothing that I can say about the C programming language that hasn't already been said thirty years ago. But it's all new to me, so just let me have this.

C is a monumental culture shock to anybody familiar with modern programming languages. C is so old that the canonical reference (Kernighan & Ritchie's *The C Programming Language*) describes it in terms of languages with which I have literally no familiarity, such as FORTRAN and Pascal. For example, the book carefully explains that the notation `a[i, j]` for accessing elements of multidimensional arrays - notation which I have *never* seen *anywhere* - is illegal. At the same time, K&R singularly fail to make clear that in C it is *impossible to choose the size of an array at run time*. Once this fact dawned on me it made life a lot easier and, in retrospect, it is obvious that dynamic arrays would be a feature only of dynamic programming languages, which C isn't. But the book was written before dynamic programming languages existed, or the world even had a word for them. Dynamic array sizing just wasn't there at the time, in the same way that nowadays you would never stop to explain that no, this car cannot fly.

It is enlightening and relaxing to finally find out where things like `printf()` and `fopen()` are actually invariably cribbed from; to find out what `malloc` actually is and why (see above) it is even necessary. It's valuable to learn the true power and practical use of memory pointers. It's cool to be finally exposed - if briefly, like radioactivity - to the layer where things like binary trees and linked lists are actually important to the process and need to be built manually.

It's also aggravating and frustrating. C's strict type safety is wonderful, except when it isn't. Its type declarations are confusing and strange. (why are the square brackets on the label, not on the data type? `int[10] foo` makes more sense than `int foo[10]`, doesn't it? I'm not crazy?) Handing pointers to functions around, yes! Null-terminated strings, erm, okay. Not checking array bounds? When would that *ever not* lead to a catastrophic error of some description? Here's another one: "Hey, what happens if you forget to `free` some of the memory you `malloc`ed?"

K&R deserve all the praise they have ever received for creating a language book which is so small that it is actually useful. It's the pocket Psalms and New Testament of system programming. Ever tried to use *Programming Perl* as a reference? "Dear Zarquon, Larry," I think to myself every time I try, "stop punning and get to the point!" But K&R also seem to have an unbecoming fetish for extreme terseness in their programs. Their persistent omission of braces is alarming. And I hope C isn't the first programming language you learn and K&R isn't the first programming language book you read, because unscrambling

```
while((s[i++] = t[j++]) != '0')
```

into something legible will take you a solid week, and it'll be a lifetime before anybody forces you to unlearn nonsense like

```
isdigit(s[++i] = c = getch())
```

. Never in my life have I thought that I would have to write this down, but **don't do more than three things simultaneously in one line of code!** I never imagined that there was anybody in the world who needed to be told that, but here we are. It's not even faster code! It's just the idiotic idiomatic style!

Method

The basic pseudocode for the brute force algorithm is this:

```
BOOLEAN player_can_always_force_win(well w) {
for each piece p {
for each landing state of piece p in well w {
well w2 = result
if(
that_makes_a_line(w2)
or player_can_always_force_win(w2)
) {
return TRUE
}
}
}
return FALSE
}

print player_can_always_force_win(empty_well)
```

Since it would be preferable to know *who* wins, and *how*, what I actually implemented is a little more complicated, and a full solution is returned in each case.

1. If the player wins, then the solution is a list of 7 "magic landing spots", one for each possible supplied piece. A magic landing spot need not immediately create a line. It may be part of a strategy to force a line later.
2. If the AI wins, then the solution is simply the identity of a single "killer piece". A killer piece need not immediately result in a player loss. It may be part of a strategy to force a player loss later.

The major optimisation breakthrough I made when planning this program was to observe that when you consider all 7 possible pieces in all 19 possible orientations, from left to right and top to bottom in the well, there are only approximately 4000 possible *states* which pieces can occupy during the course of any given game. These states form a network or directed graph, joined by edges representing player input. Starting from the original spawning state, pressing Left leads to one state, Right leads to another, as do Down and Rotate. If the wall is in the way, it is possible that pressing any given key will not move the piece, resulting in an edge leading from a state back to itself. There are in fact 7 disconnected graphs of states, one for each possible piece.

Having seen this, I wrote my program to generate every single possible state in a single procedure right at the start of execution, before beginning the brute force search. During this initial stage, all the tedious calculation of bit masks, piece rotation and piece dimensions and positions and offsets is handled and then *cached* in a single big array for future reference. Also cached are the edges connecting states together.

Pre-calculating all of this makes several things easier, such as detecting a collision between a piece and the contents of the well, inserting a piece into the well and removing the piece from the well afterwards. But fastest of all is locating new landing sites. By simply "painting" every state as "unreachable" and then starting from a single guaranteed "reachable" spawn point, I wrote what basically amounts to a "flood fill" routine which explores the whole state space looking for accessible states (and indirectly ruling out inaccessible states) and incidentally collecting a list of those states which happen to be landing points.

In the first few versions of this program, I ignored duplicate wells and solved every single possibility naively. This allowed me to calculate a solution for a large selection of wells. The longest solution (width 8, height 7, player wins) took some 9 hours to solve. However, I suspected that some wells were being solved over and over again using this method. At length, I rewrote the program to store the results for each well in turn. This resulted in a program which ran much faster. Unfortunately, it now rapidly rans out of memory when solving the larger wells. Unfortunately, the new program could not generate any new results.

Code

I used Fabrice Bellard's Tiny C Compiler to compile and run my program on Windows. I'm not in a position to compare this with other C compilers, but TCC is extremely small and fast and did the job, at a time when most of the people I asked were seriously suggesting that I switch to Linux just so that I

could run "Hello world".

The program:

- tetris4.c - Uses a very small amount of memory, can theoretically handle arbitrary well sizes. Slow, and doesn't give a complete result
- tetris6.c - Caches well results, runs like the blazes, pretty code, pretty output. Uses all your memory

The only two options you should really worry about are the `WIDTH` and the `HEIGHT` of the well. These are set using `#define` statements at the top of the code. Possibly this *doesn't* provide a substantial performance improvement over taking those numbers from the command line, but it was just how I implemented the configuration to begin with and I never got around to changing it.

I'm not a C expert. I'm pretty sure, though, that this program strikes a good balance between being readable and being high-performance. The most scope for improvement would be in a re-written, more efficient *algorithm*, with a focus on intelligence and saving memory, and less emphasis on brute force.

Results so far

As mentioned, for odd-numbered widths, the AI wins by supplying an endless stream of O blocks.

Who wins?		WIDTH				
		4	6	8	10	12+
HEIGHT	0	AI	AI	AI	AI	AI
	1	AI	AI	AI	AI	AI
	2	AI	AI	AI	AI	AI
	3	AI	AI	AI	AI	?
	4	AI	AI	AI	AI	?
	5	PLAYER	AI	AI	AI	?
	6	PLAYER	PLAYER	AI	?	?
	7	PLAYER	PLAYER	PLAYER	?	?
	8+	PLAYER	PLAYER	PLAYER	?	?

Results for (width, height) = (8, 6), (8, 7) and (10, 5) were computed using `tetris4.c`. `tetris6.c` will run out of memory.

All other results can be generated using either program - `tetris6.c` is recommended as it provides more informative output.

I am pleased to have completely solved Tetris for wells of width 4, 6, and 8 (and, trivially, 5, 7 and 9). I'm disappointed not to have been able to solve Tetris for width 10, but I remain firmly convinced that a player victory is inevitable there too.

What's next?

- I considered a method for avoiding having to "flood fill" the whole state space every time a new piece was added. Instead of painting every state as unreachable every time, we would keep the "reachability" attribute as a persistent attribute. First, we would also create and cache a list of "predecessor" states for each state. (In other words, the same digraph but with all the edge directions reversed.) For each state, we would also collect a list of other states overlapping that state.

  Then, whenever a new piece is locked into the well, we would mark that lock state *and any states listed overlapping it* as unreachable. Then, we could do a sort of smaller "reverse flood fill", looking for all states which have now become unreachable because all their predecessors are unreachable too. When the new piece is removed from the well, we mark its lock state and all the states listed as overlapping it as reachable again and perform a conventional flood fill to revert everything to the way it was.

  It's not clear whether this "flood fill / flood unfill" procedure would be any quicker than doing a conventional flood fill from the top every time. The "reverse flood fill" is much slower than a flood fill because every state needs checking multiple times. And just implementing the pre-caching of overlapping states *and nothing else* occupied much more memory and made my program take substantially longer to run.

- I realised that if the player can force a line *at row 2* (for example) in a well of width 4, then the player can simply do this twice side by side in a well of width 8. This would result in a complete line at row 2 spanning the whole well, and this would in turn result in a general solution for wells of all widths divisible by 4: 4, 8, 12, 16 and so on. If the same could be done for a well of width 6, then 4+6=10, 4+4+6=14, and so on, so we would have a general solution for the whole game!

  Unfortunately I ran some tests and proved that the AI, if so inclined, can deliberately prevent a line from forming at any specific row it so pleases, even while the player creates lines at other rows. This is achieved by supplying O and I blocks in particularly mean combinations. There might be some more meat in this approach but I can't get to it.

- There's limitless scope for smarter investigation of all the possibilities, and for being choosy about what to cache, freeing up solutions to solved wells when they're no longer needed in memory, and so on. But this isn't easily accomplished in C and it needs a better mathematician than me to implement. Frankly, I've lost the will to continue. If you discover any simple modification to the code which results in more results, particularly if you can prove that you can force a player victory for width 10 at any depth, please let me know. To say that Tetris has been "solved" would be awesomely cool.

- Although I remain convinced that the player can always force a win in a standard Tetris well, I'm strongly considering taking the optimised C code that was developed here and rolling it into a

newer, more hate-filled version of HATETRIS at some point in the future. I need a break right now though, so that won't happen immediately.

Original URL:

http://qntm.org/tetris